Programming Lab - II

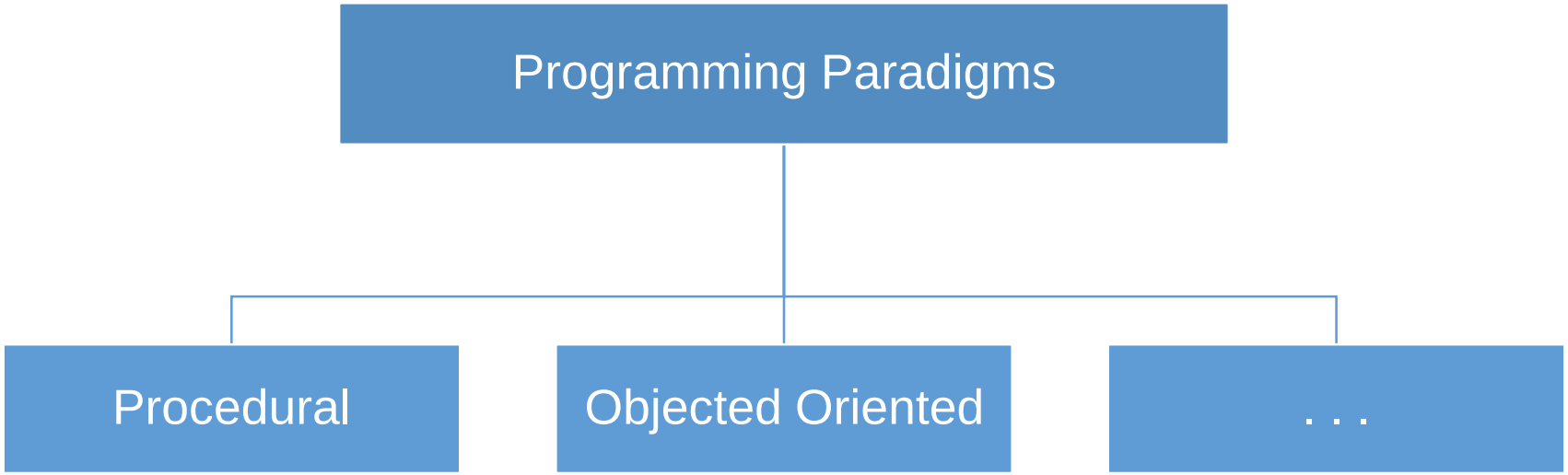
Object Oriented Programming (OOP)

Kaushik Pal

7th Mar, 2017

National Institute Of Technology, Rourkela

Programming Paradigms



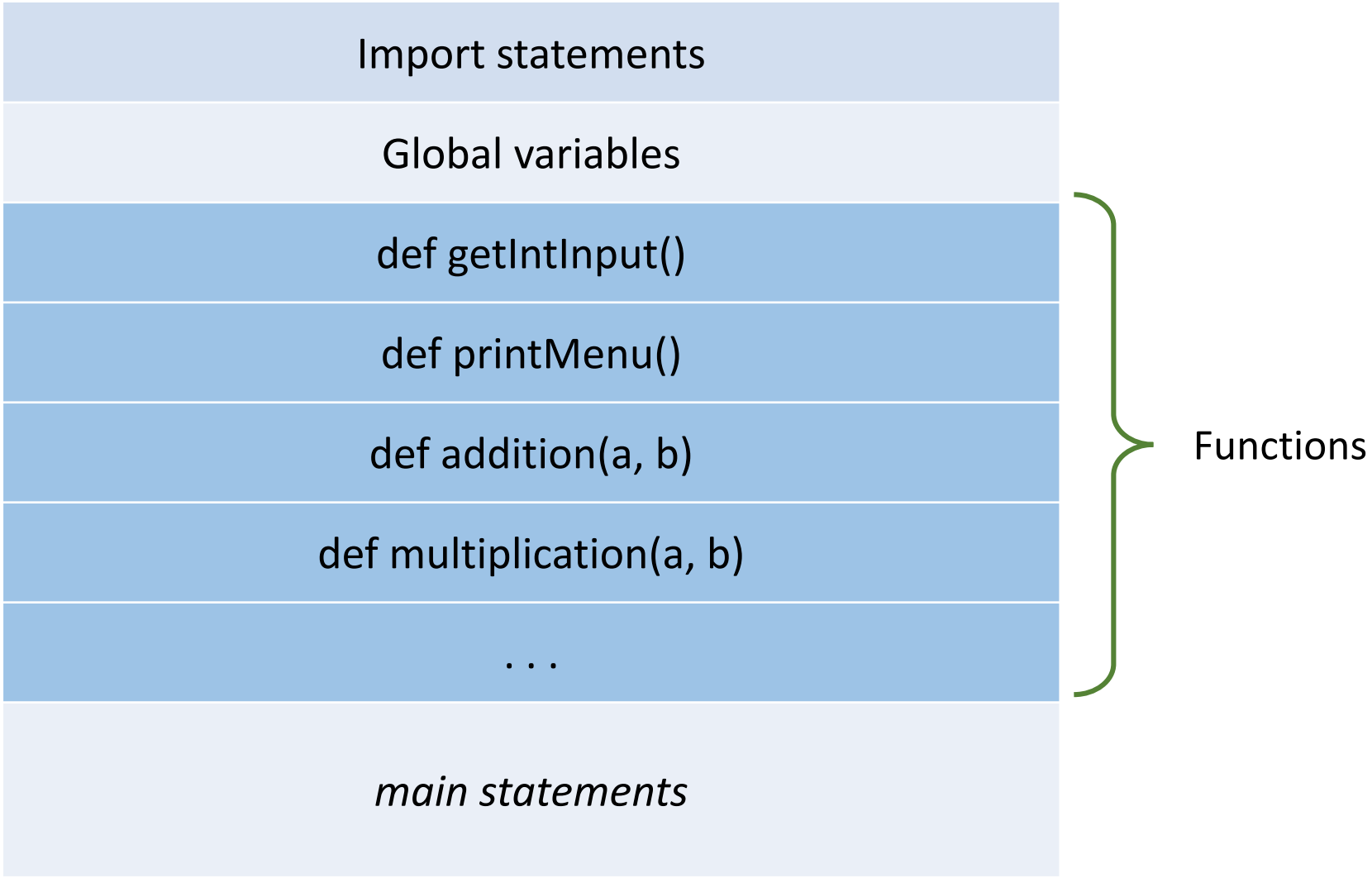
Programming Paradigms - Procedural

Procedural Programming

It is based on the concept of using procedures(a.k.a. *functions*).

- Procedure is a sequence of commands to be executed.
- Any procedure can be called from any point within the general program, including other procedures or even itself.
- Data/variable scope:
  - *Global*
  - *Local*

Programming Paradigms - Procedural



Programming Paradigms - OOP

Object-Oriented Programming

It is based on the concept of using ***classes*** and its ***objects***.

- Inspired from the real-world.
- **Class**: It is a blueprint of the properties & behaviour. It is a ***data-type***.
- **Object**: It is an instance of a particular class.
- ***Variable*** and ***function*** scope:
  - *Public*
  - *Private*
  - *etc.*

OOP - Introduction




OOP - Introduction

Name	Pluto	Scooby Doo	Droopy	Spike
Skin Color	Yellow	brown	white	grey
Ear length	long	short	long	short
Is spotted	no	yes	no	no

Attributes

Values

OOP - Introduction

class: ***Dog***




OOP - Introduction

class: ***Dog***




- Name
- Skin color
- Ear length
- Is spotted

Attributes

OOP - Introduction

class: ***Dog***




- Name
- Skin color
- Ear length
- Is spotted

Attributes

Behaviour or Methods

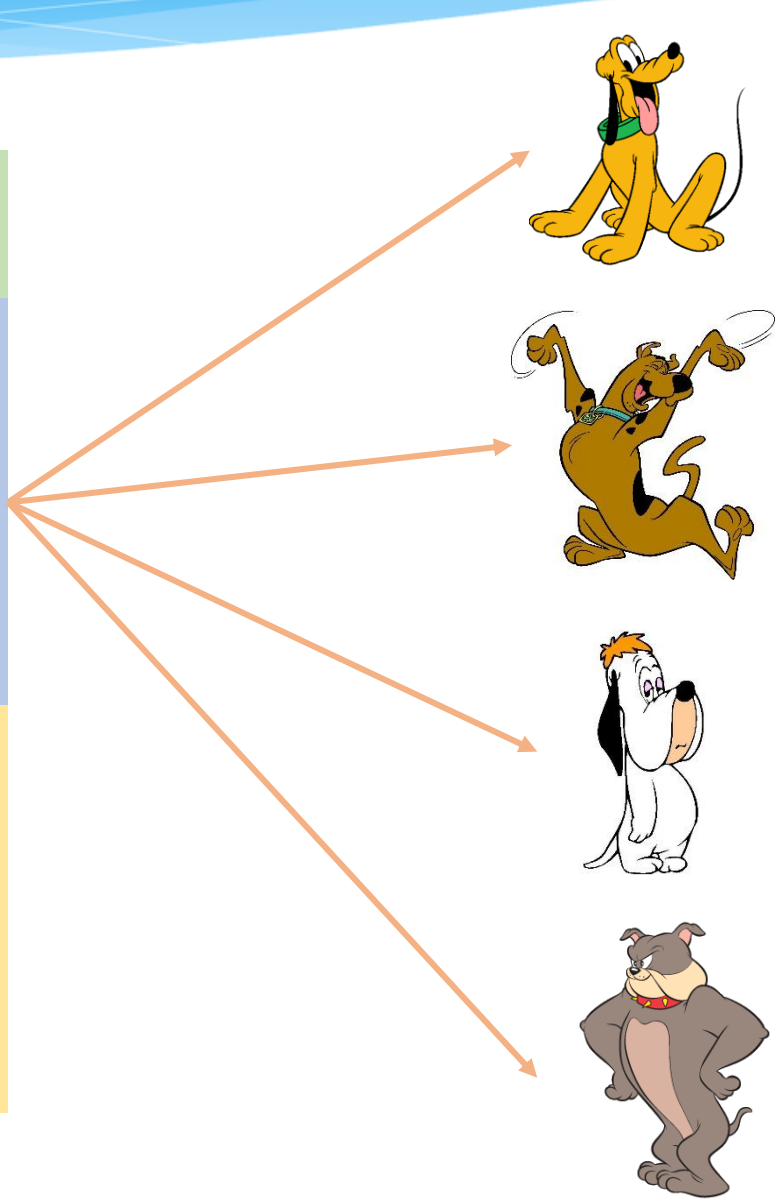
- Walk
- Eat
- Sleep
- Chase cat

OOP - Introduction

class: ***Dog***

- Name
- Skin color
- Ear length
- Is spotted

- *Walk*
- *Eat*
- *Sleep*
- *Chase cat*



Objects *of the* ***class*** *Dog*

OOP - Introduction

class: ***Dog***

- Name
- Skin color
- Ear length
- Is spotted

- *Walk*
- *Eat*
- *Sleep*
- *Chase cat*

Name	Pluto
Skin Color	Yellow
Ear length	long
Is spotted	no

Name	Scooby Doo
Skin Color	brown
Ear length	short
Is spotted	yes

Name	Droopy
Skin Color	white
Ear length	long
Is spotted	no

Name	Spike
Skin Color	grey
Ear length	short
Is spotted	no

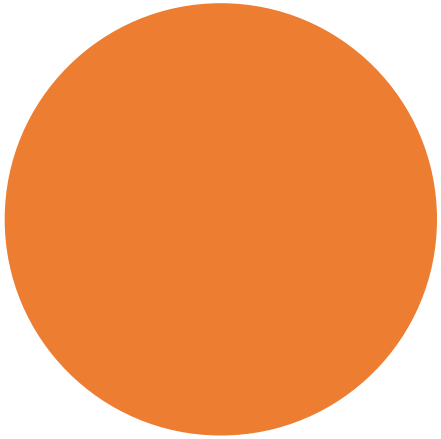
Traditional Procedural language

class: ***Dog***

- Name
- Skin color
- Ear length
- Is spotted

- *Walk*
- *Eat*
- *Sleep*
- *Chase cat*

New



Name	-
Skin Color	-
Ear length	-
Is spotted	-

Traditional Procedural language

class: ***Dog***

- Name
- Skin color
- Ear length
- Is spotted

- *Walk*
- *Eat*
- *Sleep*
- *Chase cat*

New

Name	Mr. Vodafone
Skin Color	cream
Ear length	short
Is spotted	no

Syntax

```
class < class name >:

    def __init__( self, a1, a2, a3 ):
        self.attr1 = a1
        self.attr2 = a2
        self.attr3 = a3

    def < method name >( self, p1 ):
        self.attr1 = p1
```

*Declare a **class***

```
#-----------------------------------------------------

obj1 = <class name>( param1, param2, param3 )
obj1.<method name>( 24 )
```

*Create an **object** of a **class***

*Use the **object***

Syntax

```
class < class name >:

    def __init__( self, a1, a2, a3 ):
        self.attr1 = a1
        self.attr2 = a2
        self.attr3 = a3

    def < method name >( self, p1 ):
        self.attr1 = p1
#-------------------------------------------------

obj1 = <class name>( param1, param2, param3 )
obj1.<method name>( 24 )
```

Will run only once
(during object creation)

Syntax

```
class Dog:

    def __init__( self, n, sc, el, spot ):
        self.name = n
        self.skinColor = sc
        self.earLength = el
        self.isSpotted = spot
        print 'Dog created !!'

    def walk( self ):
        print "{0} is walking!".format(self.name)

    def eat( self ):
        print "{0} is eating!".format(self.name)
```

Syntax

```
doggie1 = Dog( 'Scooby Doo', 'brown', 'short', True )
doggie1.walk()

doggie2 = Dog( 'Droopy', 'white', 'long', False )
doggie2.walk()
```

Output:

```
Dog created !!
Scooby Doo is walking!

Dog created !!
Droopy is walking!
```

Assignment - I

WAP to implement the ***Dog class*** as shown below.

Create ***3 Dog objects*** and show the output of all the methods.

class: ***Dog***

- Name
- Skin color
- Ear length
- Is spotted

- *Walk*
- *Sleep*
- *printInfo*

Output of sleep:

```
Scooby Doo is sleeping...
Zzzzz...
```

Output of printInfo:

```
Dog info:
    Name: Scooby Doo
    Skin color: brown
    Ear length: short
    Is spotted: True
```

OOP Principles

There are ***3 principles*** that provide mechanisms to help implement the object oriented model:

I. Encapsulation

II. Inheritance

III. Polymorphism

OOP Principles - Encapsulation

I. Encapsulation

The encapsulation mechanism:

- *Binds together code and the data it manipulates.*
- *Keeps both (data and code) safe from outside interference & misuse.*

*In other words, it acts like a **protective wrapper** that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.*

OOP Principles - Encapsulation

class: ***Dog***

- name
- lifespan
- age

```
class Dog:

    def __init__( self, n, age ):
        self.name = n
        self.lifespan = 20
        self.age = el
        print 'Dog created !!'
```

OOP Principles - Encapsulation

class: ***Dog***

- name
- lifespan
- age

```
class Dog:

      def __init__( self, n, age ):
            self.name = n
            self.lifespan = 20
            self.age = el
            print 'Dog created !!'
#---------------------------------

dog1 = Dog( 'Tommy', 15 )

dog1.lifespan = 1000
```

OOP Principles - Encapsulation

class: ***Dog***

- name
- lifespan
- age

```
class Dog:

    def __init__( self, n, age ):
        self.name = n
        self.lifespan = 20
        self.age = el
        print 'Dog created !!'
#---------------------------------

dog1 = Dog( 'Tommy', 15 )
dog1.lifespan = 1000
```

X

OOP Principles - Encapsulation

class: ***Dog***

- name
- lifespan
- age

```
class Dog:

    def __init__( self, n, age ):
        self.name = n
        self.lifespan = 20
        self.age = el
        print 'Dog created !!'
#---------------------------------

dog1 = Dog( 'Tommy', 15 )
dog1.lifespan = 1000
dog1.age = 50
```

X

OOP Principles - Encapsulation

class: ***Dog***

- name
- lifespan
- age

```
class Dog:

    def __init__( self, n, age ):
        self.name = n
        self.lifespan = 20
        self.age = el
        print 'Dog created !!'
#---------------------------------

dog1 = Dog( 'Tommy', 15 )
X dog1.lifespan = 1000
X dog1.age = 50
```

OOP Principles - Encapsulation

To restrict the access of the data(attributes) and the code(methods), their scope of access can be set as:

- ***Private***: Can be accessed from only within the class definition.
- ***Public***: Can be accessed from within as well as outside the class definition.

OOP Principles - Inheritance

II. Inheritance

It is the process by which one object acquires the properties of another object.

- *Helps in presenting hierarchical classification.*

OOP Principles - Inheritance

There can be many similar classes.

class: ***Dog***
▪ Name ▪ Skin color ▪ Ear length ▪ Is spotted
➢ *Walk* ➢ *Eat* ➢ *Sleep* ➢ *Chase cat*

class: ***Cat***
▪ Name ▪ Skin color ▪ Ear length ▪ Is spotted
➢ *Walk* ➢ *Eat* ➢ *Sleep* ➢ *Run from Dog* ➢ *Chase mouse*

class: ***Mouse***
▪ Name ▪ Skin color ▪ Ear length ▪ Is spotted
➢ *Walk* ➢ *Eat* ➢ *Sleep* ➢ *Run from Dog* ➢ *Run from Cat*

OOP Principles - Inheritance

*The similar classes have **same** attributes and methods.*

class: ***Dog***	class: ***Cat***	class: ***Mouse***
▪ Name ▪ Skin color ▪ Ear length ▪ Is spotted	▪ Name ▪ Skin color ▪ Ear length ▪ Is spotted	▪ Name ▪ Skin color ▪ Ear length ▪ Is spotted
➢ *Walk* ➢ *Eat* ➢ *Sleep*	➢ *Walk* ➢ *Eat* ➢ *Sleep*	➢ *Walk* ➢ *Eat* ➢ *Sleep*
➢ *Chase cat*	➢ *Run from Dog* ➢ *Chase mouse*	➢ *Run from Dog* ➢ *Run from Cat*

OOP Principles - Inheritance

*Separate-out **the common attributes & methods** into a logical class.*

class: ***Animal***
▪ Name ▪ Skin color ▪ Ear length ▪ Is spotted
➢ *Walk* ➢ *Eat* ➢ *Sleep*

OOP Principles - Inheritance

class: ***Animal***

- Name
- Skin color
- Ear length
- Is spotted

- *Walk*
- *Eat*
- *Sleep*

*The class "**Animal**" is **inherited** by all other class which are supposed to share similar attributes.*

class: ***Dog***

- *Chase cat*

class: ***Cat***

- *Run from Dog*
- *Chase mouse*

class: ***Mouse***

- *Run from Dog*
- *Run from Cat*

OOP Principles - Inheritance

class: ***Animal***

Super-class

- Name
- Skin color
- Ear length
- Is spotted

- *Walk*
- *Eat*
- *Sleep*

Sub-class

class: ***Dog***

- *Chase cat*

Sub-class

class: ***Cat***

- *Run from Dog*
- *Chase mouse*

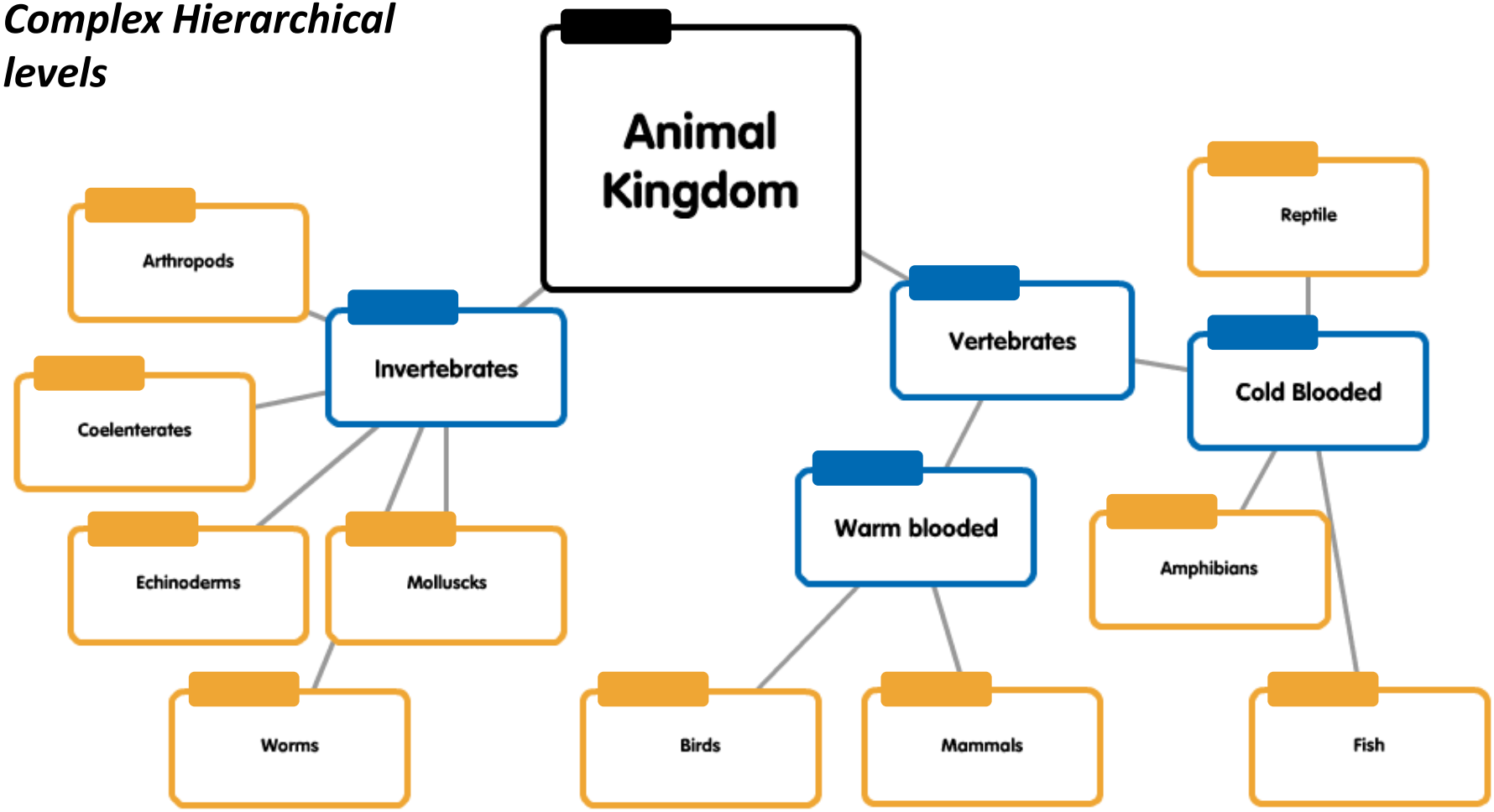
Sub-class

class: ***Mouse***

- *Run from Dog*
- *Run from Cat*

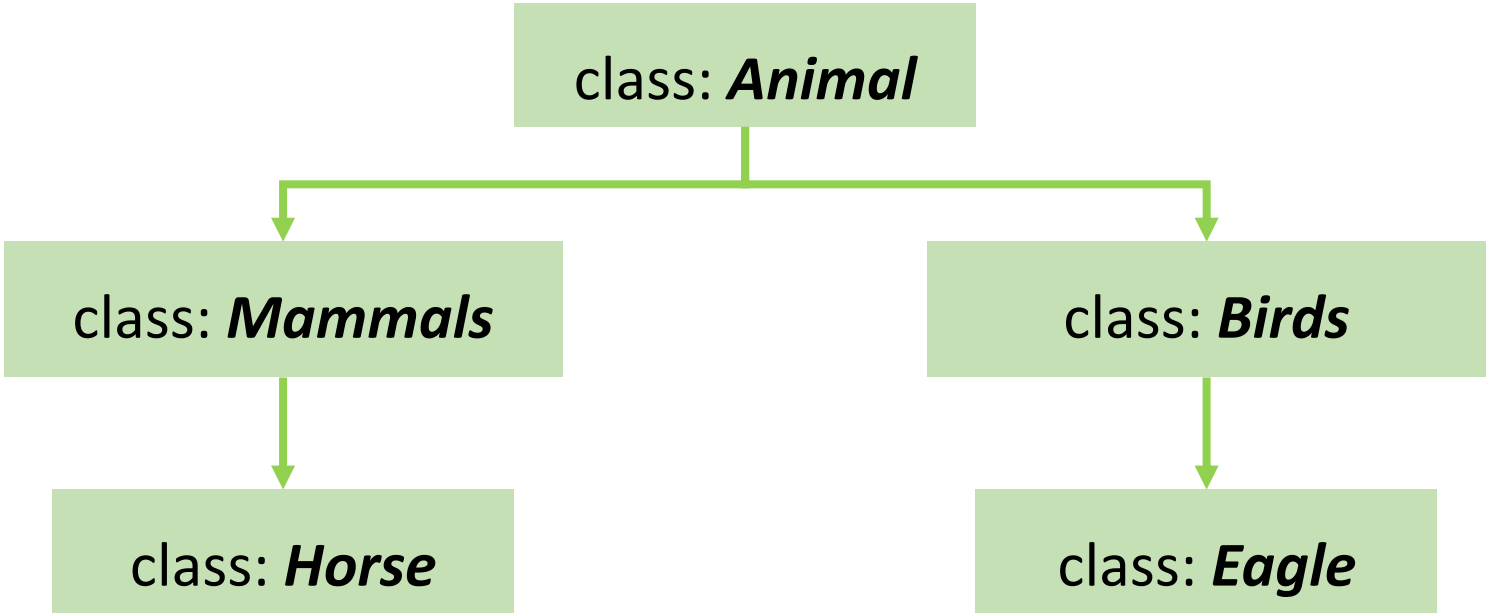
OOP Principles - Inheritance

Complex Hierarchical levels



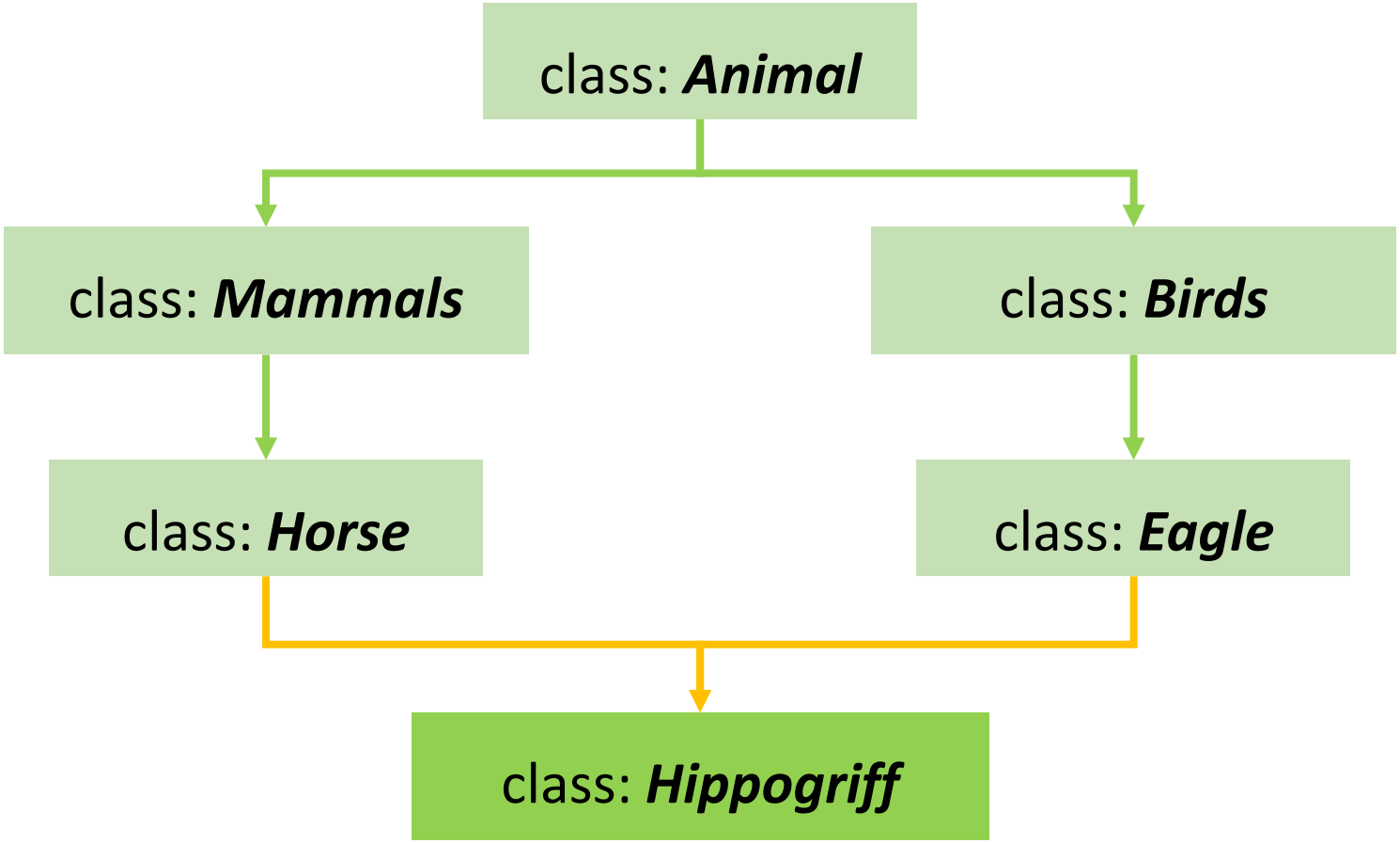
OOP Principles - Inheritance

Multiple Inheritance



OOP Principles - Inheritance

Multiple Inheritance



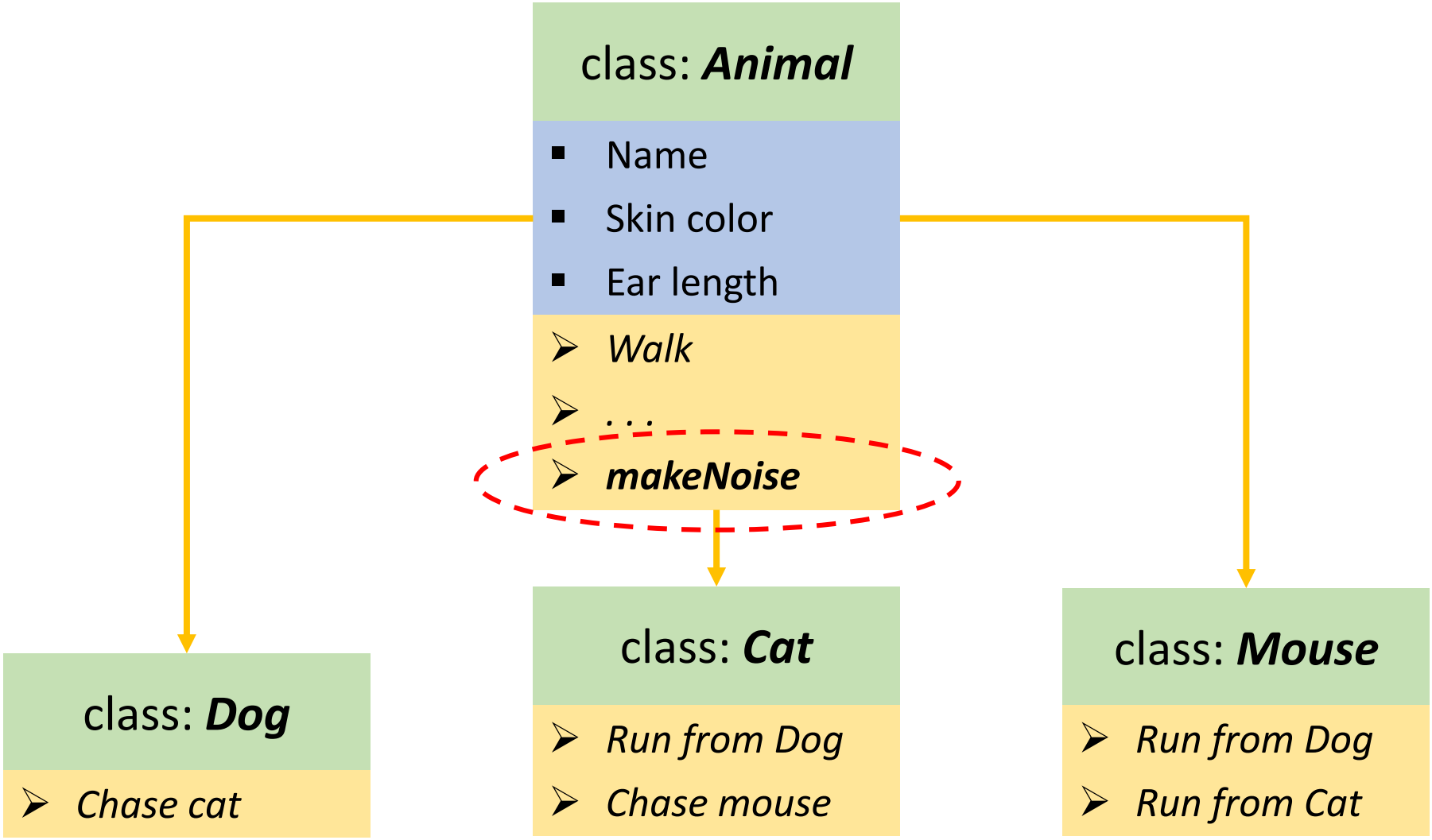
OOP Principles - Polymorphism

III. Polymorphism

It is a feature that allows one interface to be used for a general class of actions.

- *The specific action is determined by the exact nature of the situation.*
- *"Polymorphism" -> Greek -> "many forms"*

OOP Principles - Polymorphism



OOP Principles - Polymorphism

```
dog1 = Dog( 'Spike', ... )
cat1 = Cat( 'Tom', ... )
mouse1 = Mouse( 'Jerry', ... )


dog1.walk()
cat1.walk()
mouse1.walk()
```

OOP Principles - Polymorphism

```
dog1 = Dog( 'Spike', ... )
cat1 = Cat( 'Tom', ... )
mouse1 = Mouse( 'Jerry', ... )

dog1.walk()
cat1.walk()
mouse1.walk()
```

Output:

```
Spike is walking!
Tom is walking!
Jerry is walking!
```

OOP Principles - Polymorphism

```
dog1 = Dog( 'Spike', ... )
cat1 = Cat( 'Tom', ... )
mouse1 = Mouse( 'Jerry', ... )


dog1.makeNoise()
cat1.makeNoise()
mouse1.makeNoise()
```

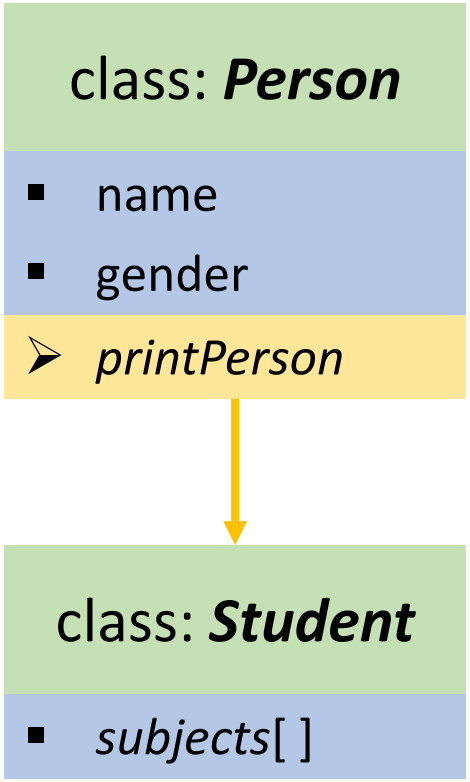
OOP Principles - Polymorphism

```
dog1 = Dog( 'Spike', ... )
cat1 = Cat( 'Tom', ... )
mouse1 = Mouse( 'Jerry', ... )

dog1.makeNoise()
cat1.makeNoise()
mouse1.makeNoise()
```

Output:

```
Spike says ... Bow bow!!
Tom says ... Meaow!!
Jerry says ... Sshhh!!
```

Inheritance - Syntax



Inheritance - Syntax

```
class Person:
      def __init__( self, n, g ):
            self.name = n
            self.gender = g
      def printPerson(self):
            print "{0}, {1}".format(self.name, self. gender)
```

Inheritance - Syntax

```
class Person:
      def __init__( self, n, g ):
            self.name = n
            self.gender = g
      def printPerson(self):
            print "{0}, {1}".format(self.name, self. gender)


class Student(Person):
      def  __init__( self, n, g, listSubjects ):
            self.name = n
            self.gender = g
            self.subjects = listSubjects
```

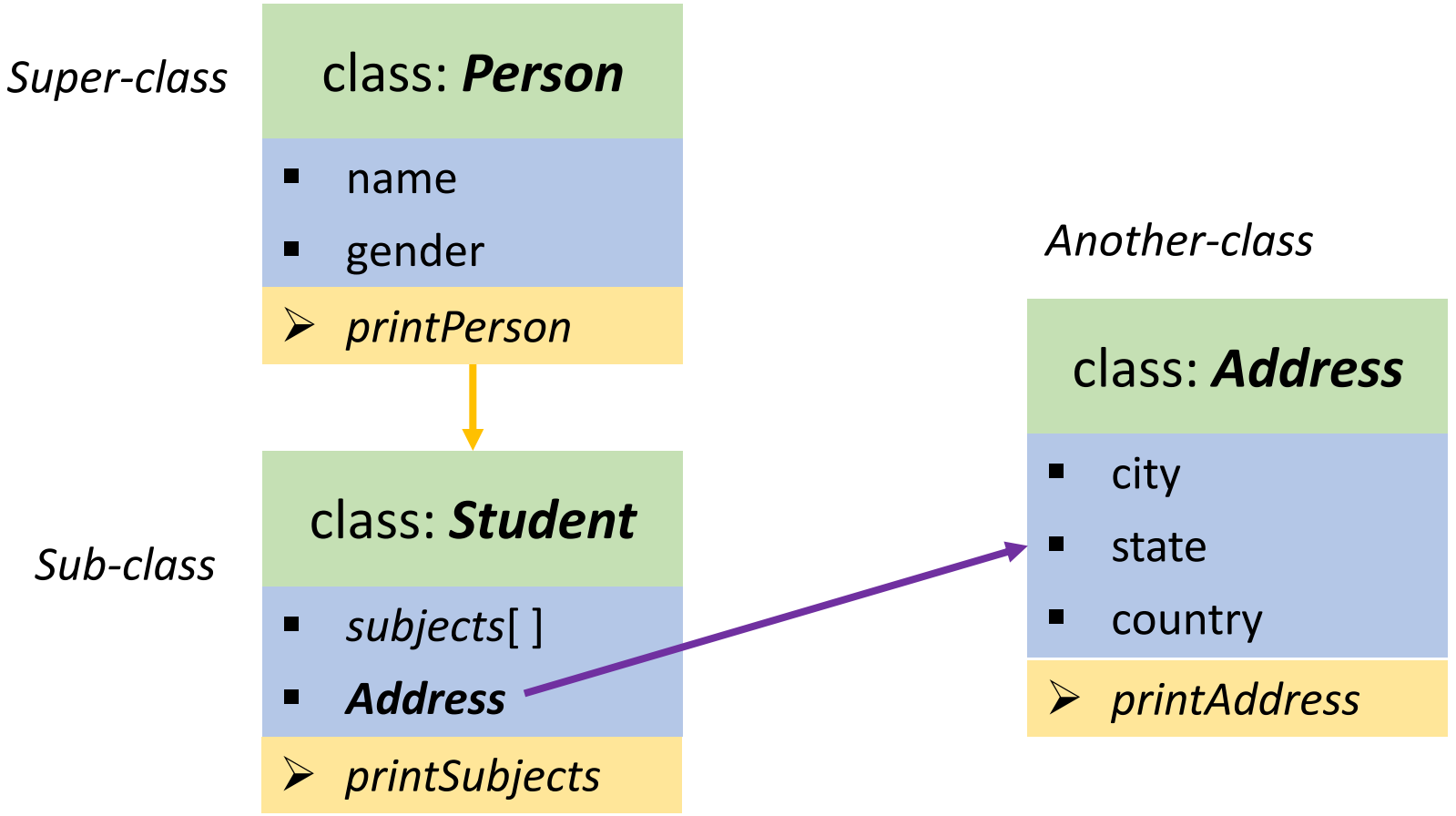
Inheritance - Syntax

```
class Person:
      def __init__( self, n, g ):
            self.name = n
            self.gender = g
      def printPerson(self):
            print "{0}, {1}".format(self.name, self. gender)


class Student(Person):
      def  __init__( self, n, g, listSubjects ):
            self.name = n
            self.gender = g
            self.subjects = listSubjects


s1 = Student( 'Shyam', 'Male', ['physics', 'chemistry'] )
s1.printPerson()
```

Assignment - II

WAP to implement the ***classes*** as shown below.

Create ***1 student object*** and print info & address.



Assignment - II

Desired output :

```
Name: Shyam
Gender: Male
Studies:
[ physics, chemistry ]
Lives at:
Rourkela, Odisha, India
```

Thank You !

National Institute Of Technology, Rourkela