Essa é uma versão dos Guias de Estudo desenvolvida para auxiliar o processo de impressão e para facilitar a leitura dos guias por programas que fornecem a leitura automatizada para suporte a todos os alunos que necessitem. Dessa forma, não apresentaremos ilustrações nesse arquivo.* *#ParaTodosLerem****.*

Módulo #3

Guias de estudo

Introdução

Agora que você já tem uma familiaridade com o mundo dos códigos de programação, imagine a complexidade de um sistema de uma grande empresa.

Esse sistema consegue gerenciar todos os setores e possui diversas funcionalidades, tais como: *realizar cadastro de usuários, cadastrar os funcionários, permitir que os usuários façam login no sistema, cadastrar produtos, cadastrar compras, cadastrar vendas, fazer todo gerenciamento de logística* e muito mais!

Além disso, esses sistemas têm diferentes implementações para cada setor da empresa. As necessidades do time de RH são uma; para o time de compras são outras, e assim é com cada setor. Imagine a quantidade de códigos que um sistema desse tem implementado! Como organizar tudo isso?

Foi aí que surgiu a ideia da **programação orientada a objetos** (**POO**).
Vamos entender mais sobre esse assunto?

Programação Orientada a Objetos (POO)

A programação orientada a objetos (POO) *é uma das maneiras que se tem de programar que chamamos também de paradigmas de programação. Como o próprio nome já diz em programação orientada a objetos temos a presença de objetos que são todas as entidades do mundo real que codificamos e que possuem características e executam ações.*

Suponha um jogo de corrida de automóveis, umas das classes que teremos que modelar no sistema são os carros. As classes são um molde de qualquer coisa do mundo real que abstraímos para o mundo virtual por meio da programação, elas devem ter todas as características e ações que os objetos de seu tipo possam realizar.

Ou seja, na classe carro, abstraímos todas as características e ações que qualquer objeto carro pode ter e/ou fazer, por exemplo: ano, modelo, marca, cor, placa e chassi; acelerar, frear, ligar e desligar. Cada carro existente é um objeto que pertence à classe carro. Vamos então implementar a classe Carro:

Ano, modelo, marca, cor, placa e chassi são as características/atributos da classe. Ligar(), Desligar(), Acelerar() e Frear() serão as ações, ou seja, os métodos a serem implementados.

Primeiro criamos a classe como nome Carro:

```
class Carro:
```

Agora vamos criar o construtor. Em python usamos o `__init__` que é executado sempre que um objeto da classe é criado. Ele será o responsável por definir os atributos de todo objeto da classe carro.

```
    def __init__(self, ano, modelo, marca, cor, placa, chassi):
        self.ano = ano
        self.modelo = modelo
        self.marca = marca
        self.cor = cor
        self.placa = placa
        self.chassi = chassi
```

Definiremos no `__init__` também a velocidade e a quantidade de passageiros atuais do carro.

```
        self.velocidade=0
        self.passageiros=0
```

O método `adicionarPassageiro` recebe como parâmetro a quantidade de passageiros a serem adicionados no carro. Então, verifica-se se esta quantidade não excede 4. Se não exceder, o atributo passageiros recebe a quantidade enviada como parâmetro, caso contrário, a mensagem `Favor não transportar mais de 4 passageiros` é exibida na tela.

```
    def adicionarPassageiro(self, quantidade):
      if quantidade<=4:
        self.passageiros+=quantidade
      else:
        print('Favor não transportar mais de 4 passageiros.')
```

O método `ligar` mostra na tela a mensagem `O carro ligou!!` e define a velocidade do carro como 0, uma vez que o carro foi ligado e ainda não está em movimento.

```
    def ligar (self):
        print('O carro ligou!!')
        self.velocidade=0
```

O método `desligar` mostra na tela a mensagem `O carro desligou!!` e define a velocidade do carro como 0, uma vez que o carro foi desligado e não está em movimento.

```
    def desligar (self):
        print('O carro desligou!!')
        self.velocidade=0
```

O método `acelerar` incrementa o atributo velocidade aumentando de 1 em 1. Sempre que quiser acelerar um objeto carro é preciso chamar este método. Sempre que um carro acelera aparece na tela a velocidade atual.

```
    def acelerar(self):
        self.velocidade+=1
        print('Velocidade: ',self.velocidade)
```

O método `frear` é bem parecido com o método `acelerar`, porém ele decrementa o atributo velocidade diminuindo de 1 em 1. Sempre que quiser frear um objeto carro é preciso chamar este método. Sempre que um carro freia aparece na tela a velocidade atual.

```
    def frear(self):
        self.velocidade-=1
        print('Velocidade: ',self.velocidade)
```

Okay, a classe carro foi criada. Vamos então criar dois objetos do tipo Carro:

```
carro1 = Carro(2017, 'Palio', 'Fiat', 'Prata', 'OWX2121',
'1CWZIZ389VT007299')
```

```
carro2 = Carro(2020, 'Edge', 'Ford', 'Branco', 'PNM1277',
'9BWZZZ377VT004251')
```

O primeiro carro foi chamado de `carro1` e o segundo de `carro2`. Os atributos foram enviados como parâmetros para serem definidos pelo construtor.

Feito isso, é possível acessar qualquer atributo dos objetos criados. Se quiser acessar a placa do carro1 por exemplo, devemos acessar da seguinte maneira:
`carro1.placa`

Ou seja, o objeto seguido de ponto ".", seguido do nome do atributo. Podemos também executar algumas ações com nossos objetos, por exemplo:

Aqui vamos adicionar 3 passageiros no `carro2`:

```
carro2.adicionarPassageiro(3)
```

Aqui vamos ligar, acerar, frear e desligar o `carro1`:

```
carro1.ligar()
carro1.acelerar()
carro1.acelerar()
carro1.acelerar()
carro1.acelerar()
carro1.acelerar()
carro1.frear()
carro1.frear()
carro1.frear()
carro1.frear()
carro1.frear()
carro1.desligar()
```

*Um dos pilares da programação orientada a objetos é a **herança**. Ela ocorre quando um objeto possui todas as características e métodos de uma determinada classe e também possui outras características particulares. Em outras palavras, cria-se uma classe a partir de outra classe.*

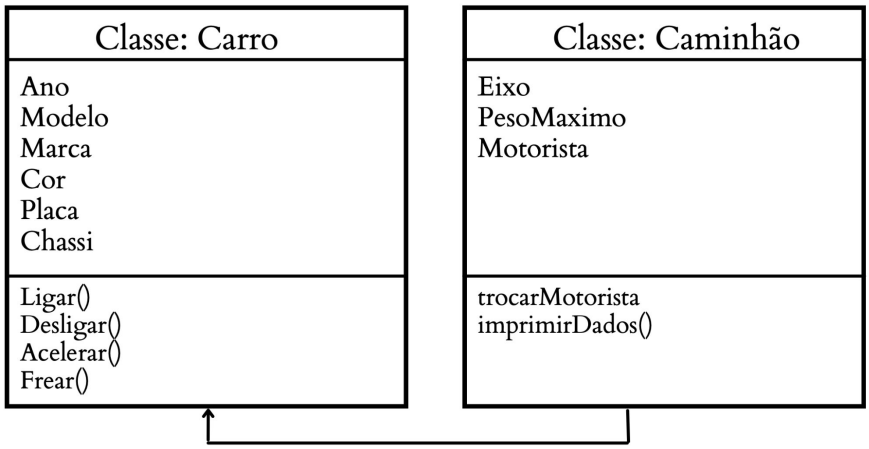
Vamos supor que no exemplo acima, do jogo de corrida, além da opção de correr com um carro, teremos a opção de correr com um caminhão.

Se pensarmos bem um caminhão, assim como um carro, tem Ano, modelo, marca, cor, placa e chassi e também pode Ligar(), Desligar(), Acelerar() e Frear().

Será que precisamos mesmo implementar tudo isso novamente na classe caminhão? A resposta é: Você pode, mas não precisa em virtude da herança que nos permite herdar a implementação de outra classe. Isso faz com que não percamos tempo implementando código repetido e isso facilita manutenções futuras.

Suponha que o padrão de placas mude novamente, se alterarmos isso na classe carro não é preciso alterar também na classe caminhão, porque esta segunda está herdando os códigos da primeira.

Vamos partir para a prática para fixarmos melhor este conceito?



No desenho anterior, o caminhão, além de ter todas as características e executar todas as ações da classe carro, tem suas particularidades que o carro não tem. São elas: *eixo, peso máximo de carga, motorista atual e permitir trocar de motorista e imprimir os dados do caminhão.*

Primeiro criamos a classe como nome Caminhão:

```
class Caminhao(Carro):
```

Agora vamos criar o construtor:

```
    def __init__(self,ano, modelo, marca, cor, placa, chassi, eixo,
pesoMaximo,motorista):
        super().__init__(ano, modelo, marca, cor, placa, chassi)
        self.eixo = eixo
        self.pesoMaximo = pesoMaximo
        self.motorista= motorista
```

Observe que dentro do construtor herdamos o construtor da super classe carro. Por isso usamos `super().__init__` ou seja, o da `__init__` superclasse. Os atributos ano, modelo, marca, cor, placa e chassi serão definidos para o objeto **caminhão** usando a implementação do `__init__` int da classe carro que estamos herdando.

Já os atributos que são apenas do objeto **caminhão** serão definidos no novo construtor que estamos desenvolvendo agora.

Como herdamos da classe carro, o caminhão pode ligar, desligar, acelerar, frear e adicionar passageiros. Mas aí teremos um pequeno problema. Um carro comum comporta 4 passageiros e um caminhão comum 2 passageiros, se herdamos o método `adicionarPassageiro` da classe carro estaremos permitindo adicionar 4 passageiros na classe caminhão?

Bom, se usarmos apenas o conceito de herança sim, mas temos também em **POO** o conceito de polimorfismo o qual permite que uma classe aproveite os métodos de outra classe porém implementando de maneira diferente. Veja:

```
def adicionarPassageiro(self, quantidade):
  if self.passageiros+quantidade<=2 :
    self.passageiros+=quantidade
  else:
    print('Favor não transportar mais de 2 passageiros.')
```

Criamos o método `adicionarPassageiro` exatamente com o mesmo nome do que foi criado na classe carro, desse modo, na classe caminhão ele sobrescreverá a classe que adiciona passageiros herdada da classe carro. Deste modo não será possível adicionar 4 passageiros no caminhão. Você pode estar se perguntando, e se eu colocar outro nome no método, funciona? **Não funciona!**

Se o método chamar `adicionarPassageiroNoCaminhão` a classe caminhão terá o método `adicionarPassageiroNoCaminhão` e o método `adicionarPassageiro` que foi herdado. Assim, será possível adicionar 4 passageiros no caminhão.

Agora vamos implementar o método `trocarMotorista` que não existia na classe carro:

```
def trocarMotorista (self, motorista):
    self.motorista= motorista
```

O nome do motorista é enviado como parâmetro e atribuímos este nome ao atributo motorista.

Vamos também criar o método `imprimirDados` para imprimir os dados do caminhão:

```
    def imprimirDados(self):
      print('Ano: ',self.ano)
      print('Modelo: ',self.modelo)
      print('Marca: ',self.marca)
      print('Cor: ',self.cor)
      print('Placa: ',self.placa)
      print('Chassi: ',self.chassi)
      print('Eixo: ',self.eixo)
      print('pesoMaximo: ',self.pesoMaximo)
      print('motorista: ',self.motorista)
```

Pronto, finalizamos a criação da classe caminhão. Vamos criar um objeto do tipo Caminhão:

```
caminhao1 = Caminhao(2020, 'L-312', 'Mercedes-Benz', 'Amarelo',
'JNM2287', '1BOZZZ300VT004778', 'Eixo Simples com Rodagem Simples',
6000, 'Milena')
```

Os atributos foram enviados como parâmetros para serem definidos pelo construtor. Feito isso, é possível acessar qualquer atributo dos objetos criados. Se quiser acessar o chassi do caminhao1 por exemplo, devemos acessar da seguinte maneira:

```
caminhao1.chassi
```

Ou seja, o objeto seguindo de ponto ".", seguido do nome do atributo. Igual fizemos com os objetos do tipo Carro. Podemos também executar algumas ações com nosso novo objeto, por exemplo:

Aqui vamos adicionar 1 passageiro no `caminhao1`:

```
caminhao1.adicionarPassageiro(1)
```

Aqui vamos imprimir os dados do `caminhao1`:

```
caminhao1.imprimirDados()
```

Por fim, vamos ligar e desligar o `caminhao1`:

```
caminhao1.ligar()
caminhao1.desligar()
```