## **O que são Classes Abstratas?**

Classes Abstratas são um tipo especial de classe que impõe restrição: não é permitido instanciar objetos de uma classe dita abstrata. Isso parece pouco inteligente, afinal, para que serve uma classe que não pode gerar objetos? A questão é que, em certos casos, queremos restringir os tipos de objetos que podem existir no nosso programa.

Além da possibilidade de criarmos objetos a partir da superclasse Pessoa, o fato de podermos ter objetos do tipo Aluno e Professor, sendo apontados por referências de Pessoa, é bastante útil para trabalharmos com métodos herdados dessa superclasse. Mas existem casos em que precisamos limitar que tipo de objeto pode existir dentro do sistema.

Por exemplo, se para um sistema que utiliza as classes citadas no exemplo apresentado só forem previstas entidades do tipo Aluno e Professor, ou seja, todos os objetos devendo, necessariamente, ser de um desses tipos, não faz sentido para o sistema ter um objeto do tipo Pessoa sendo instanciado, pois, por ele ser genérico demais, não representa algo que o sistema deva manusear. Para o sistema, não existem pessoas genéricas, apenas Alunos e Professores. Essa é uma condição lógica do programa, chamada Regra de Negócio, a qual pode ser facilmente violada com um trecho de código, como o do exemplo a seguir (observe na próxima página).



**Saiba mais!**

Em Java, assim como em outras linguagens, é possível utilizar alguns caracteres curinga, afim de facilitar a formatação de textos nos Consoles. Eles são interpretados como um único caractere, de acordo com a tabela ASCII e realizam algumas tarefas comuns de formatação. Os mais utilizados são: ‘\n’ – quebra de linha; ‘\t’ – tabulação na mesma linha

## **Como criar Classes Abstratas?**

Para fazer com que uma classe se torne uma classe abstrata, basta adicionar a palavra-chave abstract antes da palavra-chave class. A partir desse momento, a classe em questão se torna abstrata e não pode instanciar objetos diretamente. Assim, para o primeiro exemplo mostrado, se fizermos com que a classe Pessoa se torne uma classe abstrata, não poderíamos criar um novo objeto a partir dela. A tentativa de criar um objeto do tipo Pessoa resultaria em um erro apontado pelo compilador, como mostra o exemplo:

public abstract class Pessoa {

private String nome;

}

Agora que conhecemos o conceito de classes abstratas, podemos utilizar a nomenclatura formal para separá-las das demais. Dizemos que uma classe marcada com a palavra-chave abstract é uma classe abstrata e que todas as outras classes são concretas, as únicas que permitem a instanciação de objetos.  
  
 Quando modificamos a classe Pessoa tornando-a abstrata, estamos explicitamente proibindo a instanciação de objetos do tipo Pessoa e, portanto, garantindo a Regra de Negócio, que diz que não deve haver objetos que não sejam, nesse caso, do tipo Aluno ou Professor no sistema. Essa é a principal função do uso de classes abstratas. Classes abstratas não devem ter "vida", exceto no caso em que elas são estendidas, ou seja, possuem subclasses concretas. Observe o diagrama a seguir:  
  
 Exemplos de classes abstratas e concretas.  
 A imagem mostra um diagrama formado por cinco retângulos. Na parte inferior, temos três retângulos dispostos lado a lado, que trazem uma palavra no seu interior, da esquerda para a direita, respectivamente, temos escrito: Triângulo, Quadrilátero e Pentágono. De cada retângulo sai uma seta que aponta para outro logo acima, que traz a palavra Polígono. Uma seta sai desse retângulo e aponta para outro acima, que traz a palavra FormaGeometrica.

   
  
 Para o exemplo da figura, apenas as classes Triangulo, Quadrilatero e Pentagono representam entidades que podem ser instanciadas no sistema. Poligono e FormaGeometrica são classes abstratas e não podem gerar objetos diretamente, por não representarem uma forma definida, por exemplo.

## **Métodos Abstratos**

Uma vez que uma classe é dita abstrata, ela pode conter métodos abstratos. Um método abstrato não tem um corpo definido e existe para ser implementado em subclasses concretas de uma classe abstrata. Na verdade, todos os métodos marcados como abstratos devem ser, obrigatoriamente, implementados em uma subclasse concreta que herda da classe abstrata.

public abstract class Poligono extends FormaGeometrica {

private int quantidadeLados;

public abstract double area();

}

Baseado no que foi dito anteriormente, vamos imaginar uma classe Quadrado que estende a classe Poligono. Nesse caso, a classe Quadrado é obrigada a conter uma implementação do método area. O próximo exemplo apresenta a implementação da classe Quadrado, que é subclasse da classe Poligono.

É importante dizer que métodos abstratos são opcionais em uma classe abstrata, ou seja, uma classe abstrata pode conter métodos concretos. Além disso, não é possível definir construtores abstratos, uma vez que eles não são herdados, logo, nunca poderiam ser implementados.

public class Quadrado extends Poligono {

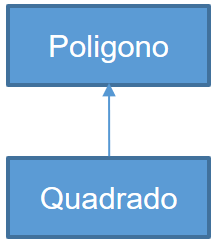
private double lado;

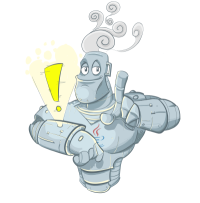
public double area(){

return lado\*lado;

}

}





Agora, para aplicar o que aprendeu, tenha como base os exemplos apresentados neste tópico e pratique-os no ambiente de programação da plataforma. Lembre-se de que a teoria e a prática caminham juntas e o exercício é muito importante. Por isso, não deixe de praticar os exemplos desta aula. Se preferir, também pode criar e praticar seus próprios exemplos referentes aos assuntos estudados aqui.

## **O que são Interfaces?**

Interfaces são um tipo especial de classe completamente abstrata. Elas permitem que criemos um conjunto de comportamentos, ou seja, métodos que desejamos que façam parte de nossas classes, mas que sejam independentes de uma hierarquia de heranças de classes. Dizemos que uma classe X implementa uma interface I quando relacionamos X e I através da palavra-chave implements, como será apresentado em breve.

As Interfaces são definidas pela palavra-chave interface e constituídas de métodos sempre públicos (mesmo que estes não possuam o modificador public, serão públicos) e sem corpo definido. Isso quer dizer que elas definem uma série de métodos que as classes que as implementarem terão, mas sem dizer como esses métodos trabalharão. O corpo dos métodos sempre terá de ser definido nas classes concretas que as implementarem. Observemos o exemplo a seguir:

public [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) Organizavel {

void ordemCrescente();

void ordemDecrescente();

String listarItens();

}

No exemplo anterior, temos a [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) Organizavel. Uma boa prática é sempre definir os nomes de uma [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) com uma palavra que indique capacidade, que será injetada nas classes que implementarem essa [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface). No caso, Organizavel tenta nos dizer que uma classe que implementá-la será, de alguma forma, organizável. Essa classe deverá definir corpo, ou seja, implementar os métodos ordemCrescente, ordemDecrescente e listarItens de acordo com as suas necessidades. A classe Lista, no exemplo a seguir, é um modelo de implementação da [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) Organizavel.

public class Lista implements Organizavel {

int numeros[];

boolean continuar;

public void ordemCrescente(){

// Organiza os elementos do vetor

// numeros em ordem crescente

continuar = true;

while(continuar == true){

continuar = false;

for(int i=0; i<(numeros.length-1); i++){

if(numeros[i] > numeros[i+1]){

int variavelAuxiliar = numeros[i+1];

numeros[i+1] = numeros[i];

numeros[i] = variavelAuxiliar;

continuar = true;

}

}

}

}

public void ordemDecrescente(){

// Organiza os elementos do vetor

// numeros em ordem decrescente

continuar = true;

while(continuar == true){

continuar = false;

for(int i=0; i<(numeros.length-1); i++){

if(numeros[i] <= numeros[i+1]){

int variavelAuxiliar = numeros[i+1];

numeros[i+1] = numeros[i];

numeros[i] = variavelAuxiliar;

continuar = true;

}

}

}

}

public String listarItens(){

String itens = "";

for(int i=0; i<numeros.length; i++)

itens = numeros[i] + "" ;

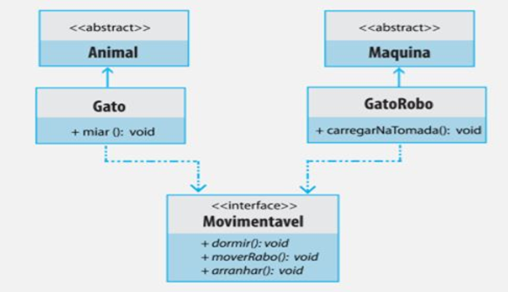
return itens;

}

}

Dentro da classe Lista, temos um vetor de inteiros chamado numeros. Implementamos a [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) Organizavel utilizando a palavra-chave implements, logo após o nome da classe, seguido do nome da [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface). Então, para efetuar os trabalhos de organização, implementamos, sobre o vetor números, cada um dos três métodos listados na [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface). Os métodos ordemCrescente e ordemDecrescente utilizam o [algoritmo Bolha](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=algoritmo%20Bolha) com [flag](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=flag) para colocar os valores do vetor em ordem. O método listaItens monta um String com a lista de todos os valores do vetor.

Na aula passada, comentamos que Java não permite herança múltipla, como a linguagem C++. O uso de interfaces complementa o recurso de herança da linguagem Java, permitindo que classes não relacionadas por herança compartilhem comportamentos.



Nesse caso, temos as classes Gato e GatoRobo que implementam a mesma [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) Movimentavel, que lhes fornece os métodos dormir, moverRabo e arranhar. Assim, as classes Gato e GatoRobo possuem esses métodos, os quais elas não poderiam herdar de uma segunda classe, uma vez que cada uma já possui uma superclasse e, de outra forma, só seria possível se as classes Animal e Maquina já possuíssem esses métodos.

Não existe um limite de quantidade para a implementação de interfaces por parte de uma classe. Isso significa que uma mesma classe pode implementar várias interfaces, enquanto só pode estender (ser subclasse) de apenas uma superclasse. Para implementar várias interfaces, basta listar os seus nomes após a palavra-chave interface, separando-as por vírgula, como no exemplo a seguir.

public class Triangulo implements Desenhavel, Imprimivel, Calculavel {

// Triangulo deve implementar os métodos Desenhavel, Imprimivel e Calculavel.

}

## **Características das interfaces**

Interfaces podem estender outras interfaces, ou seja, você pode utilizar a palavra-chave extends para construir interfaces a partir de outras. Observe o seguinte exemplo.

public [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) InterfaceA {

int metodoX();

int metodoY();

}

public [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) InterfaceB extends InterfaceA {

void metodoZ(int param1; int param2);

}

Para o exemplo mostrado, uma classe que implementar InterfaceB terá de definir o corpo de metodoX, metodoY e metodoZ. Logo, referências de interfaces podem apontar para objeto de classes concretas que as implementam. Isso facilita o manuseio de listas de objetos que possuem uma determinada interface em comum. Por exemplo: se as classes Ferro, Bronze e Prata implementam a interface Moldavel, podemos ter uma array do tipo Moldavel com objetos do tipo Ferro, Bronze e Prata dentro, todos juntos.

Além de métodos, as Interfaces ainda podem definir constantes que serão acessíveis nas classes que as implementarem como membros locais. É possível definir uma interface sem métodos a serem implementados, apenas com constantes para servirem de parâmetro para outras classes. O uso da palavra-chave final é opcional, assim, todos que não são métodos definidos em uma interface são considerados constantes e não podem ser modificados nas classes que as implementam. Observe o exemplo de declaração de constantes a seguir.

public [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) InterfaceX{

public final int valorReferencia = 50;

public final String categoria = "LQSDK";

}

public class TesteInterface implements InterfaceX {

public static void main(String args[]){

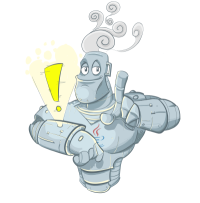
System.out.println("Valor de referência -" +valorReferencia);

System.out.println("Categoria" +categoria); } }

**Interfaces importantes da API Java**

A API (Application Programming [Interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Interface)) Java possui uma grande quantidade de interfaces que ajudam no desenvolvimento dos nossos programas. Utilizaremos muitas delas durante o desenvolvimento de interfaces gráficas, ou seja, programas com janelas, botões, caixas de texto etc. Observe algumas interfaces interessantes:

* java.lang.Cloneable: permite a clonagem de um objeto. Isso significa criar uma nova instância do seu mesmo tipo, copiando os seus valores para um novo objeto, um a um. Uma classe que implementa a [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) Cloneable deve sobrescrever o método clone.
* java.lang.Comparable: permite que possamos comparar objetos de forma especializada. Uma classe que implementa a [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) Comparable deve sobrescrever o método compareTo, que recebe como parâmetro uma referência de um objeto do mesmo tipo da classe. Dentro do método compareTo, pode-se utilizar uma lógica personalizada para efetuar a comparação entre objetos e informar qual, entre eles, é o maior, o menor ou se eles são idênticos.
* java.awt.event.\*: muitas interfaces desse pacote, como ActionListener, MouseListener e KeyListener, fazem papel de “observadores”, ou seja, ficam atentas a eventos relacionados à [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) gráfica, como cliques, teclas pressionadas no teclado, movimento de janela, entre outros. Quando um evento desse tipo ocorre, temos condição de tratá-lo nas nossas classes dentro de alguns dos métodos da [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) que estejam sobrescritos.



Você pode consultar a API Java através do link:<http://docs.oracle.com/javase/7/docs/api/> e ler um pouco mais sobre essas interfaces.

A seguir, vamos observar um exemplo de utilização da [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) Comparable. Imaginemos uma classe Quadrado que possua uma medida de lado e um método para cálculo de área. Para que essa classe implemente a [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) Comparable, precisamos sobrescrever o método compareTo. O objetivo do método compareTo é receber um objeto do tipo Quadrado e fazer uma comparação dele com o objeto que invocou o método. No caso, vamos fazer a comparação de áreas para verificar se um objeto é maior ou menor que o outro, como mostrado neste exemplo:  
  
public class Quadrado implements Comparable{

private double lado;

public Quadrado(double lado){

this.lado = lado;

}

public double area(){

return lado\*lado;

}

public int compareTo (Quadrado q){

double diferencaDeAreas = this.area() - q.area();

if(diferencaDeAreas > 0)

diferencaDeAreas = 1;

else if(diferencaDeAreas < 0)

diferencaDeAreas = -1;

return (int)diferencaDeAreas;

}

}

public class TesteComparacao{

public static void main(String[] args){

Quadrado q1 = new Quadrado(10);

Quadrado q2 = new Quadrado(20);

int resultado = q1.compareTo(q2);

switch(resultado){

case 0: System.out.println(“q1 e q2 têm a mesma área ”); break;

case 1: System.out.println(“q1 é maior que q2 ”); break;

case -1: System.out.println(“q1 é menor que q2 ”); break;

}

}

}

Como você pôde observar, dentro do método compareTo, calculamos a diferença entre as áreas dos dois quadrados. Se os quadrados possuírem a mesma área, o valor atribuído será 0 (zero). Se o quadrado representado pelo objeto que chamou o método for maior que o outro, o valor atribuído será 1 (um). Se o quadrado representado pelo objeto que chamou o método for menor que o outro, o valor atribuído será -1 (1 negativo).

O retorno do método compareTo precisa apenas ser um valor inteiro. Mas é normal parametrizar os valores que ele pode devolver. Normalmente, mantém-se o padrão de que, se o retorno for igual a 0 (zero), significa que os objetos podem ser considerados idênticos. Para o exemplo anterior, a área de q1 será igual a 100, enquanto a área de q2 será igual a 400. Logo, a mensagem “q1 é menor que q2” será exibida.

As interfaces têm um papel importante na programação orientada a objetos. Elas permitem que classes que não possuem relação de herança compartilhem métodos e constantes que, por sua vez, trazem certos benefícios da herança, como polimorfismo e sobrecarga. Além disso, como uma classe pode implementar uma quantidade ilimitada de interfaces, podemos contornar a opção da linguagem Java de não permitir herança múltipla.



**Saiba mais!**

Quando declaramos a classe Quadrado, você deve ter percebido uma nova estrutura na própria declaração. Fizemos algo como “” junto ao nome da [interface](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=interface) Comparable. Esta estrutura circundada por < e > chamamos de Generics, que é utilizada como forma de especificar qual o tipo de dado uma classe utiliza, ao invés de generalizar um processamento. Aprenderemos mais sobre Generics posteriormente.

**Algoritmo Bolha**

É uma das metodologias utilizadas para ordenamento de listas. O algoritmo bolha consiste em comparar um elemento com seu antecessor e sucessor e realizar a troca de posição entre eles, dependendo da condição de ordenamento, crescente ou decrescente.