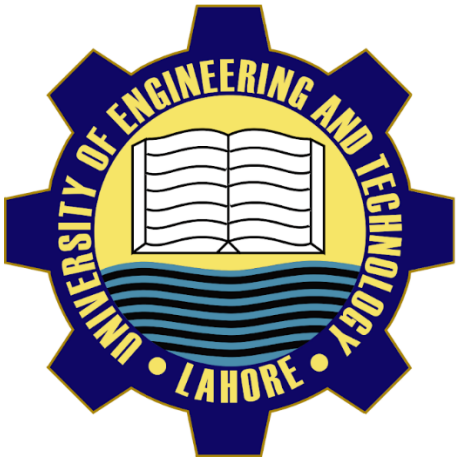
# *MCT-243 : COMPUTER PROGRAMMING-II*
# *using Python 3*
# *Object Oriented Programming*


***Prepared By:***
***Mr. Muhammad Ahsan Naeem***

**YouTube Playlist**
https://youtube.com/playlist?list=PLWF9TXck7O_zuU2_BVUTrmGMCXYSYzjku

# Lab 36: Object Oriented Programming

# Data Encapsulation

# Getters and Setters Methods

## Data Encapsulation:

**Encapsulation** in object-oriented programming is a mechanism of restricting direct access to some components of an object, so users cannot access state of the object and alter that.

In class programming there can be three types of attributes as described below:
**Public**→ These class attributes are accessible everywhere i.e. within and outside the class. The way we have used the attributes so far, they all are public.
**Private**→ These are the attributes which are accessible only inside the class and not outside.
**Protected**→ These are the attributes which are accessible within the class and other classes inherited from this class.

Unlike many other Programming Languages, all attributes in Python are Public. Encapsulation in Python is used as convention rather than enforcement. Before discussing the private attributes in Python, let's first see why there is need for a private attribute.

A six-sided rolling die is used in many games. Let's create a class for a `Die` and add basic functionality as shown here:

```
import random
class Die:
    def __init__(self):
        self.sideUp=1
    def roll(self):
        self.sideUp=random.randint(1,6)
    def __str__(self):
        return str(self.sideUp)
```

The following program uses this class to roll a die and you will see a random output every time you run this program:

```
myDie=Die()
myDie.roll()
print(myDie)
```

Getting a `6` on a roll generally has the maximum reward in most of the games. If we change the main program this way:

```
myDie=Die()
```

```
myDie.roll()
myDie.sideUp=6
print(myDie)
```

Now you will get a **6** every time you run the program. This is basically the misuse of the class by the programmer using it. The attribute **sideUp** is a public attribute and accessible everywhere. If we can make it private so that it cannot be accessed and changed outside the class, the above issue can be resolved.

In Python, there is no built-in concept of private or protected attributes. All attributes in Python are public. However, to handle situations such as above, there are two conventions used in Python and those are detailed here:

**i. Single Leading Underscore:**

As a convention, a single leading underscore before the attribute name is a way to tell other programmers that this attribute is meant to behave like a private attribute. It doesn't change the behavior of the program but a simple hint to the programmer that the attribute is intended for internal use. It is just a convention, and it cannot force the programmer for not changing the attribute.

**ii. Double Leading Underscores:**

This way of defining an attribute is more "private" compared to the first method though not purely a private type. See the **Die** class code with this approach:

```
import random
class Die:
    def __init__(self):
        self.__sideUp=1
    def roll(self):
        self.__sideUp=random.randint(1,6)
    def __str__(self):
        return str(self.__sideUp)
```

Now in Main program if we try to access this attribute as:

```
myDie=Die()
print(myDie.__sideUp)
```

This will generate following error:

```
AttributeError: 'Die' object has no attribute '__sideUp'
```

This behavior is exactly what was needed. The inside story of double underscore is quite interesting. Python interpreter basically rewrite the attribute name starting with double underscore and this process is known as **Name Mangling**. Hence outside the class the attribute name is different than what is inside it. You can see this from the above error also stating that **__sideUp** is not the attribute of **Die** class because the attribute name is different outside the class. For example, if we run the following Main program for the above class with **__sideUp** attribute:

```
myDie=Die()
myDie.roll()
myDie.__sideUp=6
print(myDie)
```

What do you think will happen on third line? An error as before? Here we are assigning a value to `__sideUp` thinking that it will update the value of `myDie` attribute inside the class. `__sideUp` inside the class has different name outside. For any object of a class, we can assign a new attribute outside the class that is not defined inside the class. So, in line `3` a new attribute named as `__sideUp` is set for the object with value `6` and it is NOT the `__sideUp` inside the class. Hence, if you run the code, you will not get an error and the output will be random as needed.

### How Python does the Name Mangling?

The rule of name mangling is not random, and we can access the new "hidden" name and change that to get "desired" behavior. Let's print the `__dict__` attribute of the `Die` class object as:

```
myDie=Die()
print(myDie.__dict__)
```

This will be the output:

```
{'_Die__sideUp': 1}
```

It means that outside the class the name of the attribute `__sideUp` is actually `_Die__sideUp`. And we still can access and change it. For example, the code below will get the value `6` every time:

```
myDie=Die()
myDie.roll()
myDie._Die__sideUp=6
print(myDie)
```

### Concluding Remarks on Data Encapsulation in Python:

In Python there is no formal mechanism of Encapsulation. But there are two conventions. The first (single leading underscore) is a soft way just to indicate to the user of the class that the attribute is meant for the internal use. The second (double leading underscore) is a bit hard way to indicate to the user not to access the attribute and he will not be able to access that with the name used inside the class, however, he still can access that using the underscore class-name before the attribute name.

# Getter and Setter methods:

We have been using the dot notation to get and set the attributes of the class objects. Python provides with built-in function which we can use for getting and setting the attributes outside the class and these are **getattr** and **setattr** functions. One example use of these functions for the **MechaStudent** class is given here:

```
from Student import MechaStudent
std1=MechaStudent('Anwar','Ali','MCT-UET-01')
std2=MechaStudent('Akbar','Khan','MCT-UET-02')

print(getattr(std1,'fName'))          # Will print Anwar
setattr(std2,'reg','MCT-UET-05')
print(getattr(std2,'reg'))            # Will print MCT-UET-05
```

Input arguments of **getattr** are the object and the name of the attribute as string while for **setattr** there is third input argument as the value we want to set for the given attribute.

### Why to use getattr and setattr instead of dot notation?

Although these are not used quite often but in a couple of scenarios these can be helpful. Suppose that the name of the attribute we want to get, or set is stored in a string variable then we cannot use that variable with the dot notation as shown here:

```
from Student import MechaStudent
std1=MechaStudent('Anwar','Ali','MCT-UET-01')

att='fName'
print(std1.att)      # Will generate an error
```

But we can do this using getattr and setattr as shown here:

```
from Student import MechaStudent
std1=MechaStudent('Anwar','Ali','MCT-UET-01')

att='fName'
print(getattr(std1,att))          # Will print Anwar
```

Secondly, in case the attribute does not exists, the dot notation format will generate the **AttributeError** but the **getattr** allows to pass third input argument as fall back value in case the attribute does not exists, this third argument will be returned. However, this will not be added as new attribute. See the code here:

```
from Student import MechaStudent
std1=MechaStudent('Anwar','Ali','MCT-UET-01')

print(getattr(std1,'uni','UET'))     # Will print UET
```

There is also a built-in function **hasattr** which we can use to verify if the object or the class contains some attribute. This function returns **True** if the object has that attribute and **False** otherwise. So, we

can also use this function to get the feature of **getattr** when the provided attribute is not set for the object. It is shown here:

```
from Student import MechaStudent
std1=MechaStudent('Anwar','Ali','MCT-UET-01')

if(hasattr(std1,'uni')):
    print(std1.uni)
else:
    print('UET')
```

# Customized Getter and Setter methods:

We can have customized getters and setters methods for the attributes of the class. There is a common practice of creating our own getter and setter for each of the attributes. **Getters** are also known as **Accessor** methods while the **Setters** are also known as **Mutator** methods. We will see why we need these getters and setters. There are three approaches used for these getters and setters which are explained in least common to most common order of practice:

**i. First Approach for Getters and Setters:**

If there are three attributes of an object, there will be three getter (one for each attribute) and likewise three setter methods. For example, if the three attribute names are att1, att2, att3 then by convention the names of three getters and setters should be:
get_att1, get_att2, get_att3, set_att1, set_att2, set_att3.

For a simple MechaStudent class, these methods are shown below and for simplicity just a few attributes are considered:

```
class MechaStudent:
    def __init__(self,fName,lName,reg):
        self.__fName=fName
        self.__lName=lName
        self.__reg=reg
    def get_fName(self):
        return self.__fName
    def set_fName(self,newname):
        self.__fName=newname
    def get_lName(self):
        return self.__lName
    def set_lName(self,newname):
        self.__lName=newname
    def get_reg(self):
        return self.__reg
    def set_reg(self,newreg):
        self.__reg=newreg

### Main Program ###
std1=MechaStudent('Anwar','Ali','MCT-UET-01')
```

```
print(std1.get_fName())
std1.set_fName('Akbar')
print(std1.get_fName())
```

You can see that Data Encapsulation technique is used such that the instance attributes cannot be accessed outside the class. However, we can access and change those using the getter and setter methods.

### What is advantage of these getter and setter methods over the simple dot notation?

We can set the validation rules within the setter methods so that the value is set only if it valid. In case of first and last names, it can be simply a condition that it is a string type, contains just the alphabets and the length is at least 2. With this approach the code is shown here:

```
class MechaStudent:
    def __init__(self,fName,lName,reg):
        self.set_fName(fName)
        self.set_lName(lName)
        self.set_reg(reg)

    def get_fName(self):
        return self.__fName
    def set_fName(self,newname):
        if(isinstance(newname,str) and len(newname)>=2 and newname.is
alpha()):
            self.__fName=newname
        else:
            raise ValueError('Name should contain alphabet only and a
t least 2 of those!')
    def get_lName(self):
        return self.__lName
    def set_lName(self,newname):
        if(isinstance(newname,str) and len(newname)>=2 and newname.is
alpha()):
            self.__lName=newname
        else:
            raise ValueError('Name should contain alphabet only and a
t least 2 of those!')
    def get_reg(self):
        return self.__reg
    def set_reg(self,newreg):
        if(isinstance(newreg,str) and str(newreg).startswith('MCT-
UET-')):
            self.__reg=newreg
        else:
            raise ValueError('Reg must start as MCT-UET-')

### Main Program ###
```

```
std1=MechaStudent('Anwar','Ali','MCT-01') # Will generate error!
```

Note that inside **__init__** method too, we are calling the setter method so that validity is checked when an instance is created. Finally before moving ahead, you must see carefully that the validation for **__fName** and **__lName** name is exactly same. As a good programming skill, we must always re-use the code instead of having a copy of it. Here we can create one function that will check the validity condition and that will be called inside setter of both attributes. This function to check the validity needs one input argument and will return **True** if it is a valid name. Secondly, this function will not set these attributes for the instance rather that will be done inside the setter methods. Hence, this function doesn't need the instance and nor the class and hence should be a **static** method. The class code with this approach is shown here:

```
class MechaStudent:
    def __init__(self,fName,lName,reg):
        self.set_fName(fName)
        self.set_lName(lName)
        self.set_reg(reg)
    def get_fName(self):
        return self.__fName
    def set_fName(self,newname):
        if(MechaStudent.validName(newname)):
            self.__fName=newname
    def get_lName(self):
        return self.__lName
    def set_lName(self,newname):
        if(MechaStudent.validName(newname)):
            self.__lName=newname
    def get_reg(self):
        return self.__reg
    def set_reg(self,newreg):
        if(isinstance(newreg,str) and str(newreg).startswith('MCT-
UET-')):
            self.__reg=newreg
        else:
            raise ValueError('Reg must start as MCT-UET-')
    @staticmethod
    def validName(name):
        if(isinstance(name,str) and len(name)>=2 and name.isalpha()):
            return True
        else:
            raise ValueError('Name should contain alphabet only and a
t least 2 of those!')
```

### ii. Second Approach for Getters and Setters:

We saw **property()** function through which we can decorate a method and it will become a property i.e. data attribute. In another way, we can use this **property()** function on getter and setter and it will make it possible to call these via the dot notation as we had been doing previously. Hence, outside the

class we will be using the dot notation as earlier, but the getter and setter will get called giving us the benefits of validity and preserving the compatibility of the code outside the class.

```
class MechaStudent:
    def __init__(self,fName,lName,reg):
        self.fName=fName
        self.lName=lName
        self.reg=reg
    def __get_fName(self):
        return self.__fName
    def __set_fName(self,newname):
        if(MechaStudent.validName(newname)):
            self.__fName=newname
    fName=property(__get_fName,__set_fName)
    def __get_lName(self):
        return self.__lName
    def __set_lName(self,newname):
        if(MechaStudent.validName(newname)):
            self.__lName=newname
    lName=property(__get_lName,__set_lName)
    def __get_fullName(self):
        return f'{self.fName} {self.lName}'
    def __set_fullName(self,newname):
        f,l=newname.split(' ')
        self.fName=f
        self.lName=l
    fullName=property(__get_fullName,__set_fullName)
    def __get_reg(self):
        return self.__reg
    def __set_reg(self,newreg):
        if(isinstance(newreg,str) and str(newreg).startswith('MCT-
UET-')):
            self.__reg=newreg
        else:
            raise ValueError('Reg must start as MCT-UET-')
    reg=property(__get_reg,__set_reg)
    @staticmethod
    def validName(name):
        if(isinstance(name,str) and len(name)>=2 and name.isalpha()):
            return True
        else:
            raise ValueError('Name should contain alphabet only and a
t least 2 of those!')

### Main Program ###
std1=MechaStudent('Anwar','Ali','MCT-UET-01')
print(std1.fName)
```

```
print(std1.__dict__)

print(std1.fullName)
std1.fullName='Hasan Sarwar'
print(std1.fName)
print(std1.lName)
```

### iii. Third Approach for Getters and Setters:

The third way we can have getter and setter for the instance attributes can be achieved using the **property** decorator. We used this decorator at very begging of our discussion on Object-Oriented programming. Here is one starting example where we used this decorator:

```
@property
    def fullName(self):
        return f'{self.fName} {self.lName}'
```

Basically, this is a getter method for the attribute **fullName**. Whenever we will try to access this attribute i.e. **fullName** for any instance of the class, this method gets called. We studied earlier that it was important to have it as getter method so that if **fName** or **lName** is changed, **fullName** will get the latest updated values. Using the same property decorator, we can have setter method as well with the format as **@fName.setter** for **fName** and likewise for others. The code is shown here:

```
class MechaStudent:
    def __init__(self,fName,lName,reg):
        self.fName=fName
        self.lName=lName
        self.reg=reg

    @property
    def fName(self):
        return self.__fName
    @fName.setter
    def fName(self,newname):
        if(MechaStudent.validName(newname)):
            self.__fName=newname

    @property
    def lName(self):
        return self.__lName
    @lName.setter
    def lName(self,newname):
        if(MechaStudent.validName(newname)):
            self.__lName=newname

    @property
    def reg(self):
        return self.__reg
```

```
    @reg.setter
    def reg(self,newreg):
        if(isinstance(newreg,str) and str(newreg).startswith('MCT-
UET-')):
            self.__reg=newreg
        else:
            raise ValueError('Reg must start as MCT-UET-')

    @property
    def fullName(self):
        return f'{self.fName} {self.lName}'
    @fullName.setter
    def fullName(self,newname):
        f,l=newname.split(' ')
        self.fName=f
        self.lName=l

    @staticmethod
    def validName(name):
        if(isinstance(name,str) and len(name)>=2 and name.isalpha()):
            return True
        else:
            raise ValueError('Name should contain alphabet only and a
t least 2 of those!')

### Main Program ###
std1=MechaStudent('Anwar','Ali','MCT-UET-01')
print(std1.fName)
print(std1.__dict__)

print(std1.fullName)
std1.fullName='Hasan Sarwar'
print(std1.fName)
print(std1.lName)
```

It is important to note that we should have the getter method if we want to have setter method. A getter method can be added without setter method but not the reverse.

# Deleter Method:

With getter and setter there is another method known as the **deleter** method which is used to delete the attribute. The deleter method for `fullName` attribute is shown here:

```
    @property
    def fullName(self):
        return f'{self.fName} {self.lName}'
    @fullName.setter
    def fullName(self,newname):
```

```
        f,l=newname.split(' ')
        self.fName=f
        self.lName=l
    @fullName.deleter
    def fullName(self):
        self.fName=None
        self.lName=None
```

If we run the following main program with deleter method added for **fullName**:

```
std1=MechaStudent('Anwar','Ali','MCT-UET-01')

del std1.fullName # Will generate an error
print(std1.fName)
print(std1.lName)
```

This will generate an error. Because the code written inside the deleter method is assigning **None** to **fName** and **lName** that in return calls the setter method of those attributes where we incorporated the validity check on the name being set that it should be alphabet. **None** is not alphabet and hence it will generate the error:

```
Name should contain alphabet only and at least 2 of those!
```

For understanding let's add some example names within the deleter method as:

```
    @property
    def fullName(self):
        return f'{self.fName} {self.lName}'
    @fullName.setter
    def fullName(self,newname):
        f,l=newname.split(' ')
        self.fName=f
        self.lName=l
    @fullName.deleter
    def fullName(self):
        self.fullName='First Second'
```

And now run the following main program:

```
std1=Student('Anwar','Ali','MCT-UET-01')

del std1.fullName
print(std1.fName)    # Will print First
print(std1.lName)    # Will print Second
```

You can see that with the power of setter for **fullName**, the attributes **fName** and **lName** have been set to values assigned in deleter method.

# Final Code of MechaStudent Class:

After incorporating the concepts we studied in this Lab Session, here is the complete code of the **`MechaStudent`** class:

```
class MechaStudent:
    'This class defines a Student for Mechatronics Department'
    _department='Mechatronics'
    _offSubjects=['Mech','LA','ES','CP-II','MOM','Proj']
    _allStudents=[]
    def __init__(self,fName,lName,reg):
        self.fName=fName
        self.lName=lName
        self.reg=reg
        self.email=f'{self.reg.lower()}@uet.edu.pk'
        self._courses=['Proj']
        self._groupMember=None
        self.fullName=f'{self.fName} {self.lName}'
        MechaStudent._allStudents.append(self)

    ## Getters and Setters
    @property
    def fullName(self):
        return f'{self.fName} {self.lName}'
    @fullName.setter
    def fullName(self,newname):
        f,l=newname.split(' ')
        self.fName=f
        self.lName=l
        self._fullName=newname
    @property
    def fName(self):
        return self._fName
    @fName.setter
    def fName(self,newname):
        if(MechaStudent.validName(newname)):
            self._fName=newname
        else:
            raise ValueError('Name should contain alphabet only and
at least 2 of those!')
    @property
    def lName(self):
        return self._lName
    @lName.setter
    def lName(self,newname):
        if(MechaStudent.validName(newname)):
```

```
            self._lName=newname
        else:
            raise ValueError('Name should contain alphabet only and
at least 2 of those!')
    @property
    def reg(self):
        return self._reg
    @reg.setter
    def reg(self,newreg):
        if(isinstance(newreg,str) and str(newreg).startswith('MCT-
UET-')):
            self._reg=newreg
        else:
            raise ValueError('Reg must start as MCT-UET-')

    ## Static Methods ##
    @staticmethod
    def validName(name):
        if(isinstance(name,str) and len(name)>=2 and name.isalpha()):
            return True
        else:
            return False
    ## Instance Methods ##
    def registerSubject(self,*sub):
        for s in sub:
            if s not in MechaStudent._offSubjects:
                raise ValueError(f'{s} is not offered!')
            if s in MechaStudent._offSubjects and s not in
self._courses:
                self._courses.append(s)
    def setGroupMember(self,other):
        if(self._groupMember!=None):
            raise ValueError(f'{self} already has {self._groupMember}
as group member')
        elif(other._groupMember!=None):
            raise ValueError(f'{other} already has
{other._groupMember} as group member')
        else:
            self._groupMember=other
            other._groupMember=self
    def dropGroupMember(self,other):
        if(self._groupMember==None and other._groupMember==None):
            return
        elif(self._groupMember!=other):
            raise ValueError(f'{self} is not group member of
{other}.')
        else:
```

```
            self._groupMember=None
            other._groupMember=None
    ## Class Methods ##
    @classmethod
    def notRegSub(cls):
        a=set()
        for std in cls._allStudents:
            s=set(std._courses)
            a.update(s)
        return list(set(cls._offSubjects).difference(a))
    @classmethod
    def withoutGroupMembers(cls):
        return list(filter(lambda s:
s._groupMember==None,cls._allStudents))

    ## Magic Methods ##
    def __repr__(self):
        return f'{self.lName}-{self.reg[-2:]}'
```

# References:


**[1]** https://youtu.be/WZbgr14jyTk
**[2]** https://youtu.be/ml7pl2a-wK0