## **O que é** [**Polimorfismo**](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo)**?**

O [Polimorfismo](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo) pode ser interpretado como a possibilidade de um grupo possuir componentes com características diferentes para o mesmo comportamento, ou seja, com "várias formas". Os animais que voam podem ser um exemplo disso, pois cada um voa de uma forma diferente. Em Programação Orientada a Objetos, [Polimorfismo](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo) é a ocorrência de comportamento especializado para classes que derivam de um mesmo tipo (superclasse), de modo que a manipulação de objetos dessa hierarquia ocorra de maneiras diferentes ao receber uma mesma mensagem (método).

Vamos entender melhor este conceito a partir de um exemplo. Suponha a classe Animal, que possui as subclasses SerHumano, Cavalo e Sapo. Observe, no exemplo a seguir, a definição de cada uma dessas classes.

public class Animal {

public void moverSe(int qtAndar){

System.out.println("Sem forma de " +

"mover-se definida. Utilize " +

"um animal específico...");

}

}

public class Cavalo extends Animal {

public void moverSe(int qtAndar){

System.out.println("Cavalo galopando " +

"de quatro patas ");

for(int i = 0; i<=qtAndar; i++ ){

System.out.println("Galope "+i);

}

}

}

public class SerHumano extends Animal {

public void moverSe(int qtAndar){

System.out.println("Ser Humano andando a pé ");

for(int i = 0; i<=qtAndar; i++ ){

System.out.println("Passo "+i);

}

}

}

public class Sapo extends Animal {

public void moverSe(int qtAndar){

System.out.println("Sapo pulando e agachando");

for(int i = 0; i<=qtAndar; i++ ){

System.out.println("Pulo "+i);

}

}

}

O exemplo apresentado ilustra a ocorrência de sobrescrita (override). Observe que o método moverSe foi definido na superclasse Animal e cada uma das subclasses tem uma implementação diferente para o método moverSe.

Agora vamos criar uma classe TesteAnimais para instanciar objetos a partir de Animal e suas subclasses e invocar o método moverSe, como no exemplo a seguir.

public class TesteAnimais {

2 public static void main(String[] args){

3 // Instanciando um objeto a partir de uma classe

4 Animal a1 = new Animal();

5

6 //Instanciando objetos de subclasses para referência de superclasse

7 Animal a2 = new SerHumano();

8 Animal a3 = new Cavalo();

9 Animal a4 = new Sapo();

10

11 a1.moverSe(2);

12 a2.moverSe(2);

13 a3.moverSe(2);

14 a4.moverSe(2);

15 }

16 }

Perceba que a1 é uma referência do tipo Animal, que referencia um objeto também do tipo Animal. Nesse caso, a chamada feita ao método moverSe, a partir de a1, na linha 11, constitui uma ocorrência de ligação precoce, pois a relação entre a chamada do método e o método que será chamado é feita em tempo de compilação.

Note também que a2, a3 e a4 são referências do tipo Animal, porém os objetos referenciados por a2, a3 e a4 são de tipos diferentes (a2 referencia um objeto do tipo SerHumano, enquanto a3 referencia Cavalo e a4 referencia Sapo). Nesse caso, as chamadas feitas ao método moverSe, a partir de a2, a3 e a4, nas linhas 12, 13 e 14, constituem ocorrências de ligação tardia, pois a relação entre a chamada do método e o método que será chamado é feita em tempo de execução.

01 public class TesteAnimais {

02 public static void main(String[] args){

03 // Definição de referência de superclasse

04 Animal a;

05 System.out.println("1 - Ser Humano\n2 - Cavalo \n3 - Sapo");

06 int opc = 1;

07 System.out.println("Opção escolhida: " + opc);

08 // Instanciando objeto de uma das subclasses para referência da superclasse

09 if(opc == 1)

10 a = new SerHumano();

11 else

12 if(opc == 2)

13 a = new Cavalo();

14 else

15 if(opc == 3)

16 a = new Sapo();

17 else

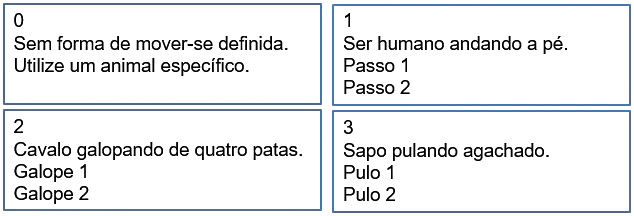
18 a = new Animal();

19 a.moverSe(2);

20 }

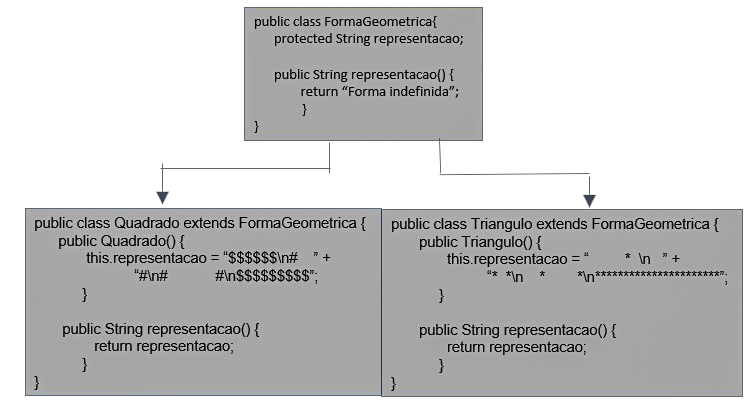
21 }

Vamos analisar o código da página anterior por partes. Inicialmente, definimos os atributos que iremos precisar para o nosso código. São eles: uma referência do tipo Animal a, uma instância de Scanner, chamada entrada, e um atributo, chamado opc. Em seguida, o programa solicita a entrada de dados para opc, que será utilizada para decidir o tipo de animal que irá mover-se (1- Ser Humano, 2- Cavalo ou 3- Sapo).  
  
 No código entre as linhas 15 e 22, temos decisões baseadas no atributo opc, que instanciam um objeto específico, baseado no valor de opc. Na linha 24, o método moverSe é chamado a partir de a; porém, em tempo de compilação, não podemos identificar o comportamento resultante dessa chamada. Somente em tempo de execução, após a digitação do valor de opc, é que será decidido o tipo de objeto instanciado e, consequentemente, o método que será executado na chamada a moverSe da linha 24.  
  
 Nesse caso, temos um exemplo de [Polimorfismo](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo) envolvendo o método moverSe, que está sendo chamado uma única vez e que, em tempo de execução, pode resultar em várias formas.



Perceba que, se digitarmos 1, o resultado é a execução do método moverSe de SerHumano. Se digitarmos 2, o resultado é a execução de moverSe de Cavalo. Se digitarmos 3, o resultado é a execução de moverSe de Sapo. Para qualquer outro valor digitado, o resultado é a execução de moverSe de Animal. Lembrando mais uma vez que temos uma única chamada na classe de execução para o método moverSe; isso é, portanto, um exemplo da concretização do conceito de [Polimorfismo](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo).

Vamos analisar mais um exemplo de [Polimorfismo](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo). Inicialmente, será apresentada a hierarquia de classes, envolvendo formas geométricas:



Para esse exemplo, temos uma classe chamada FormaGeometrica como superclasse de Triangulo e Quadrado. Cada subclasse define a sua representação de acordo com seu tipo, utilizando caracteres de texto, como asterisco (\*) e tralha (#).

Nas páginas anteriores, mostramos uma maneira de fazer [Polimorfismo](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo), no entanto, há outras formas de alcançarmos [Polimorfismo](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo). Mostraremos a ocorrência de [Polimorfismo](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo), de forma diferente, envolvendo as classes FormaGeometrica, Quadrado e Triangulo apresentadas anteriormente. Neste exemplo, criaremos o método desenhar, que executará a chamada polimórfica do método representacao.

import java.util.Scanner

2

3 public class TesteFormas {

4 public static void desenhar(FormaGeometrica f){

5 // chamada polimórfica

6 System.out.println(f.representacao());

7 }

8

9 public static void main(String[] args){

10 // Definindo os atributos necessários

11 Scanner entrada = new Scanner(System.in);

12 int opc;

13

14 System.out.println("Digite a opção de figura desejada: \n - Quadrado" +"\n - Triangulo");

15 opc = entrada.nextInt();

16

17 // Instanciando objetos de subclasses para referências da superclasse

18 if(opc == 1)

19 desenhar(new Quadrado());

20 else if(opc == 2)

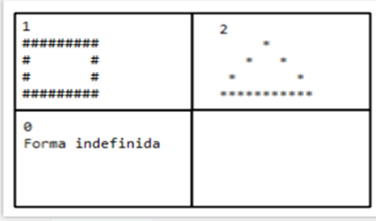
21 desenhar(new Triangulo());

22 else

23 desenhar(new FiguraGeometrica());

24 }

25 }

Se executarmos o código da classe TesteFormas, ocorrerá um dos cenários mostrados nesta figura:  
 Figura 1: [Polimorfismo](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo) utilizando as formas geométricas. 

Repare que, se digitarmos 1, o resultado é a execução do método representação de Quadrado. Se digitarmos 2, o resultado é a execução de representação de Triangulo. Para qualquer outro valor digitado, o resultado é a execução do método representação de Forma Geometrica. Lembrando, mais uma vez, que temos uma única chamada na classe de execução para o método representação, isso é, portanto, um exemplo da concretização do conceito de [Polimorfismo](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo).

Cabe lembrar que este é um conceito independente de linguagem de programação e que está intimamente ligado à herança, uma vez que só podemos sobrescrever um método se ele for herdado de uma superclasse. No tópico 2, iremos conferir como obter vantagens do uso de Polimorfismo em classes Java.

## **A** [**aplicação**](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=aplica%C3%A7%C3%A3o) **de** [**Polimorfismo**](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo) **em classes Java**

Vamos imaginar um jogo em Java, no qual um personagem principal deve atravessar um cenário cheio de inimigos. No jogo, existem diferentes inimigos que podem atacar aleatoriamente o personagem principal, enquanto ele percorre o cenário. Uma das abordagens que podemos usar para implementar essa funcionalidade do jogo é termos alguns vetores contendo objetos que representam inimigos e, de tempos em tempos, “sorteamos” um deles para executar o método ataque, como podemos conferir no exemplo a seguir:

01 public class Jogo{

02 public static void main(String[] args){

03

04 Personagem heroi = new Personagem();

05

06 Aranha [ ] aranhas = new Aranha[50];

07 Cobra [ ] cobras = new Cobra[50];

08 Leao [ ] leoes = new Leao[50];

09

10 //Neste local os vetores de inimigos devem ser populados

11 while(heroi.vivo()){

12

13 int tipoInimigo = (int)(Math.random()\*10)%3;

14 if(tipoInimigo == 0){

15 int qualInimigo = (int)(Math.random()\*10)%aranhas.length;

16 aranhas[qualInimigo].atacar();

17 } else if(tipoInimigo == 1){

18 int qualInimigo = (int)(Math.random()\*10)%cobras.length;

19 cobras[qualInimigo].atacar();

20 } else if(tipoInimigo == 2){

21 int qualInimigo = (int)(Math.random()\*10)%leoes.length;

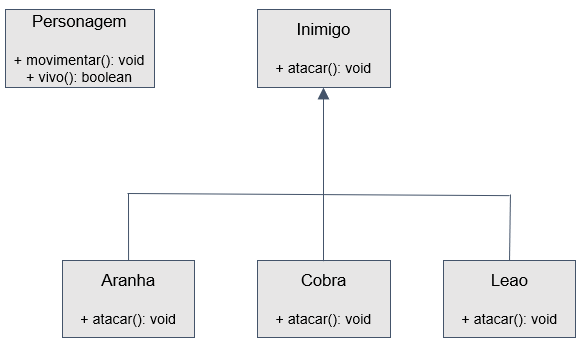
22 leoes[qualInimigo].atacar();

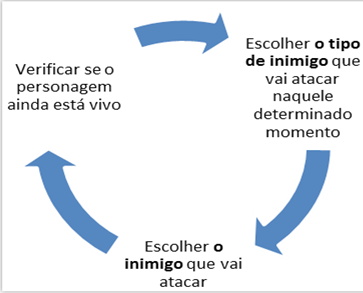
23 }

24 }

25 }

26 }

Para compreender melhor o código apresentado anteriormente, confira, a seguir, como representamos a hierarquia de heranças das classes do jogo:  
 

Ciclo de seleção de inimigos para ataque.   
 A figura 2 representa o ciclo do laço while, descrito dentro do método principal da classe Jogo (omitimos a criação dos objetos dos tipos Aranha, Cobra e Leao). Temos que gerar dois números aleatórios, um para escolher o tipo de inimigo que vai atacar e outro para escolher o inimigo a cada rodada. Isso acontece porque temos três vetores de tipos diferentes de inimigos.  
  
 E se precisássemos de 10 tipos de inimigos diferentes?  
  
 Para solucionarmos problemas desse tipo, podemos aplicar o [Polimorfismo](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo) de forma a diminuir a quantidade de vetores de inimigos e escrever o laço while do método principal de forma mais enxuta. Se as classes Aranha, Cobra e Leao são subclasses de Inimigo, todas elas “são” do tipo Inimigo e possuem o método atacar(). Então podemos colocar todos os objetos que representam inimigos dentro de um mesmo vetor, mas que seja de um tipo genérico para todos eles: um vetor do tipo Inimigo. A nossa mudança deixará o código mais simples, como neste exemplo:

01 public class Jogo {

02 public static void main(String[] args) {

03 Personagem heroi = new Personagem();

04 int cont = 0;

05 Inimigo inimigos[] = new Inimigo[150];

06 //Preenchendo aleatoriamente o vetor de inimigos

07 while(cont < 150) {

08 int tipoInimigo = (int)(Math.random()\*10)%3;

09 if(tipoInimigo == 0)

10 inimigos[cont] = new Aranha();

11 else if(tipoInimigo == 1)

12 inimigos[cont] = new Cobra();

13 else if(tipoInimigo == 2)

14 inimigos[cont] = new Leao();

15 else

16 inimigos[cont] = new Inimigo();

17 cont++;

18 }

19 // Execução Polimórfica do método atacar

20 while(heroi.vivo()) {

21 int qualInimigo = (int)(Math.random()\*100)%inimigos.length;

22 inimigos[qualInimigo].atacar();

23 }

24 }

25 }

Reduzimos o laço while de 13 para 2 linhas, sem perda de funcionalidades, aproveitando uma propriedade da herança. Um vetor do tipo Inimigo pode receber objetos das classes Aranha, Cobra e Leao, pois eles são derivados, ou seja, especializações da classe Inimigo. Assim, simplificamos o [código-fonte](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=c%C3%B3digo-fonte), mantendo apenas um vetor para armazenar todos eles e, pelo fato de todos possuírem o método atacar(), não precisamos da estrutura de decisão if/else encadeada para decidir que tipo de objeto iremos manusear. Essa ação de invocar métodos de uma subclasse, a partir de uma referência da superclasse, se chama [Downcasting](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Downcasting). O [Downcasting](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Downcasting) já foi utilizado quando abordamos o conceito de ligação tardia no tópico 1 desta aula. Para esclarecer, [Downcasting](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Downcasting), em POO, é o ato de criar uma referência de uma superclasse para uma de suas subclasses. Você faz um [Downcasting](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Downcasting) quando quer "descer" na árvore de herança. Vamos conferir este exemplo:

01 public class Cachorro {

02 // Conteúdo da classe Cachorro

03 }

04

05 public class PastorAlemao extends Cachorro {

06 // Conteúdo da classe PastorAlemao

07 }

08

09 public static void main(String[] args){

10

11 /\*

12 \* Cachorro é a classe pai de PastorAlemao. O objeto

13 \* cachorro receberá um valor do tipo PastorAlemao;

14 \*/

15 Cachorro cachorro = new PastorAlemao();

16 /\*

17 \* O trecho abaixo é sempre permitido e sem restrições, já que o

18 \* objeto instanciado pastoralemao é do tipo de uma subclasse da

19 \* classe do objeto Cachorro.

20 \*/

21 PastorAlemao pastoralemao = (PastorAlemao) cachorro;

22 }

Perceba que essa é a forma com a qual o [Polimorfismo](https://leadfortaleza.com.br/ead2pcd/app/glossario?termo=Polimorfismo) nos ajuda a simplificar nossas classes. Aproveitando-se da propriedade de que uma referência de uma superclasse pode apontar para objetos de qualquer uma das suas subclasses, economizamos em linhas de código para representar instâncias das subclasses. Outro fator positivo é que subclasses necessariamente herdam métodos públicos e protegidos das superclasses, consequência da herança, e, portanto, é possível invocar de forma homogênea esses métodos para objetos de subclasses a partir de referências da superclasse, como fizemos anteriormente.  
  
 Existem situações, porém, nas quais precisamos distinguir o tipo de objetos dentro de um vetor, sem abrir mão do fato de esse vetor ser genérico, ou seja, que receba objetos de tipos diferentes. Saberemos como fazer isso a seguir.

## **O operador instanceof**

Imagine que, para o jogo exemplificado, nós queiramos que a cor do jogo mude, caso haja um inimigo do tipo Cobra atacando. A gerência das cores do jogo não é responsabilidade do objeto do tipo Cobra, por isso deve ser feito fora do método atacar() dele. Mas, se temos um vetor único com todos os objetos do tipo Aranha, Cobra e Leao “misturados”, como vamos saber qual é qual?

Para fazer esse teste de tipo de objeto, usaremos um operador especial instanceof. Ele é um operador binário, como os operadores de soma, subtração, igualdade entre outros, e retorna um valor booleano, ou seja, verdadeiro ou falso. O operador pode ser usado como no modelo a seguir:

objeto instanceof Classe

O operador instanceof é normalmente utilizado em condicionais (if). Se o objeto antes do operador é do mesmo tipo da classe mencionada após o operador, a expressão resultará verdadeira. Assim, podemos implementar o novo comportamento proposto anteriormente, como no exemplo a seguir:

01 public class Jogo{

02 public static String cor;

03 public static void main(String[] args){

05 Personagem heroi = new Personagem();

07 Inimigo inimigos[ ] = new Inimigo[150];

09 //neste local, o vetor de inimigos deve ser populado de forma semelhante à figura 11

11 while(heroi.vivo()){

13 int qualInimigo = (int)(Math.random()\*100)%inimigos.length;

14 inimigos[qualInimigo].atacar();

16 if(inimigos[qualInimigo] instanceof Cobra)

17 corJogo(“Verde”);

18 }

19 }

22 public static void corJogo(String novaCor){

23 cor = novaCor;

24 }

25 }

26 }