

```
var = ghost)  
  
/960  
  
var = ghost)  
  
add (12.201 sub  
format avg  
(type cell = var/960)  
add (12.555401 subtract  
|  
format avg  
(type cell = var/960) file copy Duvir
```



# CLASSES E OBJETOS

# SUMÁRIO

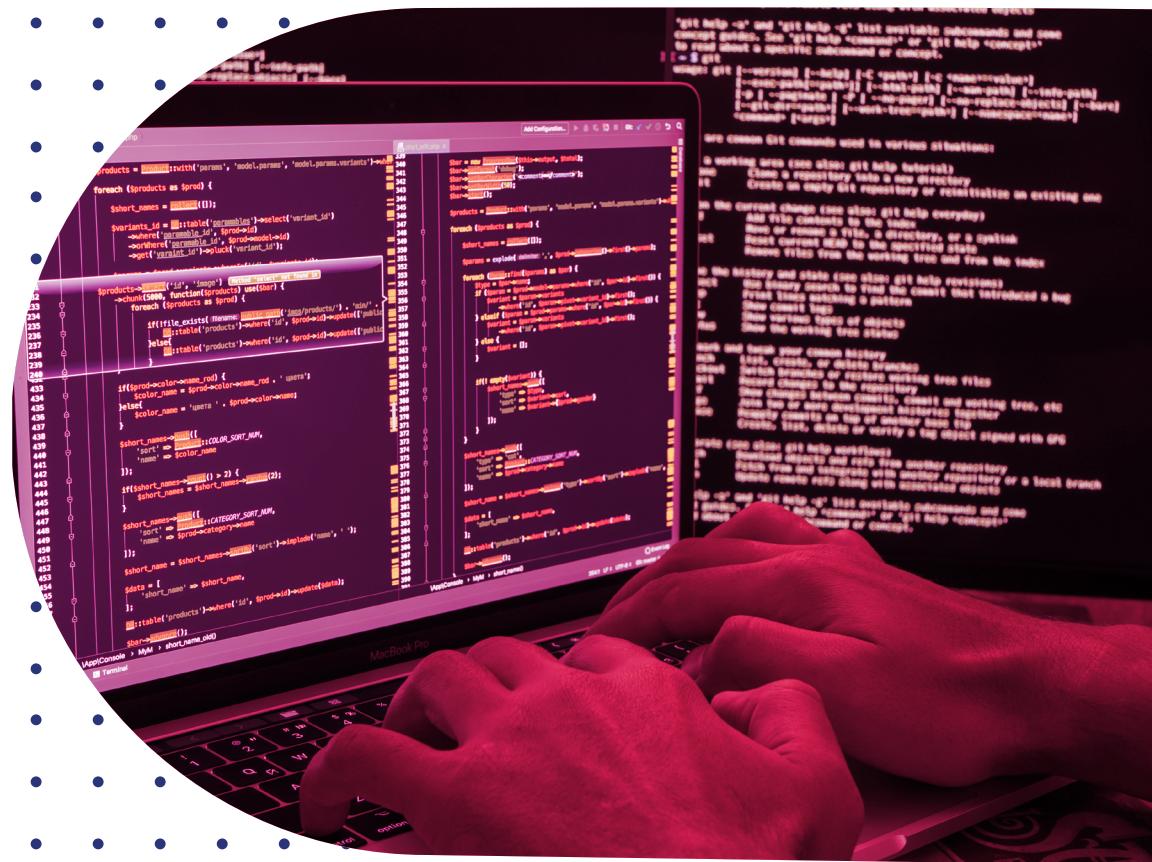
<b>Paradigmas de Programação e primeiro contato com Orientação a Objetos .....</b>	<b>3</b>
Programação Imperativa .....	4
Programação Funcional .....	4
Programação Lógica .....	4
Programação Orientada a Objetos.....	5
<b>Objetos em Java (tipos por referência).....</b>	<b>13</b>
<b>Atributos em Java (valores default, objetos como atributos) .....</b>	<b>17</b>
<b>Construtores .....</b>	<b>20</b>
<b>Encapsulamento.....</b>	<b>23</b>
<b>Getters &amp; Setters.....</b>	<b>26</b>
<b>Métodos e atributos estáticos.....</b>	<b>28</b>



# PARADIGMAS DE PROGRAMAÇÃO E PRIMEIRO CONTATO COM ORIENTAÇÃO A OBJETOS

Um paradigma refere-se a um padrão de pensamento, um modelo que guia a realização de um conjunto de atividades. Um paradigma de programação, portanto, pode ser visto como um padrão de resolução de problemas computacionais.

Esse termo é utilizado, muitas vezes, para classificar linguagens de programação.



Linguagens de programação são projetadas para suportar um ou mais paradigmas de programação, o que permite aos programadores escrever programas utilizando diferentes abordagens.

Alguns dos principais paradigmas de programação:

- Imperativa.
- Funcional.
- Lógica.
- Orientada a Objetos.

# Programação Imperativa

Para programar devem ser utilizadas algumas sequências ordenadas de comandos, confira a seguir:

- › Executar cálculos.
  - › Atribuir valores a variáveis.
  - › Obter entradas.
  - › Produzir saídas.
  - › Redirecionar o controle para outro ponto do programa.
- • • • • • •

Algumas linguagens de Programação Imperativa predominantes são: Cobol, Fortran, C e Perl.

• • • • • • •

# Programação Funcional

A programação funcional baseia-se no cálculo lambda e utiliza funções anônimas. É ela quem modela um problema como uma coleção de funções matemáticas, onde cada função mapeia um espaço de entrada (domínio) para um espaço de saída (imagem). Diferentemente da Programação Imperativa, não existe a noção de estado na Programação Funcional, pois não há declaração de variáveis, comandos de atribuição ou laços.

• • • • • • •

Algumas linguagens de Programação Funcional importantes: Lisp e Haskell.

• • • • • • •

# Programação Lógica

Na Programação Lógica o problema é modelado declarando qual resultado o programa deve obter, em vez de como ele deve ser obtido. Nessa forma de programação, os objetivos são expressos como uma coleção de asserções

(regras) sobre os resultados e restrições de computação. É muito empregada em inteligência artificial. Confira a seguir algumas características principais.

- › Não determinismo e *backtracking*.
  - › Permite encontrar várias soluções para um problema.
- • • • • •

Principal linguagem de programação lógica: Prolog.

• • • • • •

## Programação Orientada a Objetos

A Programação Orientada a Objetos (POO) se caracteriza como um programa definido como uma coleção de objetos. Ela surgiu como aposta para resolver problemas comuns na indústria de software, como produzir sistemas muito grandes e complexos de forma mais rápida, com maior confiabilidade e menor custo. Teve grande impacto na indústria de software e é um dos paradigmas mais usados até hoje.

• • • • • •

Suportada total ou parcialmente por muitas linguagens de programação: C++, Java, C#, Ruby e Python.

• • • • • •

Em **POO**, todos os objetos têm determinados estados e comportamentos. Esses estados são descritos pelos atributos das classes. Os comportamentos são definidos pelos métodos. Objetos interagem entre si, passando mensagens e as mensagens podem transformar o estado de um objeto.

Os objetos, muitas vezes, são representações de entidades do mundo real. Seus atributos representam as propriedades, as características e o estado daquele objeto em um determinado momento e seus métodos representam as ações e as operações que aquele objeto pode realizar.

Um conceito fundamental de POO é o encapsulamento de atributos e métodos. Trata-se, basicamente, de ocultar os valores de certos atributos e os detalhes de implementação de seus métodos.

A estrutura de um objeto, ou seja, quais atributos e métodos o objeto possui, é definida pela sua classe. Objetos são vistos como instâncias de uma classe. A classe também determina a interface do objeto, ou seja, quais atributos e métodos são visíveis para outros objetos. Objetos se comunicam entre si enviando mensagens, que são solicitações de execução, ou seja, chamadas de métodos.

Para entender melhor os conceitos de Classe, Objeto, Atributo e Método, pense em um carro. A classe Carro define que seus atributos são: cor, número de portas, tamanho das rodas, fabricante, ano etc. Também define que seus métodos são: acelerar, frear, buzinar, virar etc.

Todo objeto da classe Carro, ou seja, cada carro que for instanciado pelo nosso programa, vai possuir esses atributos (cada um com seus valores específicos) e vai poder executar esses métodos.

Por exemplo, ao instanciar um objeto “unoDeFirma” e definir que ele possui “numPortas = 2” e “cor = branco”, você pode executar os métodos unoDeFirma.acelerar() e unoDeFirma.frear().

E ao instanciar um novo objeto “sandero”, definir seus próprios atributos como, por exemplo, “numPortas = 4” e “cor = prata”, pode-se executar os mesmos métodos “acelerar” e “frear”, tendo escrito o código desses métodos apenas uma única vez na declaração da classe Carro.

A classe, sozinha, não faz nada. Ela define a estrutura que os objetos, as suas instâncias, vão possuir. A classe Carro, por si só, não é um carro. O objeto “unoDeFirma” sim, representa um carro.

A palavra classe vem da biologia. Todos os seres vivos de uma mesma classe biológica têm uma série de atributos e comportamentos em comum, mas não são iguais, podem variar nos valores desses atributos e como realizam esses comportamentos.

Outro exemplo é a famosa: receita de bolo. Ninguém come uma receita de bolo, certo? Precisamos instanciá-la, ou seja, criar um objeto “bolo” a partir dessa especificação (a classe) para utilizá-la. Podemos criar inúmeros bolos a partir dessa classe (a receita). Eles podem ser bem semelhantes, alguns até idênticos, mas são objetos diferentes.

Pode até parecer óbvio para quem já conhece essa diferença, mas é uma dificuldade bastante comum em iniciantes no paradigma de OO não saber distinguir o que é classe e o que é objeto.

Java é uma linguagem de programação que implementa o paradigma de orientação a objetos. Você vai conferir de que forma são utilizados todos os conceitos de POO em Java, na prática, criando um pequeno projeto.

Esse projeto consiste de uma Conta Bancária (classe Conta). Cada conta desse banco (objetos) possui um Nome do Titular e um Saldo (atributos). Essas contas (objetos) podem realizar certas operações, como Saque e Depósito (métodos).

Vamos começar com os atributos da classe Conta. Utilizando a IDE de sua escolha, crie um novo projeto (chamado, por exemplo, "MeuBanco"). Nesse novo projeto, vamos criar uma classe chamada Conta e definir os atributos para nome do titular e saldo, confira a seguir:



```
public class Conta {  
    String nomeDoTitular;  
    double saldo;  
}
```

Você se lembra dos tipos primitivos e tipos por referência em Java? A cada classe nova que criamos, estamos gerando um novo tipo por referência. Você pode criar quantas classes desejar no seu programa.

Então, vamos criar a classe Conta (a “receita de bolo”). Mas, como você sabe, não se pode comer uma receita. É preciso fazer um bolo a partir dessa receita, ou seja, instanciar um objeto dessa classe. Portanto, vamos criar uma classe para conter o método “main”, que é o ponto de entrada da execução dos programas Java, confira:

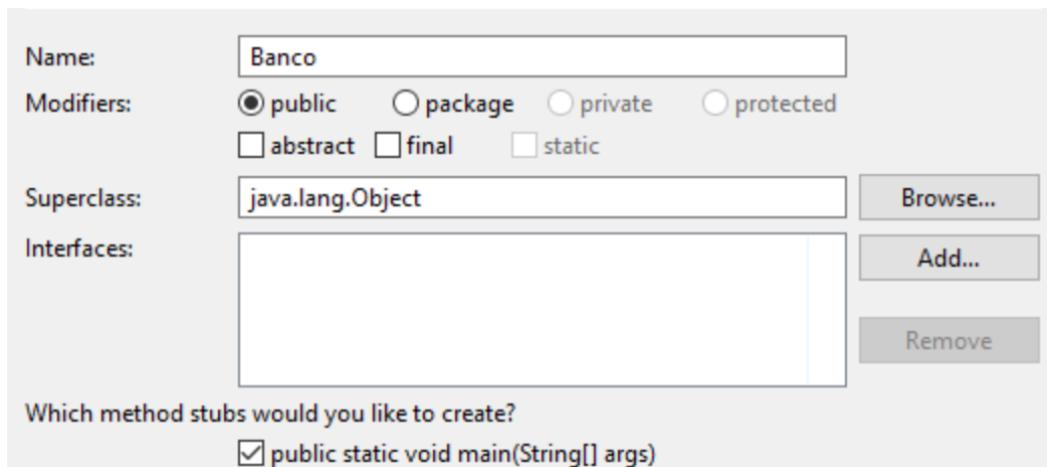


Figura 2 - Criação de nova na classe

Fonte: do Autor (2022)

Dentro do método “main”, vamos instanciar um objeto da classe Conta, utilizando a palavra-chave “new” e atribuindo a uma nova variável, confira o código:

```
public class Banco {  
    public static void main(String[] args) {  
        Conta minhaConta = new Conta();  
    }  
}
```

Lembre-se que se a classe Conta pertence ao mesmo pacote da classe Banco, então não é necessário adicionar aquela linha “import” no início do arquivo. Mas se for de pacotes diferentes, é necessário importar.

A partir de agora você já possui uma variável “minhaConta”, do tipo “Conta”, que se trata de um objeto da classe Conta. Então, pode definir os valores dos atributos do objeto recém-criado, como feito no código a seguir:

```
public class Banco {  
    public static void main(String[] args) {  
        Conta minhaConta = new Conta();  
  
        minhaConta.nomeDoTitular = "João";  
        minhaConta.saldo = 100.00;  
  
        System.out.printf("O saldo atual do cliente %s é de %.2f",  
                          minhaConta.nomeDoTitular, minhaConta.saldo);  
    }  
}
```

Em seguida, confira a sintaxe da declaração de classe:

```
public class Conta {
```

Entenda alguns importantes itens para a declaração de classe:

- › Para declarar uma classe, utilizamos um modificador de acesso, seguido pela palavra-chave “class” e, em seguida, o nome da classe.
- › Após o nome da classe, abrimos as chaves para definir o bloco de código que compreende a definição da classe, com seus atributos e métodos.

- › Esse conteúdo deve estar armazenado em um arquivo com extensão ".java", e o nome do arquivo deve ser o mesmo que o nome da classe.
- › Nomes de classe se iniciam com letra maiúscula, seguindo o padrão "PascalCase".

Agora, confira a sintaxe de declaração de atributos (ou "variáveis de instância"):

```
String nomeDoTitular;  
double saldo;
```

Perceba alguns importantes itens para a declaração de atributos a seguir:

- › Os atributos são variáveis que pertencem ao objeto e permanecem durante toda a sua existência.
- › Existem antes de um método ser chamado, durante a execução de um método e depois de o método ser finalizado.
- › São declarados dentro da classe, mas fora dos métodos.
- › Classes usualmente possuem métodos para manipular variáveis de instância (get/set).
- › Cada objeto possui sua própria cópia de cada atributo da classe, com seus próprios valores independentes.

Agora, você vai criar os métodos da nossa classe, ou seja, as operações que os objetos poderão executar. Inicialmente você vai criar o método para saque de dinheiro, dentro da definição da classe Conta, pois é uma operação específica dessa classe. Assim, todo objeto que instanciar dessa classe poderá executar esse método.

Para realizar a ação de sacar, é preciso saber a quantidade de dinheiro que o usuário deseja retirar da conta. Para isso, utilize um **parâmetro**. O parâmetro é uma variável comum, porém temporária, pois após o término da execução do método, ela deixa de existir. Vamos declarar o parâmetro com um tipo e um nome, na assinatura do método. Assim, sempre que o método for invocado, será preciso passar um valor para esse parâmetro, para que o método utilize esse valor ao realizar o saque, confira:

```
public class Conta {  
    String nomeDoTitular;  
    double saldo;  
  
    void sacar(double valor) {  
        double novoSaldo = this.saldo - valor;  
        this.saldo = novoSaldo;  
    }  
}
```

A palavra-chave “void” define que esse método não retorna nenhum valor quando for chamado.

Em seguida, declare uma variável “novoSaldo” dentro do escopo do método. Assim como o parâmetro, essa variável só existe dentro do método, não pode ser referenciada de fora dele e também deixa de existir ao fim da execução.

Perceba que para referenciar um atributo do próprio objeto dentro do método, você deve utilizar a palavra-chave “this”. Apesar de ser opcional, é uma **boa prática** sempre utilizar o “this” quando for referenciar um atributo do objeto, para deixar claro que não se trata de uma variável genérica interna do método.

Caso dentro do método você criasse uma variável com o mesmo nome de um atributo da classe, sem o “this” você estaria referenciando a variável local. Nesse caso, para referenciar o atributo, “this” não é opcional. O “this” significa “este”, ou seja, o próprio objeto que teve seu método invocado. Em outras linguagens de programação temos a palavra “self”, que na prática faz a mesma coisa que “this”. Vamos criar o método para depósito de valores na conta, confira a seguir:

```
void depositar(double valor) {  
    this.saldo += valor;  
}
```

Lembre-se que o operador “`+=`” soma o valor já existente na variável à esquerda com o valor da variável à direita, e guarda o resultado da soma na variável à esquerda. É uma abreviação da expressão “`this.saldo = this.saldo + valor`”. E se você quisesse saber o saldo atual da conta? Vamos criar um método para isso:

```
double getSaldo() {  
    return this.saldo;  
}
```

Note que na declaração do método não há mais a palavra “`void`”, mas sim o tipo “`double`”. O “`void`” indicava que o método não retornaria valor algum, já “`double`” indica que o método retorna um valor numérico real. Quando a declaração do método é diferente de “`void`”, é obrigatório retornar um valor do tipo declarado.

A palavra-chave “`return`” encerra a execução do método, retornando o valor descrito a seguir. Qualquer código escrito após a linha que contém “`return`” jamais será executado. Exceto se o “`return`” estiver dentro de um bloco de “`if...else`” ou laço (while, for).

Você poderia apenas utilizar o atributo “`saldo`” do objeto recém-criado, mas vai conferir mais adiante que utilizar um método “`get`” é uma prática bastante comum e indicada.

Agora, relembrre como declarar métodos em Java, com a imagem a seguir:

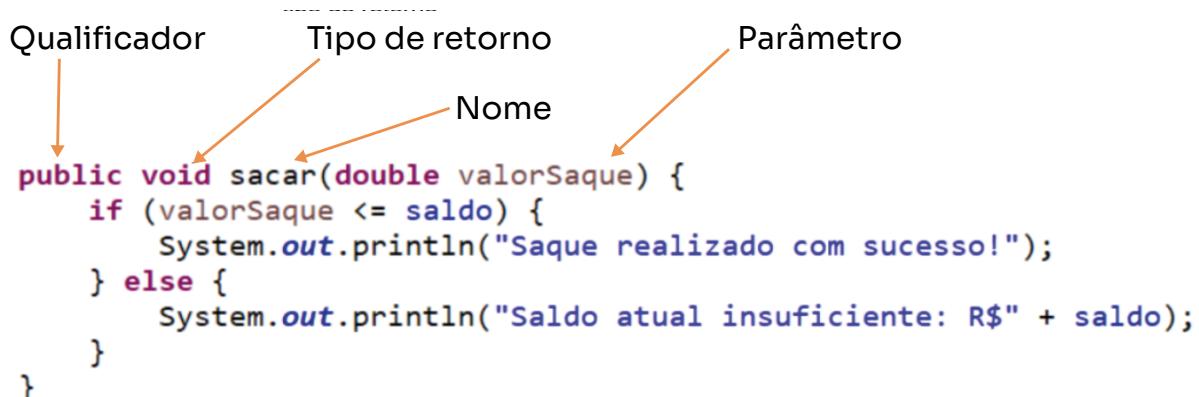


Figura 3 - Como declarar métodos em java

Fonte: do Autor (2022)

Agora, como executar os métodos da classe Conta? Para mandar uma mensagem ao objeto e pedir que ele execute um método, também use o ponto. O termo usado para isso é invocação de método, confira:

```
public static void main(String[] args) {  
    Conta minhaConta = new Conta();  
  
    minhaConta.nomeDoTitular = "João";  
    minhaConta.saldo = 100.00;  
  
    minhaConta.depositar(50);  
  
    System.out.printf("O saldo atual do cliente %s é de %.2f",  
        minhaConta.nomeDoTitular, minhaConta.saldo);  
}
```

## OBJETOS EM JAVA (TIPOS POR REFERÊNCIA)

Antes de falar sobre os objetos é importante você lembrar que em Java há:

- › Tipos primitivos (int, double, boolean, char, float, byte, short, long).
- › Tipos por referência.

Então, vamos lá. Os objetos são tipos por referência. Mas o que significa isso? Ao declarar uma variável de tipo primitivo, aquela variável guarda realmente aquele valor. Por exemplo:

```
int a = 10;  
int b = a;  
b = 20;  
System.out.printf("O valor de 'a' é: %d%n", a);  
System.out.printf("O valor de 'b' é: %d", b);
```

Ao declarar uma variável de classe, você não está guardando o objeto, mas sim uma referência para o endereço de memória onde ele está guardado. Por isso você deve utilizar a palavra-chave “new”, diferente de quando declara o valor de uma variável de tipo primitivo, confira:

```
Contas c1 = new Conta();  
Conta c2;  
c2 = new Conta();
```

O que está sendo atribuído à variável “c1” não é o valor do objeto, mas sim um novo endereço de memória onde esse dado está guardado. Por isso, o correto seria dizer que “c1 é uma referência a um objeto do tipo Conta”, e não que “c1 é um objeto do tipo Conta”. Mas para encurtar, no dia a dia, falamos da segunda forma mesmo.

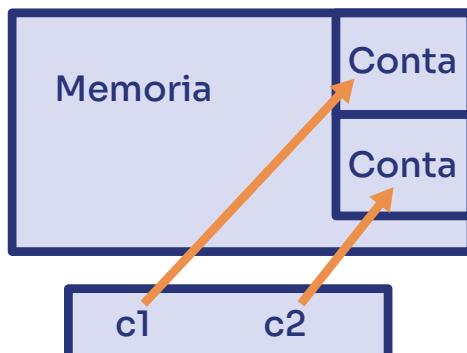


Figura 4 - Atribuição de referência a dois objetos

Fonte: do Autor (2022)

Mas o que isso significa na prática? Vamos fazer um exemplo de tipos primitivos parecido com o anterior para entender por que é importante você saber disso. Confira:

```
Conta c1 = new Conta();  
c1.depositar(100);  
  
Conta c2 = c1;  
c2.depositar(200);  
  
System.out.printf("O saldo de 'c1' é: %f\n", c1.saldo);  
System.out.printf("O valor de 'c2' é: %f", c2.saldo);
```

O que aconteceu aqui? O operador de atribuição, “=”, copia o valor da variável à direita para a variável à esquerda, certo? O que temos na variável não é o valor do objeto, mas sim o endereço na memória onde esse objeto está guardado. Então, quando você atribui a uma variável de classe (objeto) o valor de outra variável de classe (objeto), está apenas dizendo que, a partir daquela linha, essas duas variáveis vão apontar para o mesmo objeto.

Ou seja, quando c2 recebeu o valor de c1, aconteceu o que você pode perceber no código e figura a seguir:

```
Conta c1 = new Conta();  
Conta c2 = c1;
```

O que aconteceu foi que c2 passou a fazer referência ao mesmo objeto que c1 já referenciava.

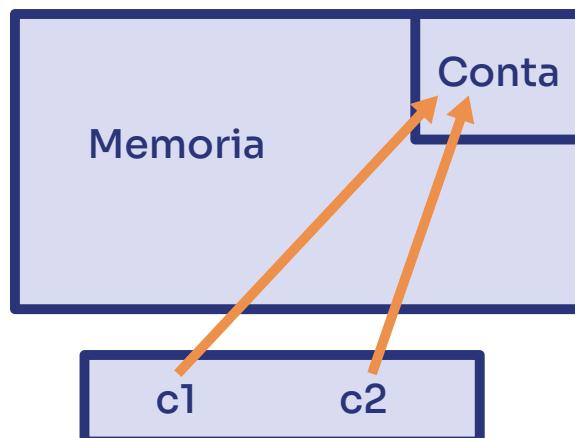


Figura 5 - Atribuições de referência a um mesmo objeto

Fonte: do Autor (2022)



Agora, ambos representam o mesmo objeto. Outra forma de entender é que você escreve apenas uma vez a palavra “new”, portanto só existe um objeto Conta. Se testar a igualdade dessas variáveis com o operador “==”, o resultado será “true”, confira.



```
Conta c1 = new Conta();
Conta c2 = c1;
System.out.println(c1==c2);
```

E se você criar dois objetos, ou seja, executar duas vezes o “new Conta()”, e ambos os objetos possuírem os mesmos valores para os atributos? Confira um exemplo no código a seguir:



```
Conta c1 = new Conta();
Conta c2 = new Conta();
c1.nomeDoTitular = "João";
c1.saldo = 100;
c2.nomeDoTitular = "João";
c2.saldo = 100;

System.out.println(c1==c2);
```

Aqui o resultado será “false”, pois são objetos diferentes. “c1” aponta para um objeto na memória, enquanto “c2” aponta para outro. E o que acontece quando você passa um objeto como argumento/parâmetro de um método?



```
c1.transferePara(c2, 10);
```

Em Java, passar um objeto como parâmetro funciona como uma atribuição (“=”). Ou seja, o parâmetro (que é uma variável interna do método) recebe uma cópia do valor da variável passada. **E qual o valor dessa variável?** O endereço onde o objeto está guardado, não o objeto em si. Por isso, qualquer alteração no objeto, realizada dentro do método, se refletirá no objeto fora do método.

# ATRIBUTOS EM JAVA (VALORES DEFAULT, OBJETOS COMO ATRIBUTOS)

Ao declarar uma variável comum em Java, você deve atribuir algum valor, do contrário não conseguirá utilizá-la. Porém, atributos funcionam de forma diferente, pois ao instanciar um objeto, seus atributos já recebem valores padrão. Numéricos recebem 0, booleanos recebem *false*, *Strings* recebem *null*. Mas também é possível definir os valores padrão quando estamos definindo a classe, confira:

```
class Conta {  
    String nomeDoTitular = "João";  
    double saldo = 10.0;
```

Assim, ao instanciar um objeto, ele já é criado com esses valores “default”. Você também pode utilizar objetos como atributos. Pense que em um sistema real, uma conta que possui apenas o nome do titular está bastante incompleta, certo? Nós, como banco, queremos mais informações sobre esse nosso cliente. Para isso, poderíamos inserir novos atributos “simples” na nossa classe, como a seguir:

```
class Conta {  
    String nomeDoTitular;  
    String sobrenomeDoTitular;  
    String cpfTitular;  
    double saldo;
```

Ao continuar dessa forma, a classe Conta terá muitos atributos que, na verdade, nem dizem respeito diretamente a uma Conta, certo? Como o próprio nome do atributo diz, eles dizem respeito ao titular da conta, ou seja, a um cliente. Então você pode criar uma nova classe Cliente, e essa sim possui os atributos nome, sobrenome e CPF, perceba no exemplo seguinte:

```
● ● ●  
class Cliente {  
    String nome;  
    String sobrenome;  
    String cpf;  
}
```

Existindo uma classe Cliente com os atributos de um cliente, podemos ter um atributo do tipo Cliente na classe Conta.

```
● ● ●  
class Conta {  
    Cliente titular;
```

• • • • • • •

Lembre-se que a variável “cli” e o atributo “titular” do objeto “minhaConta” estão apontando para o mesmo objeto a partir da quarta linha do código acima. E tanto faz atribuir os valores de nome, sobrenome e CPF diretamente à variável “cli” ou da forma feita no print, pois ambos apontam para o mesmo objeto.

• • • • • • •

Da mesma forma, podemos utilizar os atributos de clientes em outras classes que precisemos criar no nosso programa no futuro. E acessar os atributos e métodos do Cliente que está dentro da Conta utilizando o “.”, confira:

```
● ● ●  
public static void main(String[] args) {  
    Conta minhaConta = new Conta();  
    Cliente cli = new Cliente();  
    minhaConta.titular = cli;  
    minhaConta.titular.nome = “João”;  
    minhaConta.titular.sobrenome = “Oliveira”;  
    minhaConta.titular.cpf = “11122233345”;
```



**Atenção:** da mesma forma que um atributo String tem um valor padrão *null*, um atributo de qualquer tipo por referência também será *null* por padrão. O “*null*” em Java significa que aquela variável não está referenciando nenhum objeto.

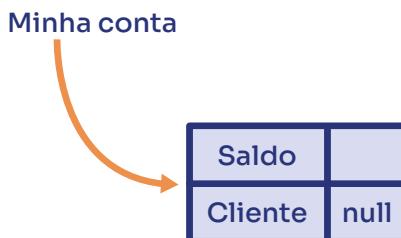


Figura 6 - Esquema de valor padrão *null*

Fonte: do Autor (2022)

Portanto, se você inicializa uma variável do tipo Conta e tenta acessar algum sinal do atributo “titular” sem inicializar um objeto Cliente, vai se deparar com uma mensagem de erro do tipo “*NullPointerException*”, como no exemplo a seguir:

```
public static void main(String[] args) {  
    Conta minhaConta = new Conta();  
    // Cliente cli = new Cliente();  
    // minhaConta.titular = cli;  
    minhaConta.titular.nome = "João";  
    minhaConta.titular.sobrenome = "Oliveira";  
    minhaConta.titular.cpf = "11122233345";  
}
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot  
assign field "nome" because "minhaConta.titular" is null  
at banco.Banco.main(Banco.java:8)
```

Você já aprendeu que ao instanciar um objeto com a palavra “new”, seus atributos são inicializados com alguns valores default: “0” para números, “false” para booleanos e “*null*” para referências. Pensando nisso, podemos definir um valor padrão para o atributo “titular”, para sempre inicializar um objeto Cliente. Dessa forma, evitamos o erro anterior e economizamos linhas de código ao instanciar um objeto Conta, entenda:

```
● ● ●  
class Conta {  
    Cliente titular = new Cliente();  
    double saldo;
```

No início, saber como o objeto x ou y está alocado na memória pode parecer bastante confuso. Mas com o tempo, você se acostuma com esses conceitos e passa a perceber rapidamente o efeito de passar referências de um lado para outro.

Nesse momento, o mais importante é você assimilar que, em Java, uma variável nunca carrega um objeto, mas sim uma referência para ele. Um sistema orientado a objetos é um grande conjunto de classes que se comunicam para realizar as funcionalidades, cada classe com sua especialidade. A classe Banco utiliza a classe Conta, que por sua vez utiliza a classe Cliente. A classe Cliente pode utilizar uma outra classe chamada Endereço, por exemplo. Os objetos dessas classes (e de várias outras mais) se comunicam entre si, invocando os métodos umas das outras, que também chamamos de “trocar mensagens”.

## CONSTRUTORES

Em um código, quando é utilizada a palavra-chave “new” para inicializar um objeto, estamos na verdade invocando o construtor da classe. O construtor é como se fosse um método que inicializa o objeto. Ele é declarado com o mesmo nome da classe, confira:

```
● ● ●  
class Conta {  
    Cliente titular = new Cliente();  
    double saldo;  
  
    Conta() {  
        System.out.println("Nova conta criada!");  
    }  
}
```

Até agora estávamos chamando de “new” sem declarar nenhum construtor. Isso é possível, pois, por padrão, o próprio Java cria um construtor vazio, caso o desenvolvedor não escreva um. Quando escrevemos um construtor próprio, esse construtor padrão do Java deixa de existir.

Mas para que serve um construtor?

O mais interessante é que um construtor pode receber parâmetros. Considerando que um construtor é sempre executado quando utilizamos a palavra-chave “new”, podemos inicializar informações do nosso novo objeto através de parâmetros para o construtor, entenda com o exemplo:

```
class Conta {  
    Cliente titular = new Cliente();  
    double saldo;  
  
    Conta(double saldo) {  
        this.saldo = saldo;  
    }  
}
```

Você pode, também, ter mais de um construtor na mesma classe:

```
class Conta {  
    Cliente titular = new Cliente();  
    double saldo;  
  
    Conta(double saldo) {  
        this.saldo = saldo;  
    }  
  
    Conta(String nomeTitular, String sobrenomeTitular, String  
    cpfTitular) {  
        this.titular.nome = nomeTitular;  
        this.titular.sobrenome = sobrenomeTitular;  
        this.titular.cpf = cpfTitular;  
    }  
}
```

Quando inicializar um novo objeto Conta agora, você precisará utilizar um dos construtores dessa classe. E dependendo dos parâmetros que utilizar, será escolhido o construtor correto automaticamente. Da mesma forma, não funcionará mais “new Conta()”, sem parâmetros, pois todos os construtores dessa classe esperam algum parâmetro, confira:

```
public static void main(String[] args) {  
    Conta minhaConta = new Conta();  
    Conta novaConta = new Conta("João", "Oliveira",  
    "1112223345");  
  
    System.out.println(novaConta.titular.nome);  
    System.out.println(minhaConta.titular.nome);  
}
```

Um construtor também pode chamar outro construtor:

```
class Conta {  
    Cliente titular = new Cliente();  
    double saldo;  
  
    Conta(String nomeTitular, String sobrenomeTitular, String  
    cpfTitular) {  
        this.titular.nome = nomeTitular;  
        this.titular.sobrenome = sobrenomeTitular;  
        this.titular.cpf = cpfTitular;  
    }  
  
    Conta(String nomeTitular, String sobrenomeTitular, String  
    cpfTitular, double saldo) {  
        this(nomeTitular, sobrenomeTitular, cpfTitular);  
        this.saldo = saldo;  
    }  
}
```

Construtores não são métodos. Um construtor não pode retornar nada. Também só é executado na construção do objeto, e nunca mais depois disso.

Então, qual a real necessidade de um construtor?

Você pode pensar na facilidade que isso nos proporciona: antes, criávamos os objetos e precisávamos ficar populando os atributos individualmente. Com um construtor, na inicialização do objeto já populamos os atributos com os valores desejados.

Mas a principal utilidade de um construtor é obrigar todos os objetos daquela classe a terem os valores mínimos necessários para seu uso. Quando definimos um construtor, qualquer outra classe que quiser utilizá-lo precisará, obrigatoriamente, fornecer esses valores iniciais.

## ENCAPSULAMENTO

Do modo que você construiu a classe Conta, ela permite que você altere o valor do atributo saldo diretamente. Isso pode gerar um problema, pois em algum lugar do código poderia ser definido o saldo de uma conta como um valor negativo, perceba:

```
public static void main(String[] args) {  
    Conta minhaConta;  
    minhaConta = new Conta("João", "Oliveira", "11122233345");  
    minhaConta.saldo = -500;  
}
```

Utilizando o método “sacar”, até você pode garantir que só será possível sacar um valor igual ou menor que o saldo, fazendo com que o saldo nunca atinja um valor negativo.

```
boolean sacar(double valorSaque) {  
    if (valorSaque <= this.saldo) {  
        this.saldo -= valorSaque;  
        return true;  
    }  
    return false;  
}
```

Agora, precisamos garantir que não seja possível alterar o valor do saldo diretamente, apenas através dos métodos “sacar” e “depositar”. Para esse fim, temos em Orientação a Objetos o que chamamos de “modificadores de acesso”, confira:

```
public class Conta {  
    private Cliente titular = new Cliente();  
    private double saldo;
```

Utilizando a palavra-chave “private” ao declarar um atributo, garantimos que em nenhum outro lugar do código será possível acessar diretamente o valor desse atributo. Assim, o valor de saldo será visível apenas para métodos internos da classe. Dessa forma, podemos ter o controle do limite de saldo dentro do método “sacar” e a certeza que o saldo nunca atingirá um valor abaixo desse limite. Concentramos todo o tratamento do atributo “saldo” nos métodos “sacar” e “depositar”, assim não temos tratamentos de saldo espalhados por todo o código da nossa aplicação.

O encapsulamento também aumenta a manutenibilidade do nosso sistema, ou seja, faz com que nosso código seja mais fácil de dar manutenção no futuro. Vamos supor que você, como desenvolvedor(a) do sistema desse banco, tenha recebido uma nova tarefa de melhoria: agora as contas podem ter saldo negativo, mas respeitando o limite de -200. Como o tratamento para não deixar que o saldo seja negativo está concentrado em um único lugar, o método “sacar”, fica fácil realizar essa alteração, entenda com o exemplo de código a seguir:

```
public boolean sacar(double valorSaque) {  
    if (this.saldo - valorSaque >= -200) {  
        this.saldo -= valorSaque;  
        return true;  
    }  
    return false;  
}
```

Perceba que alteramos apenas uma linha de código, a condição do “if” dentro do método “sacar”.

• • • • •

Agora imagine se fosse possível acessar diretamente o atributo “saldo”. Para realizar essa melhoria, precisaríamos buscar por todo o código do sistema para encontrar todos os locais onde o valor do saldo era alterado.

• • • • •

Em Orientação a Objetos é uma prática quase obrigatória declarar todos os seus atributos como “private”. Pois se entende que cada classe é responsável por seus próprios atributos. Por isso, a própria classe deve ter métodos específicos para alterar os valores de seus atributos, realizando as validações necessárias.

• • • • •

A palavra-chave “private” também pode ser utilizada na assinatura de métodos. Isso é útil quando temos um método que serve apenas à própria classe, e não a operações externas.

• • • • •

Você deve sempre expor o mínimo possível nossos atributos e métodos. Quando estiver criando atributos e métodos de uma classe, você deve pensar que a visibilidade padrão é sempre “private”.

Existe também a palavra-chave “public”. Quando definimos um atributo ou método como público, este vai ser acessível por qualquer outra classe do nosso sistema.



É comum que um programador escreva classes onde todos os atributos sejam privados e quase todos os métodos sejam públicos. Mas isso não é uma regra. Você deve sempre analisar o que faz mais sentido para cada situação específica.



No total são quatro modificadores de acesso: **private**, **public**, **default** e **protected**. O que você acabou de conferir foi uma introdução ao conceito de encapsulamento. Escondemos atributos e o corpo dos métodos para que, fora da classe, sejam visíveis apenas as assinaturas dos métodos. De fora, ninguém precisa saber como uma operação é realizada, apenas o que é realizado.

Encapsular é fundamental para que o nosso sistema tenha melhor manutenibilidade, para que seja mais suscetível a mudanças. O conjunto de métodos públicos de uma classe é chamado de “interface de classe”, já que é através desses métodos que outras partes do código da aplicação se comunicam com a classe.

É uma boa prática escrever código priorizando a interface da classe, ou seja, levando em consideração a forma como aqueles métodos vão ser utilizados, e não somente em como eles vão funcionar. A implementação do método, como ele funciona, pode mudar muitas vezes no decorrer do tempo, e por isso não é tão importante para quem o utiliza.

## GETTERS & SETTERS

Agora que o atributo “saldo” é privado, para alterá-lo nós precisamos utilizar os métodos “sacar” e “depositar”. Mas, e se quisermos **visualizar** o valor desse atributo privado? Não podemos mais acessá-lo diretamente nem para imprimir o valor na tela.

Sempre que precisamos trabalhar com um objeto, utilizamos **métodos**. Atributos são responsabilidade da própria classe, e a classe nos disponibiliza métodos para enviarmos mensagens e realizar alguma operação ou solicitar alguma informação.

Nesse sentido, vamos criar agora um método “getSaldo()” que retorna o valor do atributo saldo naquele momento, confira:

```
double getSaldo() {  
    return this.saldo;  
}
```

Nesse momento, qualquer outra classe que utilizar a classe Conta poderá visualizar o valor do atributo saldo através desse método. Para trabalhar com os atributos privados, uma prática bastante comum é criar dois métodos para cada atributo: um que retorna o valor e outro que muda o valor.

São os chamados “Getters” e “Setters”: um método “getNomeDoAtributo()”, outro “setNomeDoAtributo(algumParametro)”.

Para atributos booleanos, você pode usar “is” no lugar de “get”.

Exemplo: “isAtivo()”.

O Setter de booleanos pode ser “set” mesmo, pois faz sentido.

Você deve pensar bem antes de criar qualquer método ou atributo para nossas classes. O método que estamos criando realmente precisa existir? Conseguimos visualizar um contexto onde esse método será utilizado? A regra geral é criar um método apenas quando ele deve existir, quando enxergamos um uso para ele. Porém, Getters e Setters geralmente são criados sempre, para cada atributo.

No exemplo da conta bancária, existe necessidade de criarmos um método “setSaldo(valor)”? Possuímos um método “sacar(valor)” e um “depositar(valor)”, que fazem exatamente o que um Setter faria: alteram o valor do atributo saldo, para mais ou para menos. Esses métodos possuem contexto, ou seja, funcionam melhor para nossa classe do que um simples “setSaldo”.

Outro ponto importante, é que muitas vezes um método “get” não retorna simplesmente o valor daquele atributo. Por exemplo: uma prática comum dos bancos é mostrar o saldo do cliente como o saldo real + um certo limite de saldo negativo que o cliente pode atingir. Nesse caso, o nosso método “getSaldo()” poderia ser algo parecido com isso:

```
public double getSaldo() {  
    return this.saldo + this.limite;  
}
```

Utilizar Getters e Setters ajuda a concentrar certas lógicas em apenas um lugar, facilitando mudanças futuras. Também protege seus atributos. Isso também é encapsulamento, porque esconde a maneira como os objetos guardam seus dados.

## MÉTODOS E ATRIBUTOS ESTÁTICOS

Imagine que você recebeu a demanda por uma nova funcionalidade no seu sistema bancário. Agora a classe Conta também precisa guardar a informação de quantas contas já foram criadas. Como fazer isso?

Uma ideia interessante é ter uma variável “totalDeContas” que incrementamos sempre que um novo objeto Conta for instanciado, entenda:

```
public static void main(String[] args) {  
    int totalDeContas = 0;  
  
    Conta minhaConta = new Conta();  
    totalDeContas++;  
    Conta novaConta = new Conta();  
    totalDeContas++;  
}
```

Mas perceba, pelo exemplo, que não parece uma ideia tão boa assim. Precisaremos espalhar essa mesma linha de incremento à variável por todo o código, sempre que instanciarmos um novo objeto Conta. Quem garante que em algum lugar isso não será esquecido? Vamos tentar de outra forma, confira:

```
public class Conta {  
    private int totalDeContas;  
    private Cliente titular = new Cliente();  
    private double saldo;  
  
    Conta() {  
        this.totalDeContas++;  
    }
```

O que você acha? Existe algum problema com essa solução? O que acontece quando criarmos dois objetos Conta? Qual será o valor do atributo “totalDeContas” em cada uma delas?

Lembre-se que todos os objetos possuem os mesmos atributos, mas cada um com seu próprio valor. Sendo assim, cada objeto novo terá o atributo “totalDeContas” com o valor 1. Queremos que essa variável seja única para todos os objetos da classe. Queremos que todos os objetos compartilhem do mesmo valor para essa variável, para que quando um objeto alterar o valor dela, todos os outros possam visualizar o mesmo valor.

Para esse fim, temos em Java uma palavra-chave: “static”.

Ao declarar um atributo como “static”, ele passa a ser um atributo da classe, e não individual de cada objeto. A informação é guardada pela classe e compartilhada por todos os objetos dessa classe, entenda com o exemplo:

```
public class Conta {  
    private static int totalDeContas;
```

Para acessar um atributo estático, não usamos a palavra-chave “this”, pois não estamos mais nos referindo a um atributo do objeto. Nesse caso, utilizaremos o nome da classe:

```
Conta() {  
    Conta.totalDeContas++;  
}
```

Esse atributo estático é privado, então para acessá-lo de outra classe precisamos de um Getter, confira:

```
public int getTotalDeContas() {  
    return Conta.totalDeContas;  
}
```

Então, para saber quantas contas foram criadas:

```
public static void main(String[] args) {  
    Conta minhaConta = new Conta();  
    int total = minhaConta.getTotalDeContas();  
}
```

Agora reflita: **faz sentido precisar criar uma conta para saber quantas contas existem?**

Esse atributo é da classe, não depende de um objeto específico. Portanto, o método “get” para visualizar o valor desse atributo também deveria ser da classe, sem depender de um objeto. Para transformar um método “de objeto” em um método da classe, usamos a mesma palavra “static”, na assinatura do método:

```
public static int getTotalDeContas() {  
    return Conta.totalDeContas;  
}
```

Em seguida, acessamos esse método apenas pela classe, sem necessidade de instanciar um objeto, entenda:

```
public static void main(String[] args) {  
    int total = Conta.getTotalDeContas();  
}
```

Métodos estáticos só têm acesso a atributos estáticos. Isso faz todo sentido, já que dentro de um método estático não temos uma referência de objeto, um “this”.

Quanta coisa importante, não é mesmo? Seus estudos não acabam por aqui, siga construindo seus conhecimentos e aprenda ainda mais sobre Java.



### **João Victor Mendes de Oliveira**

Experiência em análise e desenvolvimento de software, web e desktop. Já trabalhou com SQL, Angular, Python, C# e Delphi. Atualmente desenvolve microsserviços com Java, Spring, MongoDB e Docker na Ambev Tech, e atua como mentor educacional no LAB365.

