

PROGRAMAÇÃO ORIENTADA A OBJETOS

CAPÍTULO 1 – O QUE É PROGRAMAÇÃO ORIENTADA A OBJETOS E POR QUE VOCÊ PRECISA SABER?

Handerson Bezerra Medeiros

INICIAR

Introdução

Quando falamos sobre desenvolvimento de *software*, sabemos que existem diversas linguagens de programação disponíveis no mercado. Então, fica sempre a dúvida: por que devemos escolher uma linguagem orientada a objetos? Esta é uma pergunta muito comum, quando estamos aprendendo programação.

Atualmente, as linguagens mais utilizadas no mercado são orientadas a objetos, como, por exemplo: Java, C#, C++, Python e Ruby. Quando sabemos aplicar os conceitos do paradigma da orientação a objetos, nossa programação fica mais funcional, pois conseguimos um nível de abstração muito eficaz.

Mas, o que significa orientação a objetos? Começamos a entender isso, quando percebemos que é como uma filosofia de trabalho. Não necessariamente significa que precisamos usar ferramentas X ou Y para programar, e sim, representa uma

mudança na nossa forma de pensar e enxergar os problemas, usando uma forma lógica, que foca no objeto, para chegar em possíveis soluções.

Quando utilizamos POO, nossa programação fica mais natural, pois conseguimos modelar nossos objetos e explicar como eles deverão interagir entre si. E, quando falamos de POO, falamos de um paradigma de programação de computadores, que faz uso de definições de objetos e classes como elementos principais, para representar, modelar e processar os dados que são usados no programa.

Neste capítulo, vamos entender a diferença entre programação estruturada e programação orientada a objetos. A partir disso, podemos nos aventurar nos nossos primeiros programas, reconhecendo a sintaxe e semântica da linguagem de programação chamada Java, uma das linguagens mais difundidas nos dias atuais. Por fim, vamos começar a modelar nosso programa, identificando suas classes e objetos, tornando assim nossa programação mais abstrata e fácil. E é dessa maneira que vamos compreender o mundo a nossa volta, de uma forma diferente.

Vamos estudar esse conteúdo a partir de agora, acompanhe!

1.1 Programação Estruturada vs Programação Orientada a Objetos: diferenças e similaridades

Tanto a Programação Estruturada (PE), quanto a Programação Orientada a Objetos (POO), são paradigmas de programação. Então o que as torna diferentes? Bem, cada uma se constitui a partir de um pensamento distinto. Por isso, toda vez que vamos desenvolver um programa, é necessário que façamos uma análise para identificar qual paradigma, dentre esses tipos de programação, é mais adequado ao problema que temos para solucionar.

Tanto a PE, como a POO, possuem seus pontos altos e baixos, porém a orientada a objetos tem sido identificada como o paradigma de programação de preferência dos desenvolvedores. Vamos entender um pouco o porquê de isso acontecer?

1.1.1 Programação Estruturada

O paradigma que envolve a PE tem como base, a ideia de que a programação seja mais voltada ao pensamento de máquina. Veremos que uma lógica estruturada é mais simples de criar, tendo em vista que já fazemos uso dela no dia a dia. Dessa forma, é comum optarmos por uma linguagem PE no início do nosso estudo de programação (MACHADO, 2018).

A programação estruturada é formada por três estruturas básicas (blocos de código) que se interligam: sequência, decisão (desvio) e iteração (repetição).

A **estrutura de sequência** define como as instruções serão executadas de maneira sequencial no programa. Ou seja, a sequência dita o fluxo de execução que aparece no programa.

Já a **estrutura de decisão** controla o fluxo das instruções que necessitam de alguma condição lógica para identificar qual fluxo deverá seguir no programa. Ou seja, caso a resposta da condição apresentada for verdadeira, o programa segue um determinado fluxo de instrução, mas caso a resposta da condição for falsa, o programa poderá seguir um caminho de instruções completamente diferente.

Enquanto isso, a **estrutura de iteração** (repetição) controla o fluxo das instruções, que podem ser executadas mais de uma vez no programa, obedecendo, é claro, a alguma condição lógica.

Apesar de ser uma programação de fácil entendimento, os códigos da PE são apresentados em um mesmo bloco de instruções, o que dificulta, caso seja necessário realizar alguma alteração no código. Por exemplo, se precisamos alterar alguma informação, obrigatoriamente, precisamos olhar todo o código, para verificar se nenhuma outra instrução que depende daquela informação, será prejudicada, ou terá que ser modificada também. Parece um pouco cansativo, correto?

Porém, é importante sabermos que na medida em que nosso programa for crescendo, nós podemos simplificar nosso código, criando sub-rotinas (funções ou procedimentos) que melhoram o código, dividindo-o em partes menores. Por exemplo, podemos criar uma função que receba dois números e retorne como resultado a soma. Dessa forma, toda vez que precisarmos realizar alguma soma em nosso programa, em vez de reescrever as mesmas instruções, podemos apenas chamar a função que foi criada para fazer soma. Podemos dizer que a PE nada mais é do que um passo a passo (sequência lógica), para criarmos nosso programa. Então o que é uma programação orientada a objetos? Vamos descobrir!

1.1.2 Programação Orientada a Objetos

Sabendo que a PE é focada no pensamento da máquina, a POO defende que a programação deve ser voltada mais ao pensamento humano. Precisamos desenvolver um algoritmo, de forma a ensinarmos a máquina a simular um pensamento humano (SILVA FILHO, 2010). Parece difícil? Será que é possível ensinar a máquina a pensar como uma pessoa?

A partir de agora, vamos começar a entender como isso funciona.

VOCÊ O CONHECE?

Um dos principais pesquisadores a desenvolver os conceitos de POO, foi o cientista Alan Kay, um dos criadores da linguagem Smalltalk. Enquanto trabalhava em suas pesquisas, Alan teve a ideia de que poderíamos construir um programa usando conceitos e abstrações do mundo real, como objetos e troca de mensagens. A ideia lhe valeu o Prêmio Turing, em 2003. O principal questionamento que o levou a esses conceitos foi: como seria um sistema de *software* que funcionasse como um ser vivo? A partir disso, ele começou a ter ideias sobre um sistema de *software*, fazendo uma comparação com o sistema de seres vivos (FURGERI, 2015).

Quando utilizamos POO, podemos resolver os mesmos problemas, utilizando o passo a passo tradicional (igual a uma Programação Estruturada), mas também podemos modelar o problema de uma forma que fique mais natural e parecido com o mundo real.

Como precisamos simular o nosso mundo real em um programa, precisamos relacionar alguns conceitos, por exemplo, nosso mundo real é composto de objetos, então por que não utilizarmos este mesmo conceito de objetos na nossa programação?

Então, quando formos desenvolver nossos programas, precisamos saber criar e modelar estes objetos, identificando seus relacionamentos com os outros objetos. Para ocorrer esse relacionamento entre os objetos, é necessário que ocorra um envio de mensagens, que precisa ser definido no momento de programar como cada objeto receberá essas mensagens, e como deve ser a ação a ser realizada.

Para compreender melhor o que é POO, vamos conhecer os quatro pilares que sustentam uma orientação a objetos: **abstração, encapsulamento, herança e polimorfismo.**

Quando pensamos em algo abstrato, imaginamos logo, algo que existe apenas na ideia, no conceito. A **abstração**, no universo da POO, se dá pela nossa aptidão de abstrair a complexidade de um sistema e se concentrar em apenas partes desse sistema. Ficou confuso? Para não restar dúvidas, vamos imaginar a seguinte situação: sabemos que um médico tem a opção de se tornar especialista em alguma parte do corpo humano, como por exemplo, coração (cardiologista), ossos (ortopedista) ou cérebro (neurologista). Quando o médico especialista vai tratar de um paciente, ele precisa abstrair (sem desconsiderar) as influências dos outros órgãos e focar sua atenção no órgão que está com problemas.

VOCÊ SABIA?

Devido à evolução constante das tecnologias em *softwares* e *hardwares*, podemos fazer, a cada dia, programas mais complexos. Os problemas resolvidos pela máquina nos ajudam a diminuir o esforço repetitivo e tedioso que antes eram realizados pelo homem (FONSECA FILHO, 2007). Quer saber mais? Recomendamos o artigo: <<http://www.pucrs.br/edipucrs/online/historiadacomputacao.pdf>>.

Para entender melhor, vamos imaginar um exemplo prático: imagine que precisamos desenvolver um programa para uma clínica. Essa clínica precisa ter o registro e controle de seus pacientes. Como estamos utilizando o paradigma de POO, e com isso, modelando nosso sistema baseado em um sistema real, nada mais óbvio do que existirem objetos do tipo “pacientes”, dentro do programa que estamos desenvolvendo. Portanto, esses objetos do tipo “pacientes”, deverão simular as características e ações, no mundo virtual, do que um paciente poderia realizar no mundo real.

Durante o curso, vamos entender e detalhar cada um desses quatro pilares que compõem a orientação a objeto. E agora, qual paradigma da programação melhor se encaixa com o meu problema? Podemos descobrir isso, entendendo são as vantagens e desvantagens da PE e da POO.

1.1.3 PE vs POO

Vimos, até agora, que podemos escolher entre dois tipos de programação: Estruturada ou Orientada a Objetos. Mas, para escolher qual delas usar na programação, precisamos entender um pouco mais sobre suas diferenças. Sabemos que na Programação Estruturada, construímos procedimentos ou funções que serão aplicados de maneira global no nosso programa (figura a seguir).

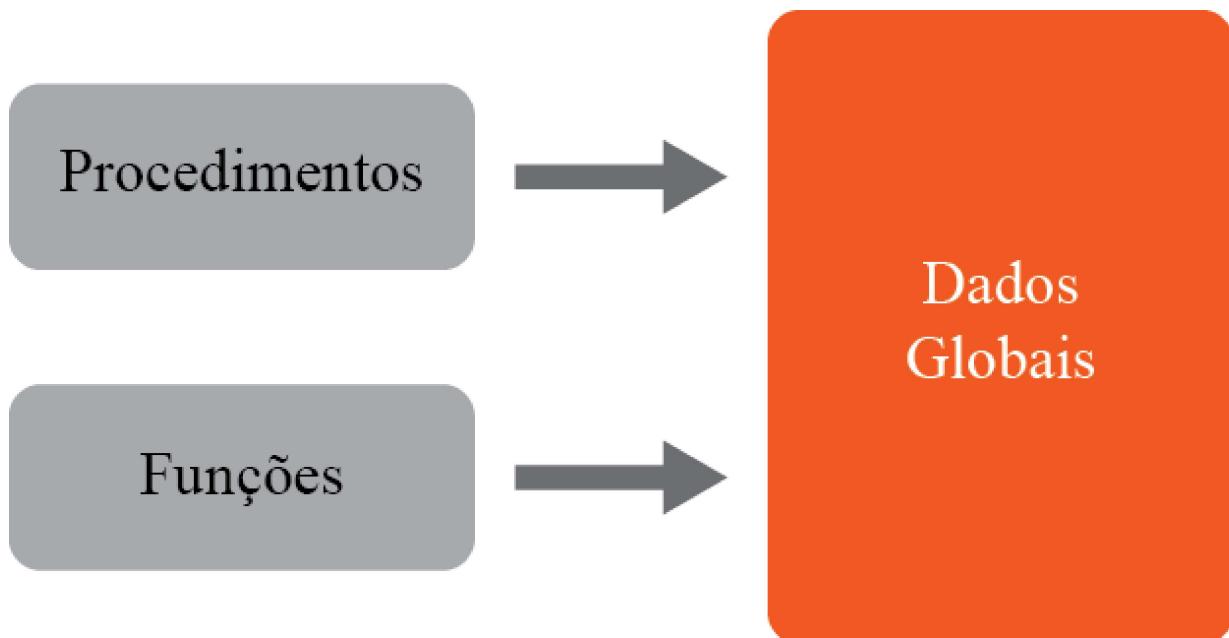


Figura 1 -

Modelo de interação de uma Programação Estruturada, no qual os procedimentos e funções agem nos dados globais do programa. Fonte: Elaborada pelo autor, 2018.

Já na Programação Orientada a Objetos, sabemos que identificamos nossos objetos e criamos métodos que vão interagir com esses objetos no programa (figura a seguir).

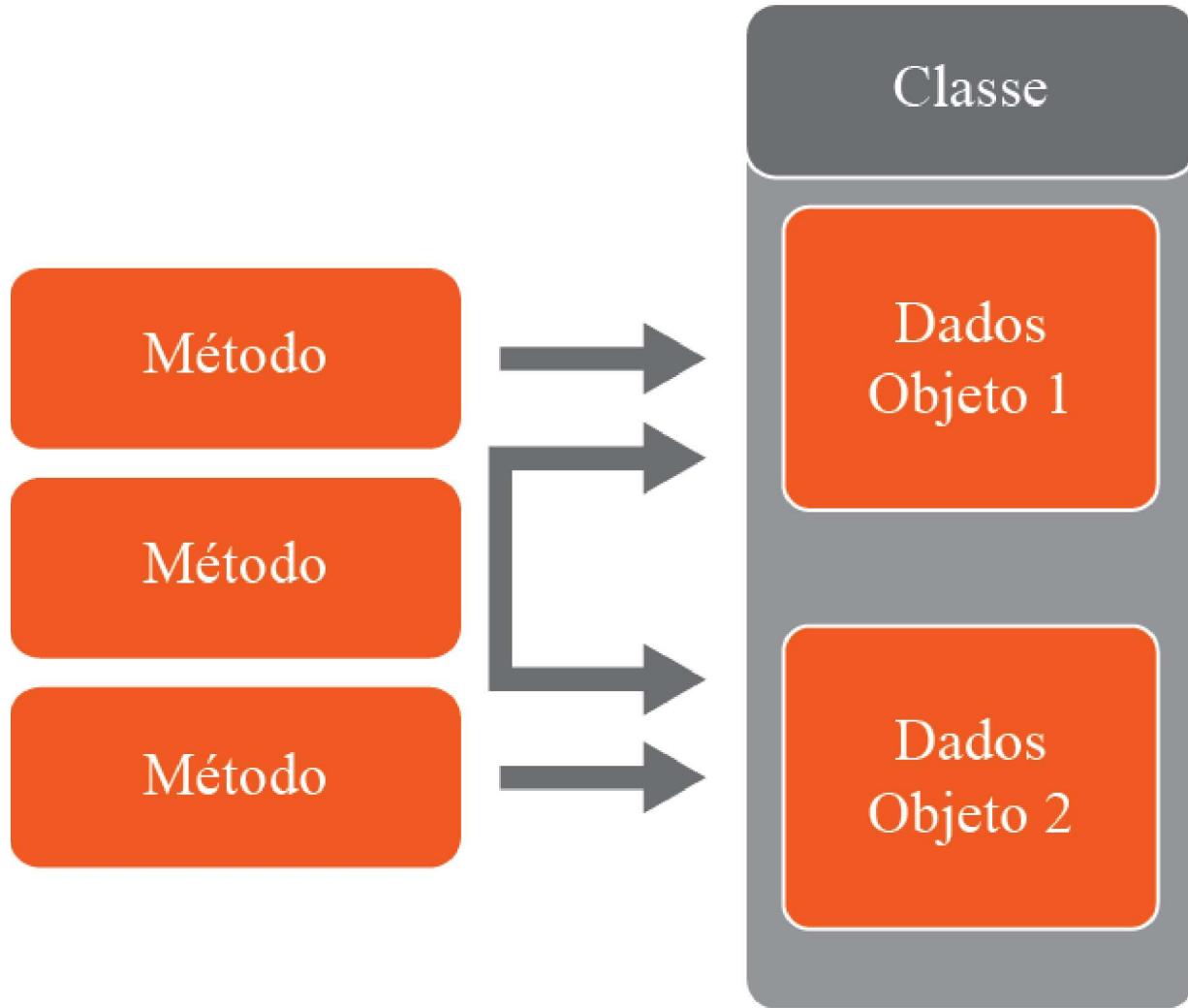


Figura 2 -

Modelo de interação de uma Programação Orientada a Objetos, no qual cada objeto diferente compartilha os mesmos métodos, se forem da mesma classe. Fonte: Elaborada pelo autor, 2018.

Vamos entender com um exemplo prático? Um dos exemplos mais clássicos sobre PE e POO é sobre o passo a passo para trocarmos uma lâmpada. Pare e analise rapidamente quantas etapas são necessárias para realizar esta atividade. Pensando de maneira estruturada e sucinta, podemos enxergar 12 passos para trocar uma lâmpada (quadro a seguir). Não se preocupe se você imaginou mais ou menos etapas, isso é natural, pois podemos chegar na mesma solução de maneira diferente.

1	Desligar o interruptor
2	Pegar uma escada
3	Montar a escada
4	Subir na escada
5	Desenroscar a lâmpada queimada
6	Descer da escada
7	Jogar a lâmpada queimada no lixo
8	Pegar uma lâmpada nova
9	Subir na escada
10	Rosquear a nova lâmpada
11	Descer da escada
12	Ligar o interruptor para verificar se a nova lâmpada acende

Quadro 1 - Exemplo lógico de uma troca de lâmpada utilizando o paradigma de programação estruturada.

Fonte: Elaborado pelo autor, adaptado de DEITEL; DEITEL, 2016.

Agora, vamos pensar em como resolver este mesmo problema, mas de maneira orientada a objetos. Sabemos que precisamos identificar o que são os objetos e quais são suas características (quadro a seguir). Neste caso, podemos identificar que lâmpada é um objeto e que algumas de suas características seriam: inteira ou quebrada.

Objetos	Características
Pessoa	Pegar, descer, subir, jogar
Lâmpada	Inteira, quebrada, ligada, desligada

Quadro 2 -

Exemplo lógico de uma troca de lâmpada utilizando o paradigma de Programação Orientada a Objeto.

Fonte: Elaborado pelo autor, adaptado de DEITEL; DEITEL, 2016.

Após realizar essa identificação, podemos verificar como os objetos irão se relacionar entre si. Por exemplo, “pessoa pega a lâmpada inteira” ou “pessoa joga a lâmpada quebrada”. Simples, né? Quando utilizamos POO, podemos reaproveitar

melhor nosso código, fazendo apenas com que os objetos se relacionem de várias maneiras.

No próximo tópico, vamos entender melhor como modelar nosso programa orientado a objetos.

1.2 Primeiros passos para trabalhar com POO

Agora, vamos começar a desenvolver nossos programas. Para isso, precisamos escolher uma linguagem de programação que seja orientada a objetos. Como comentado anteriormente, vamos utilizar a linguagem JAVA. O diferencial dessa linguagem é a utilização do conceito de máquina virtual. Basicamente, essa máquina virtual fica entre o sistema operacional e o programa, e tem a responsabilidade de interligar essas duas camadas. Interessante, né?

Neste tópico, vamos entender a sintaxe e semântica da linguagem de programação JAVA.

1.2.1 Declarando as variáveis

Em Java, todas as variáveis devem possuir um tipo, o qual não pode ser mudado durante a execução do programa. Para declararmos uma variável, é necessário que seja informado o tipo de dado que ela poderá receber e seu nome.

```
tipoDaVariavel nomeDaVariavel;
```

Por exemplo, se quisermos criar uma variável para guardar a idade do usuário, podemos chamar de idade e ela será do tipo inteiro (*int*). E uma outra variável, para guardar o nome do usuário, poderia ser nome e do tipo texto (*string*). Portanto, essas variáveis só poderão receber valores desse tipo.

```
int idade;
```

```
String nome;
```

Algumas regras e convenções precisam ser respeitadas quando declararmos nossas variáveis. Não podemos começar nossa variável com um número, mas podemos criar variáveis que contenham letras, números e caracteres sublinhados (_), por exemplo, *string nome123 ou string nome_123*.

Para a linguagem Java, se declararmos uma variável com o nome “idade” e outra com o nome “Idade”, ele reconhece como sendo duas variáveis diferentes, por Java ser uma linguagem *case sensitive*. Por padrão das boas práticas de programação, por fim, as variáveis devem ser declaradas em minúsculo, salvo seja um nome composto, colocamos todas as palavras em minúsculo, menos a primeira letra da segunda palavra, por exemplo, *int idadeAtual*. Após declaradas, as variáveis podem ser utilizadas para armazenar informações, por exemplo, *idadeAtual = 10*.

Podemos também, atribuir valores às variáveis, no momento em que elas são declaradas. Ou inicializá-las realizando algum tipo de operação aritmética (adição, subtração, divisão, multiplicação e módulo).

```
int idadeAtual = 10;
```

```
int idadeFutura = 10 + 5;
```

```
int dobraDaldade = 10 * 2;
```

A linguagem Java apresenta quatro tipos primitivos para guardar tipos de informações diferentes, veja exemplos na figura a seguir.

- *Boolean*: só admite os valores *true* ou *false*.
- *Char*: guarda apenas um caractere.
 - *String*: armazena todos os caracteres.
- Inteiros: números inteiros positivos ou negativos.
 - *Byte*: 1 byte
 - *Short*: 2 bytes
 - *Int*: 4 bytes
 - *Long*: 7 bytes
- Reais com ponto flutuante: além de armazenar números inteiros, também armazena ponto flutuante (por exemplo, frações)
 - *Float*: 4 bytes
 - *Double*: 8 bytes

```

boolean status = true;
char letra = 'a';
String nome = "Maria";
int ano = 2018;
double pi = 3.14;

```

Figura 3 - Exemplo dos possíveis valores que podem ser armazenados nos tipos de variáveis disponibilizados na linguagem. Fonte: Elaborada pelo autor, 2018.

Quando declaramos uma variável, dentro da classe e fora dos métodos, podemos atribuir mais uma especificação antes de informar o tipo e seu nome. Podendo ser: *private*, *public* ou *protect*. Se declararmos como *private* significa que a única classe que pode ter acesso aquela variável é a própria variável que a declarou. Já o atributo *public*, se declarado junto com a variável, significa que todas as classes podem ter acesso àquela variável.

VOCÊ QUER LER?

Java é uma das linguagens de programação com maior aderência no mercado e em meio aos desenvolvedores, e com certeza, a mais utilizada no mercado. Sua criação se deu para resolver alguns problemas que são encontrados em outras plataformas, como, por exemplo: utilizar ponteiros tem o

problema de consumir muita memória e precisar reescrever o código, quando muda o sistema operacional (ZANETTE, 2017). Quer conhecer mais a linguagem Java? Acesse: <<https://becode.com.br/java-linguagem-programador-rico/>> (<https://becode.com.br/java-linguagem-programador-rico/>)>.

Por fim, se quiser que a variável esteja disponível apenas para as classes nas quais a variável foi declarada, ou são do mesmo pacote, é só colocar o atributo *protected*. Mas não se preocupe, isso será melhor detalhado nos próximos tópicos.

1.2.2 Casting

pesar de termos várias opções de tipo de variáveis, precisamos ter muito cuidado quando formos decidir por seu tipo. Pois, alguns valores são incompatíveis, caso você tente fazer alguma atribuição direta.

Sabemos que podemos fazer atribuições diretas, por exemplo:

```
int saldo = 10; // saldo recebe 10
```

```
int total = saldo; // total recebe uma cópia do valor de saldo, no caso 10
```

```
saldo = saldo - 4; // saldo recebe o resultado do que já tinha em saldo - 4, no caso 6
```

No exemplo acima, as duas variáveis (saldo e total) pertencem ao mesmo tipo (inteiro), o que permite que essa atribuição seja realizada com sucesso. Porém, um número real representado por *float* ou *double*, não poderá ter seu valor ser atribuído a uma variável do tipo inteiro (int).

```
double pi = 3.14; // pi recebe 3.14
```

```
int numeroPi = pi; // erro, não compila
```

Mas podemos imaginar a seguinte situação: se o valor que for armazenado na variável do tipo *double* for inteiro (pois sabemos que um número real pode conter um número inteiro), então dará certo a atribuição dessa variável *double* com uma do tipo inteira?

```
double num = 3;           //num recebe 3  
int outroNumero = num; //erro, não compila
```

Apesar do valor 3, armazenado na variável *num*, ser um número inteiro e, em teoria, poder ser armazenado na variável inteira *outroNumero*, o compilador não tem como garantir essa conversão. E se for o contrário?

```
int num = 3;           //num recebe 3  
double outroNumero = num; //outroNumero recebe o valor armazenado por num, 3
```

Nesse caso, o código compila sem nenhum problema. Pois, já vimos anteriormente, que o tipo *double*, por se tratar de números reais, pode armazenar números inteiros, sem nenhum problema.

VOCÊ QUER LER?

Estamos percebendo que os valores numéricos na linguagem Java são representados por variáveis de vários tipos. Quando declaramos nossas variáveis com os tipos bem especificados, nosso programa armazena, na memória, exatamente o espaço referente àquele tipo que foi declarado. Mas podemos atribuir um valor 0 para esses tipos, que são chamados de *default* (RICARTE, 2001). Conheça mais em: [\(http://www.dca.fee.unicamp.br/cursos/PooJava/Aulas/poopjava.pdf\)](http://www.dca.fee.unicamp.br/cursos/PooJava/Aulas/poopjava.pdf).

E, se realmente precisarmos que ocorram essas atribuições, de forma que a transição arredonde meu valor *double*, isso é possível? Para que seja possível realizarmos essas atribuições sem que haja erro na compilação do nosso programa, podemos moldar nosso número do tipo *double*. Ou seja, ordenar que o programa arredonde para que se torne um número inteiro. A este processo, chamamos de *casting*.

```
double num = 3.14;           // num recebe 3.14
int outroNumero = (int) num; // outroNumero recebe o valor de num arredondado, 3
```

Nem todas as conversões serão possíveis em nosso programa. Com o tipo de variável *boolean* é impossível de se fazer conversão com qualquer outro tipo. Podemos verificar na figura a seguir, as possíveis conversões em Java.

Para:	<i>Byte</i>	<i>Short</i>	<i>Char</i>	<i>Int</i>	<i>Long</i>	<i>Float</i>	<i>Double</i>
De:	-----	automático	(char)	automático	automático	automático	automático
<i>Byte</i>	(byte)	-----	(char)	automático	automático	automático	automático
<i>Short</i>	(byte)	(short)	-----	automático	automático	automático	automático
<i>Char</i>	(byte)	(short)	(char)	-----	automático	automático	automático
<i>Int</i>	(byte)	(short)	(char)	(int)	-----	automático	automático
<i>Long</i>	(byte)	(short)	(char)	(int)	(long)	-----	automático
<i>Float</i>	(byte)	(short)	(char)	(int)	(long)	(float)	-----
<i>Double</i>	(byte)	(short)	(char)	(int)	(long)	(float)	-----

Quadro 3 - Possíveis conversões em Java, verificando as conversões que são automáticas e qual tipo tem prioridade. Fonte: Elaborado pelo autor, adaptado de MANZANO, 2014.

Podemos perceber, que algumas conversões são realizadas de maneira automática pela linguagem de programação. Dessa forma, não se faz necessário informar, em código, esses tipos de conversões.

1.2.3 Controle de fluxo e repetições

O controle de fluxo e repetições é essencial em qualquer linguagem de programação, por isso não poderia ser diferente para Java. Mesmo sendo uma programação orientada a objetos, é necessário ajustar a maneira como nosso programa vai realizar as tarefas. Precisamos entender como essas tarefas devem ser executadas de maneira seletiva, repetida, ou para prevenir um erro. Por exemplo, se nosso programa quer saber se o usuário é maior de idade, precisamos de algum mecanismo que após a inserção dessa informação do usuário o programa faça uma comparação para saber se aquele valor é ou não maior que 18, correto? Para realizar isso, vamos precisar de algumas instruções específicas, os comandos.

Esses comandos podem ser classificados em duas categorias: tomada de decisão (*if-else, switch-case*) e laços repetições (*for, while, do-while*).

Para darmos início, vamos entender a forma mais simples de um controle de fluxo, o comando *if-else*. Esse comando é responsável por testar se uma condição é verdadeira ou não, lembra do nosso exemplo de verificar se o usuário é maior de idade? Caso a resposta dessa condição seja verdadeira, nosso programa executa um bloco de instruções, caso for falso, nosso programa poderá executar um outro bloco de instruções ou apenas prosseguir com as próximas instruções. Vamos verificar como esse comando poderia ser desenvolvido no nosso programa de verificação de idade.

```

int idade;           // declarando a variável
idade = 18;         // idade recebe o valor 18
if(idade >= 18){    // comando de verificação do valor da idade
    System.out.println("Usuário é maior de idade!"); // executado se a condição for true
} else {
    System.out.println("Usuário é menor de idade!"); // executado se a condição for false
}

```

Agora imagine, além da verificação se o usuário é maior de idade gostaríamos de saber também se esse mesmo usuário tem menos de 60 anos, ou seja, sua faixa etária tem que ser de 18 a 60. Já sei, poderíamos fazer dois *if-else*, o primeiro verifica se é maior de idade e o segundo verifica se é menor que 60.

```

if(idade >= 18){    // comando de verificação do valor da idade
    System.out.println("Usuário é maior de idade!"); // executado se a condição for true
} else {
    System.out.println("Usuário é menor de idade!"); // executado se a condição for false
}
if(idade < 60){     // comando de verificação do valor da idade
    System.out.println("Idade menor que 60!"); // executado se a condição for true
} else {

```

```
System.out.println("Idade maior que 60!") //executado se a condição for false
}
```

O exemplo acima, parece funcionar perfeitamente, tendo em vista que o valor armazenado na variável idade é 18. Mas, caso a variável idade tivesse o valor de 10, no primeiro comando `if (idade >= 18)` teríamos como resposta false e o segundo comando `if (idade < 60)` seria true. O que nos ocasiona um problema, pois nosso programa deveria mostrar só aqueles usuários que estão na faixa etária que queremos (18 a 60). Como poderíamos resolver isso? É simples, esse tipo de comando nos permite colocar uma condição dentro de outra condição, ou seja, um `if` dentro de outro `if`.

```
if (idade > 60) { //comando de verificação do valor da idade
    System.out.println("Idade maior que 60!!") //executado se a condição for true
} else if (idade < 18) {
    System.out.println("Usuário é menor de idade!"); // executado se a condição for false
} else {
    System.out.println("Idade entre 18 a 60 anos!"); //executado se a condição for true
}
```

Esse tipo de estratégia é chamado de execução seletiva de múltiplos comandos. Pois conseguimos em um único bloco de comando executar várias instruções, de acordo com um determinado critério. Podemos ainda melhorar nosso código fazendo o uso de expressões booleanas por meio de operadores lógicos, por exemplo, E (`&&`) e OU (`||`).

```
if (idade >= 18 && idade < 60) { //comando de verificação do valor da idade
    System.out.println("Idade entre 18 a 60 anos!"); //executado se a condição for true
} else {
    System.out.println("Usuário é menor de idade!"); // executado se a condição for false
```

}

Podemos realizar essas seleções de maneira que verificamos caso a caso e identificando comandos para cada caso existente. Ficou confuso? Podemos utilizar do comando *switch-case* para testar cada condição separadamente, passando uma expressão de assume o valor de *true* para ser executada. Por exemplo, sabemos que a variável *idade* pode ter um valor de 1 a 3. Mas para cada valor, meu programa deverá mostrar para o usuário, informações diferentes.

```
int idade;  
idade = 2;  
switch(idade) { //expressão válida de entrada do comando  
case 1:  
    System.out.println("Idade é igual a 1");  
break; //comando que finaliza o switch se case 1 for verdade  
case 2  
    System.out.println("Idade é igual a 2");  
break;  
case 3:  
    System.out.println("Idade é igual a 3");  
break;  
default: System.out.println("O valor não é 1, 2 ou 3"); //se nenhum case for verdade, será  
executado a instrução que existe no default  
}
```

Podemos nos deparar com situações, nas quais é necessária uma verificação de vários valores de uma variável, e, para cada verificação, é realizado o mesmo bloco de instruções. Ou seja, precisamos criar um laço de repetição, e para isso precisamos criar uma variável que ficará responsável por contar o número de

interações. O laço mais comum é o comando *for*. Este comando apresenta uma estrutura, na qual nossa variável de controle (aquele que conta as interações, expressão 2) precisa ser informada de como inicializará o laço (expressão 1) e finalizar, enquanto aquela condição de verificação for *true*. A sua estrutura básica é simples.

for ([expressão 1]; [condição]; [expressão 2])

[bloco de instruções]

Por exemplo, vamos criar uma instrução utilizando *for* para um programa onde seja impresso os números de 1 a 10.

```
int contador;  
for (contador = 1; contador <= 10; contador++) {  
    System.out.println(contador);  
}
```

Analisando o exemplo acima, percebemos que a primeira coisa que o comando *for* realiza é inicializar nossa variável de controle com o valor 1 (*contador = 1*). Após isso, é verificado se a condição retorna *true* (*contador <= 10*). Seu retorno sendo verdadeiro, o bloco de instruções dentro do *for* é executado (ou seja, a impressão do valor atual armazenado na variável *contador*). E por fim, é realizado o incremento de +1 na variável *contador* (*contador++*), de modo que seja testado novamente a condição. Enquanto a condição retornar *true*, o comando de instrução é executado, fazendo com que ocorra um laço. Esse laço é interrompido assim que a condição retornar *false*.

Podemos realizar laços pelo comando *while*. A estrutura base é a mesma do comando *for*, porém a sequência de comandos é realizada de forma diferente.

[expressão 1]

while ([condição]) {

[bloco de instruções]

[expressão 2] //expressão responsável por alterar o valor do controle do laço
}

Então qual é a diferença entre *for* e *while*? Utilizamos o comando *while* quando não sabemos a quantidade exata de interações que deve existir, ou seja, quantas vezes o laço deverá ser executado. Por exemplo, enquanto a variável *número* for diferente 5, o programa exibirá ao usuário “o número é menor que 5”. Veja como é simples.

```
int numero = 0;
while (numero != 5) {
    System.out.println("O número é menor que 5");
    numero++;
}
```

Um outro comando de iteração (laços) que podemos utilizar é o *do-while*. A principal diferença desse comando com os outros de laço (*for*, *while*) é que, nesta estrutura, o bloco de instruções será executado pelo menos uma vez, pois diferente dos demais, esse comando realiza a verificação de condição após executada a iteração.

```
int numero = 0;
do {
    System.out.println("O número é menor que 5");
    numero++;
} while (numero != 5);
```

Para finalizar o controle de fluxo, temos dois comandos especiais que podem nos ajudar quando quisermos interromper ou continuar com um bloco de instruções. Esses comandos são *break* e *continue*. Mas vamos pensar em outro exemplo, digamos que queremos criar um laço que imprima os números inteiros inicializando

por 1 e podendo ir até 100. Sabemos que podemos utilizar o comando *for* para criar essa repetição, ok? Porém, nosso programa exige que essa interação deverá ser interrompida quando o número for 10, e agora? Para isso, podemos usar nosso comando *break*.

```
int numero;

for (numero = 1; numero <= 100; numero++){
    System.out.println("Número válido");

    if (numero == 10) break;
}
```

O comando *break* pode ser utilizado em todos os laços de repetições que vimos (*for*, *while*, *do-while*) como também nos comandos de decisão (*if*, *switch-case*). Já o comando *continue* é utilizado somente por laços de repetição, pois quando executamos este comando, nosso programa volta obrigatoriamente para o laço, a fim de testar novamente a condição. Basicamente, o comando *continue* vai ignorar o código e voltar para o laço, enquanto o comando *break* interrompe e sai do laço. Por exemplo, queremos que nosso programa imprima os valores de 1 a 100, porém não queremos que o número 10 seja impresso ao usuário. Para isso, utilizamos o *continue*.

```
int numero = 0;

while (numero < 100){

    numero++;

    if (numero == 10) continue;

    System.out.println(numero);

}
```

Se a condição do *if* retornar verdade, nosso programa ignora as próximas instruções (*System.out.println*) e retorna para o comando *while*, fazendo a nova verificação de condição.

Quando estamos desenvolvendo nosso programa, precisamos implementar a lógica que nosso algoritmo deve seguir. E, para isso, temos todos esses comandos para realizar e decidir qual é o melhor fluxo de instruções para executar nosso programa. Todas essas linhas de instruções ficam dentro de uma classe e representam as características dos nossos objetos. Vamos entender mais sobre isso?

1.3 Identificando classes e objetos

Quando utilizamos POO é necessário compreender alguns elementos que compõem a construção do nosso algoritmo. Podemos destacar os elementos: classes, objetos, atributos e métodos.

As classes são consideradas projetos de um objeto que possuem características similares. Definindo os possíveis comportamentos e estados para esses objetos. Esses possíveis comportamentos são chamados de métodos. Podemos representar nossa classe por meio de um substantivo que represente algo abstrato, por exemplo, uma pessoa, uma cidade, uma escola.

Algumas características importantes nos ajudam a lembrar, quando criamos nossa classe:

- toda classe possui um nome;
- identificamos a visibilidade (*public, private, protected*);
- verificamos seus métodos (características e ações compartilhadas por esses objetos).

Sua estrutura base é bem simples,

```
Visibilidade class NomeDaClasse{  
    //Seus atributos  
    //Seus métodos  
}
```

Para entender melhor, vamos criar uma classe que represente as pessoas. Inicialmente sabemos que precisamos criar um nome para essa classe (Pessoas), logo após isso precisamos identificar quais atributos que são compartilhados pelos objetos pessoas (nome, idade, peso, altura e gênero). Esses atributos são variáveis que nossos objetos vão possuir, podendo ser declarada por qualquer tipo (*int, String e outros*). Por fim, criamos nossos métodos que representam as ações que são compartilhadas pelos objetos ‘pessoas’ (andar, falar, comer). Ou seja, quando o programa receber a mensagem contendo o nome de algum método existente na nossa classe, aquela ação será executada no objeto correspondente.

```
public class Pessoas{  
    //Seus atributos  
    public String nome, gênero;  
    public int idade;  
    public float peso, altura;  
    //Seus métodos  
    public void andar(){  
        //ações do método andar  
    }  
    public void falar(){  
        //ações do método falar  
    }  
    public void comer(){  
        //ações do método comer  
    }  
}
```

Ou seja, as classes classificam um objeto compartilhando seus atributos e comportamentos, porém cada objeto é representado com valores distintos.

Logo no início, quando falamos sobre orientação a objetos, vimos que os objetos nada mais são do que a representação de entidades do nosso mundo real, ou seja, usamos objetos para representar elementos e/ou abstrações que vão nos ajudar na construção do nosso programa. Basicamente, os objetos são as características definidas pela nossa classe. Analisando a estrutura acima, notamos que a classe *Pessoa* não tem vida, é apenas um conceito de como os objetos que se identificam como pessoa devem ter seus atributos. Mas os objetos *Maria* e *João* possuem vida. Cada um desses objetos compartilha da mesma estrutura, porém, as suas informações são distintas (figura a seguir).

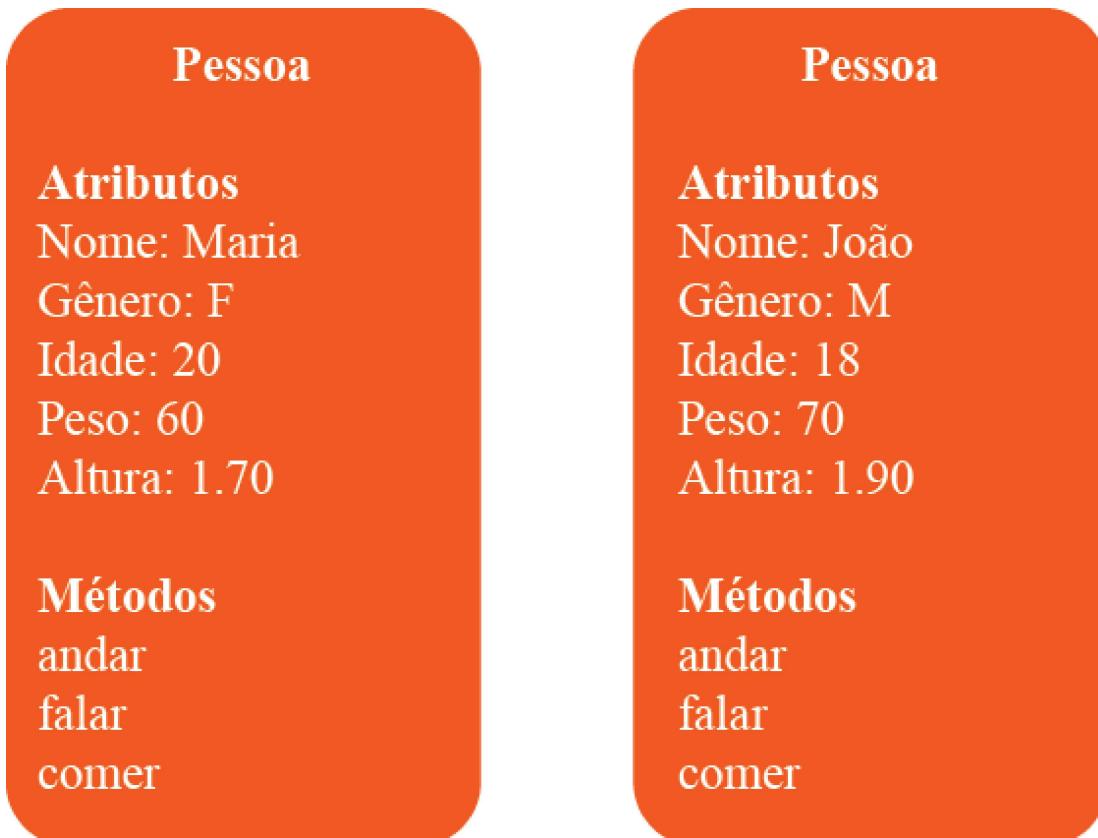


Figura 4 - Exemplo

de objetos diferentes que compartilham os mesmos atributos e métodos, pois são da classe *Pessoa*.. Fonte:

Elaborada pelo autor, 2018.

Ou seja, a classe *Pessoa* não pode andar, falar e comer, ela apenas especifica e define a estrutura de como é uma pessoa. Mas os objetos criados que são do tipo pessoa, poderão sim andar, falar e comer. Um outro exemplo entre classes e objetos, seria os carros. Podemos definir uma classe *carros*, contendo os atributos e comportamentos de qualquer carro. Porém, cada carro seria considerado um objeto daquela classe, pois possuem informações diferentes.

```

public class Carros{
    public String marca, placa, cor;
    public int anoDeFabricacao;
    public void acelerar(){
    }
    public void frear(){
    }
    public void ligar(){
    }
    public void desligar(){
    }
}

```

Então, como é que se faz para criar um objeto? Para criarmos nosso objeto precisamos instanciá-los no nosso programa. Para isso, fazemos o uso do comando `new` para construirmos o objeto na nossa classe principal, que vamos chamar de **Programa** (*main*). Por exemplo, vamos instanciar um objeto do tipo *carros*?

```

class Programa{
    public static void main(String[] args){
        Carros meuCarro; // variável meuCarro que é do tipo Carros
        meuCarro = new Carros(); //criando um novo objeto
    }
}

```

Após instanciado o novo objeto do tipo *carros*, podemos acessar este novo objeto, por meio da variável *meuCarro*, e alterar seus atributos (*marca*, *placa*, *cor* e *anoDeFabricacao*) e utilizar seus métodos (*acelerar*, *frear*, *ligar* e *desligar*).

```
class Programa{  
    public static void main(String[] args){  
        Carros meuCarro;  
        meuCarro = new Carros();  
  
        meuCarro.placa = "ABC 1234";  
        meuCarro.anoDeFabricacao = 2010;  
  
        System.out.println("A placa do carro é: " + meuCarro.placa);  
    }  
}
```

Como podemos notar, o ponto (.) é utilizado para acessar as informações do objeto *meuCarro*. Vimos que os métodos são os comportamentos que o nosso objeto pode ter, por exemplo, na nossa classe *Carros* os objetos instanciados possuem como método *acelerar*, *frear*, *ligar* e *desligar*. Por conveniência, esses métodos são nomeados por um verbo, de forma que fica de melhor entendimento, quando for realizada a chamada desses métodos pelos objetos. A melhor forma de acessar os atributos de um objeto, é utilizando os métodos *get* e *set*.

VOCÊ SABIA?

Um novo sistema criado a partir do Java EE traz estabilidade e transparência ao complexo mundo das negociações de créditos financeiros no Brasil. O sistema bancário brasileiro escolheu a linguagem Java, pois a plataforma Java para aplicativos orientados a objetos proporciona transações escaláveis, de alto desempenho e altamente seguros e confiáveis. Além de sua ampla popularidade nos círculos bancários brasileiros, o Java EE foi a única plataforma que integra o conjunto abrangente de APIs necessárias para um sistema tão complexo quanto o C3; estes incluem o Java Message Service (JMS) e o Java Open Transaction Manager (JOTM) (BILHARINHO, 2012).

Podemos utilizar o método *get*, quando quisermos ter acesso ao valor de um atributo. E dentro do método precisamos apenas informar qual será o retorno do atributo.

```
public String getNomeDoCliente(){
```

```
    return nomeDoCliente;
```

```
}
```

```
public int getIdadeDoCliente(){
```

```
    return idadeDoCliente;
```

```
}
```

Já o método *set*, é utilizado quando queremos atribuir ou alterar o valor de um atributo. Por padrão, esse método não possui retorno, apenas executa o bloco de instruções, informando dentro do seu método.

```
public void setNomeDoCliente(nome){
```

```
    this.nomeDoCliente = nome;
```

```
}
```

```
public void setIdadeDoCliente(idade){
```

```
    this.idadeDoCliente = idade;
```

```
}
```

Como podemos notar, o método *set* necessita que seja passado por parâmetro () o valor que será atribuído ou modificado naquele atributo. Utilizamos o comando *this.*, para fazer referência ao atributo da classe dentro da qual se está executando, porém, sua utilização é opcional.

Podemos criar métodos que executem um bloco de instruções e/ou retorne alguma informação para o usuário. Por exemplo, vamos criar um método que atribui uma idade futura do usuário, a partir da idade atual do usuário, armazenando essas duas informações (idade atual e idade futura).

```
public void idadeFutura(int idade){  
    this.idadeAtual = idade;  
    this.idadeFutura = idade + 2;  
}
```

O método *void* que utilizamos, representa que esse método irá apenas realizar uma alteração de informação nos atributos, e não retornará nenhuma mensagem de volta para o usuário. E se o método precisa dar um retorno ao usuário? Em vez de utilizarmos a palavra *void*, informaremos o tipo do valor de retorno daquele método.

Por exemplo, vamos imaginar que o método *idadeFutura* recebe do usuário sua idade atual e retorna ao usuário sua idade daqui a 10 anos. Sabemos que o tipo de retorno da idade deverá ser do tipo inteiro.

```
int idadeFutura(int idade){  
    return idade + 10;  
}
```

VOCÊ QUER VER?

Percebeu o quanto a linguagem Java é muito rica e simples? Que tal conhecer um pouco mais da sua história? Sabia que comparada a outras linguagens, Java é considerada uma linguagem recente? Essa linguagem foi criada em 1991, na Sun Microsystems (GUANABARA, 2015). Aprenda um pouco mais acessando o vídeo <<https://www.youtube.com/watch?v=sTX0UEplF54>> (<https://www.youtube.com/watch?v=sTX0UEplF54>).

E se quisermos inicializar nosso objeto já com alguns atributos? Será que é necessário chamar todos métodos os *get* e *set* para cada atributo?

1.4 Implementando construtores

Quando quisermos inicializar um objeto com alguns atributos no momento que estamos instanciando (criando) aquele objeto, utilizamos os construtores. O construtor é um método especial que pode inicializar e atribuir informações em um objeto. O nome utilizado em um construtor deve seguir o mesmo nome da classe na qual ele pertence. Ele deve receber, por parâmetro, os valores informados pelo programa, e após isso, atribuir esses valores aos atributos daquele objeto. É importante lembrar que, na estrutura de um construtor, não deve ter nenhuma instrução de retorno.

```
class Carros{
    public int anoDeFabricacao;
    public String marca;

    public Carros(int ano, string marca) {      //Método construtor
        this.anoDeFabricacao = ano;
        this.marca = marca;
    }
}
```

E como fica nosso programa principal? Já sabemos que para instanciar um objeto utilizamos a palavra *new*. Porém, se quisermos inicializar esse objeto com atributos é só informar os valores em parêntese após a execução do *new*.

```
public static void main (String[] args){
    Carros novoCarro;
    novoCarro = new Carros(2010, "FIAT");
}
```

Como podemos perceber, o objeto *novoCarro* foi instanciado e inicializado com os valores de *anoDeFabricacao* = 2010 e *marca* = “FIAT”. Mesmo que o compilador não declare um método construtor, se faz necessário a construção de um construtor padrão. De forma que ele não receba nenhum argumento por parâmetro e não execute nenhuma instrução.

```
public Carros(){  
}
```

Perceba que a utilização de um construtor padrão (vazio), não nos impossibilita de ter também um construtor que receba parâmetros e inicialize junto com o objeto.

```
class Carros{  
  
    public int anoDeFabricacao;  
  
    public String marca;  
  
    public Carros(){           //Método construtor padrão  
    }  
  
    public Carros(int ano, string marca) { //Construtor com parâmetros  
        this.anoDeFabricacao = ano;  
        this.marca = marca;  
    }  
}
```

Ou seja, nosso programa nos permite criar um objeto *novoCarro* sem nenhum atributo inicial, como também um objeto *outroCarro* sendo inicializado com os atributos de *anoDeFabricacao* e *marca*.

CASO

Uma empresa de venda de automóvel decidiu desenvolver um programa para cadastro e gerenciamento de seus clientes. O programa deve armazenar as seguintes informações do cliente: nome, idade, contato e endereço. Obrigatoriamente, um cliente quando criado deverá possuir, pelo menos, as informações de nome e idade.

Utilizando Programação Orientada a Objetos, os desenvolvedores identificaram que seria necessário criar uma classe chamada *Clientes*, que conterá a estrutura de atributos e comportamentos que um Cliente deve possuir. Como as informações de contato e endereço podem ser modificadas – caso o cliente se mude, por exemplo –, além dos métodos que retornam todas as informações, será necessário criar métodos que alterem essas duas informações (contato e endereço). Notou-se também, que será necessário criar um método construtor que, obrigatoriamente, receba como parâmetros os atributos de nome e idade.

```
class Clientes{  
    //Atributos  
    public String nome, endereco;  
    public int idade, contato;  
    //Construtor com 2 parâmetros  
    public Clientes (string nome, int idade) {  
        this.nome = nome;  
        this.idade = idade;  
    }  
    //Métodos sem retorno  
    public void setContatoCliente(int contato){  
        this.contato = contato;  
    }  
    public void setEnderecoCliente(string endereco){  
        this.endereco = endereco;  
    }  
    //Métodos com retorno  
    public string getNomeDoCliente(){  
        return nome;  
    }  
    public int getIdadeDoCliente(){
```

```

    return idade;
}

public int getContatoDoCliente(){
    return nome;
}

public string getEnderecoDoCliente(){
    return endereco;
}

}
}

```

Para finalizar, os desenvolvedores perceberam que na classe principal, ficará responsável por instanciar esse novo objeto *Cliente*, junto com a informação de nome e idade. Como também, após a criação desse novo cliente, apresentar na tela do usuário todas as informações daquele cliente.

```

public static void main (String[] args){
    Clientes novoCliente;
    novoCliente = new Clientes ("Maria", 20);

    //Passando as informações restantes sobre o cliente
    novoCliente.setContatoCliente (1234567890);
    novoCliente.setEnderecoCliente ("Rua Luis Sagrado, 147");

    //Imprimindo ao usuário todas as informações do cliente
    System.out.println(novoCliente.getNomeDoCliente());
    System.out.println(novoCliente.getIdadeDoCliente());
    System.out.println(novoCliente.getContatoDoCliente());
    System.out.println(novoCliente.getEnderecoDoCliente());
}

}

```

Após a finalização desse programa, a loja de venda de automóveis conseguiu agilizar e melhorar o gerenciamento dos seus clientes.

Percebemos que utilizar o paradigma de orientação a objetos nos ajuda a resolver problemas pensando no nosso mundo real. Tornando assim nosso desenvolvimento mais rápido e eficaz. Quando optamos por utilizar a linguagem de programação Java, entendemos seus conceitos iniciais e percebemos o quanto essa linguagem é simples e motivadora. Pois para entendermos os problemas, precisamos separar e definir os objetos/classe envolvidos na situação e compreender como eles se relacionam. Ou seja, essa atividade sempre será a primeira etapa de um projeto de software.

Síntese

Chegamos ao final do capítulo. E com isso, iniciamos nossos conhecimentos sobre programação orientada a objetos utilizando a linguagem de programa Java. Vimos a importância de utilizarmos o paradigma de orientação a objetos, enxergando a programação como o mundo real. Identificando seus objetos e classes, atribuindo suas características e comportamento. E assim, podemos analisar nosso problema de maneira mais eficaz e criarmos nosso programa com sucesso.

Neste capítulo, você teve a oportunidade de:

- compreender os paradigmas que envolvem uma Programação Estruturada e uma Programação Orientada a Objetos, identificando suas similaridades e diferenças;
- conhecemos a sintaxe e semântica que envolvem a linguagem de programação Java, identificando conceitos de classe, objetos, atributos e métodos e construtores;
- construímos nossos primeiros programas em Java, de forma a analisar, de maneira lógica, todo o fluxo do programa, com o auxílio de controles de repetições;
- aplicamos em exemplos práticos todos esses conceitos iniciais em problemas de casos reais.



◀ Clique para baixar o conteúdo deste tema.

Bibliografia

BILHARINHO, E. R. Utilizando Código Java em Banco de Dados Oracle. **Revista MundoJ**, edição 51, janeiro-fevereiro, 2012. Disponível em: <<http://www.univale.com.br/unisite/mundo-j/artigos/51BDOoracle.pdf>> (<http://www.univale.com.br/unisite/mundo-j/artigos/51BDOoracle.pdf>). Acesso em: 14/08/2018.

DEITEL, P; DEITEL, H. **Java: como programar**. 10. ed. São Paulo: Pearson: 2016.

FONSECA FILHO, C. F. **História da Computação**: o caminho do pensamento e da tecnologia. Porto Alegre: EDIPUCRS, 2007. Disponível em: <<http://www.pucrs.br/edipucrs/online/historiadacomputacao.pdf>> (<http://www.pucrs.br/edipucrs/online/historiadacomputacao.pdf>). Acesso em: 14/08/2018.

FURGERI, S. **Programação Orientada a Objetos**. São Paulo: Érica, 2015

GUANABARA, G. **Curso de Java #01 - História do Java - Gustavo Guanabara**. Canal Curso em Vídeo, YouTube, publicado em 2 de mar de 2015. Disponível em: <<https://www.youtube.com/watch?v=sTX0UEplF54>> (<https://www.youtube.com/watch?v=sTX0UEplF54>). Acesso em: 14/08/2018.

MACHADO, H. **Os 4 pilares da programação orientada a objetos**. DevMedia, 2018. Disponível em: <<https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264>> (<https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264>). Acesso em: 14/08/2018.

MANZANO, J. A. G.; COSTA JR., R. **Programação de Computadores com Java**. São Paulo: Érica, 2014. 127p.

MARINHO, A. L. **Programação Orientada a Objetos**. São Paulo: Pearson Education do Brasil, 2016. 167p.

SEPE, A.; MAITINO, R. N. **Programação orientada a objetos**. Londrina: Editora e Distribuidora Educacional AS, 2017. 176p.

SILVA FILHO, A. M. **Introdução à Programação Orientada a Objetos Com C++**. São Paulo: Editora Campus, 2010.

ORACLE. **The History of Java Technology. Portal Oracle**, 2018. Disponível em: <<http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html>><http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html>>. Acesso em: 14/08/2018.

RICARTE, I. L. M. **Programação Orientada a Objetos**. Universidade Estadual de Campinas. Departamento de Engenharia de Computação e Automação Industrial. Faculdade de Engenharia Elétrica e de Computação. Campinas: Unicamp, 2001. Disponível em: <<http://www.dca.fee.unicamp.br/cursos/PooJava/Aulas/poojava.pdf>><http://www.dca.fee.unicamp.br/cursos/PooJava/Aulas/poojava.pdf>>. Acesso em: 14/08/2018.

ZANETTE, A. Java, a linguagem do programador “RICO”. Entenda!. Portal BeCode, 2017. Disponível em: <<https://becode.com.br/java-linguagem-programador-rico/>><https://becode.com.br/java-linguagem-programador-rico/>>. Acesso em: 16/08/2018.