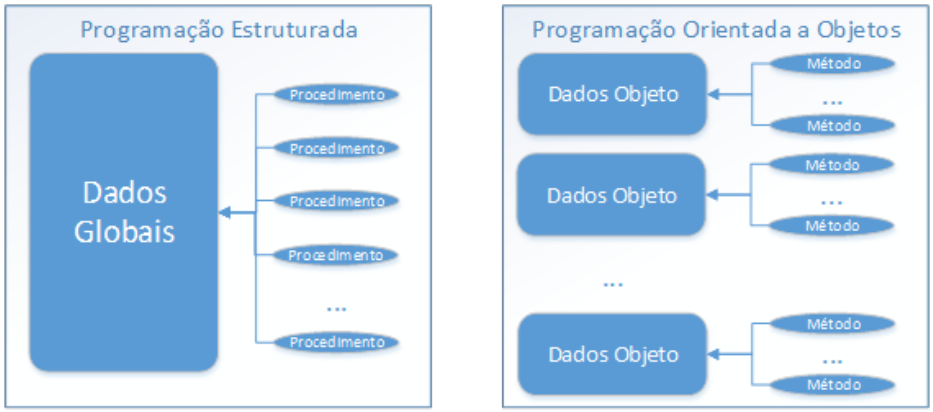
PYTHON
Programação Orientada a Objetos

O que é?

Programação Orientada a Objetos (também conhecida pela sua sigla **POO**) é um modelo de análise, projeto e programação de software baseado na composição e interação entre diversas unidades chamadas de **'objetos'**. A Programação Orientada a Objetos (POO) diz respeito a um padrão de desenvolvimento que é seguido por muitas linguagens, como C# e Java.

Programação Estruturada vs Programação Orientada a Objetos.

Na abaixo podemos observar uma comparação muito clara entre a programação estruturada e a programação orientada a objetos no que diz respeito aos dados. Repare que, no paradigma estruturado, temos procedimentos (ou funções) que são aplicados globalmente em nossa aplicação. No caso da orientação a objetos, temos métodos que são aplicados aos dados de cada objeto. Essencialmente, os procedimentos e métodos são iguais, sendo diferenciados apenas pelo seu escopo.



A linguagem C é a principal representante da programação estruturada. Se trata de uma linguagem considerada de baixo nível, que atualmente não é utilizada para projetos muito grandes. A sua principal utilização, devido ao baixo nível, é em programação para sistemas embarcados ou outros em que o conhecimento do hardware se faz necessário para um bom programa.

Os 4 pilares da Programação Orientada a Objetos.

Para entendermos exatamente do que se trata a orientação a objetos, vamos entender quais são os requerimentos de uma linguagem para ser considerada nesse paradigma. Para isso, a linguagem precisa atender a quatro tópicos bastante importantes:

Abstração

A abstração consiste em um dos pontos mais importantes dentro de qualquer linguagem Orientada a Objetos. Como estamos lidando com uma representação de um objeto real (o que dá nome ao paradigma), temos que imaginar o que esse objeto irá realizar dentro de nosso sistema. São três pontos que devem ser levados em consideração nessa abstração.

O primeiro ponto é darmos uma identidade ao objeto que iremos criar. Essa identidade deve ser única dentro do sistema para que não haja conflito. Na maior parte das linguagens, há o conceito de pacotes (ou namespaces). Nessas linguagens, a identidade do objeto não pode ser repetida dentro do pacote, e não necessariamente no sistema inteiro. Nesses casos, a identidade real de cada objeto se dá por ..

A segunda parte diz respeito a características do objeto. Como sabemos, no mundo real qualquer objeto possui elementos que o definem. Dentro da programação orientada a objetos, essas características são nomeadas propriedades. Por exemplo, as propriedades de um objeto “Cachorro” poderiam ser “Tamanho”, “Raça” e “Idade”.

Por fim, a terceira parte é definirmos as ações que o objeto irá executar. Essas ações, ou eventos, são chamados métodos. Esses métodos podem ser extremamente variáveis, desde “Acender()” em um objeto lâmpada até “Latir()” em um objeto cachorro.

Encapsulamento

O encapsulamento é uma das principais técnicas que define a programação orientada a objetos. Se trata de um dos elementos que adicionam segurança à aplicação em uma programação orientada a objetos pelo fato de esconder as propriedades, criando uma espécie de caixa preta.

A maior parte das linguagens orientadas a objetos implementam o encapsulamento baseado em propriedades privadas, ligadas a métodos especiais chamados getters e setters, que irão retornar e setar o valor da propriedade, respectivamente. Essa atitude evita o acesso direto a propriedade do objeto, adicionando uma outra camada de segurança à aplicação.

Para fazermos um paralelo com o que vemos no mundo real, temos o encapsulamento em outros elementos. Por exemplo, quando clicamos no botão ligar da televisão, não sabemos o que está acontecendo internamente. Podemos então dizer que os métodos que ligam a televisão estão encapsulados.

Herança

O reuso de código é uma das grandes vantagens da programação orientada a objetos. Muito disso se dá por uma questão que é conhecida como herança. Essa característica otimiza a produção da aplicação em tempo e linhas de código.

Para entendermos essa característica, vamos imaginar uma família: a criança, por exemplo, está herdando características de seus pais. Os pais, por sua vez, herdam algo dos avós, o que faz com que a criança também o faça, e assim sucessivamente. Na orientação a objetos. O objeto abaixo na hierarquia irá herdar características de todos os objetos acima dele, seus “ancestrais”. A herança a partir das características do objeto mais acima é considerada herança direta, enquanto as demais são consideradas heranças indiretas. Por exemplo, na família, a criança herda diretamente do pai e indiretamente do avô e do bisavô.

A questão da herança varia bastante de linguagem para linguagem. Em algumas delas, como C++, há a questão da herança múltipla. Isso, essencialmente, significa que o objeto pode herdar características de vários “ancestrais” ao mesmo tempo diretamente. Em outras palavras, cada objeto pode possuir quantos pais for necessário. Devido a problemas, essa prática não foi difundida em linguagens mais modernas, que utilizam outras artimanhas para criar uma espécie de herança múltipla.

Polimorfismo

Outro ponto essencial na programação orientada a objetos é o chamado polimorfismo. Na natureza, vemos animais que são capazes de alterar sua forma conforme a necessidade, e é dessa ideia que vem o polimorfismo na orientação a objetos. Como sabemos, os objetos filhos herdam as características e ações de seus “ancestrais”. Entretanto, em alguns casos, é necessário que as ações para um mesmo método seja diferente. Em outras palavras, o polimorfismo consiste na alteração do funcionamento interno de um método herdado de um objeto pai.

Como um exemplo, temos um objeto genérico “Eletrodoméstico”. Esse objeto possui um método, ou ação, “Ligar()”. Temos dois objetos, “Televisão” e “Geladeira”, que não irão ser ligados da mesma forma. Assim, precisamos, para cada uma das classes filhas, reescrever o método “Ligar()”.

Esses quatro pilares são essenciais no entendimento de qualquer linguagem orientada a objetos e da orientação a objetos como um todo. Cada linguagem irá implementar esses pilares de uma forma, mas essencialmente é a mesma coisa. Apenas a questão da herança, como comentado, que pode trazer variações mais bruscas, como a presença de herança múltipla. Além disso, o encapsulamento também é feito de maneiras distintas nas diversas linguagens, embora os getters e setters sejam praticamente onipresentes.

Principais vantagens da POO

A programação orientada a objetos traz uma ideia muito interessante: a representação de cada elemento em termos de um objeto, ou classe. Esse tipo de representação procura aproximar o sistema que está sendo criado ao que é observado no mundo real, e um objeto contém características e ações, assim como vemos na realidade.

A reutilização de código é um dos principais requisitos no desenvolvimento de software atual. Com a complexidade dos sistemas cada vez maior, o tempo de desenvolvimento iria aumentar exponencialmente caso não fosse possível a reutilização. A orientação a objetos permite que haja uma reutilização do código criado, diminuindo o tempo de desenvolvimento, bem como o número de linhas de código. Isso é possível devido ao fato de que as linguagens de programação orientada a objetos trazem representações muito claras de cada um dos elementos, e esses elementos normalmente não são interdependentes. Essa independência entre as partes do software é o que permite que esse código seja reutilizado em outros sistemas no futuro.

Classes em Python

Uma classe associa **dados (atributos)** e **operações (métodos)** em uma só estrutura. Um **objeto** é uma **instância** de uma classe. Ou seja, uma representação da classe. Veja abaixo como podemos definir uma simples classe em Python.

```
class Pessoa:
    pass
```

Utilizamos a palavra reservada **class** e em seguida o nome da classe. para que serve o pass? A resposta é que, em Python, ao contrário de várias outras linguagens de programação, os blocos de código NÃO são definidos com os caracteres { e }, mas sim com indentação e o caractere :. Devido a esse fato, python necessitava de algo para explicitar quando se quer definir um bloco vazio. O pass foi criado exatamente para explicitar essa situação.

Um exemplo de uma função vazia feita em linguagem Java e a mesma função vazia feita em Python:

```
public class Pessoa {

}
```

```
class Pessoa:
    pass
```

Por exemplo, Felipe é uma instância de uma classe chamada Pessoa, mas a Pessoa é a classe que o representa de uma forma genérica. Se você criar um outro objeto chamado Fábio, esse objeto também será uma instância da classe Pessoa. Veja o exemplo abaixo:

```
1. class Pessoa:
2.
3.     def __init__(self, nome):
4.         self.nome = nome
5.
6. felipe = Pessoa(‘Felipe’)
7. fabio = Pessoa('Fabio')
```

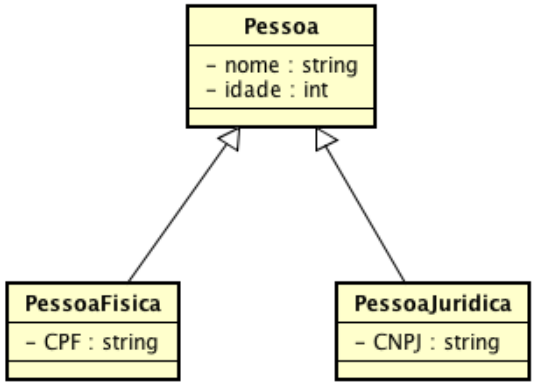
Métodos de uma classe:
Um método ou uma função no python é definido pela palavra reservada **def.** Em seguida entre parênteses são passados os parâmetros que o método pode utilizar para realizar alguma operação.

```
def metodo(self, args):
    pass
```

Note que inicialmente, é passado a palavra **self**, esse parâmetro é **obrigatório** quando estamos trabalhando com métodos internos de uma classe. O self é utilizado para fazer uma atribuição a um atributo de instância, então é necessário explicitar a que instância atribuir, self informa que a atribuição é a si mesmo. self é o this do Java.

```
1. class Pessoa:
2.
3.     def __init__(self, nome):
4.         self.nome = nome
5.
6.     def falar(self):
7.         return 'Oi me chamo' + self.nome
8.
9. felipe = Pessoa('Felipe')
10. print(felipe.falar())
```

Herança em Python
Na Programação Orientada a Objetos o conceito de herança é muito utilizado. Basicamente, dizemos que a herança ocorre quando uma classe (filha) herda características e métodos de uma outra classe (pai), mas não impede de que a classe filha possua seus próprios métodos e atributos. Na imagem abaixo podemos ver, em nível de diagrama, como esta relação ocorre:




```
1. from pessoa import Pessoa
2.
3. class PessoaFisica(Pessoa):
4.     def __init__(self, CPF, nome, idade):
5.         super().__init__(nome, idade)
6.         self.CPF = CPF
```

```
7.
8.      def getCPF(self):
9.          return self.CPF
10.
11.     def setCPF(self, CPF):
12.         self.CPF = CPF
```

Linha 1: Como vamos herdar da classe Pessoa e ela foi definida em outro arquivo (pessoa.py), precisamos importá-la. Para isso usamos a instrução import e indicamos o nome do arquivo, sem a extensão .py, seguido do nome da classe que queremos importar;

Linha 3: Para definir que uma classe herdará de uma outra classe, precisamos indicar o nome da classe pai entre parênteses após o nome da classe filha;

Linha 4: Criamos o construtor da classe filha e definimos quais atributos ela espera receber. Neste caso, o nome e idade estão definidos na classe pai, enquanto o cpf é próprio da classe filha;

Linha 5: Utilizando o método super, invocamos o construtor da classe pai. Com isso aproveitamos todo a lógica definida nesse método, que no caso faz a atribuição dos valores de nome e idade aos atributos da classe. Com isso garantimos que ao ser criada, a classe filha efetuará o mesmo processamento que a classe pai e mais alguns passos adicionais.

```
1.  from pessoa import Pessoa
2.
3.  class PessoaJuridica(Pessoa):
4.      def __init__(self, CNPJ, nome, idade):
5.          super().__init__(nome, idade)
6.          self.CNPJ = CNPJ
7.
8.      def getCNPJ(self):
9.          return self.CNPJ
```

Encapsulamento:

```
1. class ClassePai:
2.     a = 1 # atributo público
3.     __b = 2 # atributo privado a class ClassePai
4.
5. class ClasseFilha(ClassePai):
6.     __c = 3 # atributo privado a ClasseFilha
7.
8.     def __init__(self):
9.       print (self.a)
10.     print (self.__c)
11.
12. instancia_pai = ClassePai()
13. print (instancia_pai.a) # imprime 1
14.
15. instancia_filha = ClasseFilha()
16. print (instancia_filha.__b )
17. # Erro, pois __b é privado a classe ClassePai.
18. print (instancia_filha.__c)
19. # Erro, __c é um atributo privado, somente chamado pela classe.
```

Linha 1: Criamos uma superclasse e atribuímos dois atributos, a e b. Quando desejamos colocar um atributo privado usamos __ no início da variável.

Linha 3: A classe filha herda da classe pai chamada ClassePai.

Linha 6: A classe filha agora possui um atributo privado a própria classe __c

Polimorfismo:

```
1.  class Pessoa(object):
2.      def __init__(self, nome ='', idade=0):
3.          self.nome = nome
4.          self.idade = idade
5.
6.      def calcular_lucro(self, valor, valor2, taxa):
7.          return 'Podemos retornar o lucro!'
8.
9.  class PessoaFisica(Pessoa):
10.     def __init__(self, CPF, nome, idade):
11.         super().__init__(nome, idade)
12.         self.CPF = CPF
13.
14.     def __str__ (self):
15.         return '{} - {} - {}'.format(self.nome, self.idade, self.CPF)
16.
17.     def calcula_taxa(self, salario, portentual):
18.         return ((salario*portentual)/100)
19.
20.     def calcular_lucro(self, valor, taxa):
21.         return 'Podemos retornar o lucro {}'.format(valor*taxa)
22.
23. class PessoaJuridica(Pessoa):
24.     def __init__(self, CNPJ, nome, idade):
25.         Pessoa.__init__(self, nome, idade)
26.         self.__CNPJ = CNPJ
27.
28.     def __str__ (self):
29.         return '{} - {} - {}'.format(self.nome, self.__CNPJ, self.idade)
30.
31.     def calcula_lucro(self, faturamento, portentual):
32.         return ((faturamento*portentual)/100) + 20
33.
34. pessoa = PessoaFisica('122.333.332-1', 'Felipe', 28)
35. banco = PessoaJuridica('00.000.000/0001-11', 'Banco do Brasil', 435)
36.
37. print (pessoa.calcular_lucro(30,30))
38. print (pessoa.calcula_taxa(1525, 5))
39. print (banco.calcular_lucro(3,4,5))
```

Linha 6: Criamos a função calcular_lucro na classe pai Pessoa.

Linha 9: Herdamos na classe filha PessoaFisica a classe pai Pessoa

Linha 20: A classe filha PessoaFisica tem uma função calcular_lucro igual a classe pai Pessoa, porém na classe filha o lucro é calculado de forma diferente, adquirindo um novo comportamento.

Linha 23: Herdamos na classe filha PessoaJuridica a classe pai Pessoa.

Linha 31: A classe filha PessoaJuridica tem uma função calcular_lucro igual a classe pai Pessoa, porém na classe filha o lucro é calculado de forma diferente, adquirindo um novo comportamento.

Linhas 34 e 35: Criamos duas instâncias para representar uma PessoaFisica e uma PessoaJuridica

Linha 37: A partir do objeto pessoa executamos a função calcular_lucro com o comportamento atribuída na classe filha PessoaFisica.

Linha 38: A partir do objeto banco executamos a função calcular_lucro com o comportamento atribuída na classe filha PessoaJuridica.