Runtime Polymorphism or Duck Typing

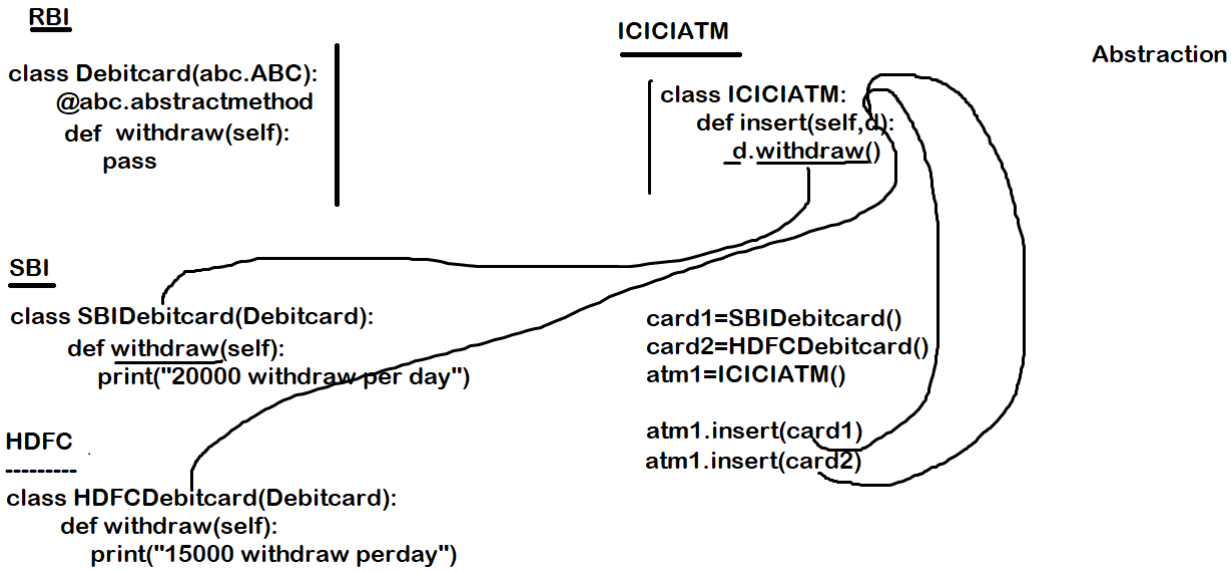
The behavior of reference variable changes based on the type of object assigned is called runtime polymorphism (OR) an ability of reference variable based on the type of object assigned.

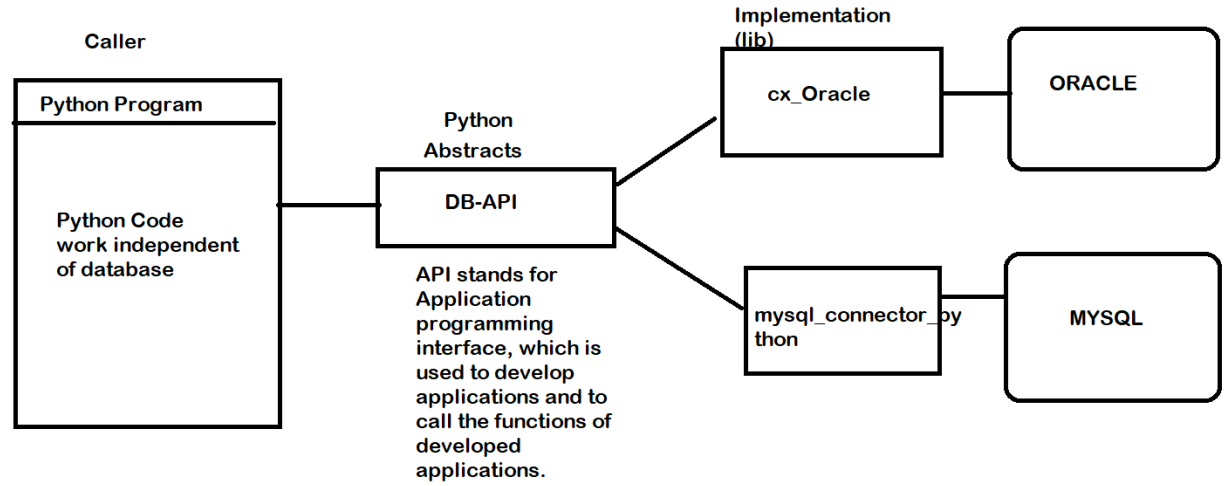
Duck Typing is a term commonly related to dynamically typed programming languages and polymorphism. The idea behind this principle is that the code itself does not care about whether an object is a duck, but instead it does only care about whether it quacks.

The advantage of runtime polymorphism is to develop loosely coupled code.

The runtime polymorphism is achieved using abstract classes and methods (OR) method overriding.

Using abstract classes and abstract methods a program can also achieve abstraction. Abstraction is process of hiding implementation details from caller, caller knows only specifications (abstracts).





Example:

```
import abc

class Driver(abc.ABC):
    @abc.abstractmethod
    def connect(self):
        pass

class HPDriver(Driver):
    def connect(self):
```

```
        print("connect to HP Printer")

class SonyDriver(Driver):
    def connect(self):
        print("Connect to Sony Printer")


class WindowOS:
    def install(self,d):
        d.connect()


windowos1=WindowOS()
d1=HPDriver()
d2=SonyDriver()
windowos1.install(d1)
windowos1.install(d2)
```

Output

connect to HP Printer
Connect to Sony Printer

1. Class
2. Object
3. Encapsulation
4. Inheritance
5. Polymorphism
6. Abstraction

Main objective of object oriented programming is developing user defined data types or custom classes.

Operator Overloading

Operator overloading is part of polymorphism.
Operator overloading is done by overriding operator magic methods. For every operator there is equal magic method exists in object class.

Binary Operators	**Magic Method or Special Function**
–	object.__sub__(self, other)
+	object.__add__(self, other)
*	object.__mul__(self, other)
/	object.__truediv__(self, other)
//	object.__floordiv__(self, other)
Comparison Operators	**Magic Method or Special Function**
<	object.__lt__(self, other)
>	object.__gt__(self, other)
<=	object.__le__(self, other)
>=	object.__ge__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)

```
class Matrix:
    def __init__(self):
        self.__m=[]
    def readMatrix(self):
        for i in range(2):
            row=[]
            for j in range(2):
                value=int(input("Enter Value"))
                row.append(value)
            self.__m.append(row)
    def printMatrix(self):
        for i in range(2):
```

```
            for j in range(2):
                print(self.__m[i][j],end=' ')
            print()

    def __add__(self, other): # overloading operator/method overriding
        m3=Matrix()
        for i in range(2):
            row=[]
            for j in range(2):
                row.append(self.__m[i][j]+other.__m[i][j])
            m3.__m.append(row)
        return m3



matrix1=Matrix()
matrix2=Matrix()
matrix1.readMatrix()
matrix2.readMatrix()
matrix1.printMatrix()
matrix2.printMatrix()
matrix3=matrix1+matrix2
matrix3.printMatrix()
```

Output

```
Enter Value1
Enter Value2
Enter Value3
Enter Value4
Enter Value5
Enter Value6
Enter Value7
Enter Value8
1 2
```

3 4
5 6
7 8
6 8
10 12

Nested Classes OR Inner Classes

Defining a class inside class is called nested class or inner class.

Advantage of nested classes

1. Hiding implementations
2. Modularity

Types of Inner classes

1. Member class
2. Local class

Member class

A class defined inside class and outside the methods is called member class. This member class can be used by other members of the class. This member class can be private, public or protected.

Syntax:

```
class <outer-class-name>:
    class <inner-class-name>:
        variables
        methods

    variable
    methods
```

Example:

```
class A: # outer class
    class B: # inner class/member class
        def m1(self):
            print("m1 of B class")
    class __C:
        def m1(self):
            print("m1 of C class")


objb=A.B()
objb.m1()
objc=A.__C()
```

Output

```
m1 of B class
Traceback (most recent call last):
  File "C:\Users\nit\PycharmProjects\project1\test59.py", line 12, in
<module>
    objc=A.__C()
         ^^^^^
AttributeError: type object 'A' has no attribute '__C'
```