MCT-243 : COMPUTER PROGRAMMING-II
using Python 3
Object Oriented Programming

Prepared By:

Mr. Muhammad Ahsan Naeem

YouTube Playlist

https://youtube.com/playlist?list=PLWF9TXck7O_zuU2_BVUTrmGMCXYSYzjku

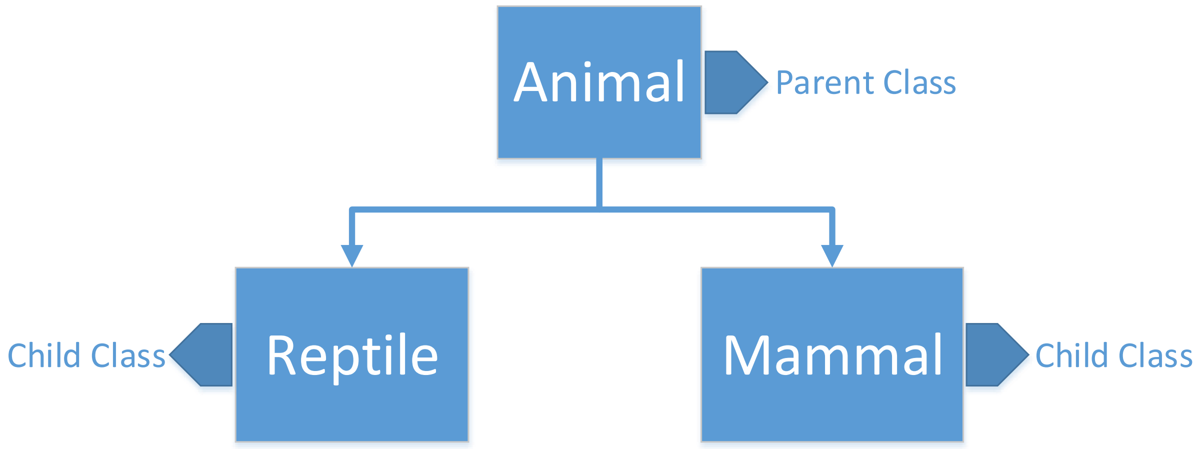
Lab 39: Object Oriented Programming

Sub-Classes or Class Inheritance

The concept of having a class as the extended version of another class is known as **Sub-Class or Class Inheritance**. The subclass inherits the features of the other class also known as the **Parent Class** or **Base Class** whereas the subclass is termed as **Child Class**. We can consider the base class as a generalized class with its own properties and the child class will be the special version of the base class having all of the properties of the base class and the other special properties of its own. A child object inherits the properties of its parent class.

When to have the subclasses:

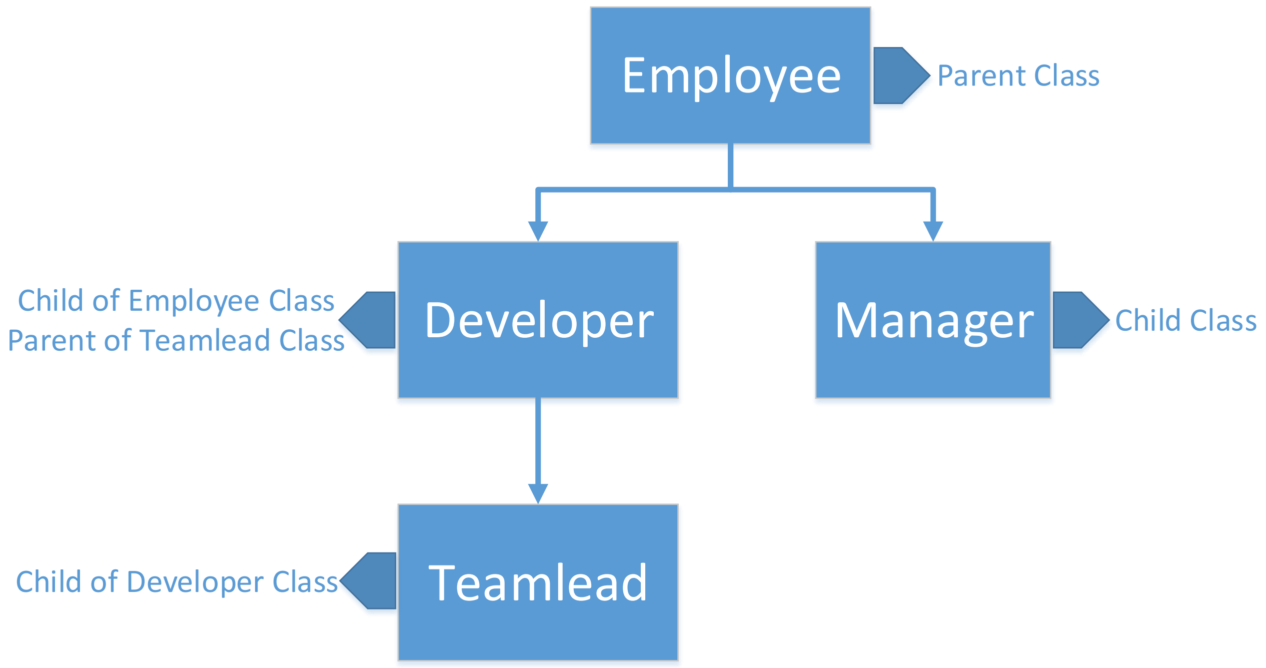
When we have a case where we want to have a class or multiple classes which are a superset of another class, we will use the subclasses. A typical way to explain the concept is through one famous example of animals. Suppose there is a class to define an animal named as **`Animal`**. The objects of this class will be the animals. Now animals can be reptiles or mammals. The two classes to define the objects of reptile and mammal named as **`Reptile`** and **`Mammal`** respectively are child classes of the **`Animal`** class. The reptiles and mammals are animals but have other properties too. The idea is shown pictorially here:



A turtle is an object of **`Reptile`** class and it is **`Animal`** too because **`Reptile`** class is the child of the **`Animal`** class. Likewise, a horse is an object of **`Mammal`** class and is an **`Animal`** too because **`Mammal`** class is also child of the **`Animal`** class. A horse is not a **`Reptile`** because there is no link between the two child classes.

Now let's consider a bit practical case. Suppose an IT-based company named as **`MCTSolutions`** has different types of employees and their records can be managed by creating class and the subclass(es). The

`Employee` will be the parent class that can have two child classes as `Developer` and `Manager`. Further, the `Developer` class can have a child class of `Teamlead`. The scenario is shown here:

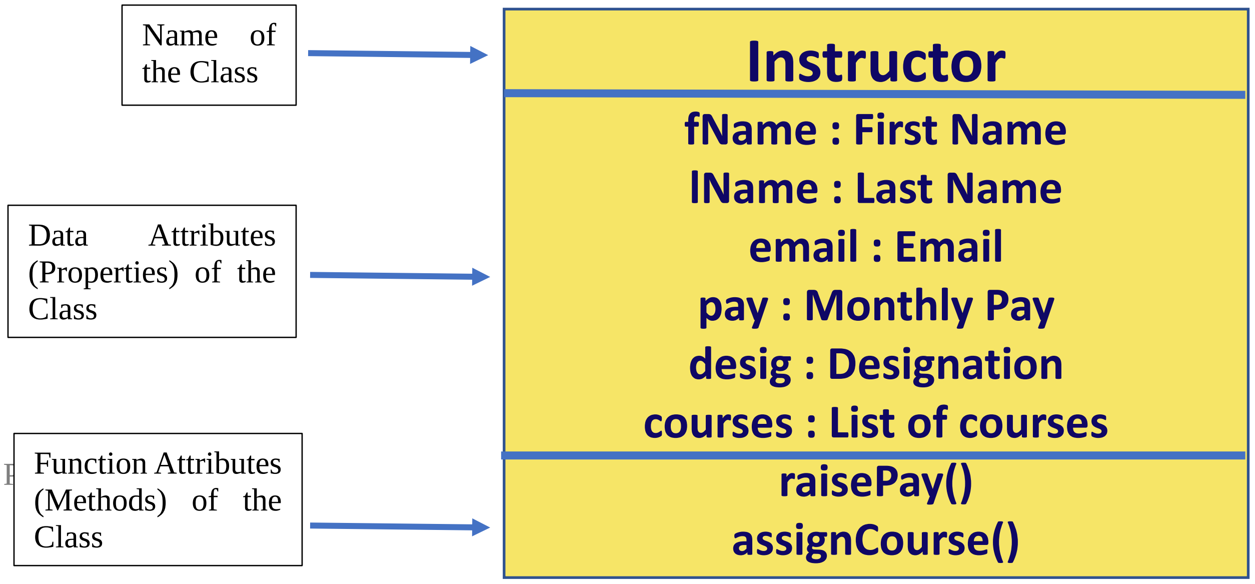


Instructor/Admin Staff Example:

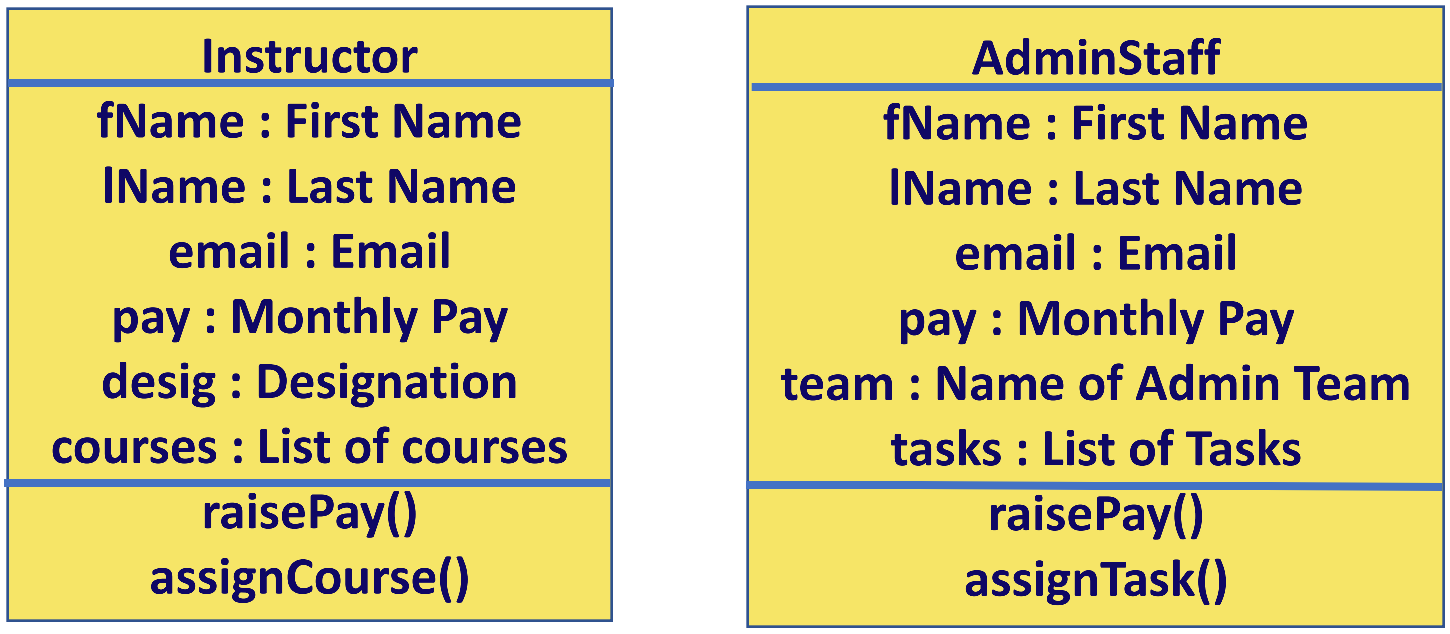
In previous lab sessions we created the classes `MechaStudent` and `Instructor` to represent student and instructor for the **Mechatronics Department**. **Instructor** is employee of the department. There can be many other employees like **Admin Staff**, **Lab Director** etc. We will create these classes linked together through class inheritance

Firstly, we can represent a class as **Unified Modeling Language (UML)** diagram. This helps the viewer see the structure of the class (name, data attributes and method attributes) without looking at the code of the class.

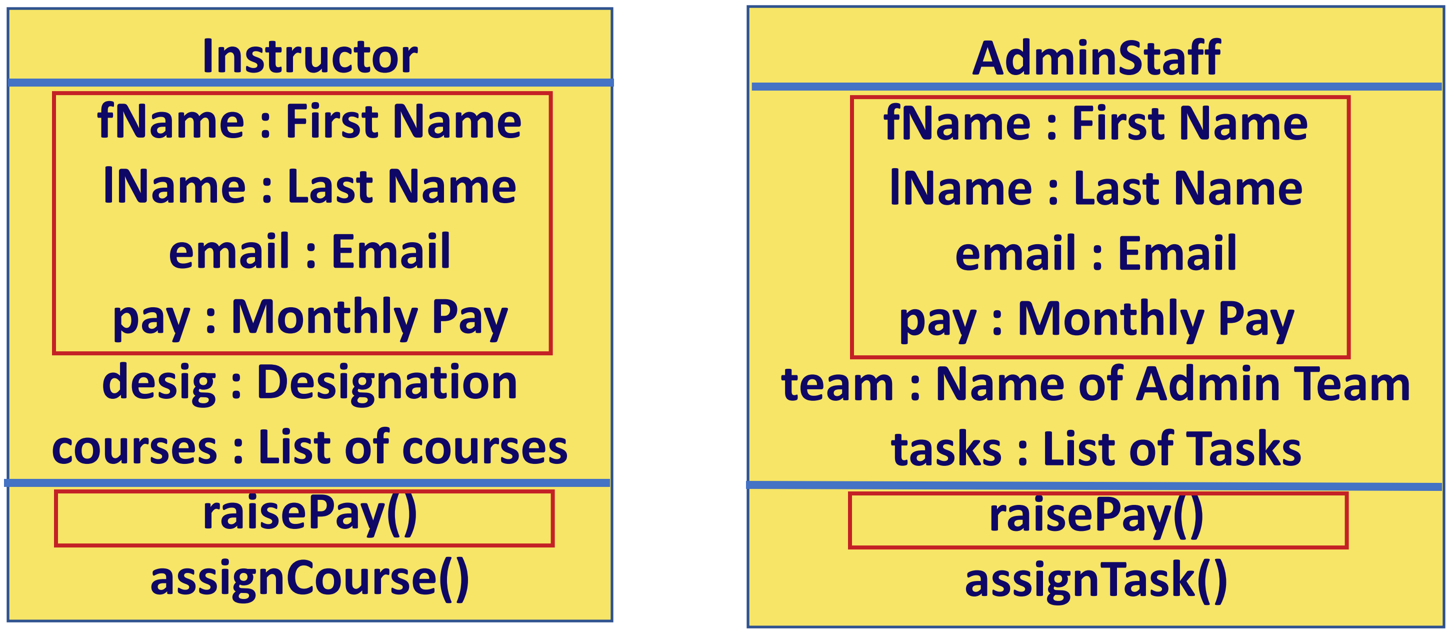
UML diagram of the `Instructor` class is shown here:



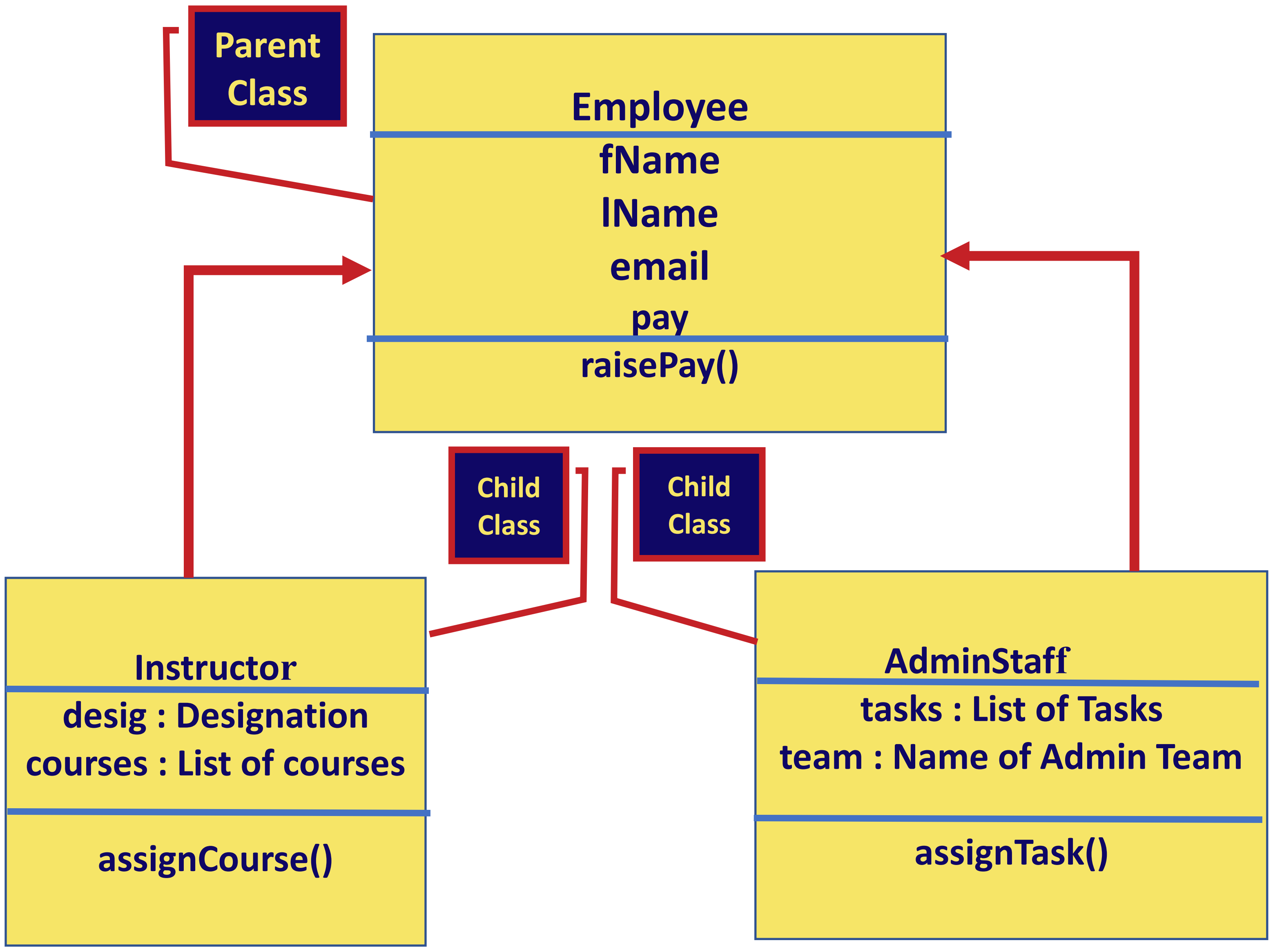
Now suppose we also want to create a class for the Admin Staff. Both Classes are shown here:



See carefully that there are many things common in the two classes as indicated below:



This is the situation where we can create a parent class comprising of these common attributes and then these two classes will be the child classes of that. With the concept of inheritance, these two classes will inherit everything (the common attributes) from the parent class and can have their own extra attributes (not common as shown in above UML diagram).



Following terms are used in **Class Inheritance**:

- Instructor **inherits** from Employee
- Instructor is a **child** of Employee
- Employee is a **parent** of Instructor
- Instructor is a **subclass** of Employee
- Employee is a **baseclass** of Instructor
- Instructor **subclasses** Employee
- Instructor **extends** Employee

- Instructor **drives** from Employee

Now let's code the complete scenario. Firstly, the **`Employee`** class is coded as:

```
class Employee:
    raise_amount=1.1 #10% annual raise in income
    total_emp=0
    def __init__(self,fName,lName,pay):
        self.fName=fName
        self.lName=lName
        self.pay=pay
        self.email=f"{self.fName.lower()}.{self.lName.lower()}@uet.ed
u.pk"
        Employee.total_emp+=1
    def raise_pay(self):
        self.pay*=self.raise_amount
    def __repr__(self):
        return f'{self.fName} {self.lName}'
```

Now we can create the **`Instructor`** and **`AdminStaff`** class as the child class of the **`Employee`** class as shown here:

```
class Instructor(Employee):
    pass
class AdminStaff(Employee):
    pass
```

Even with the blank classes, we can use all attributes of the parent class. Let's create the **`Instructor`** class object:

```
i1=Instructor('Haroon','Shahid',100_000)
print(type(i1))
print(isinstance(i1,Instructor))    #Prints True
print(isinstance(i1,Employee))      #Prints True
```

The is **"is-a"** relation between a child class and the parent class. An instructor is an employee as well. We can apply the parent class methods on the child class object as shown here:

```
i1=Instructor('Haroon','Shahid',100_000)
print(i1.pay)
i1.raise_pay()
print(i1.pay)
```

We can also create an instance of **`AdminStaff`** class. That will also be an instance of the **`Employee`** class but an **`AdminStaff`** will not be an instance of the **`Instructor`** class.

Method Over-riding:

If you see the UML diagram the `Instructor` and `AdminStaff` has properties other than specified in `Employee` class e.g. `desig` and courses for the `Instructor`. We can create the same methods in the child class with its own definition. This is the form of **Polymorphism** known as **Method Overriding**.

So, if we create the `__init__` method inside the `Instructor` class, that will be called instead of the `__init__` method of the `Employee` class. For demonstration purpose, let's create a test `__init__` method inside the instructor class as shown here:

```
class Instructor(Employee):
    def __init__(self):
        print('Inside the Instructor Class')
```

This `__init__` method does not need any input argument. So, we will create the object as:

```
i1=Instructor()
```

And you will see the output as:

```
Inside the Instructor Class
```

So, Python first looks for a method inside the class itself and if not found it will check for the method in the parent class.

Method Resolution Order (MRO):

As explained above, Python first look for a method inside the class itself and if not found it will check for the method in the parent class. There is a term known as **Method Resolution Order**, which specifies the order of the classes from which the Python will look for a method.

Let's print the help of the child class Instructor as:

```
help(Instructor)
```

The staring few lines of the output are shown here:

```
class Instructor(Employee)
 |  Method resolution order:
 |      Instructor
 |      Employee
 |      builtins.object
```

You can see the **Method Resolution Order**. Python will look for a method inside the `Instructor` class first, if not found it will go for the `Employee` class and if not found there, it will move to the `builtins` class.

There is more detail of the class in above `help` output. If you just want to see the MRO, you can also view that as:

```
print(Instructor.__mro__)
```

The output is:

```
(<class '__main__.Instructor'>, <class '__main__.Employee'>, <class
'object'>)
```

Modifying the Parent class inside the Child Class:

As we see that there is `__init__` method in the `Employee` class but we need more properties for the `Instructor` class. We can create its own `__init__` method. But notice that the properties of the `Employee` class inside the `__init__` method are needed in the `Instructor` class. There might be the getters and setters of those. So, recreating everything in the child class is not a good idea. Rather we will create the new method `__init__`, but will use the `__init__` method of the parent class for the common properties and will add more properties we need for the child class. It is shown here:

```
class Instructor(Employee):
    def __init__(self,fName,lName,pay,desig):
        Employee.__init__(self,fName,lName,pay)
        self.desig=desig
        self.courses=[]
```

Super() method:

We can use the parent class method inside the child class by using the name of the parent class. There is another way of doing the same by using the `super()` method. This method refers to the parent class and we don't need to pass the name of the parent class. We don't need to pass `self` while using the `super()` method. The above code with the `super()` method is shown here:

```
def __init__(self,fName,lName,pay,desig):
        super().__init__(fName,lName,pay)
        self.desig=desig
        self.courses=[]
```

The complete codes of the three classes are shown here:

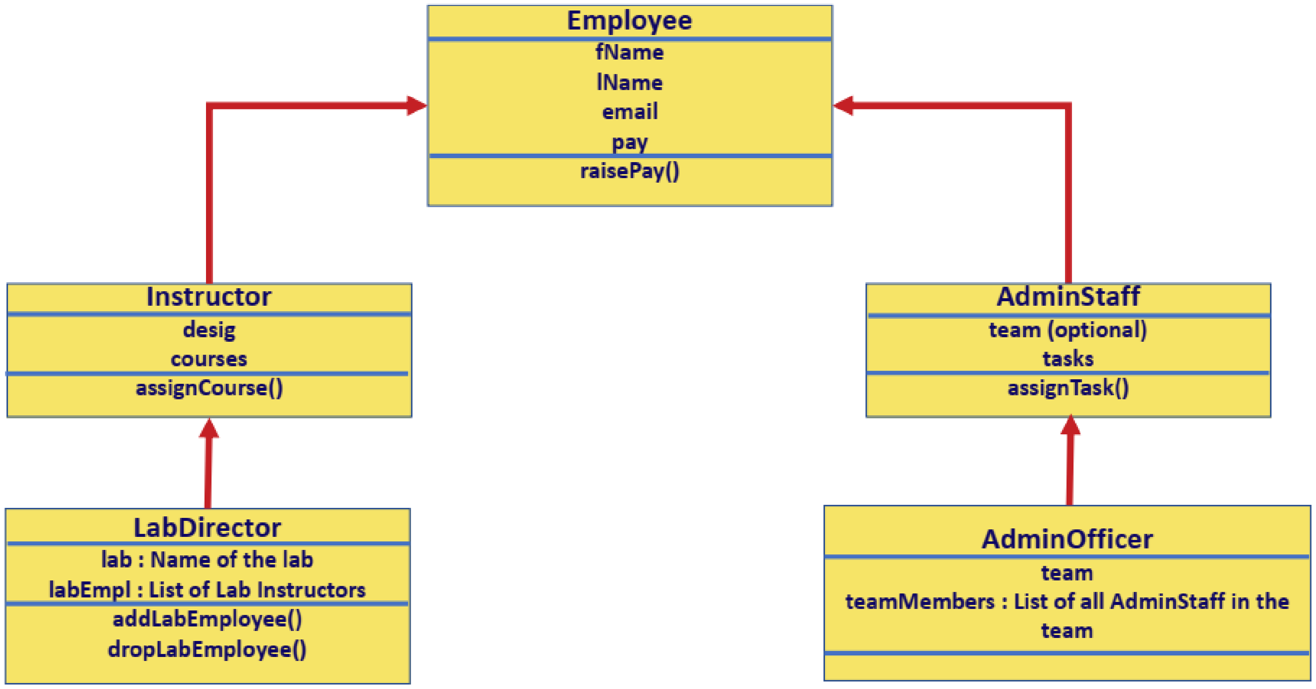
```
class Employee:
    raise_amount=1.1 #10% annual raise in income
    total_emp=0
    def __init__(self,fName,lName,pay):
        self.fName=fName
        self.lName=lName
        self.pay=pay
        self.email=f"{self.fName.lower()}.{self.lName.lower()}@uet.ed
u.pk"
        Employee.total_emp+=1
    def raise_pay(self):
        self.pay*=self.raise_amount
    def __repr__(self):
        return f'{self.fName} {self.lName}'
class Instructor(Employee):
    def __init__(self,fName,lName,pay,desig):
        # Employee.__init__(self,fName,lName,pay)
        super().__init__(fName,lName,pay)
        self.desig=desig
        self.courses=[]
    def assignCourse(self,*subj):
        self.courses=list(set(self.courses+list(subj)))

class AdminStaff(Employee):
    raise_amount=1.15
    def __init__(self,fName,lName,pay,team=None):
        super().__init__(fName,lName,pay)
        self.team=team
        self.tasks=[]
    def assignTask(self,*task):
        self.tasks=list(set(self.tasks+list(task)))

## Main Program ##
i1=Instructor('Haroon','Shahid',100000,'AP')
i1.assignCourse('CP2','CP2','Mech')
i1.raise_pay()
print(i1.pay)

as1=AdminStaff('Imran','Hanif',90000,'Exam')
as1.assignTask('exam-1','exam-2')
as1.raise_pay()
print(as1.pay)
```

One important Point in above code is explained here:

- The parent class has a method `raise_pay()` which will raise the pay by 10% using the class level variable `raise_amount` set to `1.1`.
- Both child classes can use this method and the pay will increase by 10%.
- Suppose that for the `AdminStaff` class, it is decided that their pay increase will be 15% instead of 10%. You simply need to create the class level attribute `raise_amount` and set that to `1.15`. Now when the `raise_pay()` method will be called on the `AdminStaff` object, it will use the `raise_amount` value from the `AdminStaff` class. Hence the same method will give the desired results for the `AdminSatff`.

Multi-Level Class Inheritance:

We can have the inheritance concept to multi-level. For example, an `Instructor` could also be a Lab Director. Similarly, an Admin Staff could be an Admin Officer as well. See the hierarchy below:



The Lab Director will have a list of Lab Instructors who are Instructor Class objects. If a course is assigned to a Lab Director, the same should be assigned to all Lab Instructor of that lab. The code is presented here:

```
class Employee:
    raise_amount=1.1 #10% annual raise in income
    total_emp=0
    def __init__(self,fName,lName,pay):
        self.fName=fName
        self.lName=lName
        self.pay=pay
        self.email=f"{self.fName.lower()}.{self.lName.lower()}@uet.ed
u.pk"
        Employee.total_emp+=1
    def raise_pay(self):
        self.pay*=self.raise_amount
    def __repr__(self):
        return f'{self.fName} {self.lName}'
class Instructor(Employee):
    def __init__(self,fName,lName,pay,desig):
        # Employee.__init__(self,fName,lName,pay)
        super().__init__(fName,lName,pay)
        self.desig=desig
        self.courses=[]
    def assignCourse(self,*subj):
        self.courses=list(set(self.courses+list(subj)))
class LabDirector(Instructor):
    def __init__(self,fName,lName,pay,desig,lab,*labEmp):
        super().__init__(fName,lName,pay,desig)
        self.lab=lab
        self.labEmp=list(labEmp)
    def assignCourse(self,*subj):
        super().assignCourse(*subj)
        for ins in self.labEmp:
            ins.assignCourse(*subj)
    def addLabEmployee(self,ins):
        self.labEmp.append(ins)
    def dropLabEmployee(self,ins):
        self.labEmp.remove(ins)

class AdminStaff(Employee):
    raise_amount=1.15
    allAdminStaff=[]
    def __init__(self,fName,lName,pay,team=None):
        super().__init__(fName,lName,pay)
        self.team=team
        self.tasks=[]
        AdminStaff.allAdminStaff.append(self)
    def assignTask(self,*task):
        self.tasks=list(set(self.tasks+list(task)))
class AdminOfficer(AdminStaff):
    def __init__(self,fName,lName,pay,team):
        super().__init__(fName,lName,pay,team)
    @property
    def teamMembers(self):
        return list(filter(lambda s:
s.team==self.team,AdminStaff.allAdminStaff))
```

Example Main Program use is here:

```
i1=Instructor('Ahsan','Naeem',100000,'AP')
i2=Instructor('Rzi','Abbas',95000,'Lect')
i3=Instructor('Misbah','Rehman',90000,'Lect')
i4=Instructor('Shujat','Ali',85000,'TF')

labDir1=LabDirector('Mohsin','Rizwan',120000,'AP','Embedded
Systems',i1,i2)
print(labDir1.labEmp)
labDir1.assignCourse('Robotics')
print(labDir1.courses)
print(i1.courses)
print(i2.courses)

as1=AdminStaff('Imran','Hanif',90000,'Exam')
as2=AdminStaff('Saqib','Majeed',105000,'Dues')
as3=AdminStaff('Inam','Haider',110000,'Exam')

off1=AdminOfficer('Naveed','Aslam',120000,'Exam')
print(off1.teamMembers)
```