

Programação Orientada a Objetos

Aula 09 - Polimorfismo

Apresentação

Hoje, iremos iniciar o aprendizado do último princípio que serve de base para a Programação Orientada a Objeto – o Polimorfismo. Esse princípio também aposta na ideia da reutilização para facilitar o dia a dia da programação. Ele é também bastante importante tanto para o entendimento de programas OO em Java e outras linguagens, como também é um mecanismo bastante sofisticado para permitir a reutilização e flexibilidade durante o desenvolvimento de tais programas.



Vídeo 01 - Apresentação

Objetivos

Ao final desta aula, você será capaz de:

- Entender o princípio do polimorfismo.
- Saber os tipos existentes de polimorfismo.
- Conhecer polimorfismo de sobrecarga, de sobreposição e de inclusão.
- Saber o que é conversão de tipos.

Polimorfismo

O polimorfismo deriva da palavra polimorfo, que significa multiforme, ou que pode variar a forma. Para a POO, polimorfismo é a habilidade de objetos de classes diferentes responderem a mesma mensagem de diferentes maneiras. Ou seja, **várias formas** de responder à mesma mensagem. Veja a figura a seguir para entender onde se localiza o pilar do polimorfismo dentro da Programação Orientada a Objetos.

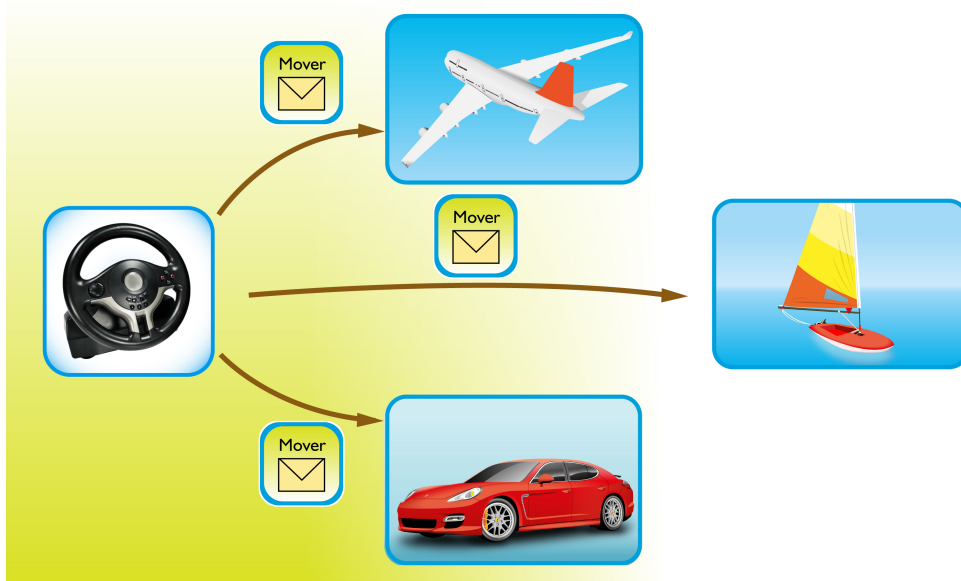
Figura 01 - Pilares da POO



Vejamos o seguinte exemplo: um dono de uma fábrica de brinquedos solicitou que seus engenheiros criassem um mesmo controle remoto para todos os brinquedos de sua fábrica. A única restrição era que cada brinquedo atendesse aos comandos específicos definidos pelo controle.

O controle remoto teria vários botões, sendo que todos eles seriam úteis para todos os brinquedos. Assim, quando o usuário clicasse no botão mover, o controle enviaria o sinal MOVER para todos os brinquedos que estivessem no raio de dois metros. A Figura 2 ilustra tal situação.

Figura 02 - Exemplo de polimorfismo



Assim, quando o brinquedo recebe o sinal MOVER, ele se move de acordo com a sua função. Para o avião, mover significa VOAR, para o barco significa NAVEGAR, e para o automóvel CORRER. Observe que os brinquedos respondem ao mesmo sinal de formas diferentes. Temos aqui então um caso de polimorfismo.



Vídeo 02 - Conceito de Polimorfismo

O **Polimorfismo** permite que diferentes objetos (avião, barco, automóvel) respondam a uma mesma mensagem (mover) de formas diferentes (voar, navegar e correr).

Atividade 01

1. Para avaliar seu entendimento, descreva em seu caderno de anotações uma situação em que aconteça o polimorfismo.

Tipos de Polimorfismo

O Polimorfismo pode ser classificado de três maneiras:

- Polimorfismo de sobrecarga
- Polimorfismo de sobreposição
- Polimorfismo de inclusão

Polimorfismo de Sobrecarga



Vídeo 03 - Sobrecarga de Polimorfismo

Polimorfismo de sobrecarga permite que um método de determinado nome tenha comportamentos distintos, em função de diferentes parâmetros (1) que ele recebe. Cada método difere no **número** e no **tipo** de parâmetros.

Exemplo

Considere uma classe **Maior** capaz de calcular e retornar o maior entre dois números de tipos diferentes. A Listagem 1 ilustra o código de tal classe.

(1) Lembrando: parâmetros são as variáveis que estão dentro dos parênteses na declaração do método.

```
1 public class Maior{
2     private int xInt;
3     private int yInt;
4     private float xFloat;
5     private float yFloat;
6     private double xDouble;
7     private double yDouble;
8
9     public int calcMaior(intx, int y){...}
10    public float calcMaior(float x, float y){...}
11    public double calcMaior(double x, double y){...}
12    public int calcMaior(double a, double b){...} //ERRO!
13 }
```

Listagem 1 - Polimorfismo de sobrecarga com tipos de parâmetros diferentes

Veja no exemplo uma aplicação prática do uso do polimorfismo de sobrecarga. Observe que temos vários métodos com o mesmo nome, no caso `calcMaior()`, cujo objetivo é indicar qual é o maior dentre dois números.

O que diferencia cada um deles é o tipo de parâmetros. No primeiro método, os parâmetros são do tipo **int**. Já no segundo método, os dois parâmetros são **float**. Finalmente, no terceiro método, os parâmetros são do tipo **double**. Esses métodos fazem a mesma coisa (calculam o maior entre dois números), mas de maneiras diferentes, pois recebem e retornam tipos diferentes.

Observe agora o último método **calcMaior()** declarado na classe `Maior`. Esse método, apesar de diferenciar do anterior pelo tipo de retorno de `double` para `int`, não será aceito pelo compilador Java como um polimorfismo de sobrecarga.

O motivo é que já existe um método com o nome `calcMaior()` e dois parâmetros do tipo **double**.

Esse método seria aceito se houvesse um número diferente de parâmetro ou um dos parâmetros tivesse o tipo diferente de **double**, como mostra a Listagem 2.

```
1 public class Maior{
2     private int xInt;
3     private int yInt;
4     private float xFloat;
5     private float yFloat;
6     private double xDouble;
7     private double yDouble;
8
9     public int calcMaior(int x, int y){...}
10    public float calcMaior(float x, float y){...}
11    public double calcMaior(double x, double y){...}
12    public double calcMaior(double a, double b, double c){...}
13    public double calcMaior(double x, int y){...}
14 }
```

Listagem 2 - Polimorfismo de sobrecarga com tipo e número de parâmetros diferentes

Observe que agora os dois últimos métodos possuem número e tipos de parâmetros diferentes dos métodos anteriores.

A Sobrecarga e os Construtores

O polimorfismo de sobrecarga normalmente acontece sobre os métodos construtores, pois é comum para uma classe ter várias maneiras de instanciá-la.

A Listagem 3 ilustra um exemplo de polimorfismo de sobrecarga nos construtores da classe Pessoa. Observe que são declarados 3 construtores e todos eles têm o mesmo nome da classe. Entretanto, cada um deles recebe um número diferente de parâmetros.

Assim, durante a criação de objetos do tipo Pessoa em um método main, por exemplo, o construtor que será chamado irá depender da quantidade e parâmetros passados durante a invocação do comando *new*.

```
1 public class Pessoa{
2     private int rg;
3     private String nome;
4
5     public Pessoa(){
6         ..
7     }
8
9     public Pessoa(String nome){
10         this.nome= nome;
11     }
12
13     public Pessoa(int rg, String nome){
14         this.rg = rg;
15         this.nome = nome;
16     }
17 }
```

Listagem 3 - Polimorfismo de sobrecarga de construtores

A Sobrecarga e a Conversão

Conversão e sobrecarga frequentemente andam lado a lado.

A **Conversão** é a capacidade de um tipo ser convertido em outro tipo de maneira automática ou pela força bruta também chamada de coerção.

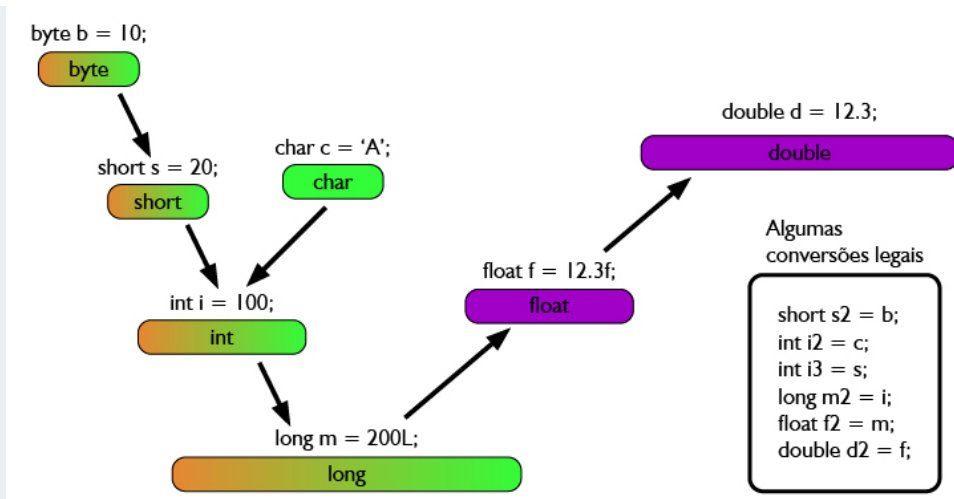
Veja o quadro abaixo sobre **conversão de tipos primitivos**.

Conversão de Tipos Primitivos

Java converte um tipo de dado em outro sempre que isso for apropriado.

As conversões ocorrem automaticamente quando há garantia de não haver perda de informação. As conversões automáticas são permitidas para tipos de maior precisão e para tipos de menor precisão.

Exemplos de Conversões Automáticas



São permitidas as conversões do tipo **byte** para o tipo **short**, de **short** para **int**, e assim sucessivamente, até chegar o tipo de maior precisão, que é o **double**.

Observe também algumas conversões permitidas baseadas nas variáveis criadas pelas letras no quadro “Algumas conversões legais”.

A conversão também pode fazer com que um método pareça como se fosse polimórfico. A conversão ocorre quando um argumento de um tipo é convertido para o tipo esperado, internamente. Por exemplo, suponha que a classe **Maior**, descrita anteriormente, tivesse apenas atributos e métodos para lidar com o tipo primitivo, conforme ilustra a Listagem 4.

```

1 public class Maior{
2     private float xFloat;
3     private float yFloat;
4
5     public float calcMaior(float x, float y){...}
6 }
  
```

Listagem 4 - Classe **Maior** com o método **calcMaior** para os tipos **int** e **float**

Agora, suponha que em uma aplicação (método **main**) usamos um objeto da classe **Maior**, mas, na chamada de seu método **calcMaior**, fossem passados, ao invés de valores do tipo **float**, valores do tipo **int**., conforme ilustra a Listagem 5.

```

1 public class Main{
2     public static void main(String[] args){
3         Maior maior = new Maior();
4         int x = 2;
5         int y = 4;
6         float z= maior.calcMaior(x,y);
7     }
8 }

```

Listagem 5 - Conversão automática de tipos primitivos

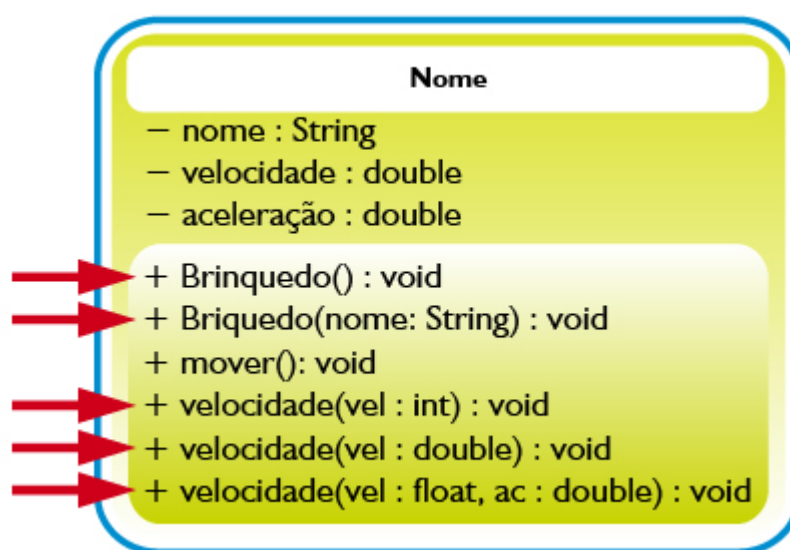
Observe que, apesar do método esperar parâmetros do tipo **float**, o compilador aceitou receber parâmetros do tipo **int**, e os converteu automaticamente para float dentro do método calcMaior da classe Maior apresentada na Listagem 5.

Nesse caso, parece um caso de polimorfismo de sobrecarga, mas, na verdade, o que acontece é uma conversão automática de tipos.

Atividade 02

1. Implemente em Java a classe Brinquedo apresentada na Figura 3 a seguir, aplicando o polimorfismo de sobrecarga nos métodos apontados pelas setas. Em seguida, escreva um método main que cria diferentes brinquedos fazendo chamadas para seus diferentes métodos construtores e chamando diferentes métodos velocidade().

Figura 03 - Esquema da classe brinquedo



Polimorfismo de Sobreposição

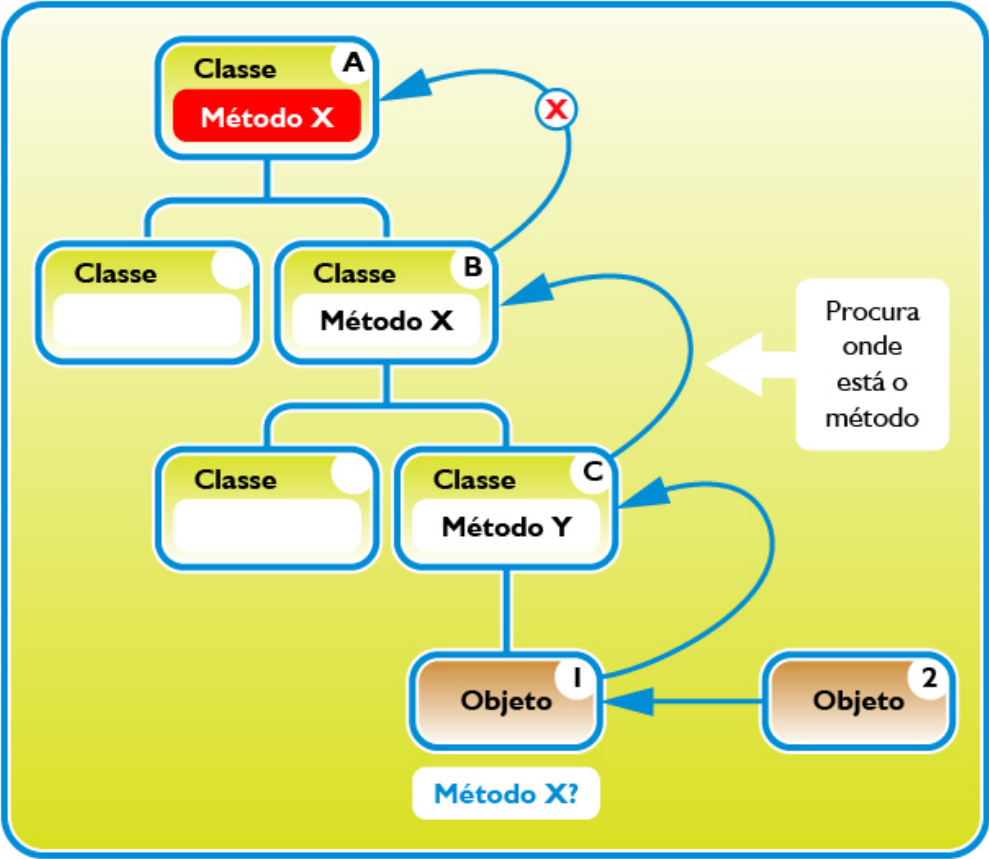


Vídeo 04 - Sobrescrita de Polimorfismo

Polimorfismo de sobreposição é a redefinição de métodos em classes descendentes. Ou seja, um método de uma classe filha com o mesmo nome de um método de uma classe mãe irá sobrepor esse último. Vejamos o exemplo da Figura 4.

Como pode ser observado na hierarquia de classes apresentada, existe: (i) uma **classeA**, que implementa um **metodoX()**; (ii) uma **classeB**, que implementa um método com o mesmo nome; e, finalmente, (iii) uma **classeC**, que implementa um **metodoY()**.

Figura 04 - Sobreposição de métodos em uma hierarquia de herança



O que aconteceria se fosse solicitado ao **Objeto1** da classeC a execução do **metodoX()**?

Conforme você viu nas aulas anteriores sobre herança, esse método será procurado na hierarquia da classe instanciada pelo **Objeto1**.

Devemos observar que, nesse caso, o **métodoX()** que também é implementado na **ClasseB**, foi encontrado primeiro que o **metodoX()** da **ClasseA**.

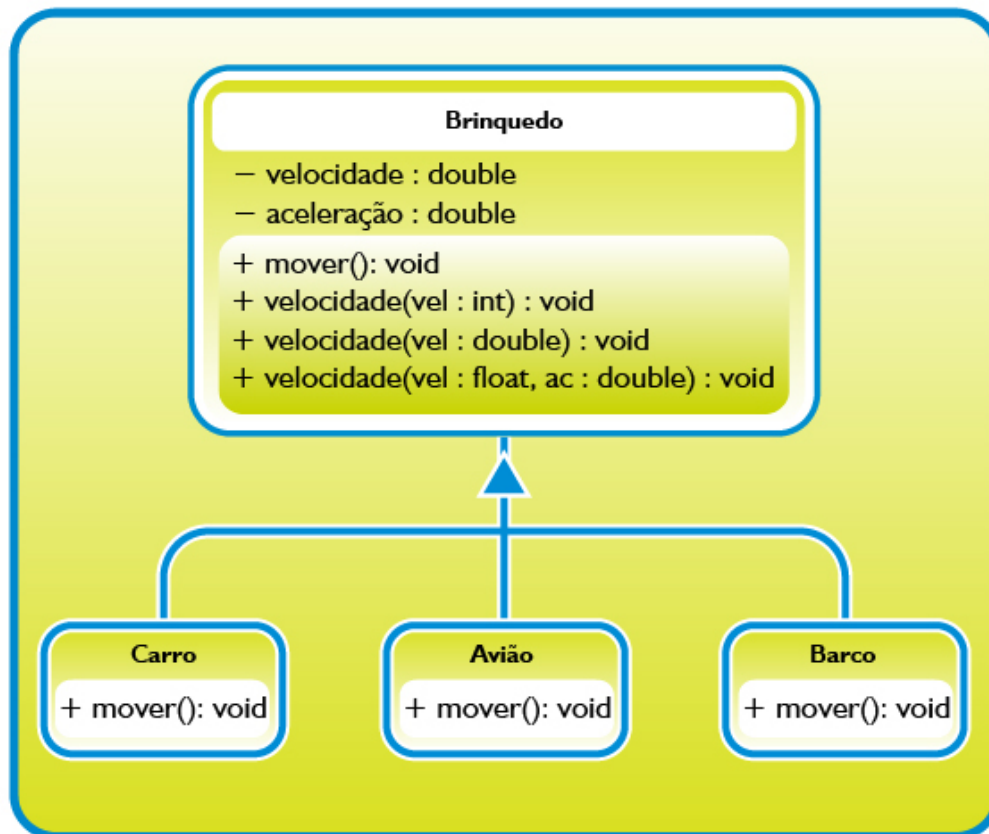
Nesse caso, o método que de fato será executado será o da ClasseB.

E, nesse caso, o **metodoX()** da **ClasseA** jamais será alcançado, a menos que seja criado um objeto da ClasseA. Dizemos então que ocorreu uma **SOBREPOSIÇÃO DE MÉTODO** ou um **POLIMORFISMO DE SOBREPOSIÇÃO**. O **métodoX** da **classeB** sobrepôs (ou redefiniu) o método de sua classe mãe.




O Polimorfismo de Sobreposição em Java

Considere que a classe **Brinquedo** (usada no exercício anterior) possui como descendentes as classes **Carro**, **Avião** e **Barco**, conforme ilustra a Figura 5.

Figura 05 - Hierarquia de herança da classe Brinquedo



Observe que as classes filhas sobrepõem o método **mover()** da classe **Brinquedo**. Vejamos então como ficam essas classes codificadas em Java na Listagem 6.

	<pre> public class Brinquedo{ ... public void mover() { System.out.println("mover brinquedo"); } ... } </pre>
	<pre> public class Carro extends Brinquedo{ ... public void mover() { System.out.println("CORRER"); } ... } </pre>
	<pre> public class Aviao extends Brinquedo{ ... public void mover() { System.out.println("VOAR"); } ... } </pre>
	<pre> public class Barco extends Brinquedo{ ... public void mover() { System.out.println("NAVEGAR"); } ... } </pre>

Listagem 6 - Implementação em Java da hierarquia da classe Brinquedo

Considerando o método **mover()** de cada classe filha, como poderíamos chamar o método mover() do brinquedo (classe) correto? Ou seja, como o **ControleRemoto** saberá que método **mover** ele deve chamar se ele tem disponível três tipos de mover diferentes (um para cada brinquedo)?

Vamos ver inicialmente como fica a implementação do Controle Remoto na Listagem 7.

```

1 public class ControleRemoto{
2     private Brinquedo brinquedo;
3     public ControleRemoto(Brinquedo b){
4         brinquedo = b;
5     }
6     public void mover(){
7         brinquedo.mover();
8     }
9 }

```

Listagem 7 - Classe ControleRemoto

Você lembra que na primeira vez que apresentamos esse exemplo dissemos que a única restrição para um controle remoto tão versátil seria que “quando criado o controle remoto, ele receberia o tipo de brinquedo que iria acionar em um dado instante”? Pois é, é aí que está o segredo para o controle remoto saber qual deve ser o método **mover()** que ele deve chamar (CORRER, NAVEGAR ou VOAR). O método construtor da classe **ControleRemoto** exige que o controle para ser inicializado receba um parâmetro do tipo **Brinquedo**. E isso acontece quando o atributo **brinquedo** do **ControleRemoto** recebe “b” (um objeto do tipo **Brinquedo**).

Então, quando o método **mover()** da classe **ControleRemoto** for acionado (isso significa dizer que o botão mover foi apertado), esse faz com que seu atributo **brinquedo** chame o método **mover()** correto, dependendo do tipo de **Brinquedo** que recebeu quando foi instanciado.

A Listagem 8 mostra como fica a classe Principal que instancia um objeto controleRemoto e um objeto carro para ser controlado automaticamente.

```

1 public class Principal{
2     public static void main(String[] args){
3         Carro carro = new Carro();
4         ControleRemoto controleRemoto = new ControleRemoto(carro);
5         controleRemoto.mover();
6     }
7 }

```

Listagem 8 - Classe Principal com o método main

Observe que criamos um brinquedo do tipo Carro, e quando criamos o **ControleRemoto**, enviamos esse objeto carro para o objeto controleRemoto através da chamada ao seu construtor. Assim, quando acionado o comando controleRemoto.mover(), será chamado o método **mover()** do **carro**.

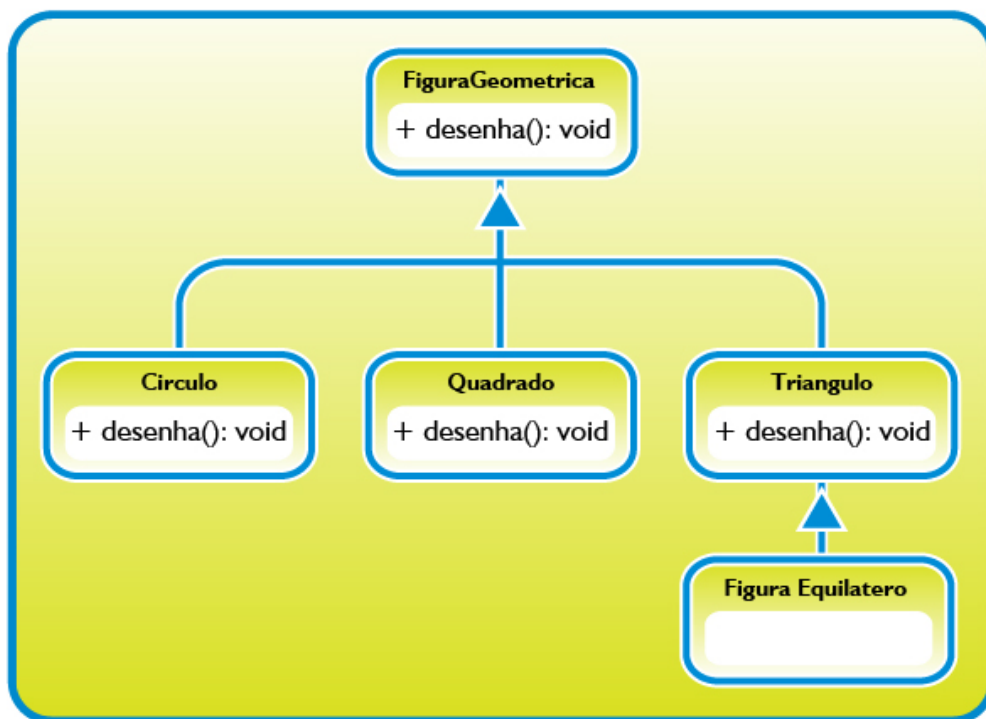
O resultado será a impressão da palavra:

CORRER

Atividade 03

1. Implemente as classes da hierarquia da classe `FiguraGeometrica` mostrada na Figura 6 abaixo em Java, aplicando o polimorfismo de sobreposição para o método `desenha()`.
2. Em seguida, crie uma classe `Principal` com um método `main` que cria um objeto de cada uma das classes e chama seus respectivos métodos `desenha()`.

Figura 06 - Hierarquia da classe `FiguraGeométrica`



Polimorfismo de Inclusão

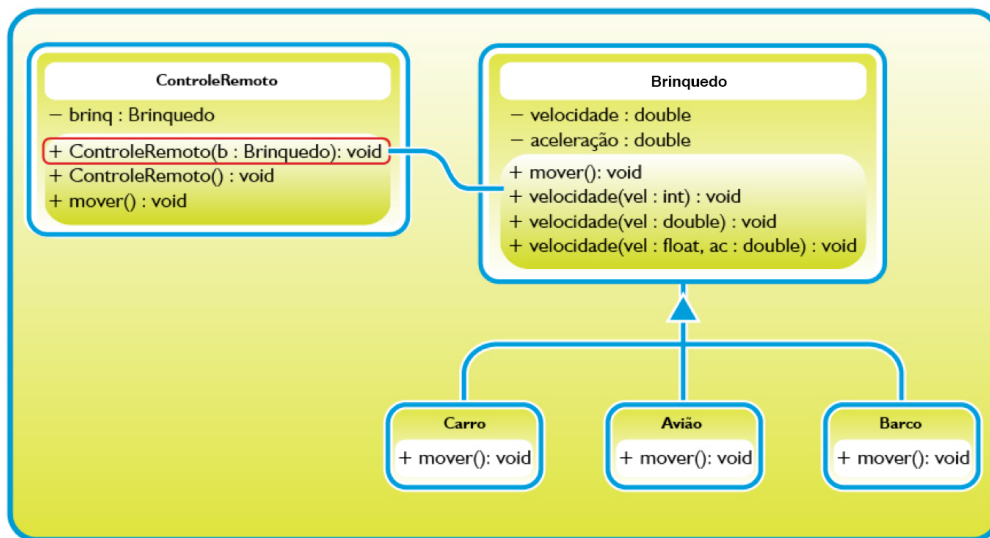
Polimorfismo de inclusão usa a capacidade de substituição da [herança](#) (Para você entender melhor, veja aula 10 (Tipos de herança: substituição de tipos).), de uma classe mãe por qualquer classe descendente, para permitir um

comportamento polimórfico nos métodos que usam a classe mãe.

No exemplo visto na seção anterior, veja a classe **Principal** na Listagem 8, onde criamos um objeto do tipo **Carro** e outro do tipo **ControleRemoto**, nós utilizamos o comportamento polimórfico do polimorfismo de inclusão. Fizemos isso quando substituímos a classe **Brinquedo** (mãe) pela classe **Carro** (filha) dentro da classe **ControleRemoto**.

Assim, o atributo interno do tipo Brinquedo da classe ControleRemoto pode receber qualquer objeto que seja de uma classe filha de Brinquedo. Para entender melhor o que aconteceu, veja a Figura 7.

Figura 07 - Hierarquia da classe Brinquedo e da classe Controle Remoto



Observe que a classe **ControleRemoto** está relacionada com a classe **Brinquedo**, pois possui um atributo do tipo **Brinquedo**. Mas, como as classes **Carro**, **Avião** e **Barco** são descendentes de **Brinquedo**, elas podem substituir a classe **Brinquedo** em qualquer método que a utilize.

Nesse caso, isso foi feito explicitamente, através da passagem de um objeto da classe **Carro** para o método construtor de **ControleRemoto** na Listagem 8.

Caso o programador deseje mudar o controle remoto para interagir com algum outro tipo de brinquedo, bastaria passar um objeto da classe **Avião** ou **Barco** na chamada ao construtor da classe **ControleRemoto**. A capacidade do objeto (`brinq`) do tipo **Brinquedo** da classe **ControleRemoto** de receber qualquer um objeto de subclasses da classe **Brinquedo** é que caracteriza o **polimorfismo de inclusão**.

Leitura Complementar

A seguir, veja os links que abordam o tema polimorfismo. Aproveite e tente encontrar a nossa classificação de polimorfismo nas diferentes abordagens de explicar o polimorfismo.

<<http://pt.wikipedia.org/wiki/Polimorfismo>>

<http://www.dsc.ufcg.edu.br/~jacques/cursos/p2/html/oo/o_que_e_polimorfismo.htm>

<http://www2.unoeste.br/~aglae/ling_pro/java_polimorfismo.htm>

Resumo

Nesta aula, você aprendeu que, com o polimorfismo, objetos de tipos diferentes podem responder à mesma mensagem (solicitação de método com o mesmo nome) de maneiras diferentes. Você estudou quais são e como funcionam os tipos de polimorfismo desde seus conceitos até sua codificação na linguagem Java. Você viu também diferentes exemplos para ilustrar as diferentes situações na qual o polimorfismo pode ser usado.

Autoavaliação

1. Sem consultar o material, responda: o que você entendeu por polimorfismo?
2. Quais são os tipos de polimorfismo?
3. Qual é a finalidade de se usar o polimorfismo de sobrecarga? Dê um exemplo.
4. O que é conversão de tipos?

5. Como funciona o polimorfismo de sobreposição? Dê exemplos.
6. Explique como funciona o polimorfismo de inclusão. Dê um exemplo.
7. Implemente o diagrama de classes representado pela Figura 8 abaixo. Para a classe **CadastroPessoas** considere o atributo **pessoas** como um array do tipo **Pessoa**.

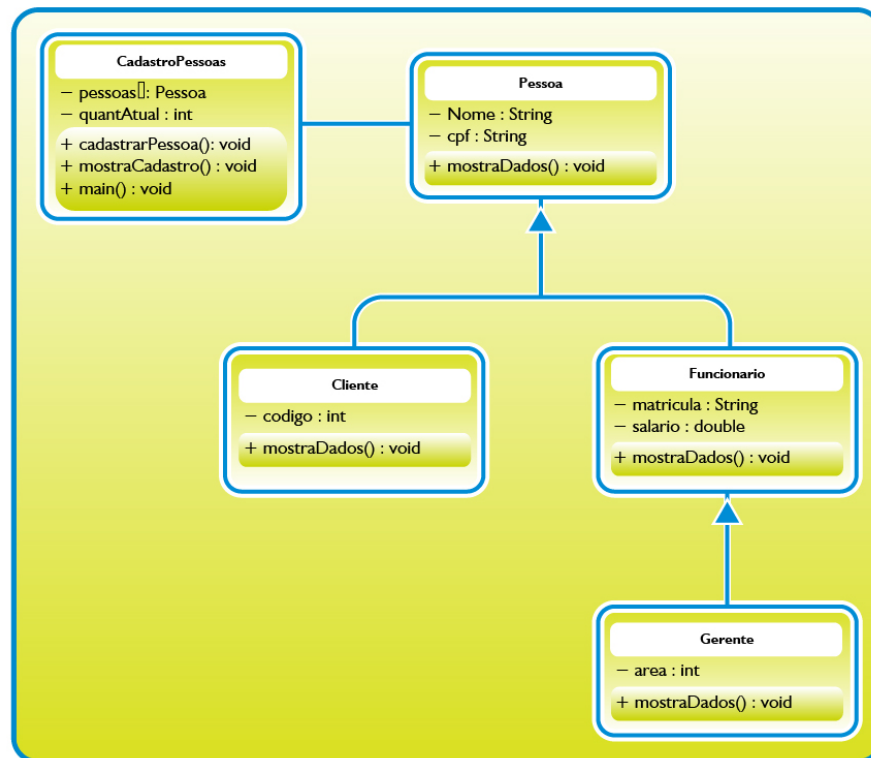
O método `cadastrarPessoa()`: deve acrescentar ao array **pessoas** um objeto descendente da classe **Pessoa**.

O método `mostraCadastro()`: deve percorrer todo o array de **pessoas** e mostrar todos os dados do descendente de **Pessoa**.

Aplice os tipos de polimorfismo em cada uma das situações solicitadas:

- a. **Polimorfismo de Sobrecarga**: crie mais de um método construtor para cada classe: **Pessoa**, **Cliente**, **Funcionario** e **Gerente**.
- b. **Polimorfismo de Sobreposição**: faça com que o método `mostraCadastro()` utilize o método `mostraDados()` correto, dependendo se a **Pessoa** é um **Cliente**, **Funcionario** ou **Gerente**.
- c. **Polimorfismo de Inclusão**: quando for adicionar ao array **pessoas** uma nova pessoa que pode ser de um dos tipos descendentes de **Pessoa**.

Figura 08 - Diagrama de classe



Referências

BARNES, David J.; KÖLLING, Michael. **Programação orientada a objetos com Java**. São Paulo: Pearson Prentice Hall, 2004.

DEITEL, H. M.; DEITEL, P. J. **Java como programar**. Porto Alegre: Bookman, 2003.

LEMAY, Laura. **Aprenda Java em 21 dias**. Tradução: Daniel Vieira. Rio de Janeiro: Campos, 2003.

SANTOS, Rafael. **Introdução à programação orientada a objetos usando Java**. Rio de Janeiro: Editora Campus, 2003.

SINTES, anthony. **Aprenda a programar orientada a objeto em 21 dias**. Tradução: João Eduardo Nóbrega Tortello. São Paulo: Pearson EducationdoBrasil, 2002.

THE JAVA tutorials. **What is inheritance?** Disponível em:
<<http://java.sun.com/docs/books/tutorial/java/concepts/inheritance.html>>. Acesso
em: 15 maio de 2017.