



Quem se prepara, não para.

Programação Orienta a Objetos

3º período

Professora: Michelle Hanne

Sumário

- ✓ Formatação com Printf
- ✓ Leitura de String
- ✓ Modificadores
- ✓ Encapsulamento
- ✓ Construtor
- ✓ Setters e Getters

Formatação com Printf

- `System.out.printf(formato, dados de saída)`
- Formato exibido na saída, são separadas por vírgula.
- Os formatos sempre começam com "%", e como eu disse a saída são separadas por vírgulas.

`%s` = String

`%d` = Inteiro

`%f` = número com ponto flutuante. Na verdade o "f" representa a vírgula.

`\t` = tabulação

`\n` = salto de linha

Exemplo:

```
System.out.printf ("%d\t%d\t%.2f\t%s",5,5,254.336,"TESTE");
```

NumberFormat

- A classe `NumberFormat`, faz parte do pacote `java.text` e **permite formatar números** conforme a localização geográfica em que você se encontra, realizando a distinção entre o sinal de ponto, milhar e de decimal, também identifica a posição do sinal do número e identifica o prefixo que indica a moeda em caso de valores monetários.

```
import java.text.NumberFormat;
```

NumberFormat

Os principais métodos do NumberFormat são:

getCurrencyInstance()

- Usado para formatar moedas

getIntegerInstance()

- Usado para formatar números ignorando casas decimais

getPercentInstance()

- Usado para formatar frações pro exemplo 0,15 é formatado e mostrado como 15%

NumberFormat

```
1 package formatanumeros;
2 import java.text.NumberFormat;
3 public class FormataNumeros {
4
5     public static void main(String[] args) {
6         System.out.println(NumberFormat.getCurrencyInstance().format(12345678.90));
7         double n[]={523.34, 54344.23 ,95845.223 ,1084.895};
8
9         NumberFormat z = NumberFormat.getCurrencyInstance();
10
11         for (int a = 0; a < n.length; a++) {
12
13             if(a != 0)
14
15                 System.out.print(", ");
16
17                 System.out.print(z.format(n[a]));
18         }
19         System.out.println();
20     }
21 }
```

Entrada de Dados do tipo String

Diferença entre Scanner vs *BufferedReader*

- Reader simplesmente lê o arquivo e te dá o conteúdo, caractere por caractere;
 - Usado para manipular arquivos
- Já o Scanner é uma ferramenta mais complexa capaz de "interpretar" o conteúdo, separando por delimitadores e convertendo para outros tipos (numérico, por exemplo).

Entrada de Dados do tipo String

```
1 package entradastring;
2
3 import java.util.Scanner;
4 import java.io.*;
5 public class EntradaString {
6
7     /**
8      * @param args the command line arguments
9      */
10    public static void main(String[] args) {
11        //Leitura via classe Scanner
12        Scanner entrada = new Scanner(System.in);
13        System.out.println("Digite um nome ");
14        String nome = entrada.nextLine();
15        System.out.println("Nome digitado " + nome);
16
17        //Leitura via classe BufferedReader
18        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
19        String userName = null;
20        System.out.println("Digite seu Username");
21        try {
22            userName = in.readLine();
23        } catch (IOException ioe) {
24            System.out.println("IO erro tentando ler o nome");
25            System.exit(1);
26        }
27        System.out.println("Seu Username é " + userName);
28    }
29 }
```

Modificadores de Métodos

- **abstract**: método abstrato, sem corpo.
- **final**: método não pode ser redefinido.
- **public**: método pode ser acessado por outras classes.
- **private**: método só pode ser acessado pela própria classe.
- **protected**: método pode ser acessado por classes dentro do mesmo pacote ou pelas subclasses.
- **static**: método compartilhado por todos os objetos da classe, com acesso a apenas campos estáticos.

Atributos Públicos

Atributos públicos

- –Pode ser acessado ou modificado por código escrito por qualquer classe.

```
class Funcionario
{
    public string nome;
    public double salario;
    public static double valeRefeicaoDiario;

    public void AumentarSalario(double aumento)
    {
        ...
    }

    public static void ReajustarValeRefeicaoDiario(double taxa)
    {
        ...
    }
}
```

```
Funcionario f1 = new Funcionario();
Funcionario f2 = new Funcionario();
Funcionario f3 = new Funcionario();

f1.salario = 1000.00;
f2.salario = -2000.00;
f3.salario = 1800.00;
```

Atributos Públicos

- Para identificar algum erro relacionado a manipulação dos salários dos funcionários, é necessário verificar o código de todos os arquivos onde a classe **Funcionario** está definida.
- Quanto maior o número de arquivos, menos eficiente será a manutenção da aplicação.
- **Portanto, devemos evitar ao máximo a definição de atributos PUBLICOS (*public*) .**

Atributos Privados

Atributos privados

- Uso do modificador de acesso ***private***.
- Controle centralizado:
- Tornar os atributos privados ; definir métodos para implementar todas as lógicas que utilizam ou modificam o valor desse atributo.

Métodos Privados

- O papel de alguns métodos pode ser o de auxiliar outros métodos da mesma classe.
- Muitas vezes, não é correto chamar esses métodos auxiliares de fora da sua classe diretamente.
- Exemplo: método **DescontarTarifa()**
 - É um método auxiliar dos métodos **Depositar()** e **Sacar()**.
 - Ele não deve ser chamado diretamente, pois a tarifa só deve ser descontada quando ocorre um depósito ou um saque.

Métodos Privados

```
private double saldo;  
private double limite;  
private int numero;  
  
public static int contador;  
  
public void Depositar(double valor)  
{  
    this.saldo += valor;  
    this.DescontarTarifa();  
}  
  
public void Sacar(double valor)  
{  
    this.saldo -= valor;  
    this.DescontarTarifa();  
}  
  
/*  
 * O método auxiliar privado DescontarTarifa deve tirar $0.10  
 * para cada operação bancária Depositar, Sacar ou Transferir  
 */  
private void DescontarTarifa()  
{  
    this.saldo -= 0.1;  
}
```

- Para garantir que métodos auxiliares não sejam chamados por código escrito fora da classe na qual eles foram definidos, podemos torná-los privados, acrescentando o modificador ***private***.

```
private double saldo;  
private double limite;  
private int numero;  
  
public static int contador;  
  
public void Depositar(double valor)  
{  
    this.saldo += valor;  
    this.DescontarTarifa();  
}  
  
public void Sacar(double valor)  
{  
    this.saldo -= valor;  
    this.DescontarTarifa();  
}  
  
/*  
 * O método auxiliar privado DescontarTarifa deve tirar $0.10  
 * para cada operação bancária Depositar, Sacar ou Transferir  
 */  
private void DescontarTarifa()  
{  
    this.saldo -= 0.1;  
}
```

```
class TesteCartaoCredito  
{  
    static void Main(string[] args)  
    {  
        Conta c = new Conta();  
        c.DescontarTarifa();  
    }  
}
```

newton

prepara, não para.

Métodos Públicos

- Os métodos que devem ser chamados a partir de qualquer parte do sistema devem possuir o modificador de visibilidade **public**.
- Exemplo: métodos **de acesso** (*accessor*), **Depositar()**, **Sacar()** e outros.

```
public void Depositar(double valor)
{
    this.saldo += valor;
    this.DescontarTarifa();
}
```

```
public void Sacar(double valor)
{
    this.saldo -= valor;
    this.DescontarTarifa();
}
```

Encapsulamento

- Uma das ideias mais importantes da orientação a objetos é o encapsulamento.
- **Encapsular significa:**
 - esconder a implementação dos objetos.
 - O encapsulamento favorece principalmente dois aspectos de um sistema:
 - A manutenção e o desenvolvimento

Por quê Encapsular?

Exemplo Carro (*uma leitura para refletir*)

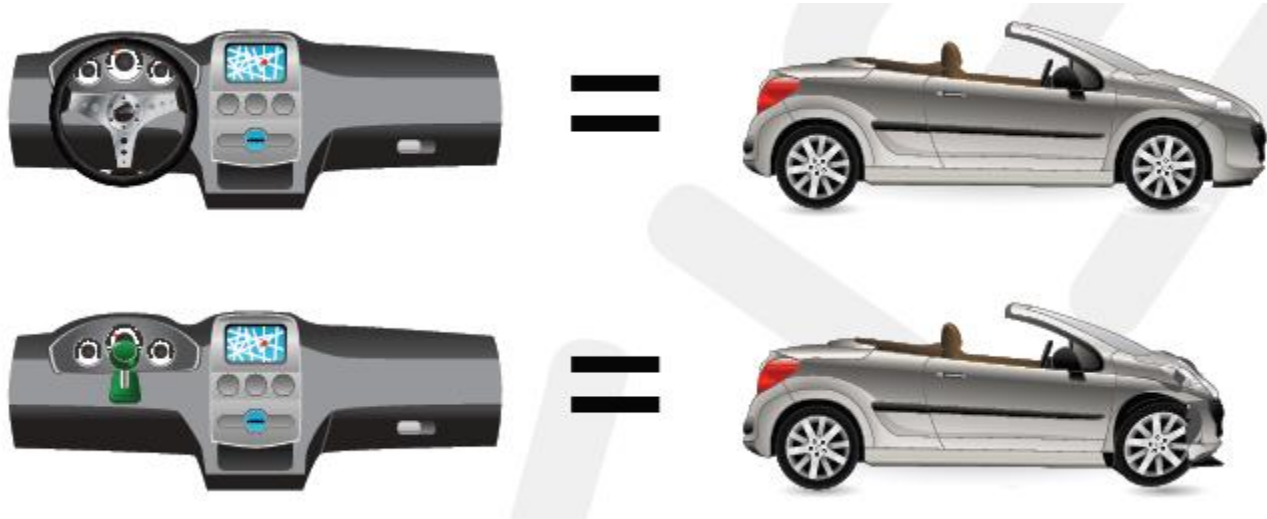
- A interface de uso de um carro é composta pelos dispositivos que permitem que o motorista conduza o veículo (volante, pedais, alavanca do câmbio, etc).
- A implementação do carro é composta pelos dispositivos internos (motor, caixa de câmbio, radiador, sistema de injeção eletrônica ou carburador, etc) e pelos processos realizados internamente por esses dispositivos.

Por quê Encapsular?

Exemplo Carro (*uma leitura para refletir*)

- Hoje em dia, as montadoras fabricam veículos com câmbio mecânico ou automático. O motorista acostumado a dirigir carros com câmbio mecânico pode ter dificuldade para dirigir carros com câmbio automático e vice-versa.
- –Quando a interface de uso do carro é alterada, a maneira de dirigir é afetada, fazendo com que as pessoas que sabem dirigir tenham que se adaptar.

Por quê Encapsular?



- *Substituição de um volante por um joystick*

Por quê Encapsular?

- No contexto da orientação a objetos, aplicando o conceito do encapsulamento, as implementações dos objetos ficam “escondidas”.
 - Dessa forma, podemos modificá-las sem afetar a maneira de utilizar esses objetos.
 - Por outro lado, se alterarmos a interface de uso que está exposta, afetaremos a maneira de usar os objetos.
- Considere, por exemplo, a mudança do nome de um método público. Todas as chamadas a esse método devem ser alteradas, o que pode causar diversos efeitos colaterais nas classes da aplicação.

Construtor

- O construtor é um método onde todas as inicializações do objeto são declaradas e possui o mesmo nome da classe.
- A seguir, temos as propriedades de um construtor:
 1. Possuem o mesmo nome da classe
 2. Construtor é um método, entretanto, somente as seguintes informações podem ser colocadas no cabeçalho do construtor:
 - Escopo ou identificador de acessibilidade (como public)
 - Nome do construtor
 - Argumentos, caso necessário

Construtor

3. Não retornam valor

4. São executados automaticamente na utilização do operador new durante a instanciação da classe

Para declarar um construtor, escrevemos:

```
[modificador] <nomeClasse> (<argumento>*) {  
  
<instrução>*  
  
}
```


Construtor Padrão (default)

- Toda classe tem o seu construtor padrão. O construtor padrão é um construtor público e sem argumentos.
- Se não for definido um construtor para a classe, então, implicitamente, é assumido um construtor padrão.
- Por exemplo, na nossa classe ***StudentRecord***, o construtor padrão é definido do seguinte modo:

```
public StudentRecord() {  
  
}
```

```
public StudentRecord() {  
  
    // qualquer código de inicialização aqui  
  
}  
  
public StudentRecord(String temp){  
  
    this.name = temp;  
  
}  
  
public StudentRecord(String name, String address) {  
  
    this.name = name;  
  
    this.address = address;  
  
}  
  
public StudentRecord(double mGrade, double eGrade, double sGrade) {  
  
    mathGrade = mGrade;  
  
    englishGrade = eGrade;  
  
    scienceGrade = sGrade;  
  
}
```

Overload de Construtores

Usando Construtores

```
public static void main(String[] args) {  
  
    // criar três objetos para o registro do estudante  
  
    StudentRecord annaRecord = new  
  
    StudentRecord("Anna");  
  
    StudentRecord beahRecord = new StudentRecord("Beah", "Philippines");  
  
    StudentRecord crisRecord = new StudentRecord(80,90,100);  
  
    // algum código aqui  
  
}
```

Usando Construtores

- O **atributo estático studentCount** têm por objetivo contar o número de objetos que são instanciados com a classe **StudentRecord**.
- Então, o que desejamos é incrementar o valor de **studentCount** toda vez que um objeto da **classe StudentRecord** é **instanciado**. Um bom local para modificar e incrementar o valor de **studentCount** é nos construtores, pois são sempre chamados toda vez que um objeto é instanciado.

Usando Construtores

```
public StudentRecord() {  
  
    studentCount++; // adicionar um estudante  
  
}  
  
public StudentRecord(String name) {  
  
    studentCount++; // adicionar um estudante  
  
    this.name = name;  
  
}
```

Usando Construtores

```
public StudentRecord(String name, String address) {  
  
    studentCount++; // adicionar um estudante  
  
    this.name = name;  
  
    this.address = address;  
  
}  
  
public StudentRecord(double mGrade, double eGrade, double sGrade) {  
  
    studentCount++; // adicionar um estudante  
  
    mathGrade = mGrade;  
  
    englishGrade = eGrade;  
  
    scienceGrade = sGrade;  
  
}
```

Utilizando o This

- Chamadas a construtores podem ser cruzadas, **o que significa ser possível chamar um construtor de dentro de outro construtor. Usamos a chamada *this()* para isso.**
- Como boa prática de programação, é ideal nunca construir métodos que repitam as instruções. Buscamos a utilização de *overloading* com o objetivo de evitarmos essa repetição.

Utilizando o This

```
public StudentRecord() {  
  
    studentCount++; // adicionar um estudante  
  
}  
  
public StudentRecord(String name) {  
  
    this();  
  
    this.name = name;  
  
}
```

```
public StudentRecord(String name, String address) {  
  
    this(name);  
  
    this.address = address;  
  
}  
  
public StudentRecord(double mGrade, double eGrade,  
double sGrade) {  
  
    this();  
  
    mathGrade = mGrade;  
  
    englishGrade = eGrade;  
  
    scienceGrade = sGrade;  
  
}
```


Variáveis de Instância, métodos set e get **Newton**

Quem se prepara, não para.

Uma classe normalmente consiste em um ou mais métodos que manipulam os atributos que pertencem a um objeto particular da classe.

Os atributos são representados como variáveis em uma declaração de classes.

São chamadas de campos e estão dentro da declaração da classe.

Métodos Set e Get

- Para que outros objetos possam modificar os nossos dados, disponibilizamos métodos que possam gravar ou modificar os valores dos atributos de objeto ou de classe. Chamamos a estes métodos modificadores. Este método é **escrito como** `set<NomeDoAtributoDeObjeto>`
- O método que retorna um valor é escrito como `get<NomeDoAtributo>`.

Métodos Set e Get

Esses métodos servem para pegarmos informações de variáveis da classe que são definidas como 'private', porém esses métodos são definidos como 'public'.

Daí surge uma pergunta natural: por que criar métodos para acessar variáveis, se podemos acessar elas diretamente?

Simples: questão de segurança

Métodos Set e Get

As variáveis 'private' só podem ser acessadas de dentro da Classe. É como se elas fossem invisíveis foram do escopo da classe/objeto.

Usamos get para obter informações. Esse tipo de método sempre retorna um valor.

Usamos set para definir valores. Esse tipo de método geralmente não retorna valores.

Métodos Set e Get

- Como visto anteriormente, o encapsulamento "protege" os atributos ou métodos dentro de uma classe, portanto devemos prover meios para acessar tais membros quando eles são particulares, ou seja, quando possuem o modificador **private**.
- Em programação orientada a objetos, esses métodos são chamados de métodos **assessores ou getters e setters**, pois eles provêm acesso aos atributos da classe, e geralmente, se iniciam com **get** ou **set**, daí a origem de seu nome.

Método Set

Nomeamos um método como ***set*** toda vez que este método for modificar algum campo ou atributo de uma classe.

Como o valor de um atributo da classe será modificado, não é necessário que este método retorne nenhum valor, por isso, os métodos **setters são void**. Porém, obrigatoriamente, eles tem que receber um argumento que será o novo valor do campo.

Método Get

Nomeamos um método como ***get*** toda vez que este método for verificar algum campo ou atributo de uma classe.

Como este método irá verificar um valor, ele sempre terá um retorno como *String, int, float, etc.* Mas não terá nenhum argumento.

Método Is

Nomeamos um método acessor com *is* toda vez que este método for verificar algum campo ou atributo de uma classe que tenha retorno do tipo *boolean*.

Exemplo 1

```
public class getSet{  
  
    public static void main(String[] args){  
  
        String nome = "Neil Peart";  
  
        int ID=2112;  
  
        double salario = 1000000;  
  
        Funcionario chefe = new Funcionario();  
  
        chefe.setNome(nome);  
  
        chefe.setID(ID);  
  
        chefe.setSalario(salario);  
  
        chefe.exibir();  
  
    }  
  
}
```

```
public class Funcionario {
```

```
    private String nome;
```

```
    private int ID;
```

```
    private double salario;
```

```
    public void exibir(){
```

```
        System.out.printf("O funcionário %s, de número %d recebe %.2f por mês",  
        getNome(),getID(),getSalario());
```

```
    }
```

```
    public void setNome( String nome ){
```

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

```
    }
```

```
    public void setID( int ID ){
```

```
        this.ID = ID;
```

```
    }
```

Exemplo 1

Exemplo 1

```
public void setSalario( double salario ){  
    this.salario = salario;  
  
}  
  
public String getNome(){  
    return this.nome;  
  
}  
  
public int getID(){  
    return this.ID;  
  
}  
  
public double getSalario(){  
    return this.salario;  
  
}  
  
}
```

Como invocar métodos de dentro do construtor

Outra maneira de inicializarmos as variáveis de um Objeto, é usando os métodos 'set', direto do construtor:

```
public class setGet{  
  
    public static void main(String[] args){  
  
        String nome = "Neil Peart";  
  
        int ID=2112;  
  
        double salario = 1000000;  
  
        Funcionario chefe = new Funcionario(nome, ID, salario);  
  
        chefe.exibir();  
  
    }  
}
```

```
public class Funcionario {
```

```
    private String nome;
```

```
    private int ID;
```

```
    private double salario;
```

```
    public Funcionario( String nome, int ID, double salario){
```

```
        setNome(nome) ;
```

```
        setID(ID) ;
```

```
        setSalario(salario) ;
```

```
    }
```

```
    public void exibir(){
```

```
        System.out.printf("O funcionário %s, de número %d recebe %.2f por mês",  
        getName(),getID(),getSalario());
```

```
    }
```

Como invocar métodos de dentro do construtor

```
public void setName( String nome ){  
  
    this.nome = nome;  
  
}  
  
public void setID( int ID ){  
  
    this.ID = ID;  
  
}  
  
public void setSalario( double salario ){  
  
    this.salario = salario;  
  
}  
  
public String getNome(){  
  
    return nome;  
  
}
```

Como invocar métodos de dentro do construtor

```
public int getID(){  
    return ID;  
}
```

```
public double getSalario(){  
    return salario;  
}  
}
```

Como invocar métodos de dentro do construtor

Exemplo 2 – Classe Carro

```
1 package questao2_eng;
2 import java.util.Scanner;
3 public class Questao2_eng {
4
5     public static void main(String[] args) {
6
7         Scanner entrada = new Scanner(System.in);
8
9         System.out.println("Digite a placa do carro ");
10        String placa = entrada.nextLine();
11
12        System.out.println("Digite o modelo do carro ");
13        String modelo = entrada.nextLine();
14
15        System.out.println("Digite a cor do carro ");
16        String cor = entrada.nextLine();
17
18        System.out.println("Digite ao numero de portas");
19        int portas = entrada.nextInt();
20
21        System.out.println("Digite o ano do carro");
22        int ano = entrada.nextInt();
23
24        System.out.println("Digite a quantidade de combustível");
25        double combustivel = entrada.nextDouble();
26
27
28        Carro obj = new Carro(portas, cor, placa, modelo, ano, combustivel);
29        System.out.println("A placa do carro possui " + obj.verificaPlaca(placa) + " vogal(is).");
30    }
31 }
```



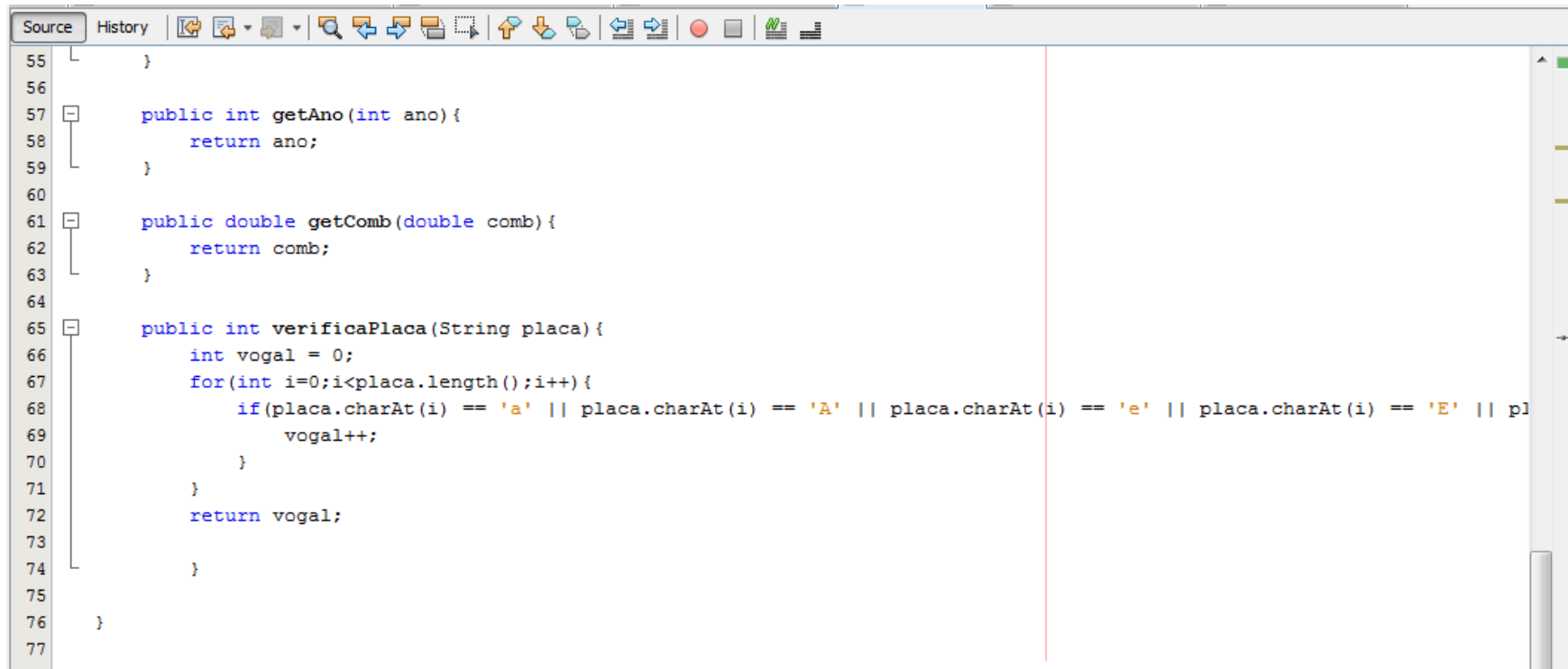
```
1 package questao2_eng;
2 public class Carro {
3     private int qtdePortas;
4     private String cor;
5     private String placa;
6     private String modelo;
7     private int ano;
8     private double qtdeCombustivel;
9     Carro (int porta, String cor, String placa, String modelo, int ano, double comb){
10         setPorta(porta);
11         setCor(cor);
12         setPlaca(placa);
13         setModelo(modelo);
14         setAno(ano);
15         setComb(comb);
16     }
17     private void setPorta(int porta){
18         this.qtdePortas = porta;
19     }
20
21     private void setCor(String cor){
22         this.cor = cor;
23     }
24
25     private void setPlaca(String placa){
26         this.placa = placa;
27     }
28
29     private void setModelo(String modelo){
30         this.modelo = modelo;
31     }
```

Exemplo 2 – Classe Carro

Exemplo 2 – Classe Carro

```
32
33 private void setAno(int ano){
34     this.ano = ano;
35 }
36
37 private void setComb(double comb){
38     this.qtdeCombustivel = comb;
39 }
40
41 public int getPorta(int porta){
42     return porta;
43 }
44
45 public String getCor(String cor){
46     return cor;
47 }
48
49 public String getPlaca(String placa){
50     return placa;
51 }
52
53 public String getModelo(String modelo){
54     return modelo;
55 }
56
57 public int getAno(int ano){
58     return ano;
59 }
60
```

Exemplo 2 – Classe Carro



```
55     }
56
57     public int getAno(int ano) {
58         return ano;
59     }
60
61     public double getComb(double comb) {
62         return comb;
63     }
64
65     public int verificaPlaca(String placa) {
66         int vogal = 0;
67         for(int i=0;i<placa.length();i++){
68             if(placa.charAt(i) == 'a' || placa.charAt(i) == 'A' || placa.charAt(i) == 'e' || placa.charAt(i) == 'E' || p
69                 vogal++;
70         }
71     }
72     return vogal;
73 }
74
75
76 }
77
```

Exercício Set e Get 2

Escreva uma classe em Java chamada Estoque. Ela deverá possuir:

- a) os atributos **nome (string)**, **qtdAtual (int)** e **qtdMinima (int)** do tipo **private**.
- b) um construtor sem parâmetros contendo os parâmetros **nome**, **qtdAtual** e **qtdMínima**. Os métodos **set** deverão ser **Private**. Os métodos **get** deverão ser **Public**

Exercício Set e Get 2

- Escreva uma classe em Java chamada Estoque. Ela deverá possuir:
 - a) os atributos **nome (string)**, **qtdAtual (int)** e **qtdMinima (int)** **do tipo private**.
 - b) um construtor sem parâmetros contendo os parâmetros **nome**, **qtdAtual** e **qtdMínima**. **Os métodos set deverão ser Private**. **Os métodos get deverão ser Public**.

Exercício Set e Get 2

c) os métodos com as seguintes assinaturas:

void darBaixa(int qtde)

String mostra()

boolean precisaRepor()

O método **darBaixa(int qtde)** recebe uma quantidade e atualiza o estoque. Porém, o estoque não poderá ficar negativo.

O método **mostra()** retorna uma **String** contendo o nome do produto, sua quantidade mínima, sua quantidade atual.

O método **precisaRepor()** retorna true caso a quantidade atual esteja menor ou igual à quantidade mínima e false, caso contrário.

Referências

SILVA, Fabricio Machado da. **Paradigmas de programação**. SAGAH, 2019. ISBN digital: 9788533500426

Barnes, David e Kölling, M. **Programação Orientada a Objetos com Java**. São Paulo: Pearson Prentice Hall, 2004.

Deitel, H. M.; Deitel, P. J. **Java - Como Programar**. 6. ed. Prentice-Hall, 2005. Capítulo 4 e 5.

PRESSMAN, Roger S. **Engenharia de Software**. Rio de Janeiro: MacGraw Hill, 2002.