- Classe Abstrata
  - Não pode ser instanciada
  - Só pode servir para herança
- Método Abstrato
  - Sem implementação
- Classe Final
  - Não pode ser herdada por outra classe
  - Obrigatoriamente folha
- Método Final
  - Não pode ser sobrescrito pelas suas subclasses
  - Obrigatoriamente herdado
- abc – abstract base class

```
from abc import ABC, abstractmethod
import math
class Poligono(ABC):
    @abstractmethod
    def area(self):
        pass
    @abstractmethod
    def perimetro(self):
        pass
    @abstractmethod
    def diametro(self):
        pass
    def __str__(self):
        aux = f"Area: {self.area()}"
        aux += f"\nPerimetro: {self.perimetro()}"
        aux += f"\nDiametro: {self.diametro()}"
        return aux
class Quadrado(Poligono):
    def __init__(self, lado):
        self.lado = lado
        return self.lado ** 2
    def perimetro(self):
        return self.lado * 4
    def diametro(self):
        return self.lado * math.sqrt(2)
q1 = Quadrado(5)
print(q1)
class Circulo(Poligono):
    def __init__(self, raio):
        self.raio = raio
    def area(self):
        return (self.raio ** 2) * 3.14
    def perimetro(self):
        return self.raio * 2 * 3.14
    def diametro(self):
        return self.raio * 2
C1 = Circulo(5)
print(C1)
    def perimetro(self):
        return 2 * (self.base + self.altura)
    def diametro(self):
        return math.sqrt(self.base**2 + self.altura**2)
R1 = Retangulo(5,8)
print(R1)
class Triangulo(Poligono):
    def __init__(self, base):
        self.base = base
    @abstractmethod
    def altura(self):
        pass
    def area(self):
        return (self.base * self.altura()) / 2
    def __str__(self):
        return super().__str__()
class TrianguloRetangulo(Triangulo):
    def __init__(self, cateto1, cateto2):
        super().__init__(base=cateto1)
        self.cateto1 = cateto1
        self.cateto2 = cateto2
    def altura(self):
        return self.cateto2
    def perimetro(self):
        return self.cateto1 + self.cateto2 + self.hipotenusa()
    def diametro(self):
        return self.hipotenusa()
    def hipotenusa(self):
        return math.sqrt(self.cateto1**2 + self.cateto2**2)
    def __str__(self):
        return super().__str__()
t1 = TrianguloRetangulo(3, 4)
print(t1)
```

- Herança Múltipla
  - Um objeto pode conter "MUITOS" outros por composição; Um objeto quer ser "muitos" outros
  - Algorítmo Diamante - ordem de resolução de métodos

```
class Terreste(object):
    def __init__(self, velocidade=100):
        self.se_move_em_terra = True
        self.velocidade_em_terra = velocidade
    def __str__(self):
        aux = '\nse_move_em_terra = '
        aux += str(self.se_move_em_terra)
        aux += '\nvelocidade_em_terra = '
        aux += str(self.velocidade_em_terra)
        return aux
class Aquatico(object):
    def __init__(self, velocidade=5):
        self.se_move_na_agua = True
        self.velocidade_agua = velocidade
    def __str__(self):
        aux = '\nse_move_na_água = '
        aux += str(self.se_move_na_agua)
        aux += '\nvelocidade_agua = '
        aux += str(self.velocidade_agua)
        return aux
class Carro(Terreste):
    def __init__(self, veloc=120, pistoes=4):
        self.rodas = 4
        self.pistoes = pistoes
        Terreste.__init__(self, veloc)
    def __str__(self):
        aux = '\nrodas = ' + str(self.rodas)
        aux += '\npistões = ' + str(self.pistoes)
        aux += super().__str__()
        return aux
class Barco(Aquatico):
    def __init__(self, veloc=6, helices=1):
        self.helices = helices
        Aquatico.__init__(self, veloc)
    def __str__(self):
        aux = '\nhélices = ' + str(self.helices)
        aux += super().__str__()
        return aux
class Anfibio(Carro, Barco):
    def __init__(self, vt=80, va=4, pi=6, he=2):
        Carro.__init__(self, veloc=vt, pistoes=pi)
        Barco.__init__(self, veloc=va, helices=he)
    def __str__(self):
        aux = Carro.__str__(self)
        aux += Barco.__str__(self)
```

- Interfaces
  - É um tipo de herança "fraca" -> é uma classe abstrata que contém SOMENTE métodos abstratos
  - São as assinaturas do métodos {enviar(), receber(), fazer()}
  - Somente é permitida a especificação de:
    - Constantes públicas e Métodos públicos e abstratos
  - São proibidos:
    - Atributos e Métodos privados e protegidos
  - Definem um comportamento -> Praticamente um "decorador"

```
class Cobrador(ABC):
    @abstractmethod
    def cobrarTaxa(self):
        pass
class ContaCorrente(Conta)
    def cobrarTaxa(self):
        self._saldo -= 10.0
```