## **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.

https://materialpublic.imd.ufrn.br/curso/disciplina/5/8/9/2

### **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.

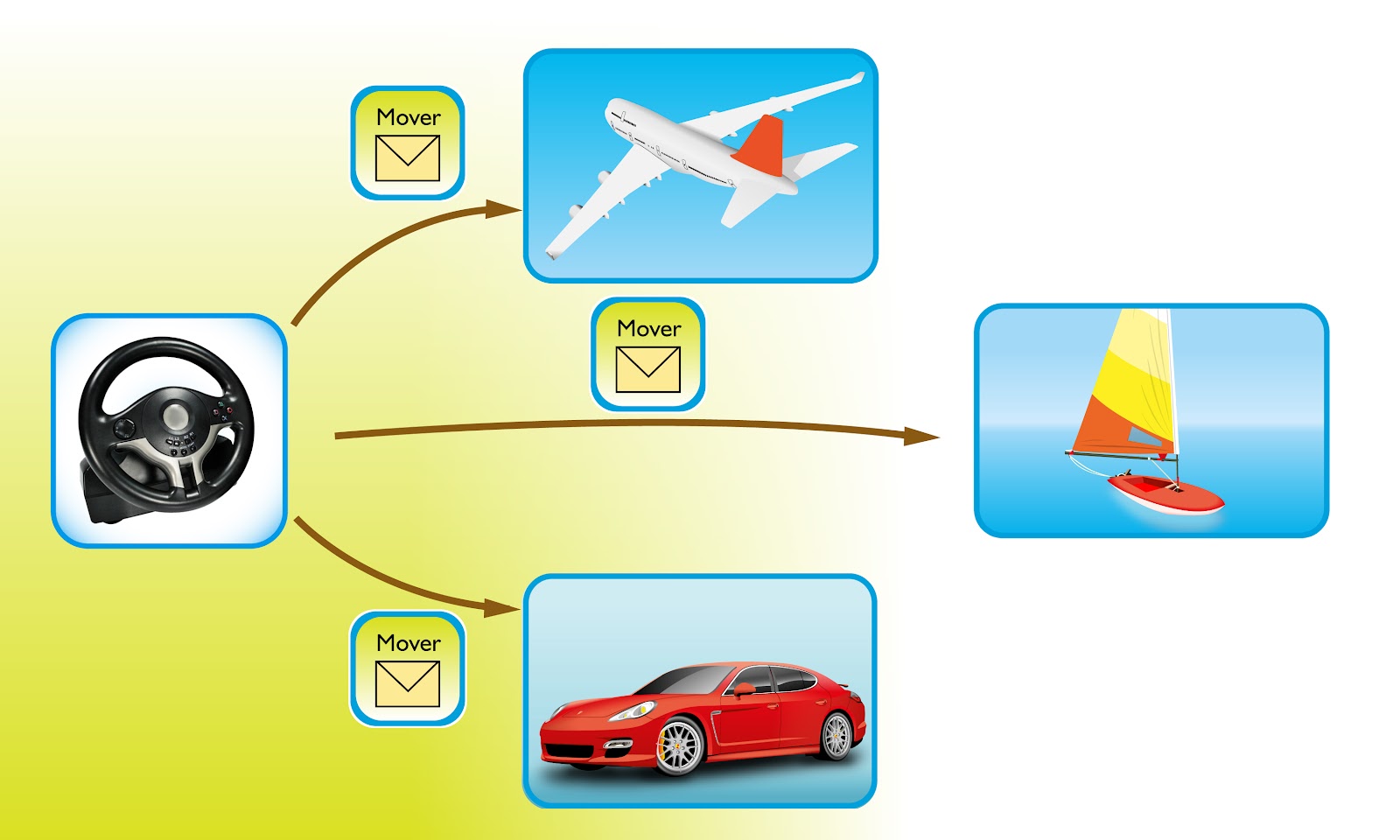
Figura 01 - Pilares da POO



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.

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

Polimorfismo pode ser classificado de três maneiras:

* Polimorfismo de sobrecarga
* Polimorfismo de sobreposição
* Polimorfismo de inclusão

#### **Polimorfismo de Sobrecarga**

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.

public class Maior{

      private int xInt;

      private int yInt;

      private float xFloat;

      private float yFloat;

      private double xDouble;

      private double yDouble;

​

      public int calcMaior(intx, int y){...}

      public float calcMaior(float x, float y){...}

      public double calcMaior(double x, double y){...}

      public int calcMaior(double a, double b){...} //ERRO!

}

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.

public class Maior{

   private int xInt;

   private int yInt;

   private float xFloat;

   private float yFloat;

   private double xDouble;

   private double yDouble;

​

   public int calcMaior(int x, int y){...}

   public float calcMaior(float x, float y){...}

   public double calcMaior(double x, double y){...}

   public double calcMaior(double a, double b, double c){...}

   public double calcMaior(double x, int y){...}

}

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.

### **Polimorfismo de Sobrecarga**

#### **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.

public class Pessoa{

   private int rg;

   private String nome;

​

   public Pessoa(){

   ..

   }

​

   public Pessoa(String nome){

      this.nome= nome;

   }

​

   public Pessoa(int rg, String nome){

      this.rg = rg;

      this.nome = nome;

   }

}

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.

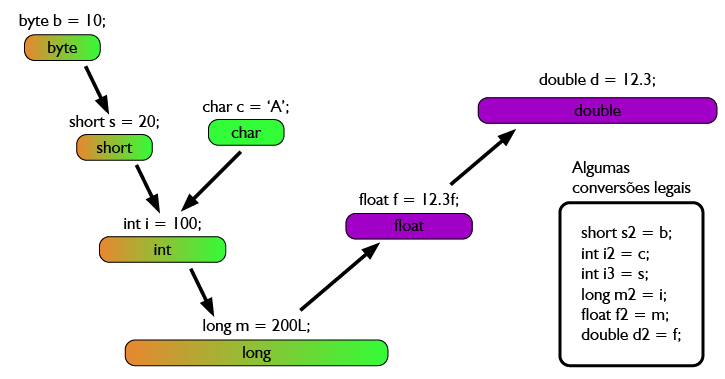
Veja o quadro abaixo sobre conversão de tipos primitivos.

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.

public class Maior{

   private float xFloat;

   private float yFloat;

​

   public float calcMaior(float x, float y){...}

}

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.

public class Main{

   public static void main(String[] args){

      Maior maior = new Maior();

      int x = 2;

      int y = 4;

      float z= maior.calcMaior(x,y);

   }

}

Listagem 5 - Conversão automática de tipos primitivos

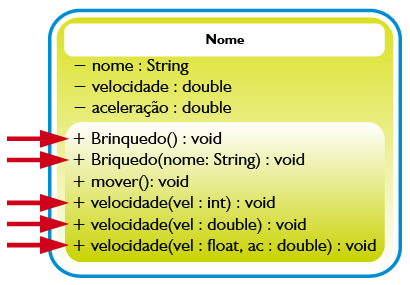
Observe que, apesar do método esperar parâmetros do tipo float, o compilador aceitou receber parâmetros do tipo int, e os converteu automaticamente para float dentro do método calcMaior da classe Maior apresentada na Listagem 5.

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**

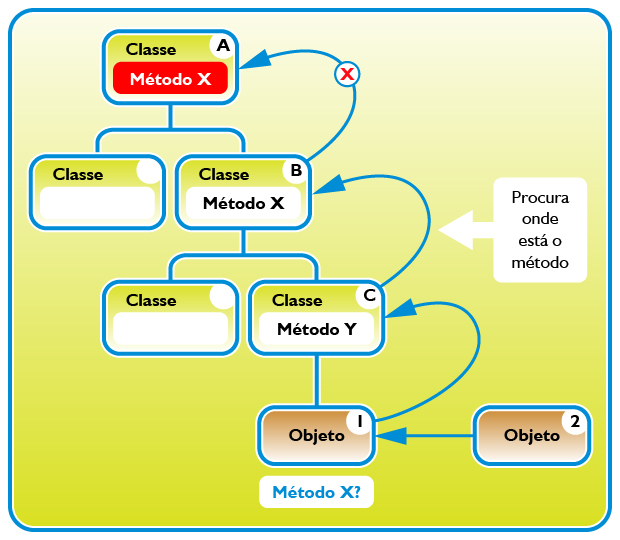
*play\_circle\_filled*

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 métodoY().

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.

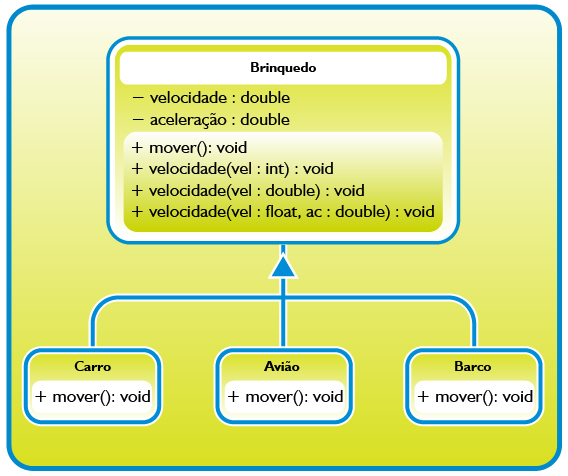
E, nesse caso, o metodoX() da ClasseA jamais será alcançado, a menos que seja criado um objeto da ClasseA. Dizemos então que ocorreu uma SOBREPOSIÇÃO DE MÉTODO ou um POLIMORFISMO DE SOBREPOSIÇÃO. O métodoX da classeB sobrepôs (ou redefiniu) o método de sua classe mãe.

### **Polimorfismo de Sobreposição**

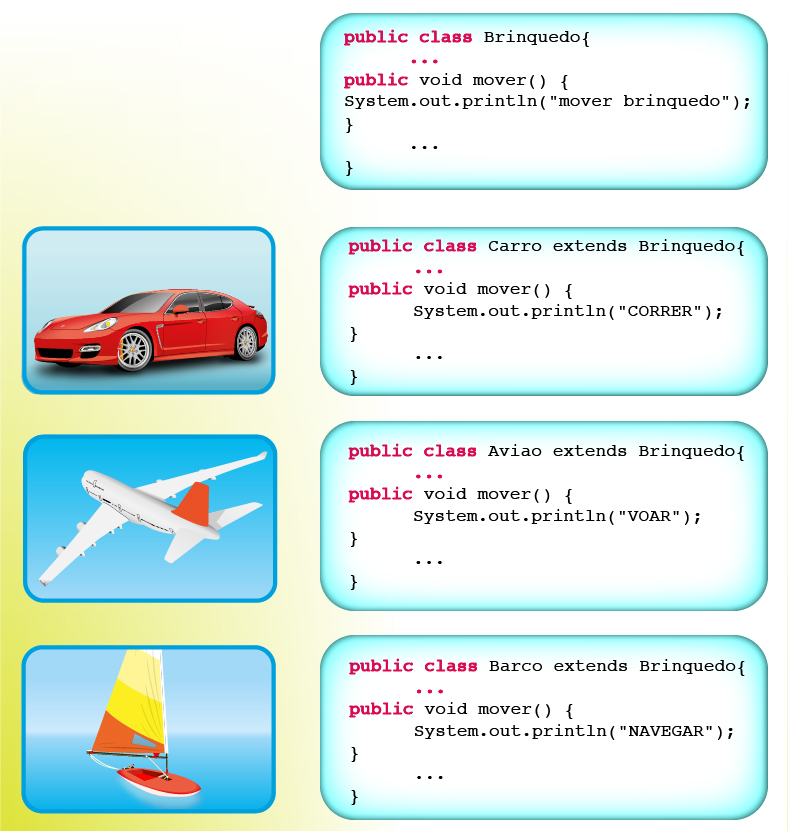
#### **O Polimorfismo de Sobreposição em Java**

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.

 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.

public class ControleRemoto{

   private Brinquedo brinquedo;

   public ControleRemoto(Brinquedo b){

      brinquedo =b;

   }

   public void mover(){

      brinquedo.mover();

   }

}

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 tipoBrinquedo. 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.

public class Principal{

   public static void main(String[] args){

      Carro carro = new Carro();

      ControleRemoto controleRemoto = new ControleRemoto(carro);

      controleRemoto.mover();

   }

}

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.

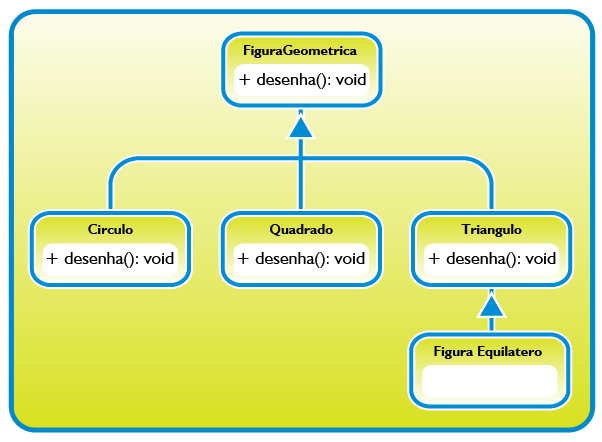
O resultado será a impressão da palavra:

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

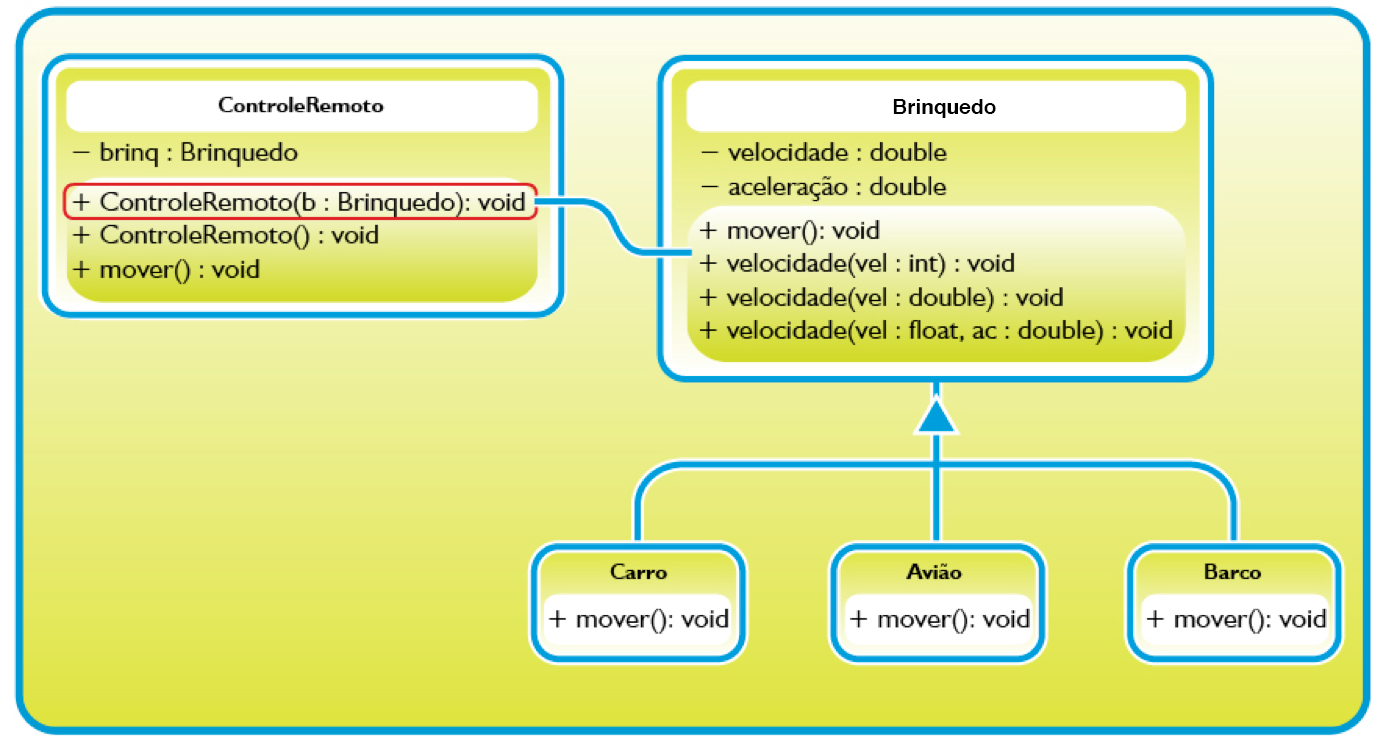


Polimorfismo de inclusão usa a capacidade de substituição da herança*sms\_failed*, de uma classe mãe por qualquer classe descendente, para permitir um comportamento polimórfico nos métodos que usam a classe mãe.

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.

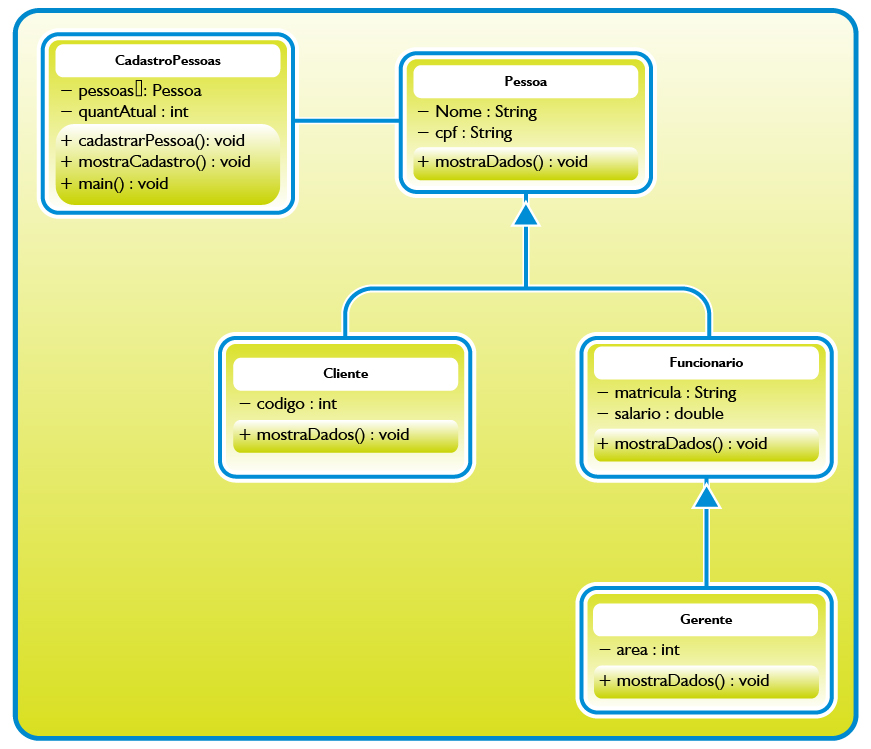
Nesse caso, isso foi feito explicitamente, através da passagem de um objeto da classe Carro para o método construtor de ControleRemoto na Listagem 8.

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.

### **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.

### **utoavaliação**

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 arraypessoas 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.  
     
   Aplique os tipos de polimorfismo em cada uma das situações solicitadas:
   1. Polimorfismo de Sobrecarga: crie mais de um método construtor para cada classe: Pessoa, Cliente, Funcionario e Gerente.
   2. Polimorfismo de Sobreposição: faça com que o método mostraCadastro() utilize o método mostraDados() correto, dependendo se a Pessoa é um Cliente, Funcionario ou Gerente.
   3. Polimorfismo de Inclusão: quando for adicionar ao arraypessoas uma nova pessoa que pode ser de um dos tipos descendentes de Pessoa.
8. Figura 08 - Diagrama de classe  
   

## **Coleções em Java**

### **Apresentação**

Em um contexto prático de programação, utilizando conceitos de orientação a objeto, precisamos constantemente manipular muitas informações e muitos objetos ao mesmo tempo. Para isso, é necessária uma estrutura que permita armazená-los e recuperá-los sempre que desejarmos. A linguagem Java oferece várias dessas estruturas de dados em um conjunto de classes chamadas de coleções. Tais coleções ajudam a armazenar e recuperar nossos objetos dentro de um sistema. É isso que você verá nesta aula.

### **Armazenando Objetos**

Vamos organizar os objetos semelhantes!

Existem várias situações durante a construção de sistemas e programas em que precisamos armazenar um número expressivo de objetos. Nesses casos, a criação e manuseio de uma variável para cada um dos objetos torna-se uma tarefa impraticável. Esse fato ocorre, por exemplo, quando precisamos armazenar uma lista de alunos de uma turma, ou mesmo da escola inteira, e em um dado momento precisamos recuperar apenas um dos objetos aluno dessa lista. Outro exemplo seria quando queremos encontrar um número de telefone e recorremos à lista telefônica, ela é nosso repositório de dados.

Imagine termos que criar 100 (cem) variáveis para 100 nomes de alunos! É possível criar um tipo especial de variável para armazenar um número específico de objetos ou valores primitivos. Esse tipo de variável é denominada de array (também conhecida como vetor) e será melhor discutida na sequência deste curso.

### **Arrays**

Considere a seguir um exemplo que motiva o uso de arrays. Constantemente, precisamos tomar nota de um valor que muda ao longo do tempo, e precisamos medir sua média, valores máximo e mínimo. Por exemplo, suponha que você é um treinador de um atleta velocista de 100 metros rasos, como parte do treinamento, você irá fazer 5 medições de tiros de 100 metros.

Para cada tentativa do atleta, você vai medir o tempo e colocar o resultado em sua planilha de rendimento.

Precisamos armazenar 5 valores numéricos reais para representar a marca de tempo obtida. Porém, não podemos criar 5 variáveis, e sim apenas uma para comportar esses valores. Veja tal código na Listagem 1.

public class TreinadorMain{

   public static void main(String[] args){

      String nomeAtleta = "Bolt";

      double[] marca= new double[5];

​

      marca[0] = 10.02;

      marca[1] = 10.0;

      marca[2] = 9.56;

      marca[3] = 9.57;

      marca[4] = 9.56;

   }

}

Listagem 1 - Declaração de um array

Conforme pode ser observado, um novo operador entrou em cena: o par de colchetes “[“ e “]”. Eles servem para indicar uma dimensão, ou seja, uma posição onde iremos colocar um valor inteiro que indica a quantidade de elementos que precisamos armazenar na variável em questão (marca). Por conta de tais colchetes, a variável marca não mais é capaz de armazenar um único valor do tipo double, mas sim irá armazenar um conjunto de valores do tipo double. Dessa forma, caracteriza-se a declaração de uma variável/atributo do tipo array (também chamada de vetor) em Java.

Observe que o atributo marca representa um array, mas é necessário o operador new para alocar espaço para armazenar o conjunto de valores. Assim, a palavra new é usada com o objetivo de indicar quantos valores do tipo Double serão necessários para armazenar tais valores. A quantidade de valores estipulada para armazenar no array, cinco (5) para o exemplo do atributo marca, é um valor que permanece fixo, após a chamada com new. Isso significa que no nosso exemplo o atributo marca será capaz de armazenar 5 elementos.

O código logo após a declaração indica uma atribuição de valor para armazenar uma determinada marca em cada posição do array. Veja que colocamos um número para indicar que posição estamos acessando. Esse valor sempre começa com 0 (zero) e vai até o comprimento do vetor menos uma unidade. No nosso caso, o array marca vai de 0 (zero) a 4 (quatro), compreendendo 5 (cinco) posições conforme foi declarado.

System.out.println("Tempo 1:"" + marca[0]);

System.out.println("Tempo 1:"" + marca[1]);

System.out.println("Tempo 1:"" + marca[2]);

System.out.println("Tempo 1:"" + marca[3]);

System.out.println("Tempo 1:"" + marca[4]);

Listagem 2 - Impressão dos valores do Array

Para exibirmos os valores, o programa da Listagem 1 pode ser incrementado pelas linhas de código da Listagem 2. Não indicamos a primeira tomada de tempo como sendo o tempo 0 (zero), pois naturalmente costumamos realizar contagens a partir do número 1 (um).

Há outra forma de definir os valores de um array, no momento de sua declaração podemos indicar seus valores, sendo que já precisamos conhecê-los de antemão. Veja a Listagem 3. Observe que não precisarmos indicar o comprimento do vetor, pois ele sabe de antemão pela quantidade de elementos definidos na sua inicialização.

double[] marca = {10.02, 10.0, 9.56, 9.57, 9.56};

Listagem 3 - Outra forma de definir os valores de um Array

Tanto na definição do array quanto no acesso para leitura ou escrita de suas posições, é possível usar um comando de repetição para acessá-lo diretamente. Na verdade, na programação diária, isso é o mais comum.

Vejamos como ficaria a exibição dos valores acima com um comando FOR na Listagem 4.

marca[0] = 10.02;

marca[1] = 10.0;

marca[2] = 9.56;

marca[3] = 9.57;

marca[4] = 9.56;

​

for(int i = 0;i<5;i++){

   System.out.println ("Marca"+(i+1)+": " +marca[i]);

}

Listagem 4 - Impressão dos valores de um array usando comando FOR

### **Atividade 01**

1. Para praticar o conceito de array, crie um programa que declara uma lista (array) capaz de armazenar 10 (dez) nomes de ferramentas para uso de um mecânico. Inicialize cada uma das ferramentas armazenadas no array, em seguida, imprima cada um deles.

### **Arrays Bidimensionais**

Primeiramente, vamos definir o que é uma dimensão de um array. A dimensão, ou quantidade de dimensões, é o conjunto de valores que precisamos definir para localizar uma informação. Por exemplo, uma lista de alunos de 0 a 100 pode ser organizada em um array de uma dimensão, pois para localizar um aluno nessa lista basta indicar um valor da sequência.

Já para localizar uma peça em um tabuleiro de xadrez precisamos de duas coordenadas, linha e coluna. Assim, para representarmos um tabuleiro de xadrez com arrays, são necessárias duas dimensões. A Listagem 5 mostra esse exemplo codificado em Java. Suponha que iremos representar um tabuleiro de xadrez com linhas e colunas de 0 (zero) a 7 (sete), compreendendo 8 valores de cada. Vamos também supor que cada valor representa uma String com o nome da peça que ocupa a casa. Observe a utilização de dois valores distintos para localizar uma casa no tabuleiro, e para cada valor um par de colchetes, um par para cada dimensão do array.

String[][] tabuleiro = new String[8][8];

​

tabuleiro[0][0] = "Torre branca";

tabuleiro[0][1] = "Cavalo branco";

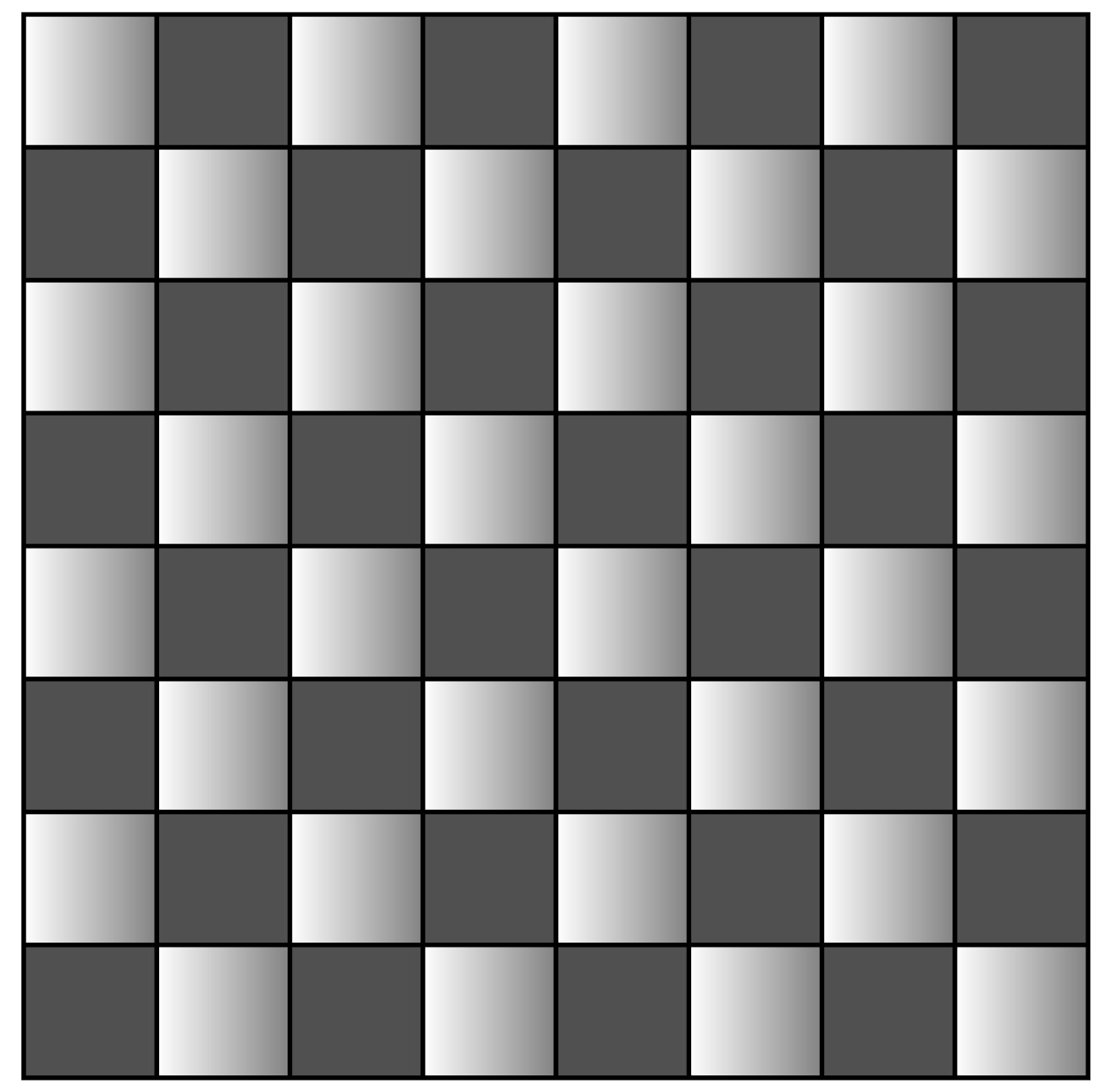
tabuleiro[1][0] = "Peão branco";

tabuleiro[0][3] = "Rainha branca";

Listagem 5 - Exemplo de Array bidimensional

É possível definir n dimensões, porém, na prática, não é comum nem recomendável trabalhar com tantas dimensões. Na prática, é extremamente comum trabalharmos com apenas 1 (uma), e algumas vezes com 2 (duas) e, quase nunca com 3 (três) dimensões, pois há outros recursos na programação orientada a objetos que desencorajam e oferecem alternativas melhores a essa prática.

### **Atividade 02**

1. Observe o tabuleiro da Figura 1 abaixo e crie um programa em Java que o represente através de um array de duas dimensões, em que cada casa irá conter a sua cor BRANCA ou PRETA, como sendo valores do tipo String. Imprima cada uma das casas do array, após iniciá-los.   
   Dica: não tente definir uma a uma, pois serão 64 linhas de códigos. Ao invés disso, crie estruturas com comandos de repetição para preenchê-lo e imprimi-lo.  
     
   Figura 01 - Tabuleiro  
   

### **Arrays como Objetos**

Vimos que arrays são estruturas para armazenar objetos, tais como Strings. Mas, é preciso dizer que arrays são também objetos, por isso é usado o comando new() para alocar espaço de armazenamento para ele. Graças a sua capacidade de se comportar como um objeto, os arrays possuem métodos e um atributo muito útil, o length, que indica seu comprimento.

É necessário conhecê-lo para não passarmos do limite e para realizar operações de manutenção em seus dados. Pois bem, vamos alterar o exemplo anterior (Listagem 1) que exibe os valores do array marca para utilizar seu atributo length. Tal programa modificado é apresentado na Listagem 6.

marca[0] = 10.02;

marca[1] = 10.0;

marca[2] = 9.56;

marca[3] = 9.57;

marca[4] = 9.56;

​

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

   System.out.println("Marca"+(i+1)+": " +marca[i]);

}

Listagem 6 - Exemplo de uso do atributo length de um array