UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Topic 10:
Advanced Object-Oriented Programming

CSGE601020 - Dasar-Dasar Pemrograman 1

Lintang Matahari Hasani, S.Kom., M.Kom. | Dr.Eng. Lia Sadita, S.Kom., M.Eng. | Raja Damanik, M.Sc.

Acknowledgement

This slide is an adapted version of **Advanced OOP** slides used in DDP1 Course (2020/2021) by Hafizh Rafizal Adnan, M.Kom.

Some additional contents, illustrations and visual design elements are provided by **Lintang Matahari Hasani, M.Kom.**

In this session, you will learn ...

- Inheritance
- Polymorphism
- Python Operator Overloading

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Inheritance

Inheritance

We have learned OOP in the previous topic. However, it is not enough to help you solving many real world problems efficiently.

In real world we can find many objects. Some of them have **similarities and relationships** that form hierarchies

Inheritance Real Life Example

Cat

Attributes: `name`, `age`, `breed`
Method: `eats()`, `meow()`

Dog

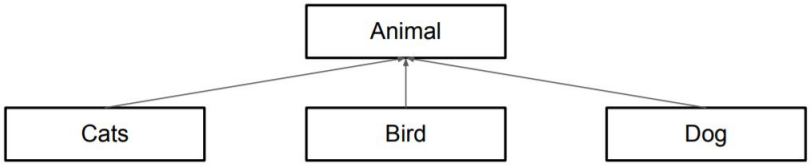
Attributes: `name`, `age`
Method: `eats()`

Bird

Attributes: `name`, `age`, `feather_color`
Method: `eats()`, `flies()`

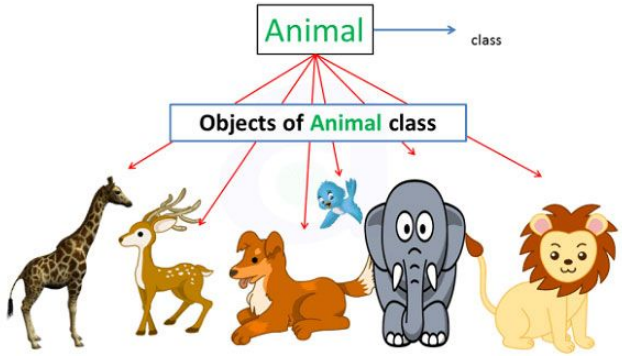
Any similarities?

Inter-Class Relations



the class hierarchy imposes an **is-a** relationship between classes

- ➔ Cats is-a (or **is a kind of**) Animal
- ➔ Animal has three subclasses, cats, rats, and tiger
- ➔ Cats has a super class: Animal



https://www.atnyla.com/library/images-tutorials/class-and-object-in-java.PNG

Code Without Inheritance

```
class Birds():
      def __init__(self, name, age, feather_color):
            self.name = name
            self.age = age
            self.breed = feather_color

      def eat(self):
            print(self.name ," is eating...")

      def flies(self):
            print(self.name, " is flying...")
```

```
class Cat():
      def __init__(self, name, age, breed):
            self.name = name
            self.age = age
            self.breed = breed

      def eat(self):
            print(self.name ," is eating...")

      def meow(self):
            print(self.name, " is meowing...")
```

```
class Dog():
      def __init__(self, name, age):
            self.name = name
            self.age = age

      def eat(self):
            print(self.name ," is eating...")
```

Inheritance in Python

Inheritance is the capability of one class **to derive or inherit the properties** from another class.

The benefits of inheritance are:

- ➔ It represents real-world relationships well.
- ➔ It provides reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- ➔ It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Simple Class Example

```
class MyClass (object):
     pass

class Child1Class (MyClass):
     pass

class Child2Class (MyClass):
     pass
```

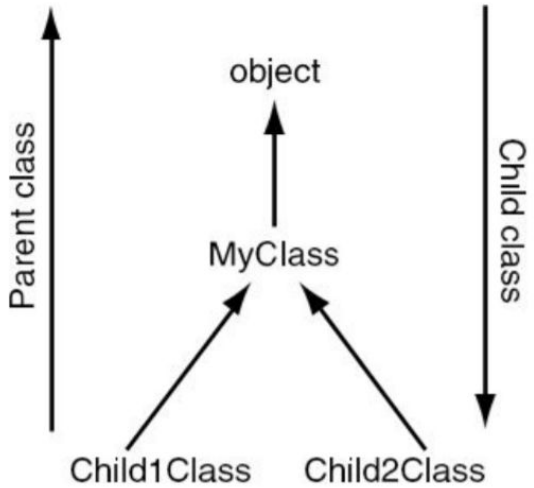


FIGURE 12.1 A simple class hierarchy.

Code with Inheritance: `Animal`, `Cat`, and `Dog`

The superclass `Animal`

```
class Animal():
      def __init__(self, name, age):
            self.name = name
            self.age = age

      def eat(self):
            print(self.name ," is eating...")
```

The subclass `Cat` and `Dog` inherit the properties of `Animal`

```
class Cat(Animal):
      def __init__(self, name, age, breed):
            super().__init__(name, age)
            self.breed = breed

      def meow(self):
            print(self.name, " is meowing...")
```

```
class Dog(Animal):
      pass
```

Code the class `Bird` that inherits `Animal` and has a function:

```
def flies(self):
      print(self.name, " is flying...")
```

```
class Animal():
      def __init__(self, name, age):
            self.name = name
            self.age = age

      def eat(self):
            print(self.name ," is eating...")
```

Share your answer

Code with Inheritance: `Animal` and `Birds`

```
class Animal():
      def __init__(self, name, age):
            self.name = name
            self.age = age

      def eat(self):
            print(self.name ," is eating...")
```

```
class Bird(Animal):
      def __init__(self, name, age, feather_color):
            super().__init__(name, age)
            self.feather_color = feather_color

      def flies(self):
            print(self.name, " is flying...")
```

Multiple Inheritance

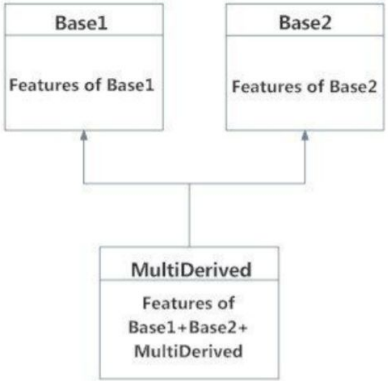
A class can be derived **from more than one base class** in Python, similar to C++. This is called multiple inheritance.

In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance

```
class Base1:
      pass

class Base2:
      pass

class MultiDerived(Base1, Base2):
      pass
```



Multilevel Inheritance

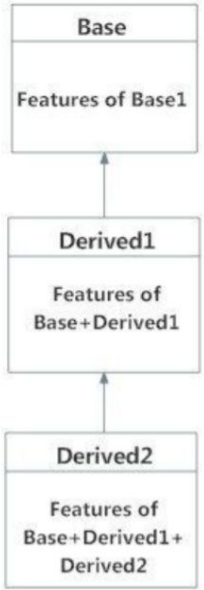
We can also **inherit from a derived class**. This is called multilevel inheritance. It can be of any depth in Python.

In multilevel inheritance, features of the base class and the derived class are inherited into the new derived class

```
class Base:
      pass

class Derived1(Base):
      pass

class derived2(Derived1):
      pass
```



UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

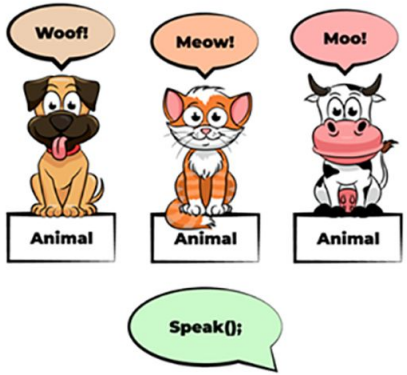
Polymorphism

Polymorphism

The word polymorphism means having many forms. In programming, polymorphism means **same function name (but different signatures) being uses for different types**.

In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class.

This process of re-implementing a method in the child class is known as **Method Overriding**



https://codegym.cc/groups/posts/polymorphism-in-java

Polymorphism Examples

Python built-in polymorphic feature

```
# Python program to demonstrate in-built poly-
# morphic functions

# len() being used for a string
print(len("geeks"))

# len() being used for a list
print(len([10, 20, 30]))
```

Polymorphism with inheritance (overriding)

```
class Bird:
        def intro(self):
                print("There are many types of birds." )

        def flight(self):
                print("Most of the birds can fly but some
        cannot.")

class Sparrow(Bird):
        def flight(self):
                print("Sparrows can fly." )

class Ostrich(Bird):
        def flight(self):
                print("Ostriches cannot fly." )
```

`flight()` methods in `Sparrow` and `Ostrich` overrides the original `flight()` method inherited from the class `Bird`

Polymorphism (Unique Case)

Method Overloading is an example of Compile time polymorphism. In this, more than one method of the same class shares the same method name having different signatures. Method overloading is used to add more to the behavior of methods and there is no need of more than one class for method overloading.

Note: Python does not support method overloading. We may overload the methods but can only use the latest defined method.

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Python Operator Overloading

Operator Overloading

Python operators work for **built-in classes**. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

So what happens when we use them with objects of a user-defined class?

```
class Point:
      def __init__(self, x=0, y=0):
            self.x = x
            self.y = y

p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1+p2)
```

It will raise `TypeError`

Python Special Function

Class functions that begin with **double underscore __** are called special functions in Python.

Using special functions, we can make our class **compatible with built-in functions**.

We have used some special functions such as `__init__` and `__str__`

Overriding + Operator

```
class Point:
     def __init__(self, x = 0, y = 0):
          self.x = x
          self.y = y

     def __str__(self):
          return "({0},{1})".format(self.x, self.y)

     def __add__(self, other):
          x = self.x + other.x
          y = self.y + other.y
          return Point(x, y)

p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1 + p2)
```

Special Function List

Operator	Expression	Internally
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder (modulo)	`p1 % p2`	`p1.__mod__(p2)`
Bitwise NOT	`~p1`	`p1.__invert__()`

More: https://docs.python.org/3/reference/datamodel.html#special-method-names

Overriding + Operator

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __lt__(self, other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag
```

```
p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)

# use less than
print(p1)
>True
print(p2)
>False
print(p1)
>False
```

Special Function List

Operator	Expression	Internally
Less than	`p1 < p2`	`p1.__lt__(p2)`
Less than or equal to	`p1 <= p2`	`p1.__le__(p2)`
Equal to	`p1 == p2`	`p1.__eq__(p2)`
Not equal to	`p1 != p2`	`p1.__ne__(p2)`
Greater than	`p1 > p2`	`p1.__gt__(p2)`
Greater than or equal to	`p1 >= p2`	`p1.__ge__(p2)`

More: https://docs.python.org/3/reference/datamodel.html#special-method-names

Overriding

Method overriding di suatu class adalah pengimplementasian method yang sudah diwariskan dari class-class parentnya.

Method fun() di class B meng-*override* method fun() yang diwariskan dari kelas A.

Efeknya adalah setiap kali objek class B memanggil fun(), yang akan digunakan adalah fungsi fun() yang memang ada di kelas B.

Fungsi fun() A sudah di-”timpa” (di-override) oleh fungsi fun() kelas B.

```
class A:
    def fun():
        pass
class B(A):
    def fun(): #overriding
        pass
```

Fungsi fun() di class A tidak di-override.

Jika fungsi fun() tidak di-override di class B, maka akan digunakan fungsi fun() yang diwariskan dari parent classnya B, yaitu A.

```
class A:
    def fun():
        pass
class B(A):
    pass
```

Pada kasus di samping, fungsi fun() tidak diimplementasikan di class B.

Class B juga tidak dibuat sebagai anak class dari class A.

Kalau objek B memanggil fungsi fun(), akan eror, karena fungsi fun() di A tidak diwariskan ke B.

Kita tidak bisa bilang fungsi fun() tidak di-override, karena **class B tidak punya hubungan apa-apa dengan class A.**

Konsep override hanya masuk akal saat kedua class memiliki hubungan child-parent atau child-grandparent…

```
class A:
    def fun():
        pass
class B:
    pass
```

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Q&A Session