Python Course Cheat Sheet 9

Method Overriding in Python

- Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes
- When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class
- The version of a method that is executed will be determined by the object that is used to invoke it.
- If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

```
# Defining parent class
class Parent():

    # Constructor
    def __init__(self):
        self.value = "Inside Parent"

    # Parent's show method
    def show(self):
        print(self.value)

# Defining child class
class Child(Parent):

    # Constructor
    def __init__(self):
        self.value = "Inside Child"

    # Child's show method
    def show(self):
        print(self.value)
```

Test

```
obj1 = Parent()
obj2 = Child()

obj1.show()
obj2.show()
```

```
# Multilevel
class Parent1():

    # Parent's show method
    def show(self):
        print("Inside Parent1")

# Defining Parent class 2
class Parent2():

    # Parent's show method
    def display(self):
        print("Inside Parent2")


# Defining child class
class Child(Parent1, Parent2):

    # Child's show method
    def show(self):
        print("Inside Child")


# Driver's code
obj = Child()

obj.show()
obj.display()
```

```
# Python program to demonstrate
# calling the parent's class method
```

```
# inside the overridden method using
# super()


class Parent():

    def show(self):
        print("Inside Parent")

class Child(Parent):

    def show(self):

        # Calling the parent's class
        # method
        super().show()
        print("Inside Child")

# Driver's code
obj = Child()
obj.show()
```

Previous example Cont’d

We reached to Diamond problem, continue from there

Everything related to productivity should be together in one module and everything related to payroll should be together

```
class ProductivitySystem:
    def track(self, employees, hours):
        print('Tracking Employee Productivity')
        print('==============================')
        for employee in employees:
            result = employee.work(hours)
            print(f'{employee.name}: {result}')
        print('')

class ManagerRole:
    def work(self, hours):
        return f'screams and yells for {hours} hours.'

class SecretaryRole:
    def work(self, hours):
        return f'expends {hours} hours doing office paperwork.'
```

```
class SalesRole:
    def work(self, hours):
        return f'expends {hours} hours on the phone.'

class FactoryRole:
    def work(self, hours):
        return f'manufactures gadgets for {hours} hours.'
```

Productivity module implements the ProductivitySystem class, as well as the related roles it supports. The classes implement the work() interface required by the system, but they don’t derived from Employee

```
class PayrollSystem:
    def calculate_payroll(self, employees):
        print('Calculating Payroll')
        print('==================')
        for employee in employees:
            print(f'Payroll for: {employee.id} - {employee.name}')
            print(f'- Check amount: {employee.calculate_payroll()}')
            print('')

class SalaryPolicy:
    def __init__(self, weekly_salary):
        self.weekly_salary = weekly_salary

    def calculate_payroll(self):
        return self.weekly_salary

class HourlyPolicy:
    def __init__(self, hours_worked, hour_rate):
        self.hours_worked = hours_worked
        self.hour_rate = hour_rate

    def calculate_payroll(self):
        return self.hours_worked * self.hour_rate

class CommissionPolicy(SalaryPolicy):
    def __init__(self, weekly_salary, commission):
        super().__init__(weekly_salary)
        self.commission = commission

    def calculate_payroll(self):
        fixed = super().calculate_payroll()
        return fixed + self.commission
```

PayrollSystem, which calculates payroll for the employees. It also implements the policy classes for payroll. As you can see, the policy classes don’t derive from Employee anymore.

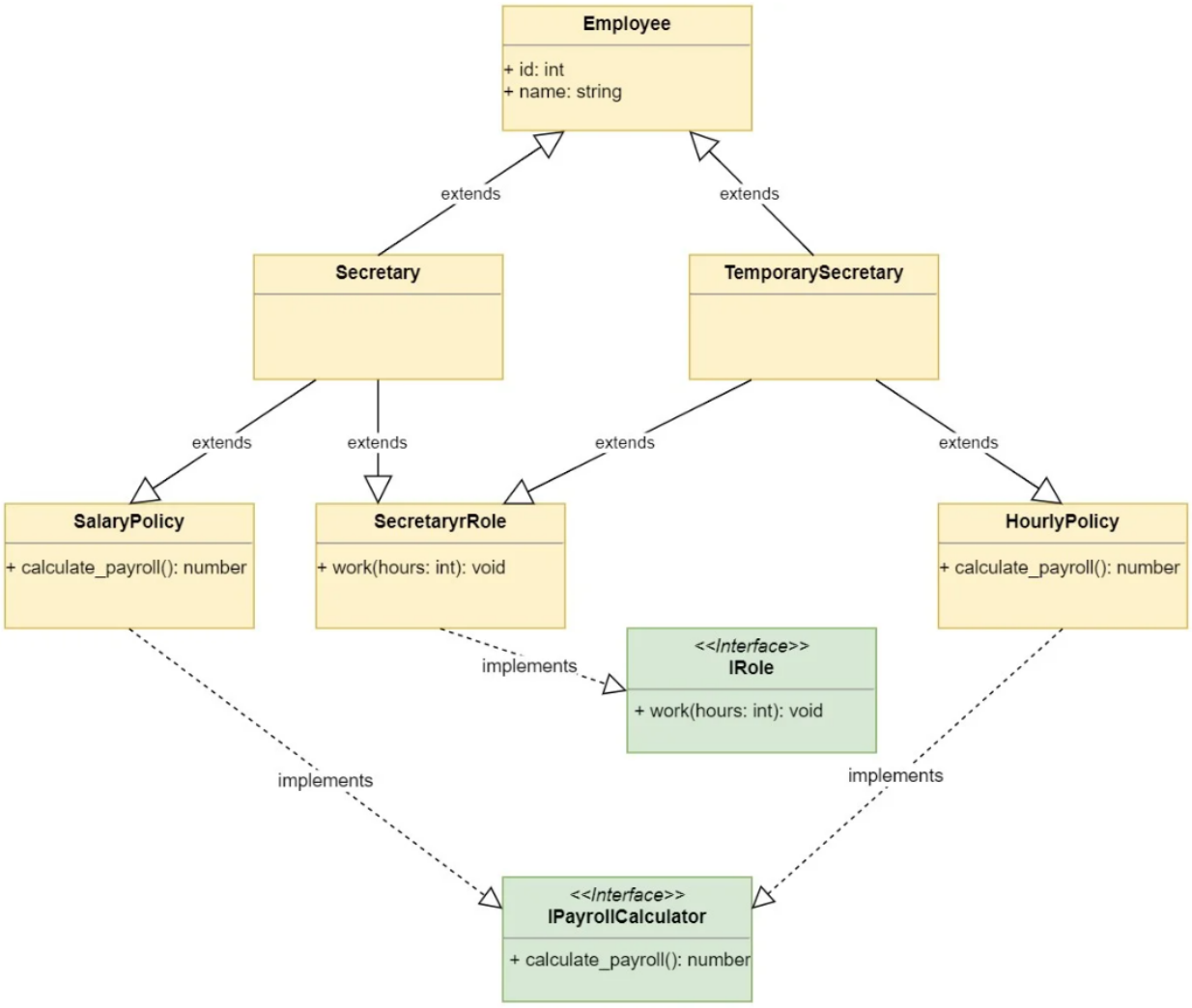
```
class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name
#        self.address = None
class Manager(Employee, ManagerRole, SalaryPolicy):
    def __init__(self, id, name, weekly_salary):
        SalaryPolicy.__init__(self, weekly_salary)
        super().__init__(id, name)

class Secretary(Employee, SecretaryRole, SalaryPolicy):
    def __init__(self, id, name, weekly_salary):
        SalaryPolicy.__init__(self, weekly_salary)
        super().__init__(id, name)

class SalesPerson(Employee, SalesRole, CommissionPolicy):
    def __init__(self, id, name, weekly_salary, commission):
        CommissionPolicy.__init__(self, weekly_salary, commission)
        super().__init__(id, name)

class FactoryWorker(Employee, FactoryRole, HourlyPolicy):
    def __init__(self, id, name, hours_worked, hour_rate):
        HourlyPolicy.__init__(self, hours_worked, hour_rate)
        super().__init__(id, name)

class TemporarySecretary(Employee, SecretaryRole, HourlyPolicy):
    def __init__(self, id, name, hours_worked, hour_rate):
        HourlyPolicy.__init__(self, hours_worked, hour_rate)
        super().__init__(id, name)
```



- The employees module imports policies and roles from the other modules and implements the different Employee types
- Using multiple inheritance to inherit the implementation of the salary policy classes and the productivity roles
- The implementation of each class only needs to deal with initialization
- Need to explicitly initialize the salary policies in the constructors