

# Classes Abstratas em Java

## Exercício

**Vinicius Takeo Friedrich Kuwaki**

Universidade do Estado de Santa Catarina

# Seções

Exemplo

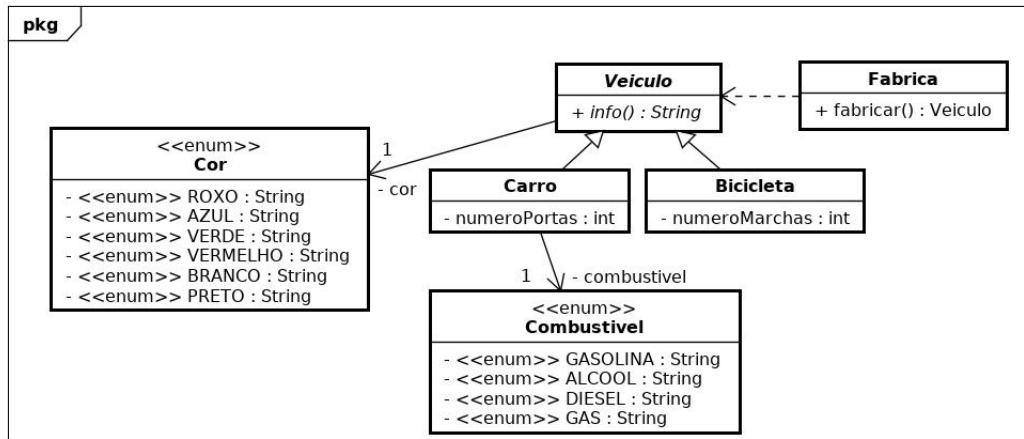
Resolução

Exercício

## Exemplo

- Uma determinada Fábrica de Veículos produz Carros e Bicicletas;
- Todos os veículos possuem uma cor;
- Mas somente os Carros possuem portas e um tipo de combustível;
- Enquanto que as Bicicletas possuem um número de marchas;
- Observe um diagrama UML representando essa Fábrica:

# Exemplo



## Exemplo

- Essa fábrica apenas fabrica veículos nas cores: roxo, azul, verde, vermelho, branco e preto;
- Os carros possuem apenas as opções de gasolina, álcool, diesel e gás como tipos de combustíveis;
- A classe Veículo é abstrata e possui um método abstrato **info()** a ser implementado pelas classes Carro e Bicicleta;
- Esse método **info()** retorna as informações referentes ao veículo;
- A classe Fábrica possui uma dependência com a classe Veículo apresentada no método **fabricar()**.
- O método **fabricar()** cria veículos escolhendo aleatoriamente suas cores, tipos de combustível, número de portas e número de marchas;
- Teremos um classe Main que ficará gerando novos veículos até que o usuário se canse de ver valores aleatórios piscando em sua tela;

# Seções

Exemplo

Resolução

Exercício

## Enum Cor

- **Enum** são classes próprias para enumerar valores finitos que as instâncias dessa classe podem assumir;
- **Enum** possuem um método **getValues()** que retorna os valores enumerados pela classe;
- Implementaremos a classe Cor como uma **enum**;
- Essa classe possui um atributo do tipo String chamado cor;
- Além de possuir um construtor privado que recebe a String cor a ser setada no atributo da classe;
- Também possui um método get para esse atributo cor;
- Note que a palavra **enum** é reservada em Java;
- Observe o código-fonte:

# Enum Cor

```
public enum Cor {  
    ROXO("roxo"), AZUL("azul"), VERDE("verde"), VERMELHO("vermelho"), BRANCO("branco"),  
    PRETO("preto");  
  
    private String cor;  
  
    private Cor(String cor) {  
        this.cor = cor;  
    }  
  
    public String getCor() {  
        return cor;  
    }  
}
```



## Classe Veículo

- Agora podemos definir a nossa classe Veículo;
- Ela é abstrata e não pode ser instanciada!
- Terá um atributo do tipo Cor com seus respectivos getters e setters;
- Além da assinatura do método **info()**;

# Classe Veículo

```
public abstract class Veiculo {  
    private Cor cor;  
  
    public Cor getCor() {  
        return this.cor;  
    }  
  
    public void setCor(Cor cor) {  
        this.cor = cor;  
    }  
  
    public abstract String info();  
}
```

## Enum Combustível

- Seguindo a mesma lógica do Enum Cor, iremos implementar o Enum Combustível;
- Nossa fábrica irá fabricar apenas carros movidos a gasolina, alcool, diesel e gás;
- Nossa classe possuirá um atributo do tipo String chamado tipo;
- Um método getTipo() publico e o construtor da classe como sendo private;
- Observe o código-fonte:

# Enum Combustivel

```
public enum Combustivel {  
    GASOLINA("gasolina"), ALCOOL("alcool"), DIESEL("diesel"), GAS("gas");  
  
    private String tipo;  
  
    private Combustivel(String tipo) {  
        this.tipo = tipo;  
    }  
  
    public String getTipo() {  
        return tipo;  
    }  
}
```

## Classe Carro

- Já a classe Carro, possui um atributo que indica o número de portas e outro que indica o seu tipo de combustível;
- Esses atributos são do tipo inteiro e do tipo Combustível, respectivamente;
- Note a palavra reservada **extends** indicando a herança da classe Carro;

```
public class Carro extends Veiculo {  
    private int numeroPortas;  
    private Combustivel combustivel;
```

- Ambos possuem métodos getters e setters para manter o encapsulamento;
- E a classe também implementa o método **info()**, definido na superclasse Veículo;

# Classe Carro

```
public int getNumeroPortas() {  
    return this.numeroPortas;  
}  
  
public void setNumeroPortas(int numeroPortas) {  
    this.numeroPortas = numeroPortas;  
}  
  
public Combustivel getCombustivel() {  
    return this.combustivel;  
}  
  
public void setCombustivel(Combustivel combustivel) {  
    this.combustivel = combustivel;  
}  
  
public String info() {  
    return "\nCarro\n" + "Cor: " + this.getCor() + "\n" + "Numero de portas: " +  
        numeroPortas + "\n"  
        + "Tipo de combustivel: " + combustivel + "\n";  
}  
}
```

## Classe Bicicleta

- Já a classe Bicicleta também estende a classe Veículo e implementa seu próprio método **info()**;
- Além de possuir um atributo que representa o número de marchas com seu getter e setter;

```
public class Bicicleta extends Veiculo {  
  
    private int numeroMarchas;  
  
    public int getNumeroMarchas() {  
        return this.numeroMarchas;  
    }  
  
    public void setNumeroMarchas(int numeroMarchas) {  
        this.numeroMarchas = numeroMarchas;  
    }  
  
    public String info() {  
        return "\nBicicleta\n" + "Cor: " + this.getCor() + "\n" + "Numero de marchas: "  
        + this.numeroMarchas + "\n";  
    }  
}
```

## Classe Fabrica

- Agora iremos implementar a classe Fabrica;
- Esta não possui nenhum atributo, como especificado no diagrama de classes pelo relacionamento de dependência;
- Ela possui apenas um método **fabricar()** que retorna ou uma instância da Classe Carro ou uma instância da classe Bicicleta;
- Todos os valores serão gerados aleatoriamente;
- Observe o código-fonte:



# Classe Fabrica

```
public Veiculo fabricar() {  
    Random r = new Random();  
    if (r.nextInt(2) == 1) {  
        Carro c = new Carro();  
        c.setCor(Cor.values()[r.nextInt(Cor.values().length)]);  
        c.setNumeroPortas(2 + 2 * r.nextInt(2));  
        c.setCombustivel(Combustivel.values()[r.nextInt(Combustivel.values().length)]);  
    }  
    return c;  
} else {  
    Bicicleta b = new Bicicleta();  
    b.setCor(Cor.values()[r.nextInt(Cor.values().length)]);  
    b.setNumeroMarchas(r.nextInt(28));  
    return b;  
}  
}
```

## Classe Main

- Agora iremos implementar um método **main()** para ficar gerando os Carros ou Bicicletas aleatoriamente;
- O método ficará gerando Carros e Bicicletas até que o usuário digite -1 para finalizar;

```
public static void main(String[] args) {  
  
    int fim = -1;  
    Fabrica fabrica = new Fabrica();  
    Scanner s = new Scanner(System.in);  
  
    do {  
  
        System.out.println(fabrica.fabricar().info());  
        System.out.println("Digite 0 para interromper a produção");  
        System.out.println("Digite qualquer número para continuar");  
        fim = s.nextInt();  
  
    } while (fim != 0);  
}
```

## Código-fonte

- O código-fonte está disponível nesse [link](#).

# Seções

Exemplo

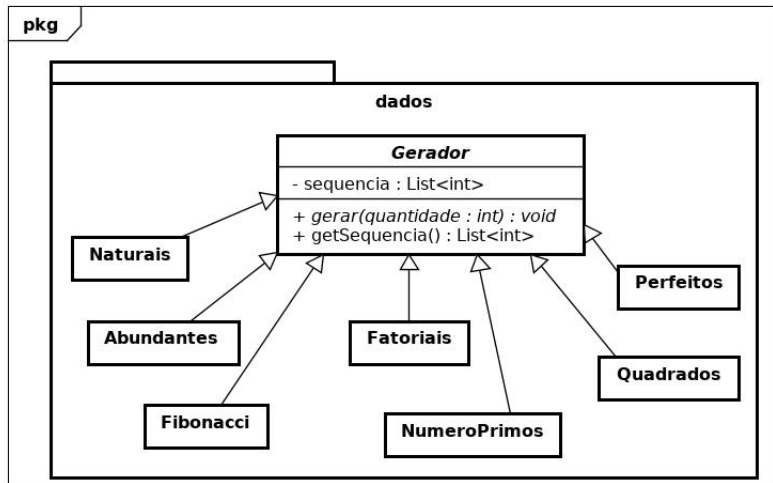
Resolução

Exercício

## Exercício

- Implemente as classes descritas no Diagrama UML de Classes a seguir;
- Não é necessário implementar uma interface via console com o usuário e nem uma classe para administrar o uso das classes especificadas no diagrama;

## Exercício - Diagrama



## Exercício - Descrição

- A classe Gerador é abstrata e requer a implementação do método **gerar()** pela classe que a extender;
- A classe que implementar o método gerar() deve gerar a sequencia que dá nome a classe;
- Por exemplo, a classe Naturais irá gerar os n números naturais até a quantidade passada como parâmetro pelo método;
- A seguir serão descritos como são geradas as sequências;


## Exercícios - Sequências

- **Naturais:** todos os números inteiros positivos até  $n$ ;
- **Abundantes:** números menores que a soma de seus divisores;
  - 12 ( $1+2+3+4+6=16$ );
  - 18 ( $1+2+3+4+6+9=25$ );
  - 20 ( $1+2+4+5+10=22$ );
  - 24 ( $1+2+3+4+6+8+12=36$ );
  - ...
- **Fibonacci:**  $0, 1, 1, 2, 3, 5, 8, \dots$  onde  $F_n = F_{n-1} + F_{n-2}$
- **Fatoriais:** produto de todos os números anteriores a  $n$  maiores que 1;
  - 1 ( $1! = 1$ );
  - 2 ( $2! = 2 \cdot 1$ );
  - 6 ( $3! = 3 \cdot 2 \cdot 1$ );
  - 24 ( $4! = 4 \cdot 3 \cdot 2 \cdot 1$ );
  - ...



## Exercícios - Sequências

- **Números Primos:** todo número que é divisível somente por 1 e por ele mesmo;
  - 2;
  - 3;
  - 5;
  - ...
- **Quadrados:** todo número  $n$  cuja raiz quadrada é inteira;
  - 2;
  - 4;
  - 16;
  - 25;
  - ...
- **Perfeitos:** todo número  $n$  cuja a soma de seus divisores é igual a ele mesmo;
  - 6 ( $1+2+3=6$ );
  - 28 ( $1+2+4+7+14=28$ );
  - ...

 KUWAKI, V. T. F. Modelo de slides udesc lattex. In: . [S.l.]: Disponível em: <<https://github.com/takeofriedrich/slidesUdescLattex>>. Acesso em: 24 jan. 2020.

Duvidas:  
Vinicius Takeo Friedrich Kuwaki  
[vinicius.kuwaki@edu.udesc.br](mailto:vinicius.kuwaki@edu.udesc.br)  
[github.com/takeofriedrich](https://github.com/takeofriedrich)



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA