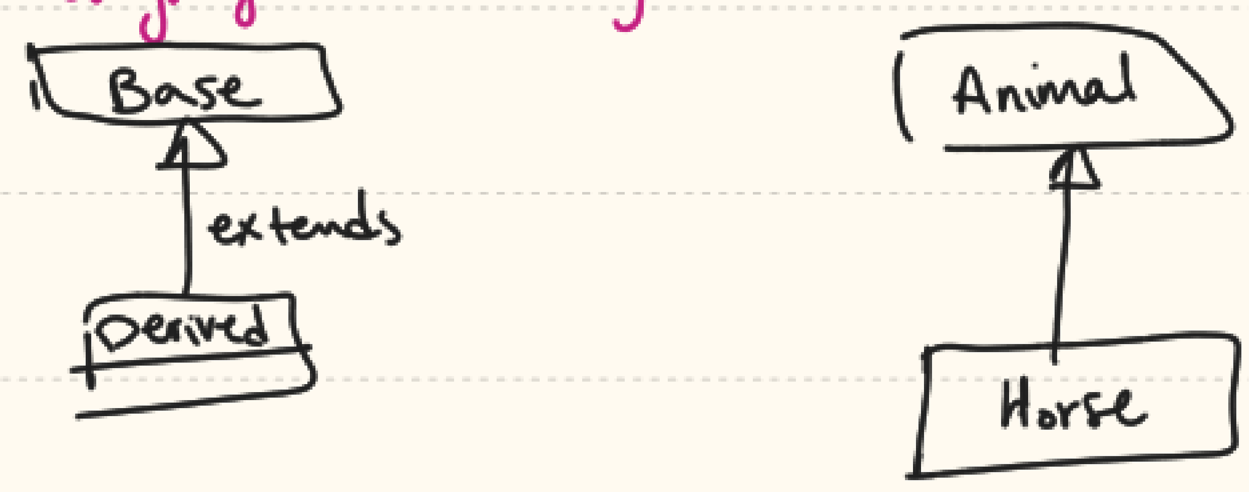
Inheritence & Composition

✓ two important concepts of OOP that model the relationship between two classes.

✓ building blocks of object oriented design

✓ used for writing reusable codes

Unified Language Modeling

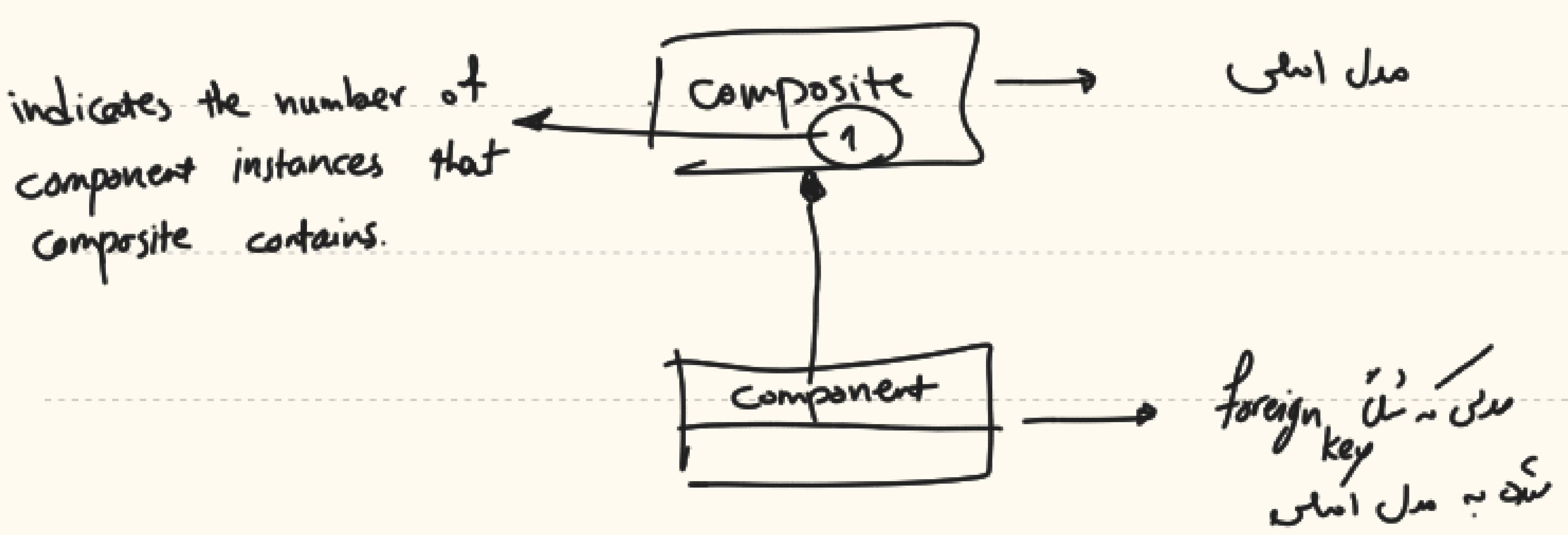


Liskov substitution principle

if S is a subtype of T, then replacing objects of type T with objects of type S doesn't change the program's behavior.

what's Composition?

relationship between models.



An overview of inheritence in python.

everything in python is an object. — modules, class, functions — objects created from classes are objects too.

→ also supports multiple inheritence.

→ every class that you create in python implicitly derives from object.

به غیر از exception ها، وقتی یک خطای خاص custom exception بسازم باید از کلاس exception، inherit شه نه از object.

```
Class customException(Exception):
        pass
raise CustomException( )
```

reminder → 1) everything in python is an object.

2) the dir() function : to return a list of valid attributes and methods of an object.

__module__ : attribute, a string representing the module in which the class is defined.

__main__

نکته: if you want to write a custom exception, it should inherit from Exception class explicitly, not object class implicitly.

```python
# Python                                   program.py
import hr

salary_employee = hr.SalaryEmployee(1, "John Smith", 1500)
hourly_employee = hr.HourlyEmployee(2, "Jane Doe", 40, 15)
commission_employee = hr.CommissionEmployee(3, "Kevin Bacon", 1000, 2

payroll_system = hr.PayrollSystem()
payroll_system.calculate_payroll(
    [salary_employee, hourly_employee, commission_employee]
)
```

(salary_employee →) یک instance از employee هست که بصورت SalaryEmployee هست.

(hourly_employee →) یک instance از employee که بصورت hourly حقوق میگیره.

payroll_system رو به عنوان یک interface برای calculate_payroll استفاده کرده و سه تا از کلاس های hourly_employee، salary_employee، commission_employee یک instance ساخته و به عنوان ورودی به کلاس payrollsystem در متد calculate_payroll داده.

Implementation Inheritance vs Interface Inheritance

the derived class inherits both:

1) the base class interface: all the methods, properties and attributes of the base class.

2) the base class implementation: the code that implements the class interface

Modern Programming Languages allow you to inherit from a single class, but you can implement multipule interfaces.

Any object that implements the desired interface can be used in place of another object: duck typing: if it walk like a duck and quack like a duck, then it must be a duck

الان میتونیم مثل payroll system آره یک کلاس دیگه بسازیم که از employee inherit نمیکنه ولی name, id و calculate_payroll function داره و میتونیم به عنوان ورودی به payroll system بدیم.

```python
# Python                                   disgruntled.py
class DisgruntledEmployee:
    def __init__(self, id, name):
        self.id = id
        self.name = name

    def calculate_payroll(self):
        return 1_000_000
```

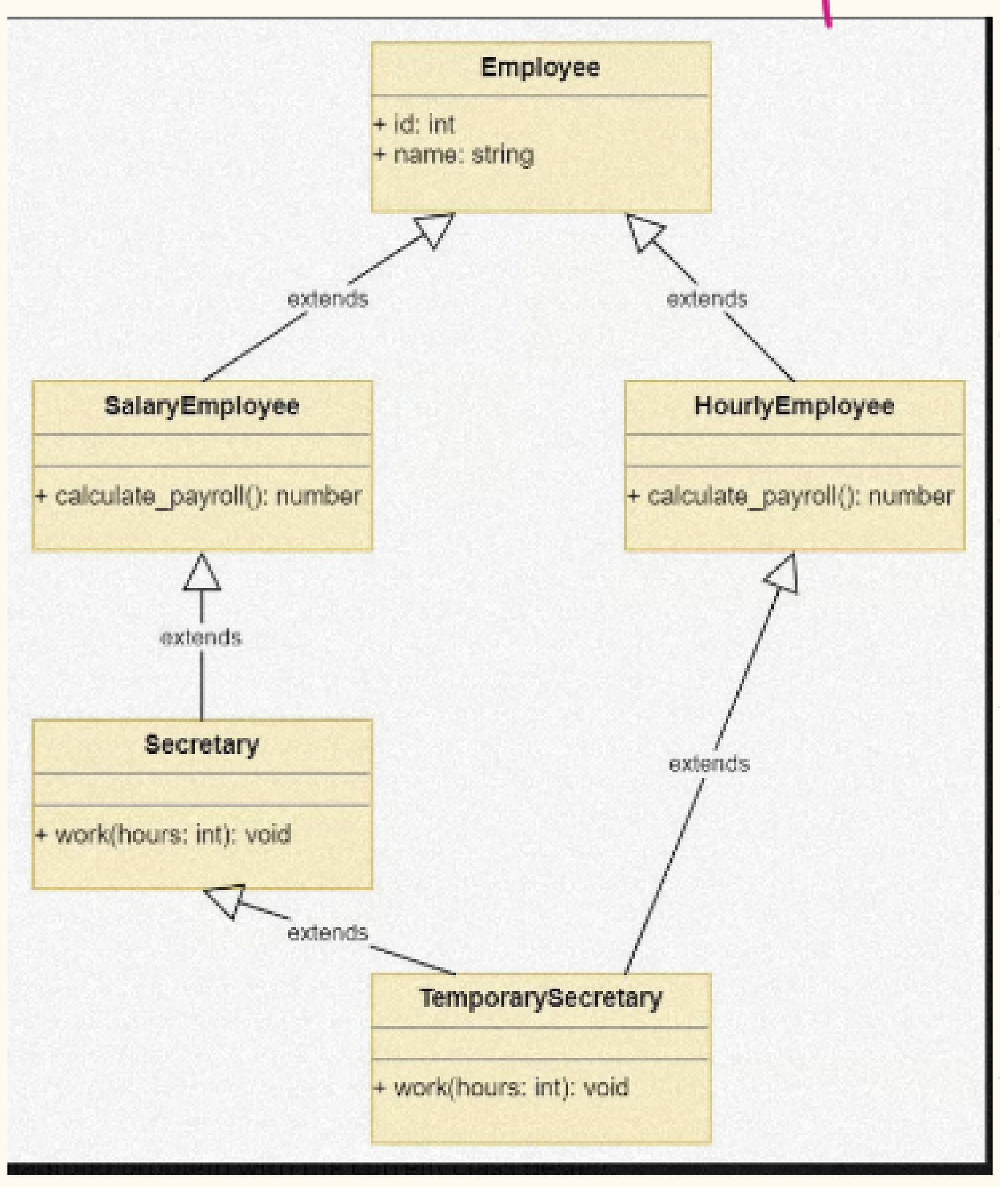
the class explosion Problem

when the your hierarchies become so big that they'll be hard to understand and maintain.

you can implement multipule interfaces : multipule inheritance درباره نشد
↳ methods, attributes, ..

but you can't inherit the implementation of multipule classes.
↳ use composition in order to do that.

the diamond problem



when you use multipule inheritance to derive from two classes that ultimately also derive from Employee class. this causes two paths to reach the employee base class. avoid this.
it can cause the wrong version of a method to be called.

reminder → composition : "has-a" relationship. composite class has a component of another class (مثل ربات)

مثال "\n". join(lines)
↳ to combine a sequence of strings stored in the list (lines) into a single string, with each string separated by a newline character (\n)

مثال Composition ⇒ is a loosely coupled relationship that often doesn't require the composite class to have knowledge of the component.

⇒ composition is more flexible than inheritance because it models a loosely coupled relationship

⇒ changes to a component class have minimal or no effects on the composite class.

```python
class CommissionPolicy(SalaryPolicy):
    def __init__(self, weekly_salary, commission_per_sale):
        super().__init__(weekly_salary)
        self.commission_per_sale = commission_per_sale

    @property
    def commission(self):
        sales = self.hours_worked / 5
        return sales * self.commission_per_sale

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

→ چرا اینجا از @property استفاده کرده؟

The .commission_per_sale is used to calculate the .commission, which is implemented as a property so it gets calculated when requested

اگه از @property استفاده نمی‌کردیم باید به .commission به صورت یک متد دسترسی پیدا می‌کردیم ← CommissionPolicy.commission()

attribute access ← دسترسی داریم Commission Policy. commission به
with function_like behavior.

policy based design

✓ in python you can acheive it through composition

✓ classes are composed of policies and they delegate to

those policies to do the work.

Customizing Behavior with Composition

✓ if your design relies on Inheritance: change the object to change its behavior

But with Composition: change the policy that the object uses to

Liskove's substitution principle (on Inheritance)

if you are able to justify the relationship between two classes

both ways then you shouldn't derive one class from another.

(the rectangle and square example)

A `Square` **is a** `Rectangle` because its area is calculated from the product of its `.height` times its `.length`. The constraint is that `Square.height` and `Square.length` must be equal.

It makes sense. You can justify the relationship and explain why a `Square` **is a** `Rectangle`. Now reverse the relationship to see if it makes sense.

A `Rectangle` **is a** `Square` because its area is calculated from the product of its `.height` times its `.length`. The difference is that `Rectangle.height` and `Rectangle.width` can change independently.

It also makes sense. You can justify the relationship and describe the special constraints for each class. This is a good sign that these two classes should never derive from each other.

Mixing features with Mixin Classes

✓ A mixin class: provides methods to other classes but is not a base class

✓ A mixin allows other classes to reuse its interface and implementation without becoming a superclass.

✓ Mixins are similar to composition but they creat a stronger relationship.

inherited from object class.

use for dicts = use for assignment

↑ ↑

Python representations.py

Key = attribute name

value

```python
class AsDictionaryMixin:
    def to_dict(self):
        return {
            prop: self._represent(value)
            for prop, value in self.__dict__.items()
            if not self._is_internal(prop)
        }

    def _represent(self, value):
        if isinstance(value, object):
            if hasattr(value, "to_dict"):
                return value.to_dict()
            else:
                return str(value)
        else:
            return value
```

represents the name of the attribute

Returns a dictionary of the object's attr with key = attribute, value = value

convert the attr's value using _represent.

iterate over all attributes

skip internal/privet attributes

checks if the value is an object (all are in python)

check if the object has a "to_dict" method

if so, call "to_dict"

otherwise, convert the value to a string

if the value is not an object, return it as it is.

```python
    def _is_internal(self, prop):
        return prop.startswith("_")
```

check if the attribute name strats with "_"

.to_dict() method => returns a dictionary non-internal attribute with their represented values.

._represent method => helper method, determines how each attribute's value should be represented.

._is_internal method => identifies which attributes are considered "internal" or "privet" and should be excluded from the dictionary. (also a helper method)

* the mixins : used for seralizing data for APIs or debugging.

purpose of the leading underscore (_)

✓ it's a convention

✓ indicates the method or attribute is intended for internal use only.

what does "internal use" mean?

✓ the method or attr is meant to be used only inside the class or its subclasses.

✓ it's not part of the public API, should not be accessed or used by external code.

✓ signals to developers that this is an implementation detail

Python Name Mangling with "__"

✓ is usually reserved for when you want to avoid name conflict in subclasses.

✓ it should start with "__" and not end with "__".

✓ __privet_method in Parent → renamed to _Parent__privet_method

you can access it like this.