

Fundamentos da Programação Orientada a Objetos

Aluno: Sander Gustavo Piva
Professor: Emerson Assis de Carvalho

1- Programação Orientada a Objeto (POO)

A programação iniciou-se em baixo nível (linguagem de máquina), depois evoluiu para alto nível (linear), com comandos sem desvios ou rotinas internas, seguindo para programação estruturada onde pequenos pedaços do programa eram organizados de modo a permitir desvios (ênfase: sequência, decisão e iteração), até a programação modular (baseada em cápsulas). Algumas dificuldades ainda eram percebidas, pois na linguagem não orientada a objeto existia um volume significativo de dados globais que poderia ser utilizado pelos procedimentos do sistema, todavia isto despendia tempo e recursos para ‘filtrar’ quais seriam os dados mais coerentes. Uma vez que fosse possível abstrair previamente os dados úteis ou dados objetos seria possível contornar essa dificuldade, inclusive esses objetos poderiam se relacionar entre si otimizando ainda mais os processos. Logo, nascia a primeira linguagem orientada a objeto com o matemático e biólogo Alan Kay, o Smalltalk. A POO, portanto, tem por finalidade aproximar o mundo digital do mundo real. Existem inúmeras linguagens de programação orientada a objeto, sendo que é possível destacar: Java, C++, C#, Python, e PHP, porém os principais conceitos de POO e exemplos, inclusive com demonstração de código (Anexo), será baseado em Java. A seguir, os principais conceitos serão abordados.

2- Conceitos de POO

- Classes

Refere-se a um ‘molde’ ou ‘forma’ para definir os atributos (características) e métodos (comportamentos) comuns que serão compartilhados por um objeto (coisas materiais ou abstratas). Exemplo: Um carro tem marca, modelo, cor e motor que podem variar, ou seja, posso ter um objeto carro 1 da marca ‘Ford’, modelo ‘Fiesta’, cor ‘azul’, motor ‘1.3’ ou um objeto carro 2 da marca ‘Volks’, modelo ‘UP’, cor ‘preto’, motor ‘1.0’, porém todos são carros, pertencem a classe Carro.

Um modelo de sintaxe: ‘public class Carro {...}’. Lembrando que na programação, as classes precisam ser públicas para que o acesso ocorra.

- Objetos

Refere-se a uma coisa material ou abstrata percebida pelos sentidos e descrita por meio de suas características (atributos), comportamentos (métodos) e estado (características e/ou comportamentos definidos, em um certo momento). Exemplo: Um objeto Volkswagen UP, verde, 4 portas, motor 1.0 é oriundo da classe Carro. Esse objeto tem marca ‘Volkswagen’, modelo ‘UP’, cor ‘Verde’ e motor ‘1.0’ e tem comportamentos: acelera, freia, sobe ladeiras com dificuldade, etc.

Um modelo de sintaxe: `'... Carro c1 = new Carro ();'`, onde `'c1'` é o objeto da classe `'Carro'`, `'instanciado'` pela palavra reservada `'new'`. A instanciação ou materialização do objeto cria um espaço na memória e faz referência ao objeto `'c1'`.

- Modificadores e tipagem?

Os modificadores definem se um atributo ou método podem ser acessados fora da classe respectiva (`'public'`), apenas pela classe atual e suas subclasses, por exemplo, no caso de Herança (`'protected'`) ou ainda apenas pela classe respectiva (`'private'`). Os tipos de atributos/variáveis podem ser, dentre outros, `'String'`, no caso de caractere, `'int'`, inteiro, `'double'`, ponto flutuante e `'boolean'`, tipo lógico. Exemplo de sintaxe:

Sintaxe para método: `'private void calculaPreco () {...}'`, onde `'private'` é o modificador, ou seja, só a classe respectiva desse método tem acesso. Sintaxe para atributo: `'protected int valor;'`, onde `'protected'` diz que esse atributo é acessível pela sua classe respectiva e subclasses.

- Métodos

Refere-se, basicamente, ao comportamento do objeto, ou seja, o que o objeto faz? Exemplo: Com uma caneta BIC azul ponta grossa cuja classe é Caneta posso escrever, pintar, desenhar, etc. Possuem modificadores (`public`, `private`, `protected`, `()` nenhum), tipos com retorno: `String`, `int`, `double`, `boolean`, dentre outros e sem retorno (`void`), nomes, e podem ou não ter parâmetros. Os principais métodos podem ser personalizados (concretos), especiais, abstratos e estáticos.

Os métodos `'personalizados'` são definidos pelo desenvolvedor, respeitando boas práticas de programação e lógica. Uma sintaxe: `'public int valor (int x) { ... }'`, onde: `'public'` (modificador) `int` (tipo) `valor` (nome) e `(int x)` é o parâmetro `'x'` do tipo `'int'`, logo o retorno precisa ser um inteiro.

Os métodos especiais podem ser:

- Métodos Modificadores (Setters): recebem um ou mais parâmetros que podem passar por uma validação prévia e depois atribuídos às variáveis respectivas da classe. Exemplo de sintaxe: a partir da classe Main: `'... c1.setDadosCaneta(marca, cor);'` agora na classe Caneta: `'public void setDadosCaneta(String marca, String cor){this.marca = marca; this.cor = cor;}'`. Esse método evita a tentativa de definir um atributo tipo `private` da classe Caneta diretamente pela classe Main, como: `c1.marca = "BIC"`, o que iria gerar um erro pelo compilador e o código não iria rodar.
- Métodos Acessores (Getters): retornam o valor do atributo, geralmente privado, de uma classe para a chamada. Exemplo de sintaxe: `'... Caneta c1 = new Caneta();'` -> [objeto `'c1'`, da classe Caneta, instanciado pelo construtor], `'c1.getMarcaCaneta()'`; -> [chamada ao método `'public String getMarcaCaneta() {...}'` que irá retornar o valor da marca da caneta: Ex: `'BIC'`]. Os métodos (Getters) contribuem com a segurança do sistema.

- Método Construtor: Ao instanciar um objeto o método construtor cria um espaço na memória e faz referência ao objeto. Nesse espaço é permitido armazenar, operar e manipular dados. O construtor pode ser com ou sem parâmetros. Exemplo de sintaxe: ‘instanciando o objeto – construtor sem parâmetros: `Caneta c1 = new Caneta();`’ ou ‘instanciando o objeto – construtor com parâmetros: `Caneta c1 = new Caneta (marca, cor);`’, logo, na classe Caneta o método construtor sem parâmetros fica: `public Caneta(){...}`, já com parâmetros: `public Caneta(String tipo, String cor){...}`.

Todo método assessor, modificador e construtor é público.

Os métodos abstratos e estáticos: Os métodos abstratos não possuem corpo ou implementação. São característicos de classes abstratas e interfaces. Uma sintaxe: `public abstract void calculaValor ();`. Já os estáticos são aqueles que pertencem a classe e não dependem de nenhuma variável de instância. Exemplo: `public static final void emiteRelatorio () {...}`.

- Atributos

Refere-se às características de um objeto, ou seja, como esse objeto é e o que tem? Exemplo: Máquina de lavar Brastemp Cinza capacidade de 11 kg com recursos multimídia. Atributos: marca ‘Brastemp’, cor ‘Cinza’, capacidade ‘11kg’, recursos multimídia ‘sim’. Sintaxe em programação: `private String marca = “Brastemp”;` `private String cor = ”Cinza”`. Existem ainda os atributos estáticos que pertencem à classe e não ao objeto. Sintaxe: `private final double taxa = 1.3;`.

- Pilares da POO

A base da POO é formada pela: Abstração, Encapsulamento, Herança, e Polimorfismo. Este é o grande diferencial da programação orientada a objeto. Vantagens: Responsável por implementar códigos consistentes, oportunos, manipuláveis, extensíveis, reutilizáveis e naturais, pois visa aproximar o mundo digital do mundo real.

- Abstração

Empregada para definir atributos úteis para um objeto. Exemplo: Os atributos peso, altura, raça, são atributos importantes para um objeto modelo de uma classe Modelos, todavia, pode não ser para um objeto encanador de uma classe Funcionário.

- Encapsulamento

Oculta partes independentes da implementação, permitindo controlar partes invisíveis ao mundo exterior. Exemplo: Quando o motorista de um carro pisa no pedal do freio, basta saber que freia o veículo, não precisa saber como funciona o sistema de freio em si. Tudo isso está protegido

(não visível) ao usuário ou motorista, logo sua interação é possível através de uma ‘interface: pedal de freio’, sendo que a partir dele ocorre a troca de informações (mensagens) necessárias ao processo de frenagem, permitindo o funcionamento correto do sistema com segurança.

Na programação, ao definir os modificadores de certos atributos e métodos como ‘private’ o propósito é impedir o acesso direto do usuário a eles, protegendo a parte interna do código contra inconsistências e quebra de integridade. Em código, quando define-se que o atributo marca da classe Caneta é do tipo private, não é possível acessá-lo pela classe Main e atribuir diretamente outro valor, ou seja, precisa de um método ‘public String getMarca()’ para ter o retorno desse valor. Um método private da classe Caneta também só é acessível por outros métodos da mesma classe, logo não é possível acessá-la pela classe Main. O encapsulamento não é obrigatório, porém recomendado. Vantagens: mudanças invisíveis ao usuário, código pode ser reutilizado, sistemas mais seguros.

- Herança

O princípio de Herança em POO permite a criação de novas classes (classes filhas ou subclasses) a partir de outras criadas previamente (classe mãe ou superclasse). Uma superclasse pode ter várias classes filhas, todavia uma subclasse só pode herdar ‘diretamente’ de uma única superclasse. Exemplo: Sistema bibliotecário de uma escola: o usuário pode ser aluno, mas também pode ser professor, logo, nota-se que ‘Usuario’ é a classe mãe ou superclasse enquanto ‘Aluno’ e ‘Professor’ são classes filhas ou subclasses. A superclasse pode ser ou não abstrata.

No caso de abstrata, ou seja, não é possível instanciar um objeto a partir dela, apenas pelas suas subclasses. A palavra reservada ‘extends’ é utilizada pelas subclasses para herdar atributos e métodos da superclasse. Uma subclasse pode adicionar novos métodos e atributos, sobrescrever métodos da superclasse, todavia não pode remover atributos e métodos da superclasse. Sintaxe de exemplo: ‘public abstract class Mamifero {...}’, sendo a superclasse e ‘public class Cao extends Mamifero {...}’ e ‘public class Baleia extends Mamifero {...}’ as suas subclasses.

- Polimorfismo

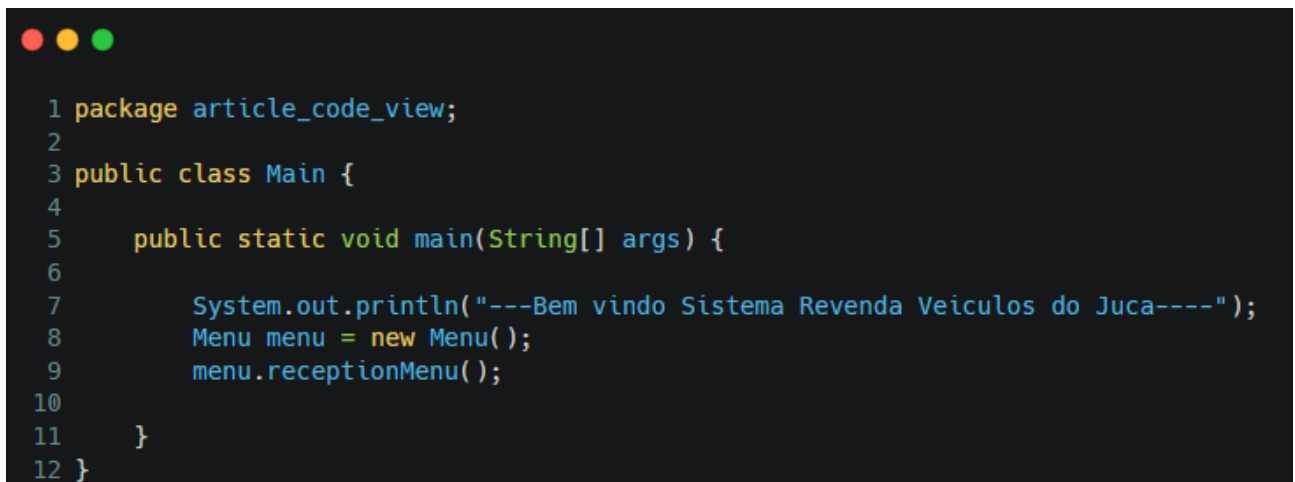
O Polimorfismo significa várias formas, ou seja, na POO é possível utilizar um elemento de formas diferentes. Pode ser estático (sobrecarga) ou dinâmico (sobreposição). Na sobrecarga, mais de um método de mesmo nome, porém com parâmetros diferentes, são criados na mesma classe. Exemplo: ‘public Caneta (String nome)’, ‘public Caneta (double ponta)’. Na sobreposição, uso mais comum, ocorre em casos de Herança, isto é, a subclasse sobrepõe o método original da superclasse. Exemplo no anexo (final do artigo).

3- Conclusão:

A Programação orientada a objetos revolucionou a maneira de codificar, permitindo uma implementação de software cada vez mais consistente, segura e de fácil manutenção. Em termos de interatividade, o cliente tem mais liberdade de escolher as funções desejadas, ao contrário de um modelo estruturado. O pilar da POO ainda proporciona criar sistemas reutilizáveis e extensíveis, por exemplo, conectando mais de uma subclasse à superclasse, mantendo sua integridade, funcionando como um organismo vivo.

Anexo

Nesse anexo, cada imagem (img1-img8) do código terá uma explicação demonstrando na prática sua funcionalidade e os principais conceitos de POO relacionados.

A screenshot of a code editor with a dark background and light-colored text. The code is in Java and defines a package and a main class. The package is 'article_code_view' and the class is 'Main'. The 'main' method prints a welcome message and creates a 'Menu' object, then calls its 'receptionMenu' method. The code is numbered from 1 to 12.

```
1 package article_code_view;
2
3 public class Main {
4
5     public static void main(String[] args) {
6
7         System.out.println("---Bem vindo Sistema Revenda Veiculos do Juca---");
8         Menu menu = new Menu();
9         menu.receptionMenu();
10
11     }
12 }
```

img1

O programa inicia com a classe 'Main', linha 3. Dentro do método estático ('public static void main(String[] args) {...}', linha 5, um 'System.out.println(...)' realiza um 'print', linha 7, anunciando o tipo de sistema: 'Bem vindo Sistema Revenda Veiculos do Juca'. Depois o objeto 'menu' da classe 'Menu', linha 8, é instanciado a partir do construtor definido pela palavra reservada 'new': '... new Menu();'. Logo, o objeto 'menu' poderá usufruir de todos os atributos e métodos da classe Menu. Na sequência, o objeto 'menu' chama o método concreto 'receptionMenu()' dessa maneira: 'menu.receptionMenu();', linha 9. Nesse começo de sistema, aplica-se o conceito de classe (Ex.: classe 'Main', linha 3), objeto (Ex.: objeto 'menu' da classe 'Menu', linha 8) e métodos estático ('public static void main(String [] args)', linha 5) e chamada ao método concreto 'receptionMenu()', linha 9.

A seguir, as imagens (img2-img4) serão demonstradas, lembrando que todas se referem a classe Menu ('public class Menu {...}').

```

1 package article_code_view;
2
3 import article_code.model.Car;
4 import article_code.model.VehicleStock;
5 import article_code.model.Vehicle;
6 import article_code.model.Motorcycle;
7 import java.util.*;
8
9 public class Menu {
10
11     private String type;
12     private String brand;
13     private String model;
14     private String color;
15     private String engineType;
16     private double value;
17     private int year;
18
19     public void receptionMenu() {
20         int inputAnswer = 1;
21         int operationAnswer = 0;
22         Scanner scan = new Scanner(System.in);
23         VehicleStock pushInStock = new VehicleStock();
24

```

img2

Já na classe Menu ('public class Menu {...}'), após os importes do pacote 'article_code_view', das classes do pacote 'model' e do 'java.util.*;', nota-se a presença dos atributos, constituídos de seus modificadores (public, private, protected, ...) e tipos (String, double, int, ...) os quais são: 'private String type, private String brand, private String model, private String color, private String engineType, private double value, private int year' seguido do método 'public void receptionMenu() {...}', nesse caso, trata-se de um método concreto ou personalizado, definido pelo desenvolvedor.

Dentro do respectivo método, existem dois atributos: 'int inputAnswer' linha 20, e 'operationAnswer' linha 21 para receber o valor da 'resposta de entrada' e 'operação' do sistema, nessa ordem. Já a classe 'Scanner', instanciada a partir da palavra reservada 'new' assim constituída: 'Scanner scan = new Scanner (System.in)' irá permitir a leitura e captura de dados dentro de um laço de repetição 'while' que está dentro de um tratamento de exceções 'try {...} catch() {...}', o qual será demonstrado a seguir. Ainda nota-se a instância 'pushInStock' da classe 'VehicleStock' linha 23 a qual permitirá que os objetos 'c' da classe 'Car' e 'mt' da classe 'Motorcycle' sejam adicionados no método 'public void setStock(...){...}' da classe 'VehicleStock'. Nela serão registrados todos os objetos 'c' e 'mt' das classes 'Car' e 'Motorcycle'.


```

25         try {
26
27             while (inputAnswer != 0) {
28                 System.out.println("Cadastrar veículo [1] - Consulta Estoque [2] Sair?
[0]*");
29                 inputAnswer = scan.nextInt();
30                 switch (inputAnswer) {
31
32                     case 1:
33                         System.out.println("Cadastra veiculos");
34                         do {
35                             System.out.println("Cadatrar carro [1] moto [2] sair?
[0]*");
36                             operationAnswer = scan.nextInt();
37                             } while (operationAnswer != 0 && operationAnswer != 1 &&
operationAnswer != 2);
38                             if (operationAnswer == 1) {
39                                 this.type = "Carro";
40                                 registerVehicle();
41                                 Car c = new Car(this.type, this.brand, this.model);
42                                 c.setYear(this.year);
43                                 c.setColor(this.color);
44                                 c.setValue(this.value);
45                                 c.setEngineType(this.engineType);
46                                 pushInStock.setStock(c);
47                             } else if (operationAnswer == 2) {
48                                 this.type = "Moto";
49                                 registerVehicle();
50                                 Motorcycle mt = new Motorcycle(this.type, this.brand,
this.model);
51                                 mt.setYear(this.year);
52                                 mt.setColor(this.color);
53                                 mt.setValue(this.value);
54                                 mt.setEngineType(this.engineType);
55                                 pushInStock.setStock(mt);
56
57                             } else {
58                                 System.out.println("Saindo do cadastramento...");
59                             }
60                             break;
61                         case 2:
62                             System.out.println("---Consulta Estoque---\n");
63
64                             for (Vehicle aux : pushInStock.getStock()) {
65
66                                 System.out.println(aux.vehicleData());
67                                 System.out.println("Ajuste aplicado para revenda: " +
((aux.additionForSale() * 100) - 100) + "%\n");
68                                 System.out.println(aux.vehicleValue());
69                                 System.out.println("---");
70                             }
71                             break;
72                     }
73                 }
74                 System.out.println("Fim");
75             } catch (Exception excpl) {
76                 System.out.println("Erro! Sistema encerrado");
77             }
78         }

```

img3

O 'try{...}catch() {...}' empregado, tem por finalidade proteger o laço 'while' de erros de inserção de dados de tipagem diferente do solicitado, por exemplo, tratando essa exceção. No laço de repetição 'while', é iniciado de fato o sistema. A primeira pergunta indaga ao usuário se pretende 'Cadastrar veículo [1]' ou 'Consultar Estoque [2]'. Quando é respondida essa questão ('inputAnswer', linha 30), é acessada a estrutura condicional 'switch case' onde, se for um cadastro de veículos é o 'case 1', se for consulta de estoque, 'case 2', ou sair, ao digitar '0'.

Dentro do ‘case 1’, outra pergunta é feita: ‘Cadastrar carro [1] Cadastrar moto [2]?’. Se a resposta da operação deseja ‘operationAnswer’ for 1, será o Carro, logo o objeto ‘c’ de carro irá definir o atributo tipo ‘this.type’ como ‘Carro’, irá chamar o método ‘public void registerVehicle () {...}’ ou registra veículo para ler e atribuir/definir os valores de cada atributo como: marca ‘this.brand’, modelo ‘this.model’, cor ‘this.color’, etc. Uma vez definido os atributos, uma parte deles será parâmetros do construtor ao instanciar o objeto ‘c’: ‘this.type’, ‘this.brand’ e ‘this.model’ e os demais serão definidos nos métodos especiais ‘setters’ como: c.setYear(this.year), c.setValue(this.value), enfim. O objeto ‘pushInStock’ pega os objetos ‘c’ e os adiciona pelo método ‘public void setStock(...){...}’ da classe ‘VehicleStock’.

Se a resposta da operação deseja ‘operationAnswer’ for 2, será a Motocicleta (Moto), logo o objeto ‘mt’ de moto irá definir o atributo tipo ‘this.type’ como ‘Moto’, irá chamar o método ‘public void registerVehicle () {...}’ ou registra veículo para ler e atribuir/definir os valores de cada atributo como: marca ‘this.brand’, modelo ‘this.model’, cor ‘this.color’, etc. Uma vez definido os atributos, uma parte deles será parâmetros do construtor ao instanciar o objeto ‘mt’: ‘this.type’, ‘this.brand’ e ‘this.model’ e os demais serão definidos nos métodos especiais ‘setters’ como: c.setYear(this.year), c.setValue(this.value), enfim. O objeto ‘pushInStock’ pega os objetos ‘mt’ e os adiciona pelo método ‘public void setStock(...){...}’ da classe ‘VehicleStock’.

Senão for nenhuma das opções anteriores para ‘operationAnswer’, o sistema emite a mensagem ‘Saindo do cadastramento’. Vale lembrar que qualquer opção diferente de 0, 1 e 2 para a operação fará a pergunta ‘Cadastrar Carro [1] Cadastrar Moto [2]’ ser repetida, pois está dentro de um laço de repetição ‘do while’. Todavia, se for ‘case 2’, a resposta de entrada ‘inputAnswer’, será mostrado todo o estoque cadastrado com os dados pertinentes, oriundos do método ‘public List<Vehicle> getStock() {...}’ da classe ‘VehicleStock’ a partir de um laço de repetição ‘for’ que irá passar uma cópia da ‘lista’ de objetos veículos para uma variável local (aux) e a partir dela será possível ‘printar’ na tela todos os dados.

Importante: Quando instancio o objeto ‘c’ e ‘mt’ das classes ‘Car’ e ‘Motorcycle’ pelo método construtor, estou passando parâmetros para as respectivas classes que, por ‘Herança’ e o uso da instrução ‘super(...)’, permite a classe Abstrata ‘Vehicle’ ser instanciada, logo, os objetos das subclasses ‘Car’ e ‘Motorcycle’ poderão usufruir dos atributos e/ou métodos da classe abstrata. Isso será visto um pouco mais a frente. Ainda na classe Menu, podemos notar que seus atributos estão com o modificador ‘private’, deixando apenas os métodos acessíveis/públicos, logo percebe-se uma aplicação básica de encapsulamento, ou seja, da classe ‘Main’ não é possível acessar esses atributos/variáveis, evitando erros de lógica ou inconsistências no código.

Senão for ‘case 1’ ou ‘case 2’, o programa é encerrado ‘Fim’.

```

79
80     public void registerVehicle() {
81         Scanner scan = new Scanner(System.in);
82
83         try {
84             System.out.println("Marca: ");
85             this.brand = scan.nextLine();
86             System.out.println("Modelo: ");
87             this.model = scan.nextLine();
88             System.out.println("Cor: ");
89             this.color = scan.nextLine();
90             System.out.println("Tipo motor: ");
91             this.engineType = scan.nextLine();
92             do {
93                 System.out.println("Valor de compra R$: ");
94                 this.value = scan.nextDouble();
95             } while (this.value <= 0);
96             do {
97                 System.out.println("Ano: [1999->2023]: ");
98                 this.year = scan.nextInt();
99             } while (this.year < 1999 || this.year > 2023);
100             System.out.println("----");
101
102         } catch (Exception excp2) {
103
104             System.out.println("Erro no cadastramento");
105         }
106     }
107 }

```

img4

Este é o método concreto ‘public void registerVehile(){...}’ no qual o usuário irá preencher e portanto definir os valores dos atributos que serão cadastrados pelos objetos ‘c’ e ‘mt’ das classes ‘Car’ e ‘Motorcycle’.

As imagens (img2-img4) se referem à mesma classe (‘public class Menu {...}’). Como demonstrado, nota-se a aplicação dos conceitos de classe (Ex.: ‘public class Menu{...}’, linha 9), objeto (Ex.: ‘c’, da classe ‘public class Car{...}’, linha 41), atributos (Ex.: ‘private int year’, linha 17), métodos (Ex.: ‘public void registerVehicle(){...}’, linha 80) e encapsulamento (Ex.: O atributo ‘year’, linha 17, com o modificador ‘private’ ‘encapsula’ o atributo de modo que fora da sua classe, não é possível acessar seu conteúdo a não ser, nesse caso, por um método público: ‘public int getYear (){...}’ das classes ‘Car’ e ‘Motorcycle’).

```

1 package article_code.model;
2
3 public abstract class Vehicle {
4
5     private String brand;
6     private String model;
7     private String type;
8     private double value;
9
10    public Vehicle(String type, String brand, String model) {
11        this.type = type;
12        this.brand = brand;
13        this.model = model;
14    }
15
16    public double getValue() {
17        return this.value;
18    }
19
20    public void setValue(double value) {
21        this.value = value;
22    }
23
24    public abstract double additionForSale();
25
26    public String vehicleData() {
27        return "Tipo: " + this.type + "\nMarca: " + this.brand
28            + " Modelo: " + this.model;
29    }
30
31    public abstract String vehicleValue();
32
33 }
34
35

```

Img5

Aqui nota-se a classe abstrata ‘Vehicle’ recebendo os parâmetros passados pelos objetos ‘c’ e/ou ‘mt’ das classes ‘Car’ e ‘Motorcycle’ pela instrução ‘super(...)’, identificada na próxima imagem (img6), assim como o valor do veículo, o qual pode ser de um carro ou moto, está sendo definido pelo método especial ‘public void setValue(double value){...}’ e retornado pelo método ‘public double getValue() {...}’ a partir dos mesmos objetos. Isso é possível devido ao princípio de Herança, um dos pilares da POO, ou seja, as classes ‘Car’ e ‘Motorcycle’ são filhas ou subclasses da classe mãe ou superclasse que é a classe abstrata ‘Vehicle’. Toda classe abstrata precisa ter ao menos um método abstrato. Nesse código existem dois: ‘public abstract double additionForSale();’ e o ‘public abstract String vehicleValue();’, além do método concreto ‘public String vehicleData() {...}’, os quais serão sobrescritos pelas subclasses.

Aplicação dos conceitos de classe (Ex.: ‘public abstract class Vehicle {...}’, na linha 3), atributos (Ex.: ‘private String brand’, linha 5), métodos (Ex.: ‘public double getValue() {...}’, linha 17) e encapsulamento (Ex.: uso do modificador ‘private’ no atributo ‘value’, linha 8, evitando acesso direto fora de sua classe). Classe abstrata não tem instância.

```

1 package article_code.model;
2
3 public class Car extends Vehicle {
4
5     private String color;
6     private int year;
7     private String engineType;
8     private final double ADDITIONforCarSale;
9
10    public Car(String type, String brand, String model) {
11        super(type, brand, model);
12        this.ADDITIONforCarSale = 1.3;
13    }
14
15    public int getYear() {
16        return this.year;
17    }
18
19    public void setYear(int year) {
20        this.year = year;
21    }
22
23    public String getEngineType() {
24        return engineType;
25    }
26
27    public void setEngineType(String engineType) {
28        this.engineType = engineType;
29    }
30
31    public String getColor() {
32        return this.color;
33    }
34
35    public void setColor(String color) {
36        this.color = color;
37    }
38
39    @Override
40    public double additionForSale() {
41        return ADDITIONforCarSale;
42    }
43
44    @Override
45    public String vehicleData() {
46        return super.vehicleData() + "\nCor: " + this.color + "\nAno: " + this.year + "\nMotor: " +
47        this.engineType;
48    }
49
50    @Override
51    public String vehicleValue() {
52        return "Valor venda do Carro: " + "R$" + (super.getValue() * ADDITIONforCarSale);
53    }
54 }

```

img6

Na subclasse ‘Car’, reconhece-se a Herança da classe ‘Vehicle’ pela palavra reservada ‘extends’ permitindo o objeto ‘c’ acessar atributos e/ou métodos da superclasse ‘Vehicle’. Apesar do objeto ‘c’ herdar atributos e/ou métodos superclasse ‘Vehicle’, o objeto ‘c’ usufrui de atributos e métodos próprios da classe como os métodos ‘getters e setters’ dos atributos year ‘this.year’, color ‘this.color’, engineType ‘this.engineType’. Percebe-se ainda que os métodos abstratos, assim como o método concreto ‘public String vehicleData() {...}’ estão sendo sobrescritos, ou seja, estão sendo utilizados de maneira específica dentro da classe ‘Car’, do mesmo modo que dentro da classe ‘Motorcycle’ tais métodos também serão sobrescritos, porém de ‘forma diferente’. Portanto, nota-se aqui o emprego do conceito de polimorfismo tipo sobrescrita.

No caso da classe 'Car', o método 'public double additionForSale(){...}' retorna o atributo estático 'ADDITIONforCarSale' com valor '1.3'. O método 'public String vehicleData(){...}' retorna os dados/atributos gerais do carro, oriundos da superclasse, adicionados com os dados/atributos específicos da classe 'Car'. O método 'public String vehicleValue(){...}' retorna o valor da aquisição do carro 'this.value' multiplicado pelo fator de ajuste/acréscimo 'ADDITIONforCarSale', pois deseja-se um valor final, nesse caso, com 30% de aumento.

Aplicação dos conceitos de classe (Ex.: 'public class Car {...}', na linha 3, lembrando que a classe 'Car' também pode ser considerada uma subclasse da classe abstrata 'Vehicle' no momento que usar a palavra reservada 'extends'), objeto (quando o objeto 'c' acessa a classe 'Car' pelo construtor 'public Car(...){...}', na linha 10), atributos (Ex.: 'private String engineType', na linha 7), métodos (Ex.: 'public void setColor(...){...}', na linha 35), encapsulamento (Ex.: uso do modificador 'private' no atributo 'engineType', na linha 7, evitando acesso direto fora de sua classe), herança (Ex.: 'public class Car extends Vehicle {...}', na linha 3) e polimorfismo (Ex.: quando o método concreto 'public String vehicleData(...){...}', na linha 45, é sobrescrito na classe 'Car' ou ainda, na mesma classe, quando o método abstrato 'public double vehicleValue(){...}', na linha 50, também é sobrescrito).

```

1 package article_code.model;
2
3 public class Motorcycle extends Vehicle {
4
5     private String color;
6     private int year;
7     private String engineType;
8     private final double ADDITIONforMotoSale;
9
10    public Motorcycle(String type, String brand, String model) {
11        super(type, brand, model);
12        this.ADDITIONforMotoSale = 1.2;
13    }
14
15    public int getYear() {
16        return this.year;
17    }
18
19    public void setYear(int year) {
20        this.year = year;
21    }
22
23    public String getEngineType() {
24        return this.engineType;
25    }
26
27    public void setEngineType(String engineType) {
28        this.engineType = engineType;
29    }
30
31    public String getColor() {
32        return this.color;
33    }
34
35    public void setColor(String color) {
36        this.color = color;
37    }
38
39    @Override
40    public double additionForSale() {
41        return this.ADDITIONforMotoSale;
42    }
43
44    @Override
45    public String vehicleData() {
46        return super.vehicleData() + "\nCor: " + this.color + "\nAno: " + this.year + "\nMotor: " +
47        this.engineType;
48    }
49
50    @Override
51    public String vehicleValue() {
52        return "Valor venda da moto: " + "R$" + (super.getValue() * ADDITIONforMotoSale);
53    }
54 }

```

img7

Na subclasse ‘Motorcycle’, reconhece-se a Herança da classe ‘Vehicle’ pela palavra reservada ‘extends’ permitindo o objeto ‘mt’ acessar atributos e/ou métodos da superclasse ‘Vehicle’. Apesar do objeto ‘mt’ herdar atributos e/ou métodos superclasse ‘Vehicle’, o objeto ‘mt’ usufrui de atributos e métodos próprios da classe como os métodos ‘getters e setters’ dos atributos year ‘this.year’, color ‘this.color’, engineType ‘this.engineType’. Percebe-se ainda que os métodos abstratos, assim como o método concreto ‘public String vehicleData() {...}’ estão sendo sobrescritos, ou seja, estão sendo utilizados de maneira específica dentro da classe ‘Motorcycle’, do mesmo modo que dentro da classe ‘Car’ tais métodos também serão sobrescritos, porém de ‘forma diferente’. Portanto, nota-se aqui o emprego do conceito de polimorfismo tipo sobrescrita.

No caso da classe `Motorcycle`, o método `public double additionForSale() {...}` retorna o atributo estático `ADDITIONforMotoSale` com valor `1.2`. O método `public String vehicleData() {...}` retorna os dados/atributos gerais da moto, oriundos da superclasse, adicionados com os dados/atributos específicos da classe `Motorcycle`. O método `public String vehicleValue() {...}` retorna o valor da aquisição da motocicleta/moto `this.value` multiplicado pelo fator de ajuste/acréscimo `ADDITIONforMotoSale`, pois deseja-se, nesse caso, um valor final com 20% de aumento.

Aplicação dos conceitos de classe (Ex.: `public class Motorcycle {...}`, linha 3, lembrando que a classe `Motorcycle` também é uma subclasse da classe abstrata `Vehicle` no momento que usar a palavra reservada `extends`), objeto (quando o objeto `mt` acessa a classe `Motorcycle` pelo construtor `public Motorcycle (...){...}`, na linha 10), atributos (Ex.: `private String engineType`, linha 7), métodos (Ex.: `public void setColor(...){...}`, na linha 35), encapsulamento (Ex.: uso do modificador `private` no atributo `engineType`, na linha 7, evitando acesso direto fora de sua classe), herança (Ex.: `public class Motorcycle extends Vehicle {...}`, na linha 3) e polimorfismo (Ex.: quando o método concreto `public String vehicleData(...){...}`, na linha 45, é sobrescrito na classe `Motorcycle` ou ainda, na mesma classe, quando o método abstrato `public double vehicleValue() {...}`, na linha 50, também é sobrescrito).


```

1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class VehicleStock {
5
6     private List<Vehicle> vehicleList = new ArrayList<Vehicle>();
7     private List<Vehicle> copyVehicleList = new ArrayList<Vehicle>();
8
9     public void setStock(Vehicle vehicle) {
10
11         this.vehicleList.add(vehicle);
12     }
13
14     public List<Vehicle> getStock() {
15         for (Vehicle aux : vehicleList) {
16
17             copyVehicleList.add(aux);
18         }
19         return this.copyVehicleList;
20     }
21 }
22

```

img8

Na classe ‘VehicleStock’ nota-se a presença de um atributo do tipo ‘ArrayList’ da classe ‘Vehicle’, pois quando o método ‘setStock(...) {...}’ é acionado este recebe um objeto ‘c’ ou ‘mt’ que é do tipo veículo ou pertence a superclasse ‘Vehicle’. Uma vez definida a lista ‘vehicleList’, o método concreto ‘public List<Vehicle> get Stock() {...}’ armazena os objetos numa variável local (aux) do tipo ‘Vehicle’ e o ‘aux’ é adicionado na lista ‘copyVehicleList’ que é retornada para o método que a chamou na classe ‘Menu’.

Aplicação dos conceitos de classe (Ex.: ‘public class VehicleStock {...}’, na linha 4), objeto (quando o objeto ‘pushInStock’ acessa a classe ‘VehicleStock’ pelo método concreto ‘public void setStock (Vehicle vehicle){...}, na linha 9, passando como parâmetro um objeto ‘vehicle’ do tipo/classe ‘Vehicle’), atributos (Ex.: ‘private List<Vehicle> vehicleList’, linha 6), métodos (Ex.: ‘public List<Vehicle> getStock(...) {...}, na linha 14)’, encapsulamento (Ex.: uso do modificador ‘private’ no atributo ‘List<Vehicle> vehicleList’, na linha 6, evitando acesso direto fora de sua classe).