Mastering Python

الدرس #5

OOP

بايثون والبرمجة نحو الشيء

By:

Hussam Hourani

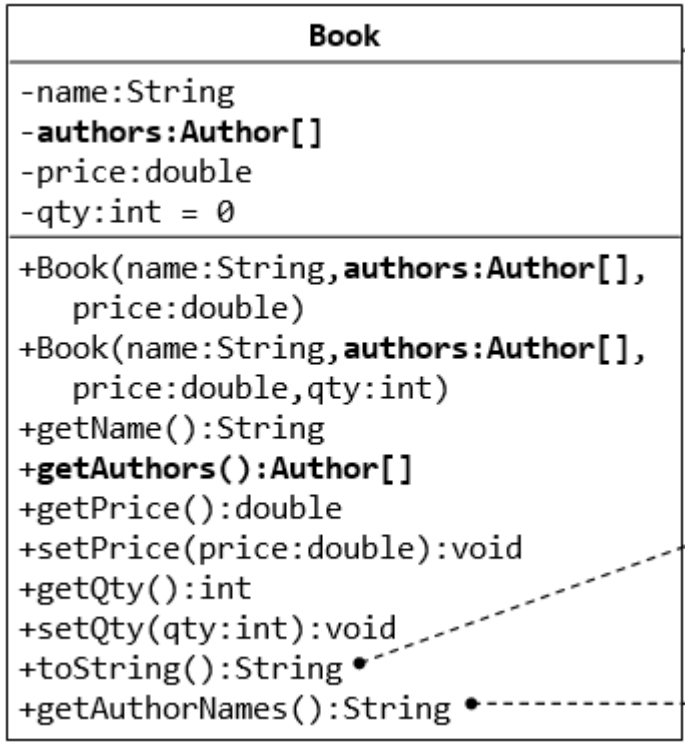
V1.0 - NOV 2019

Agenda

- What Is Object-Oriented Programming (OOP)?
- OOP key concepts
- Advantage of OOPs over Procedure-oriented
- Creating object and classes
- Constructor & Destructors
- Operator Overloading
- Inheritance
- Overriding methods
- Data Hiding and Encapsulation
- Class and Instance Variables

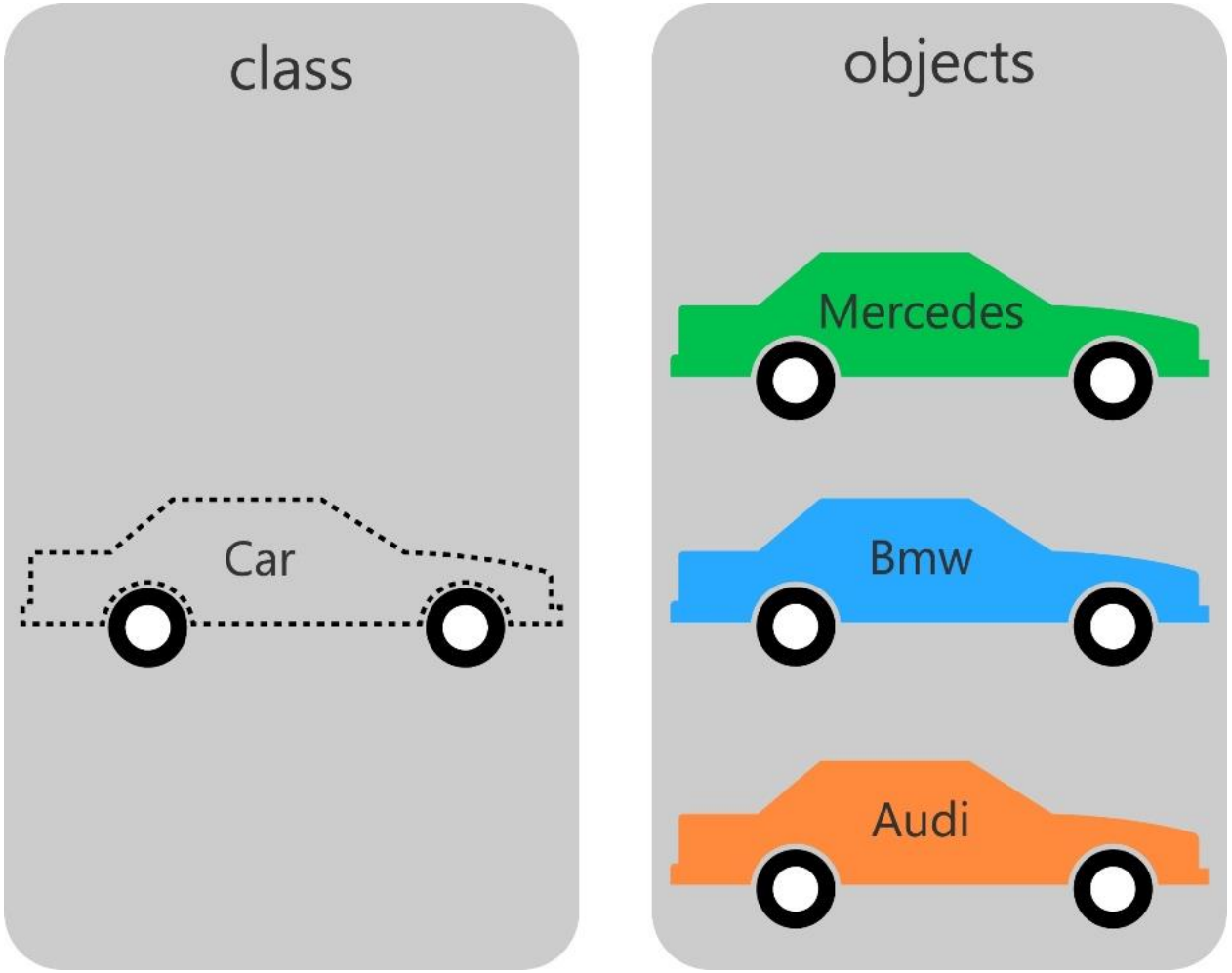
What Is Object-Oriented Programming (OOP)?

- Object-oriented Programming, or OOP for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.



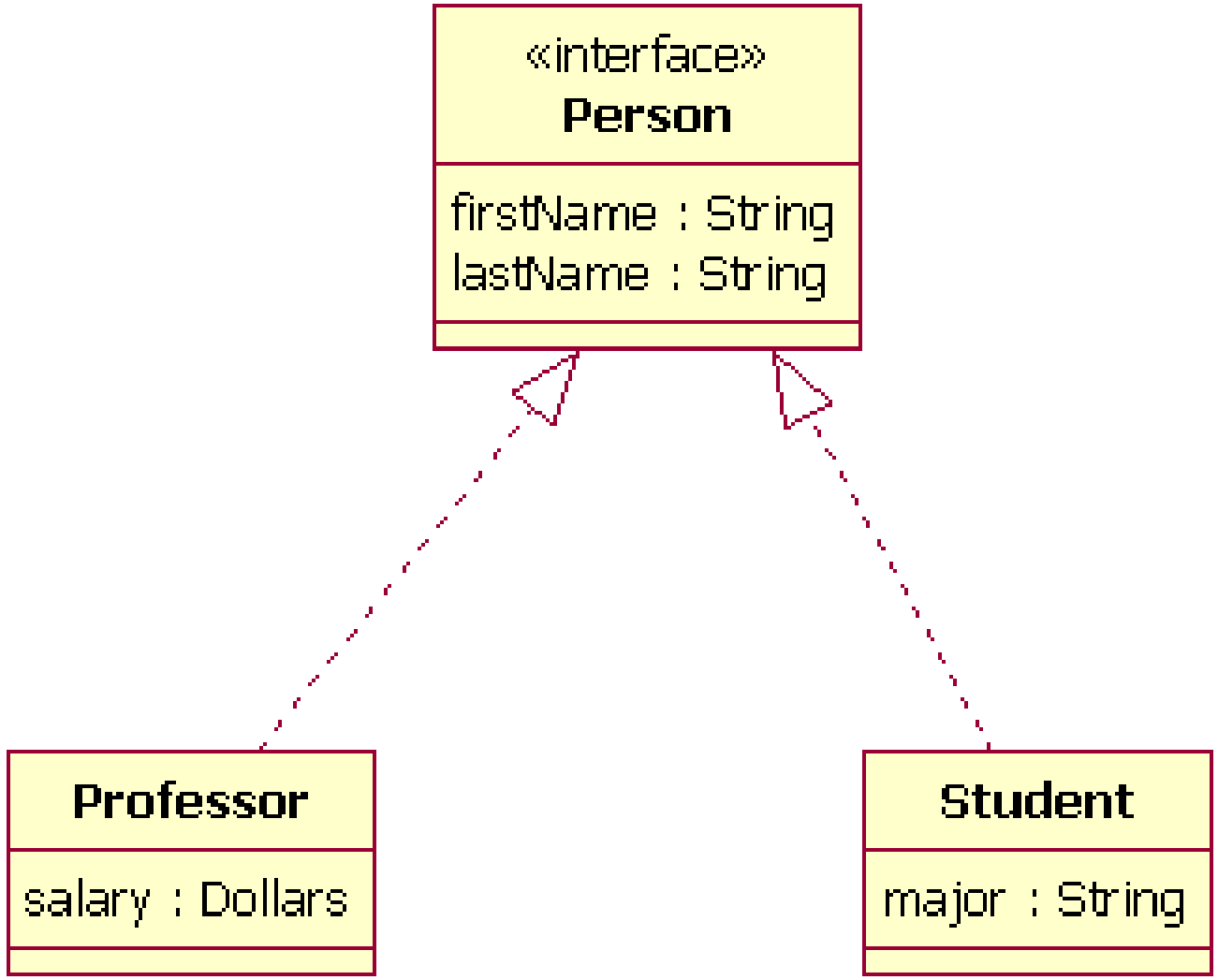
https://realpython.com/python3-object-oriented-programming/

Class vs Object

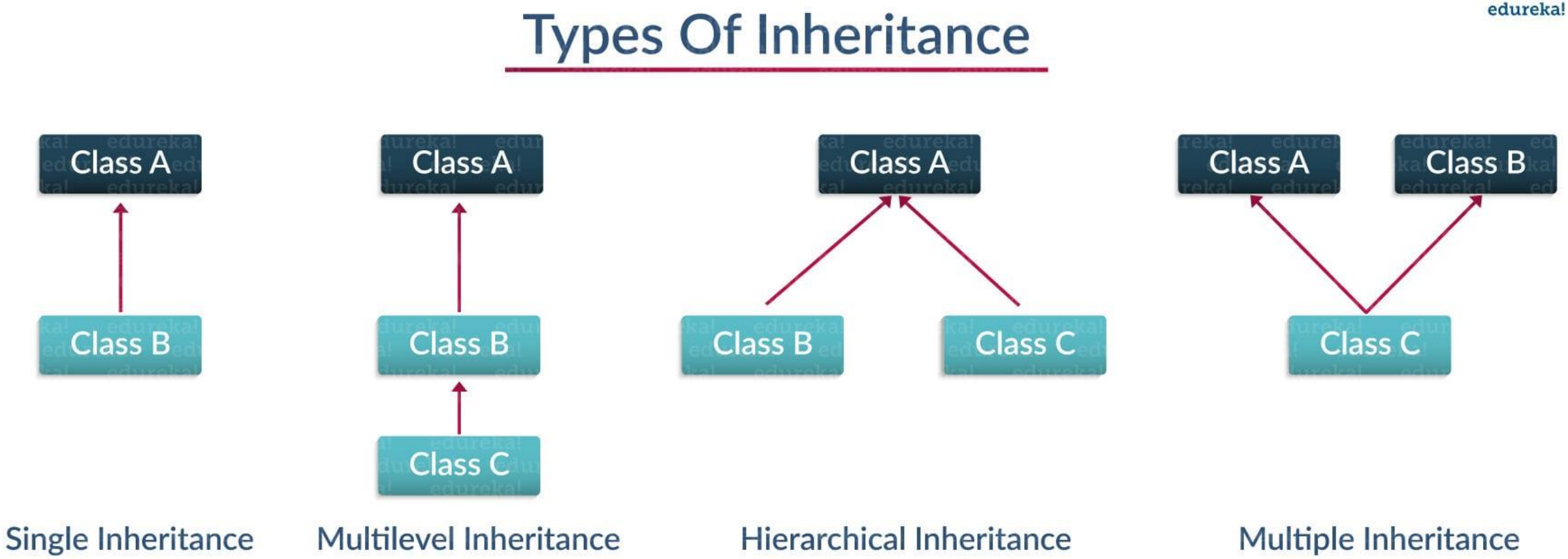


https://phpenthusiast.com/theme/assets/images/articles/classes_and_objects.jpg

Inheritance



Inheritance Types



https://www.edureka.co/blog/object-oriented-programming-python/

OOP key concepts

A class is a blueprint, a model for its objects (type of object).

A class defines a data type, it contains attributes and properties and methods

An instance is an object of a class created at run-time

Encapsulation :Binding (or wrapping) code and data together into a single unit


Capsule

A class can inherit attributes and behavior (methods) from other classes, called super-classes

Polymorphism : lets you use the same word to mean different things in different contexts

Data Hiding : prevent access to methods or variables outside the class

OOP is a design philosophy.

https://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep

https://www.python-course.eu/object_oriented_programming.php

Advantage of OOPs over Procedure-oriented

- OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
- OOPs provides data hiding and encapsulation whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
- OOPs provides ability to simulate real-world event much more effectively (like inheritance, Polymorphism,..) . We can provide the solution of real word problem if we are using the Object-Oriented Programming language.
- Object-oriented programming fosters reusability. A computer program is written in the form of objects and classes, which can be reused in other projects as well.

https://www.javatpoint.com/java-oops-concepts

Creating a class and an object

```
class Person:
    def say_hello(self):
        print('Hello')

p = Person()
p.say_hello()
```

Output

```
Hello
```

Note:
Class must have an extra parameter(self) that has to be added to the beginning of the parameter list, but you do not give a value for this parameter.

http://thepythonguru.com/python-inheritance-and-polymorphism/

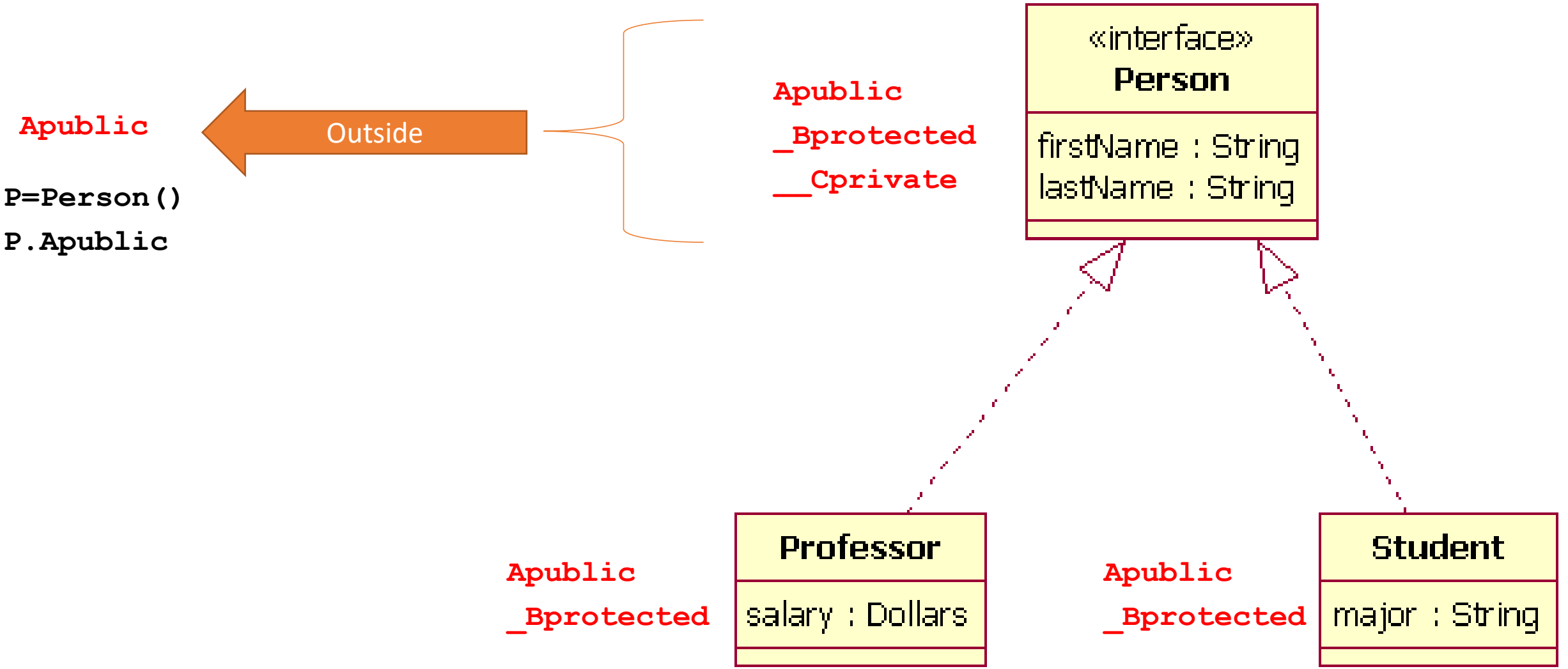
Constructor & Destructors

```
class Person:

       # constructor or initializer
      def __init__(self, name):
            self.name = name

      # method which returns a string
      def whoami( self ):
           return "I am " + self.name

     # destructors
      def __del__( self ):
            print ( 'I have been deleted')


p1 = Person('tom')
print(p1.whoami())
print(p1.name)
Del p1
```

Output

```
I am  tom
Tom
I have been deleted
```

Data Hiding and Encapsulation



https://www.python-course.eu/object_oriented_programming.php

Data Hiding and Encapsulation

```
class Encapsulation(object):
    def __init__(self, a, b, c):
        self.Apublic = a
        self._Bprotected = b
        self.__Cprivate = c
    def getprivate(self):
         return self.__Cprivate

x = Encapsulation(11,13,17)
print ( x.Apublic )
print ( x._Bprotected )
print ( x.__Cprivate) #->>>> Error
print ( x.getprivate())
```

Output

```
11
13
23
Error : AttributeError: 'Encapsulation'
object has no attribute '__private'
```

Name	Notation	Behavior
Name	Public	Can be accessed from inside and outside. All member variables and methods are public by default in Python
_name	Protected	Like a public member, but they shouldn't be directly accessed from outside. This effectively prevents it to be accessed, unless it is from within a sub-class.
__name	Private	Can't be seen and accessed from outside

https://www.python-course.eu/object_oriented_programming.php

Inheritance

```
class Parent(object):
    def __init__(self, name,age,salary):
        self.name = name
        self._age = age
        self.__salary = salary

    def public(self):
        print("Calling public")

    def _protected(self):
        print("Calling Protected")

    def __private(self):
        print("Is it really private?")

class Child(Parent):
    def foo(self):
        self.public()
        self._protected()
        print(self.name)
        print(self._age)

c = Child("Hussam",40,100)
c.foo()
c.public()
```

Output

```
Calling public
Calling Protected
Hussam
40
Calling public
```

Inheritance

```
class Parent(object):
    def public(self):
        print("Calling public")

    def _protected(self):
        print("Calling Protected")

    def __private(self):
        print("Is it really private?")

class Child(Parent):
    def foo(self):
        self.public()
        self._protected()

    def bar(self):
        self.__private() <<Error

c = Child()
c.foo()
c.public()
```

Output

```
Calling public
Calling Protected
Calling public
```


By : Hussam Hourani

Data Hiding and Encapsulation

public

```
class Cup:
    def __init__(self):
        self.color = None
        self.content = None

    def fill(self, beverage):
        self.content = beverage

    def empty(self):
        self.content = None

redCup = Cup()
redCup.color = "red"
redCup.content = "tea"
redCup.empty()
redCup.fill("coffee")
```

protected

```
class Cup:
    def __init__(self):
        self.color = None
        self._content = None

    def fill(self, beverage):
        self._content = beverage

    def empty(self):
        self._content = None

cup = Cup()
cup._content = "tea"
```

Warning!

private

```
class Cup:
    def __init__(self, color):
        self._color = color
         self.__content = None

    def fill(self, beverage):
        self.__content = beverage

    def empty(self):
        self.__content = None

redCup = Cup("red")
redCup.__content = "tea"
```

Error

https://www.python-course.eu/object_oriented_programming.php

Inheritance

```
class MySuperClass1():
    def method_super1(self):
        print("method_super1 method called")

class ChildClass( MySuperClass1):
    def child_method(self):
        print("child method")

c = ChildClass()
c.method_super1()
c.child_method()
```

Output

```
method_super1 method called
child method
```


http://thepythonguru.com/python-inheritance-and-polymorphism/

Inheritance

```
class A(object):
    def __init__(self):
        print("world")

class B(A):
    def __init__(self):
        print("hello")

b1=B()
```

Output

```
hello
```

```
class A(object):
    def __init__(self):
        print("world")

class B(A):
    def __init__(self):
        print("hello")
        super().__init__()
        A.__init__(self)
b1=B()
```

Output

```
hello
world
```

Inheritance

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width

class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)

class Cube(Square):
    def surface_area(self):
        face_area = super().area()
        return face_area * 6

    def volume(self):
        face_area = super().area()
        return face_area * self.length
c=Cube(2)

print(c.surface_area())
print(c.volume())
```

Output

```
24
8
```

Inheritance

```
class MySuperClass1():
    def method_super1(self):
        print("method_super1 method called")

class MySuperClass2():
    def method_super2(self):
        print("method_super2 method called")

class ChildClass( MySuperClass1, MySuperClass2 ):
    def child_method(self):
        print("child method")

c = ChildClass()
c.method_super1()
c.method_super2()
c.child_method()
```

Output

```
method_super1 method called
method_super2 method called
child method
```

Overriding methods

```
class A():

    def __init__(self):
        self.__x = 1

    def m1(self):
        print("m1 from A")


class B(A):

    def __init__(self):
        self.__y = 1

    def m1(self):
        print("m1 from B")

c = B()
c.m1()
```

Output

```
m1 from B
```


http://thepythonguru.com/python-inheritance-and-polymorphism/

Overriding methods

```
# Create Class Vehicle
class Vehicle:
    def print_details(self):
        print("This is parent Vehicle class method")

# Create Class Car that inherits Vehicle
class Car(Vehicle):
    def print_details(self):
        print("This is child Car class method")

# Create Class Cycle that inherits Vehicle
class Cycle(Vehicle):
    def print_details(self):
        print("This is child Cycle class method")

car_a = Vehicle()
car_a. print_details()

car_b = Car()
car_b.print_details()

car_c = Cycle()
car_c.print_details()
```

Output

```
This is parent Vehicle class method
This is child Car class method
This is child Cycle class method
```


http://thepythonguru.com/python-inheritance-and-polymorphism/

Operator Overloading

OPERATOR	FUNCTION	METHOD DESCRIPTION
+	__add__(self, other)	Addition
*	__mul__(self, other)	Multiplication
-	__sub__(self, other)	Subtraction
%	__mod__(self, other)	Remainder
/	__truediv__(self, other)	Division
<	__lt__(self, other)	Less than
<=	__le__(self, other)	Less than or equal to
==	__eq__(self, other)	Equal to
!=	__ne__(self, other)	Not equal to
>	__gt__(self, other)	Greater than
>=	__ge__(self, other)	Greater than or equal to
[index]	__getitem__(self, index)	Index operator
in	__contains__(self, value)	Check membership
len	__len__(self)	The number of elements
str	__str__(self)	The string representation

http://thepythonguru.com/python-inheritance-and-polymorphism/

Operator Overloading example

```
class Circle:
    def __init__(self, radius):
        self.__radius = radius

    def setRadius(self, radius):
        self.__radius = radius

    def getRadius(self):
        return self.__radius

    def __add__(self, another_circle):
        return Circle( self.__radius + another_circle.__radius )

c1 = Circle(4)
print(c1.getRadius())
c2 = Circle(5)
print(c2.getRadius())
c3 = c1 + c2
print(c3.getRadius())
```

Output

```
4
5
9
```


http://thepythonguru.com/python-inheritance-and-polymorphism/

Polymorphism

```python
class Document:
    def __init__(self, name):
        self.name = name

class Pdf(Document):
    def show(self):
        return 'Show pdf contents!'

class Word(Document):
    def show(self):
        return 'Show word contents!'

documents = [Pdf('Document1'), Pdf('Document2'), Word('Document3')]

for document in documents:
    print (document.name + ': ' + document.show())

d1 = Pdf('Document4')
d2 = Word('Document5')

for document in (d1,d2):
    print (document.name + ': ' + document.show())
```

Output

```
Document1: Show pdf contents!
Document2: Show pdf contents!
Document3: Show word contents!
Document4: Show pdf contents!
Document5: Show word contents!
```

Local Variables

```
# Creates class Car
class Car:
    def start(self):
        message = "Engine started"
        return message
car_a = Car()
print(car_a.start())
```

Output

```
Engine started
```

```
# Creates class Car
class Car:
    def start(self):
        message = "Engine started"
        return message
car_a = Car()
print(car_a.message)
```

Output

Error

```
AttributeError: 'Car'
object has no
attribute 'message'
```

Class and Instance Variables

```
class Dog:

    kind = 'canine' # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each
instance

d = Dog('Fido')
e = Dog('Buddy')
print( d.kind)                  # shared by all dogs
print(e.kind)                  # shared by all dogs
print(d.name )                 # unique to d
print(e.name )                 # unique to e
d.kind = "e"
print( d.kind)
print(e.kind)
```

Output

```
canine
canine
Fido
Buddy
e
canine
```

Class and Instance Variables

```
class Dog:

    tricks = [] # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
print( d.tricks )   # unexpectedly shared by all dog
```

Output

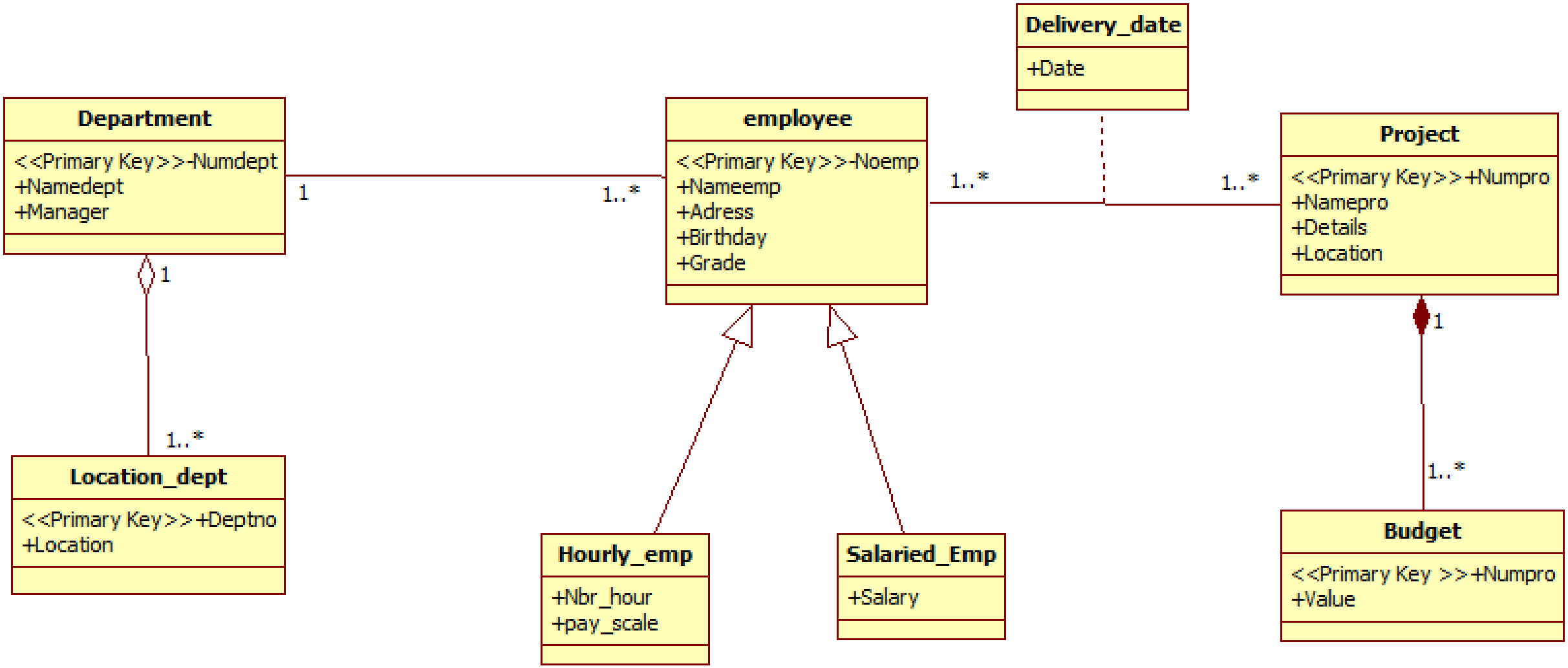
```
['roll over', 'play dead']
```

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty
list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
print(d.tricks)
Print(e.tricks)
```

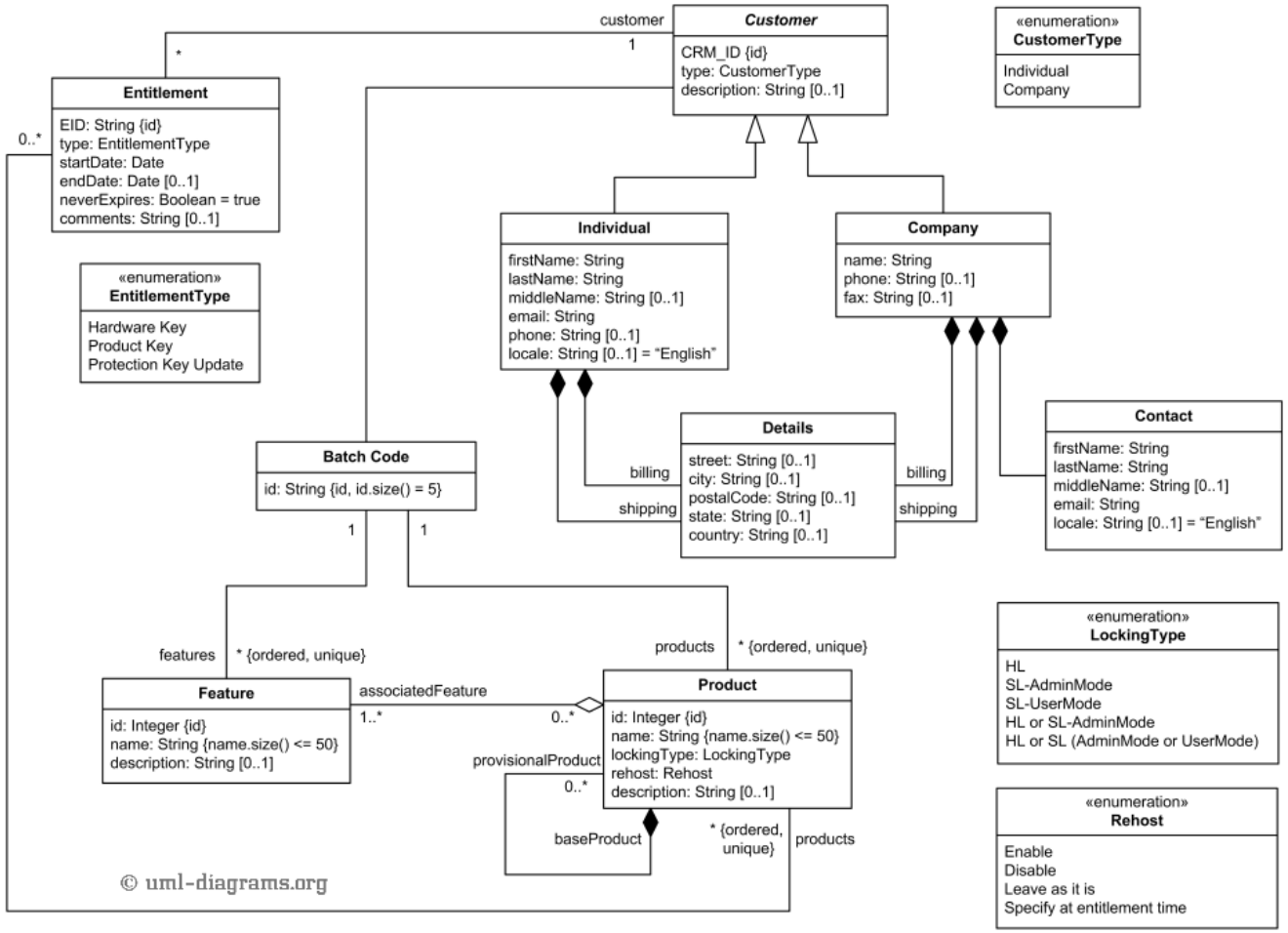
Output

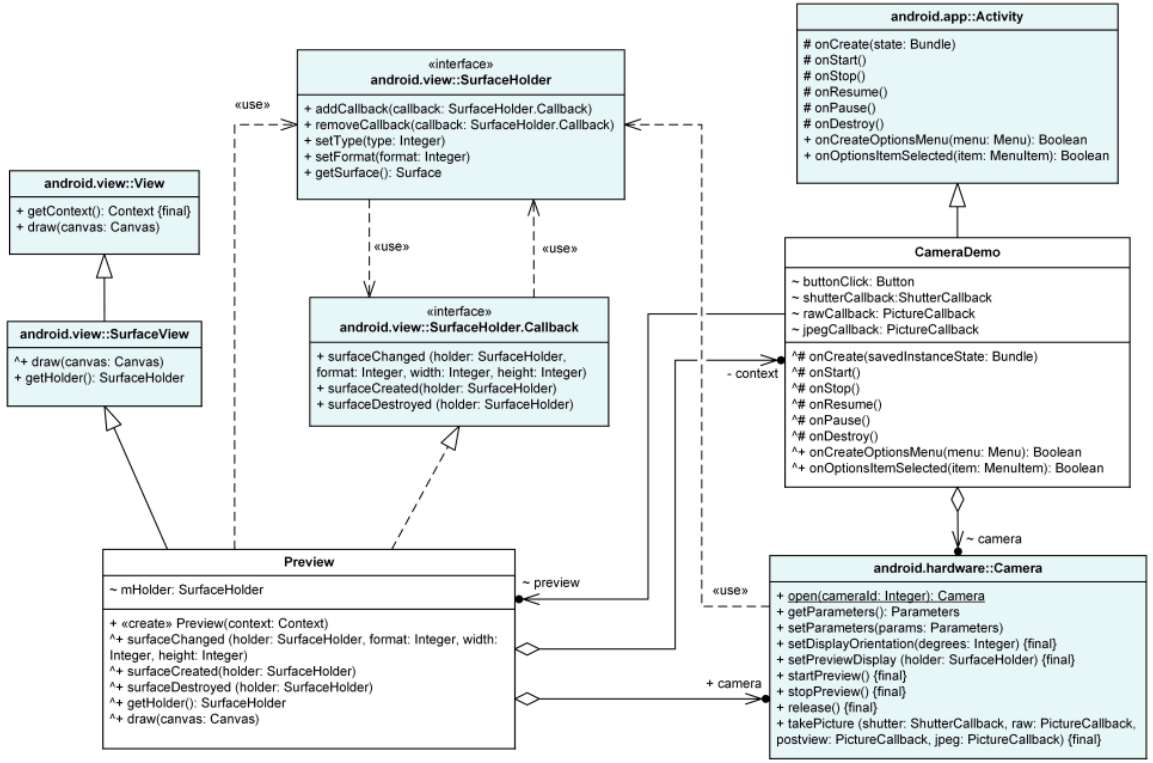
```
['roll over']
['play dead']
```

Class Diagram



Class Diagram samples





Master in Software Engineering

Hussam Hourani has over 25 years of Organizations Transformation, VROs, PMO, Large Scale and Enterprise Programs Global Delivery, Leadership, Business Development and Management Consulting. His client experience is wide ranging across many sectors but focuses on Performance Enhancement, Transformation, Enterprise Program Management, Artificial Intelligence and Data Science.