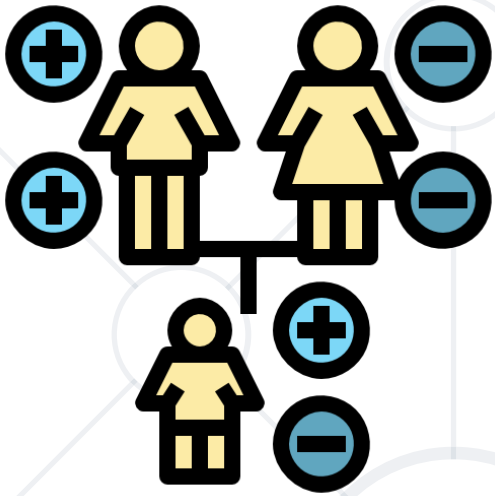


# Inheritance

Capability to Inherit Other Properties



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

[sli.do](https://sli.do)

**#python-advanced**

## 1. Inheritance

- The **super()** method

## 2. Forms of Inheritance

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance

## 3. Mixins





# Inheritance

Capability to Inherit Other Properties

# The Four Basics Concepts of OOP

- **Inheritance** - extend the functionality of the code's existing classes to eliminate repetitive code
- **Encapsulation** - stop objects from interacting with each other so classes cannot change or interact with the specific variables and functions of an object
- **Abstraction** - isolate the impact of changes made to the code so the change will only affect the variables shown and not the outside code
- **Polymorphism** - allows different classes to have methods with the same name



# Inheritance

- Inheritance is the capability of one class to **inherit** the methods and properties from another class
- Benefits of inheritance:
  - Code **reusability**
  - Add features to a class without modifying it
  - It is **transitive** in nature



# Example: Inheritance

```
class Person:
    def __init__(self, first_name,
last_name):
        self.first_name =
first_name
        self.last_name = last_name

    def get_full_name(self):
        return f'{self.first_name}
{self.last_name}'
```

Subclassing

```
class Student(Person):
    pass
```



```
# An Object of class Student
student = Student("John", "Smith")
print(student.get_full_name())
# John Smith
```

# The super() Method

- Built-in method which **returns a temporary object** of the superclass
- Allows you to call methods of the **superclass** in your **subclass**
- The primary use case of this is to **extend** the functionality of the inherited method





# Example: super() Method

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

    def get_info(self):
        return f'{self.name} is {self.age} years old.'

class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id

    def get_id(self):
        return self.student_id
```

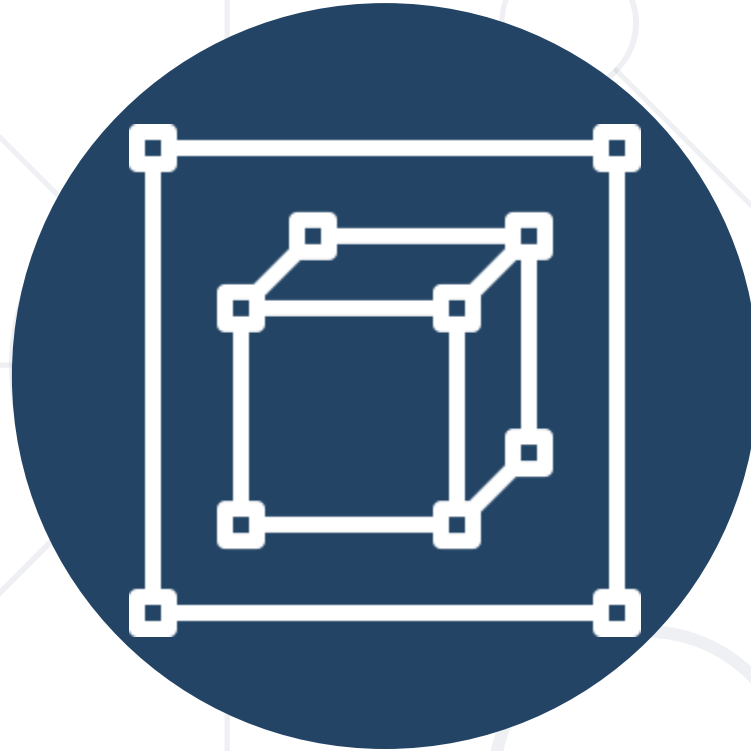
```
# Create an object of the superclass
person = Person("John", 25)
print(person.get_info())
# returns 'John is 25 years old.'

# Create an object of the subclass
student = Student("Leo", 20, 10035464)
print(student.get_info())
# returns 'Leo is 20 years old.'
print(student.get_id())
# returns 10035464
```

- Create two classes named **Food** and **Fruit**
  - **Food** will receive **expiration\_date** upon initialization
  - **Fruit** will receive **name** and **expiration\_date** upon initialization
- **Fruit** should inherit from **Food**

```
class Food:
    def __init__(self, expiration_date):
        self.expiration_date = expiration_date

class Fruit(Food):
    def __init__(self, name, expiration_date):
        super().__init__(expiration_date)
        self.name = name
```



# **Forms of Inheritance**

Single, Multiple and Multilevel

# Forms of Inheritance

- There are four types of inheritance
  - **Single**
  - **Multiple**
  - **Multilevel**
  - **Hierarchical**
- **Hybrid Inheritance** - consists of multiple types of inheritance



# Single Inheritance

- When a child class inherits properties from a **single parent** class only



```
class Parent:
    def say_hi(self):
        return "Hello!"

class Child(Parent):
    def go_school(self):
        return "I go to school."

child = Child()
print(child.say_hi())      # Hello!
print(child.go_school())  # I go to school.
```

# Problem: Single Inheritance

- Create two classes named **Animal** and **Dog**
  - **Animal** with a single method **eat()** that returns: "eating..."
  - **Dog** with a single method **bark()** that returns: "barking..."
- **Dog** should inherit from **Animal**

# Solution: Single Inheritance

```
class Animal:
    def eat(self):
        return "eating..."

class Dog(Animal):
    def bark(self):
        return "barking..."

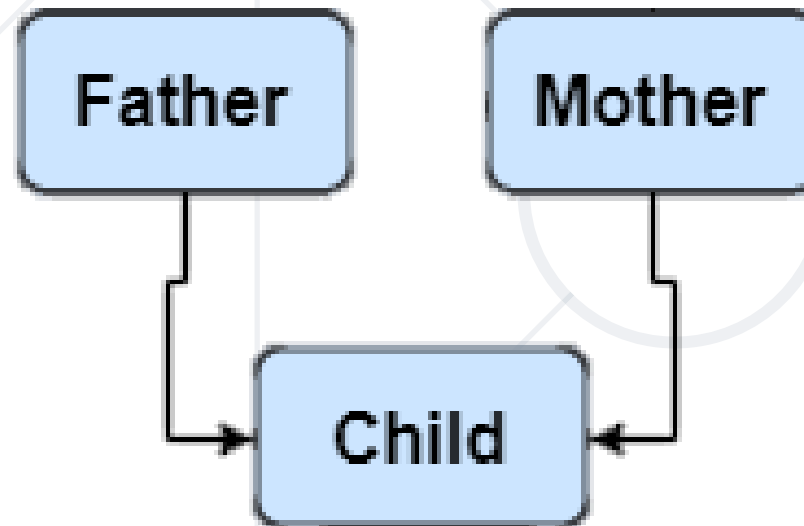
# dog = Dog()
# print(dog.eat())
# print(dog.bark())
```





# Multiple Inheritance

- When a child inherits from **more than one parent** class
- Allows modeling of **complex** relationships



# Example: Multiple Inheritance

```
class Father:
    def __init__(self):
        self.father_name = 'Taylor Evans'

class Mother:
    def __init__(self):
        self.mother_name = 'Bet Williams'

class Daughter(Father, Mother):
    def __init__(self):
        Father.__init__(self)
        Mother.__init__(self)

    def get_parent_info(self):
        return f'Father: {self.father_name},\nMother: {self.mother_name}'
```

```
child = Daughter()
print(child.get_parent_info())
# Father: Taylor Evans, Mother: Bet Williams
```

Calling constructors of both parent classes

# Problem: Multiple Inheritance

- Create three classes named **Person**, **Employee**, and **Teacher**
  - **Person** with a single method **sleep()** that returns: "sleeping..."
  - **Employee** with a single method **get\_fired()** that returns: "fired..."
  - **Teacher** with a single method **teach()** that returns: "teaching..."
- **Teacher** should inherit from **Person** and **Employee**

# Solution: Multiple Inheritance

```
class Person:
    def sleep(self):
        return "sleeping..."

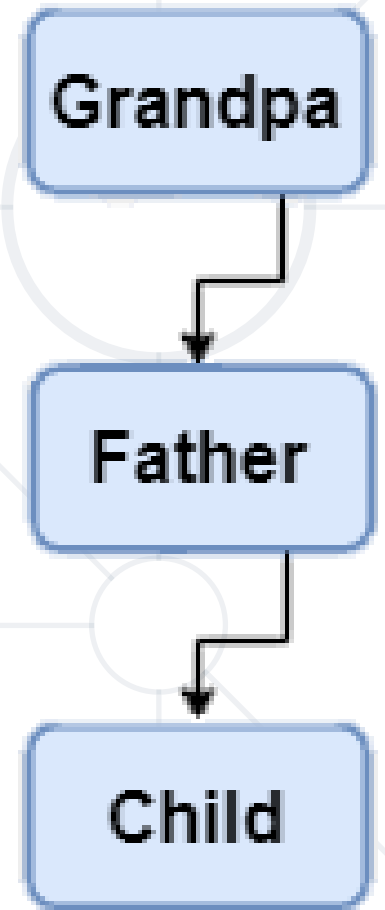
class Employee:
    def get_fired(self):
        return "fired..."

class Teacher(Person, Employee):
    def teach(self):
        return "teaching..."
```



# Multilevel Inheritance

- When a child class becomes a parent class for another child class
- In Python, multilevel inheritance can be done at any depth



# Example: Multilevel Inheritance

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

    def get_name(self):
        return self.name

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

    def get_age(self):
        return self.age
```



# Example: Multilevel Inheritance

```
class GrandChild(Child):
    def __init__(self, name, age, address):
        super().__init__(name, age)
        self.address = address

    def get_address(self):
        return self.address

grand_child = GrandChild("Grand Name", 19, "Address 15-17")
print(grand_child.name)           # Grand Name
print(grand_child.age)           # 19
print(grand_child.address)       # Address 15-17
```

# Problem: Multilevel Inheritance

- Create three classes named **Vehicle**, **Car**, and **SportsCar**
  - **Vehicle** with a single method **move()** that returns: "moving..."
  - **Car** with a single method **drive()** that returns: "driving..."
  - **SportsCar** with a single-method **race()** that returns: "racing..."
- **SportsCar** should inherit from **Car** and **Car** should inherit from **Vehicle**



# Solution: Multilevel Inheritance

```
class Vehicle:
    def move(self):
        return "moving..."

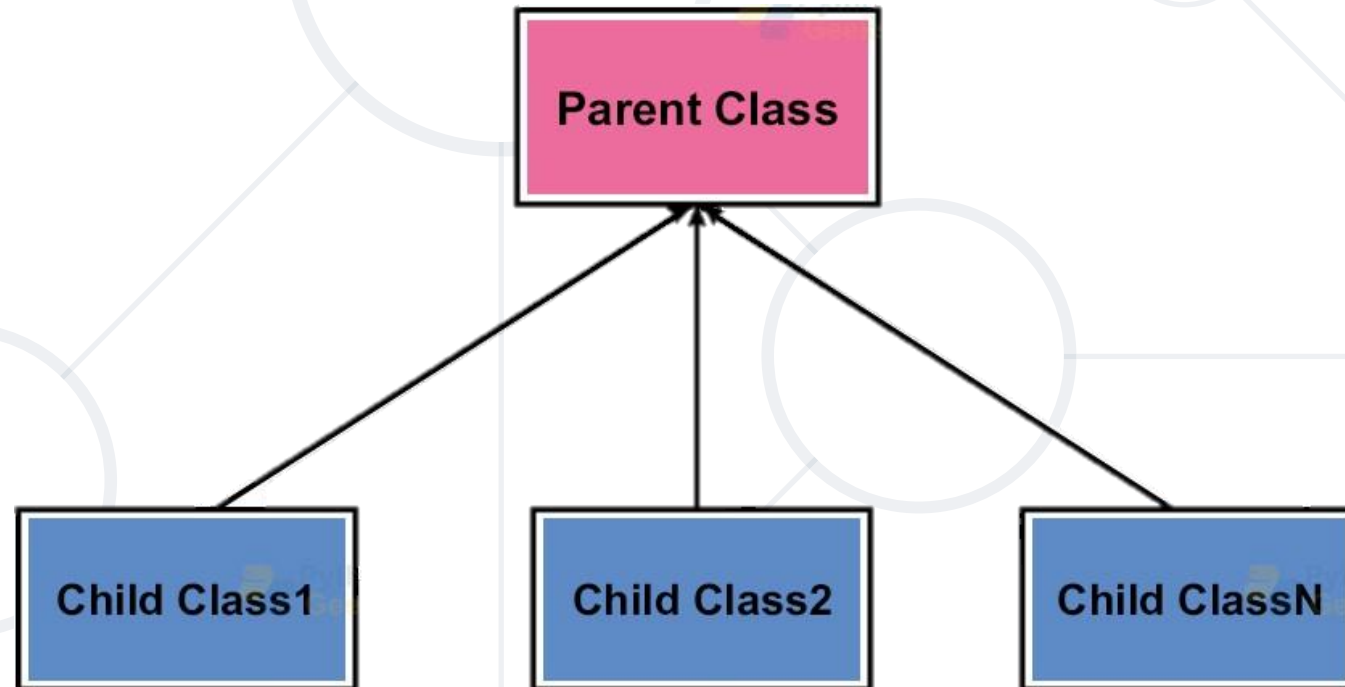
class Car(Vehicle):
    def drive(self):
        return "driving..."

class SportsCar(Car):
    def race(self):
        return "racing..."
```



# Hierarchical Inheritance

- When more than one child classes are created from a single parent class



# Example: Hierarchical Inheritance

```
class Parent:
    def init(self, name):
        self.name = name

    def say_hi(self):
        return f"Hi! I am {self.name}"
```

Both child classes reuse  
the same code

```
class Daughter(Parent):
    def __init__(self, name):
        super().__init__(name)

    def relation(self):
        return "I am my parent's daughter"

class Son(Parent):
    def __init__(self, name):
        super().__init__(name)

    def relation(self):
        return "I am my parent's son"
```

# Problem: Hierarchical Inheritance

- Create three classes named **Animal**, **Dog**, and **Cat**
  - **Animal** with a single method **eat()** that returns: "eating..."
  - **Dog** with a single method **bark()** that returns: "barking..."
  - **Cat** with a single method **meow()** that returns: "meowing..."
- Both **Dog** and **Cat** should inherit from **Animal**

# Solution: Hierarchical Inheritance

```
class Animal:
    def eat(self):
        return "eating..."

class Dog(Animal):
    def bark(self):
        return "barking..."

class Cat(Animal):
    def meow(self):
        return "meowing..."
```





# Method Resolution Order

In Python 3

# MRO in Python 3

- It is the order in which **methods should be inherited** in the presence of multiple inheritance
- Python 3 uses the **C3 linearization** algorithm for MRO
- It is possible to see the MRO of a class using **mro()** method of the class



## The Diamond Problem

```
class Parent:
    pass

class FirstChild(Parent):
    pass

class SecondChild(Parent):
    pass

class GrandChild(SecondChild, FirstChild):
    pass

print(GrandChild.mro())
# [<class '__main__.GrandChild'>, <class '__main__.SecondChild'>, <class
'__main__.FirstChild'>, <class '__main__.Parent'>, <class 'object'>]
```





# Mixins

"mix in" extra properties and methods

# Mixins

- A **mixin** is a class that is implementing a specific set of features that is needed in **many different classes**
- A mixin is a class that **has no data**, only methods
- Mixins **cannot be instantiated** by themselves
- We use mixins to **extend the functionality**



# Mixins Advantages

- Provides **non-complex** mechanisms of **multiple inheritance**
- Provides **code reusability**
- Allow inheritance and use of only **desired features** from the parent class, **not all of them**



# Example: Mixins

```
class Vehicle:
    def __init__(self, position):
        self.position = position

    def travel(self, destination):
        pass

class Car(Vehicle):
    pass

class Clock():
    pass
```

```
class RadioMixin():
    def play_song_on_station(self,
        station_frequency):
        return f'playing song on radio frequency {station_frequency}'

class Car(Vehicle, RadioMixin):
    pass

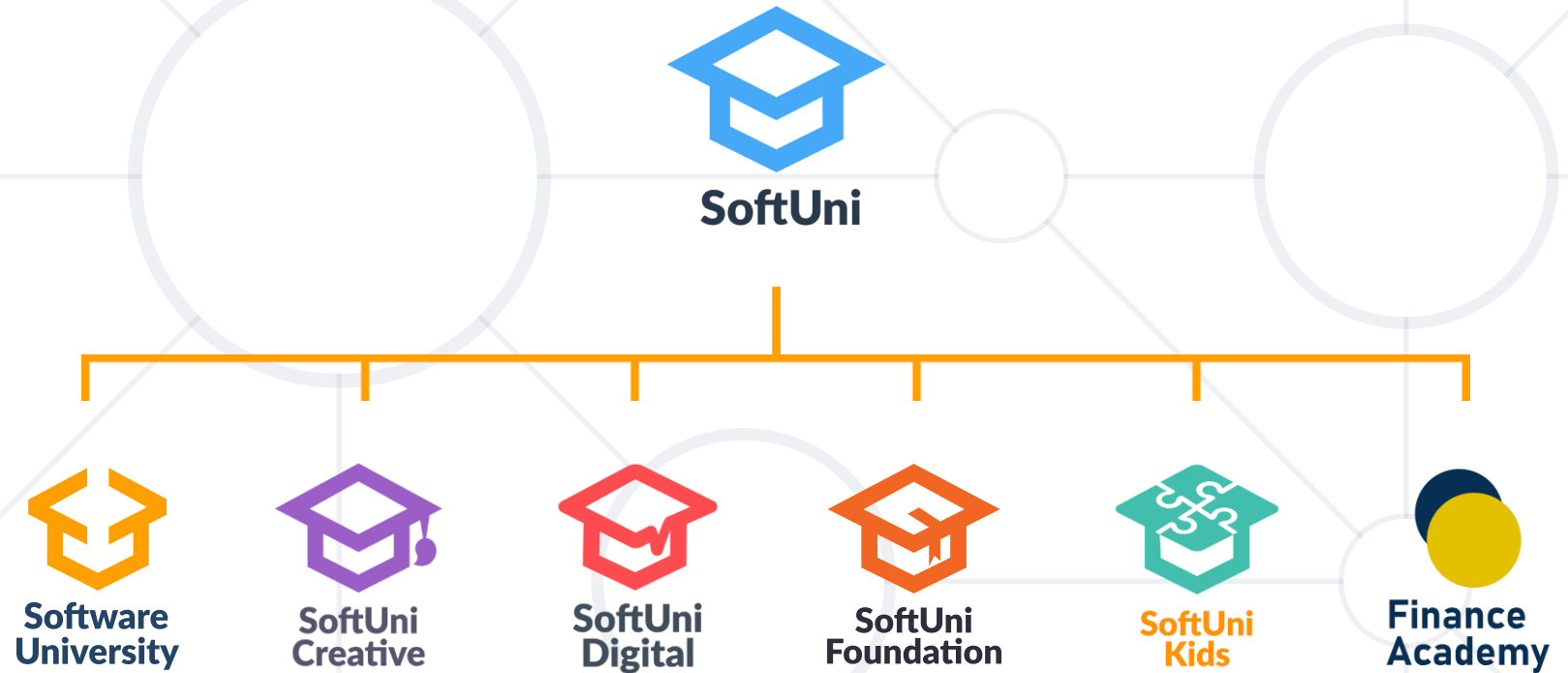
class Clock(RadioMixin):
    pass
```

```
car = Car('Sofia')
clock = Clock()
print(car.play_song_on_station(95.0))      # playing song on radio frequency 95.0
print(clock.play_song_on_station(100.3))   # playing song on radio frequency 100.3
```

- **Inheritance** is the capability to inherit the properties from some another class
- **super()** method allows us to call methods of the superclass in your subclass
- **Mixins** implement a set of features that are needed in many different classes



# Questions?



# SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg), [softuni.org](http://softuni.org)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](http://forum.softuni.bg)



Software  
University





- This course (slides, examples, demos, exercises, homework, documents, videos, and other assets) is **copyrighted content**
- Unauthorized copy, reproduction, or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>

