In [ ]:
```
MRO --> Method Resolution Order

- C3 linearization algorithm---order

- child classses are checked br=efore parent class
- Parent classes are checked in the order they appear in the inheritance list
- Order is cosistent and that avoid ambiguity
```

In [26]:
```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B,C,A): # child, child parent,  parent, parent,parent
    def method(self):
        print("Class D")

class E(D):
    def method(self):
        print("Class E")

e = E()
e.method()
```

```
Class E
```

In [27]:
```
d = D()
d.method()
```

```
Class D
```

PolyMorphism

In [ ]:
```
poly + morphism ---> Many Forms


Its the ability of different object to respond to the same interface in their ow
Same method call can behave differently depeding on the object that recieve it.
```

In [35]:
```
class Parrot:

    def fly(self):
        print("I can fly")

    def run(self):
        print("I can not run")

class Penguin(Parrot):

    def fly(self):
        print("I can not fly")
```

```
    def run(self):
        print("I can run")


p = Parrot()
p2 = Penguin()


def flying_test(cls):
    cls.fly()

flying_test(p2)
#flying_test(p2)
```

```
I can not fly
```

Type of PolyMorphism

1- Method Overloading

2- Method Overriding

In [39]:
```
#1- Method Overloading

#rule 1 : Method overloading is happen within the same name of the class
#rule 2 : Method overloading is happen within the same name of the class with di
class Parrot:

    def fly(self):
        print("I can fly")

    def run(self):
        print("I can not run")

    def fly(self):
        print("I can jump")


p1 = Parrot()
p1.fly()
```

```
I can jump
```

In [41]:
```
#1- Method Overriding

#it works on 2 different class, Method Overriding is required at least 2 class e

class Parrot:

    def fly(self):
        print("I can fly")

    def run(self):
        print("I can not run")

class Penguin(Parrot):

    def fly(self):
```

```
        print("I can not fly")

    def run(self):
        print("I can run")

p3 = Penguin()
p3.fly()
```

```
I can not fly
```

In [ ]:
```
# Encapsulation

Enacapsulation is hiding the information from outside of the world.

Enacapsulation is hiding the information form others

# denoted by
__ = private

- Acess modifiers

    1- private Acess modifiers

    2- public Acess modifiers
```

In [47]:
```
class Computer:

    def __init__(self):
        self.__maxprice = 500
        self.size = "Medium"

    def sell(self):
        print("I am started selling the computer parts")

    def price(self, price):
        self.__maxprice = price
        return self.__maxprice

    def __max_price(self):
        print("We are giving the max price")

c = Computer()

print(c.sell())
#print(c.__maxprice) #  private acess modifier
print(c.size) # public acess modifier
print(c.__max_price())
```

```
I am started selling the computer parts
None
Medium
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[47], line 22
     20 #print(c.__maxprice) #  private acess modifier
     21 print(c.size) # public acess modifier
---> 22 print(c.__max_price())

AttributeError: 'Computer' object has no attribute '__max_price'
```

In [48]:
```
class Product(Computer):
    pass

p4 = Product()
p4.__max_price()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[48], line 5
      2     pass
      4 p4 = Product()
----> 5 p4.__max_price()

AttributeError: 'Product' object has no attribute '__max_price'
```

In [ ]:
```

```