Inheritance

Welcome back to CS 2100!

Prof. Rasika Bhalerao

https://www.youtube.com/watch?v=JWzeEHlNu7k

Consider this...

We want to create a class for `Cat` (with properties like `claw_sharpness`, and methods like `knead()`).

We also want to create a class for `Lion`, which has all the functionality of `Cat`, but additional things like a `roar()` method.

Subclasses and superclasses

A *subclass* (or *child* class) is a more specific version of a *superclass* (*parent* class).

The subclass *inherits* all the methods and attributes from the *superclass* and then adds more that are specific to it.

Syntax: when declaring the subclass, put the superclass's name in parentheses: `class Lion(Cat):`

```python
class Student():
    def __init__(self, student_id: str, major: str):
        self.id = student_id
        self.major = major
        self.courses: Set[str] = set()

    def attend_lab(self, course_id: str) -> None:
        if course_id in self.courses:
            print(f'Attending {course_id}\' lab')

    def register_courses(self, courses: Set[str]) -> None:
        self.courses |= courses

class UndergraduateStudent(Student):
    def change_major(self, new_major: str) -> None:
        self.major = new_major
```

Only undergrads can switch majors.

Subclasses override methods from their superclasses

A subclass inherits all of the methods and instance variables from its superclass.

- Can add more
  - `UndergraduateStudent` adds `change_major()`
  - `Lion` adds `roar()`
- Can overwrite what it inherits from the superclass
- Actually, it inherits all of the methods and instance variables except those that are named with two underscores.

Subclasses override methods from their superclasses

```python
class Cat:
    def __init__(self, name: str):
        self.name = name
        self.food = ['tuna', 'chicken']

    def knead(self) -> None:
        print('Kneading')

    def eat(self, food: str) -> None:
        if food in self.food:
            print(f'Eating {food}')
```

```python
class Lion(Cat):
    def roar(self) -> None:
        print('Roaring')

    def eat(self, food: str) -> None:
        if food in self.food + ['zebra']:
            print(f'Eating {food}')
```

```python
class HouseCat(Cat):
    def purr(self) -> None:
        print('Purring')
```

```python
class Button:
    def __init__(self, fancy: bool):
        self.fancy = fancy

class Shirt:
    def __init__(self, size: int):
        self.size = size
        self.buttons: list[Button] = []

    def add_button(
            self, b: Button
    ) -> None:
        self.buttons.append(b)

    def fold(self) -> None:
        print('Folding')

class FormalShirt(Shirt):
    def add_button(
            self, b: Button) -> None:
        if b.fancy:
            self.buttons.append(b)
```

Poll: What happens?

```python
s = FormalShirt(500)
s.fold()
```

1. It raises an error
2. Cannot do that - won't run
3. It calls `Shirt`'s `fold()` method
4. It does nothing

Using `super()`

Look for the redundancy:

```python
class Shirt:
    def __init__(self, size: int):
        self.size = size
        self.buttons: List[Button] = []

    def add_button(self, button: Button) -> None:
        self.buttons.append(button)

class FormalShirt(Shirt):
    def add_button(self, button: Button) -> None:
        if button.fancy:
            self.buttons.append(button)
```

Same code in multiple places -> updating it requires updating in both places -> prone to typos and bugs

Calling a superclass's method

Directly call `Shirt`'s `add_button()` from within `FormalShirt` using `super()`:

```
class Shirt:
    def __init__(self, size: int):
        self.size = size
        self.buttons: List[Button] = []

    def add_button(self, button: Button) -> None:
        self.buttons.append(button)

class FormalShirt(Shirt):
    def add_button(self, button: Button) -> None:
        if button.fancy:
            super().add_button(button)
```

Any changes to `Shirt`'s `add_button()` propogate to `FormalShirt`.

Calling a superclass's method

```python
class Cat:
    def __init__(self, name: str):
        self.name = name
        self.food: set[str] = {'tuna', 'chicken'}

    def knead(self) -> None:
        print('Kneading')

    def eat(self, food: str) -> None:
        if food in self.food:
            print(f'Eating {food}')

class Lion(Cat):
    def roar(self) -> None:
        print('Roaring')

    def eat(self, food: str) -> None:
        self.food |= {'zebra'}
        super().eat(food)
```

Calling a superclass's *constructor*

```python
class Cat:
    def __init__(self, name: str):
        self.name = name
        self.food = ['tuna', 'chicken']

    def knead(self) -> None:
        print('Kneading')

    def eat(self, food: str) -> None:
        if food in self.food:
            print(f'Eating {food}')

class Lion(Cat):
    def __init__(self, name: str):
        super().__init__(name)
        self.food += ['zebra']

    def roar(self) -> None:
        print('Roaring')
```

- `eat()` method inherited from `Cat` works by default in `Lion`
- `self.food` is defined in `Cat`'s constructor, so we overwrite it with a new constructor in `Lion`...
    - one that executes `Cat`'s constructor first, and then adds `'zebra'` to `self.food`

Poll: What does this output?

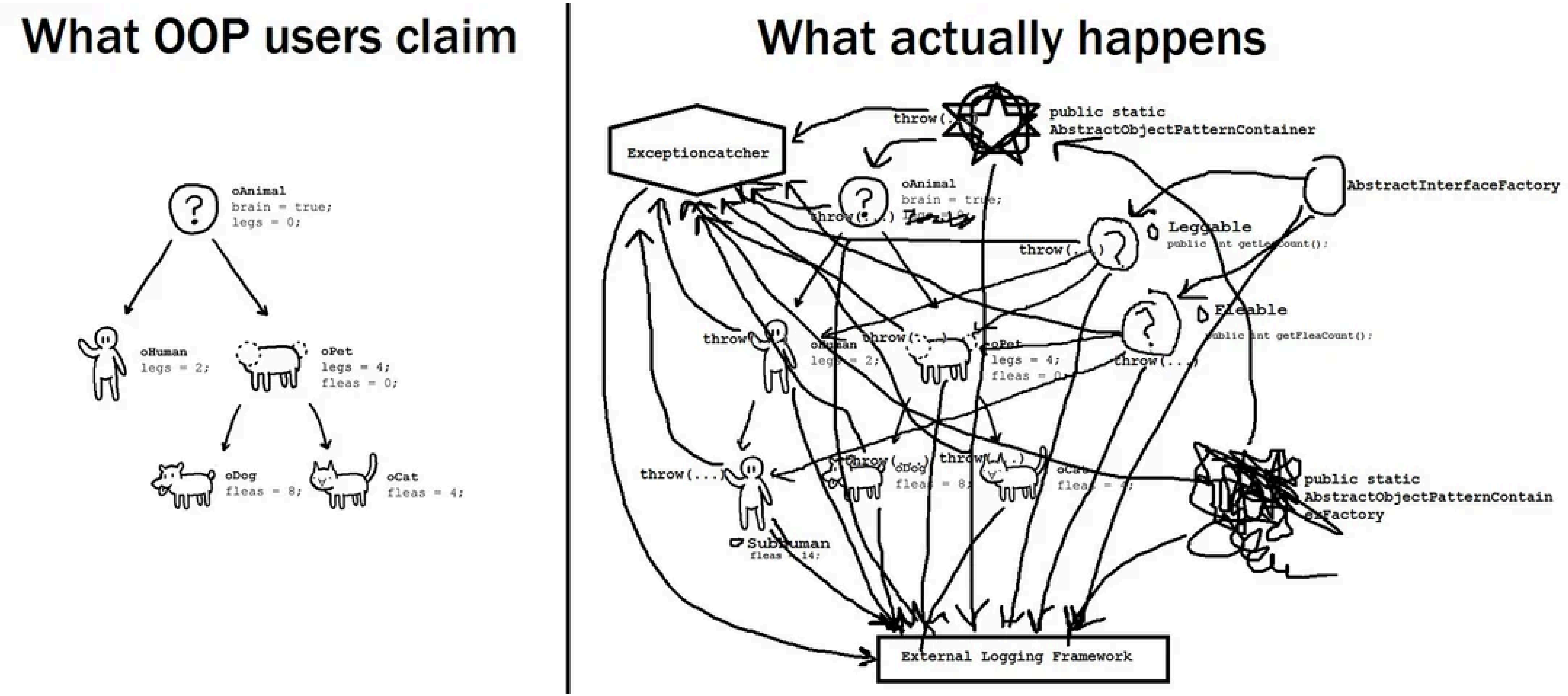
```python
class Cat:
    def __init__(self, name: str):
        self.name = name

    def knead(self) -> None:
        print('Kneading')

class Lion(Cat):
    def knead(self) -> None:
        print('I am a lion')
        super().knead()
```

```python
lion: Cat = Lion('Mini')
lion.knead()
```

1. `Kneading`
2. `I am a lion`
3. `Kneading` // `I am a lion`
4. `I am a lion` // `Kneading`



https://www.reddit.com/r/ProgrammerHumor/comments/60lm55/oop_what_actually_happens

Every class is a subclass of `object`

These two class definitions are equivalent:

```python
class MyClass: pass

class MyClass(object): pass
```

This is why every class has a built-in `__str__()` method: they inherit it from `object`

Every class is a subclass of `object`

`__eq__(self, other) -> bool` is also inherited from `object`

Before overwriting `__eq__()`:

```python
class Student():
    def __init__(self,
            student_id: str, major: str
    ):
        self.id = student_id
        self.major = major
        self.courses: Set[str] = set()

s1 = Student('s1', 'CS')
s2 = Student('s1', 'CS')
print(s1 == s2)  # False
```

After overwriting `__eq__()`:

```python
class Student():
    def __init__(self,
            student_id: str, major: str
    ):
        self.id = student_id
        self._major = major

    def __eq__(
            self, other: object
    ) -> bool:
        if not isinstance(other, Student):
            raise TypeError
        return self.id == other.id

s1 = Student('s1', 'CS')
s2 = Student('s1', 'CS')
print(s1 == s2)  # True
```

Poll: Why is this bad?

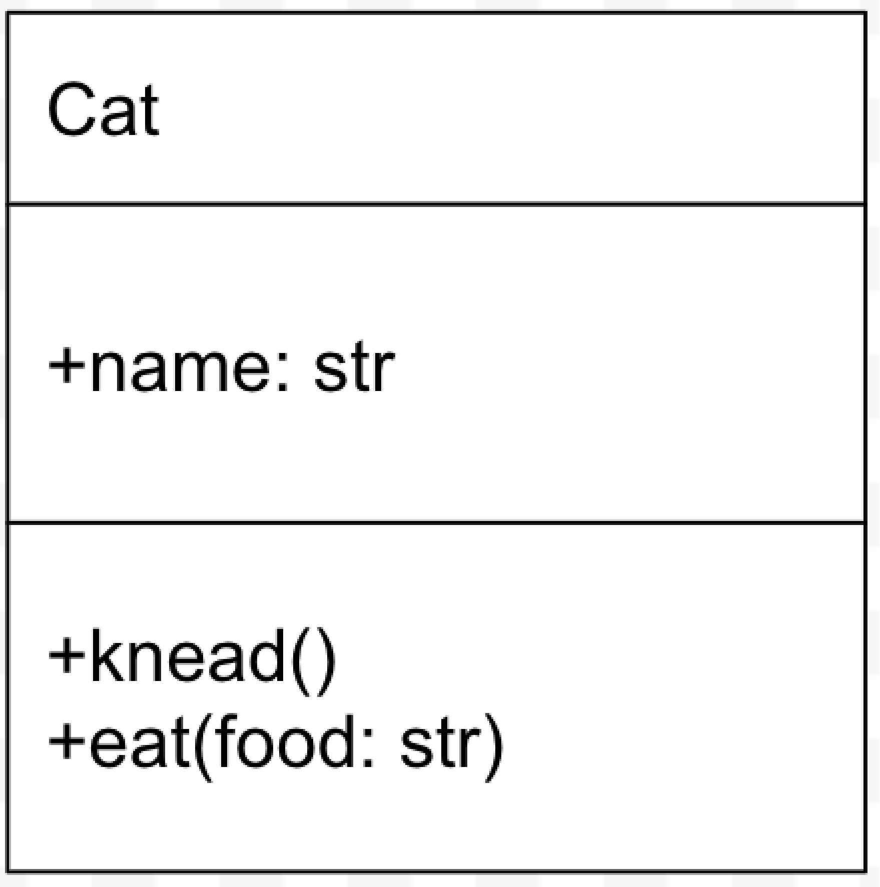
```
class Cat:
    def __init__(self, name: str):
        self.name = name
        self.food: List[str] = ['tuna', 'chicken']

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Cat):
            raise ValueError
        return other.name in self.food
```

1. It's possible for `cat_a` to equal `cat_b` today, but for `cat_a` to not be equal to `cat_b` tomorrow (with no code changes)
2. It's possible for `cat_a` to not equal itself
3. It's possible for `cat_a` to equal `cat_b`, and `cat_b` to not equal `cat_a`
4. All cats will be equal, making the `__eq__()` function useless

UML (Unified Modeling Language) Diagrams

A UML diagram visually shows us the classes and their relationships in a program.



This UML diagram says:

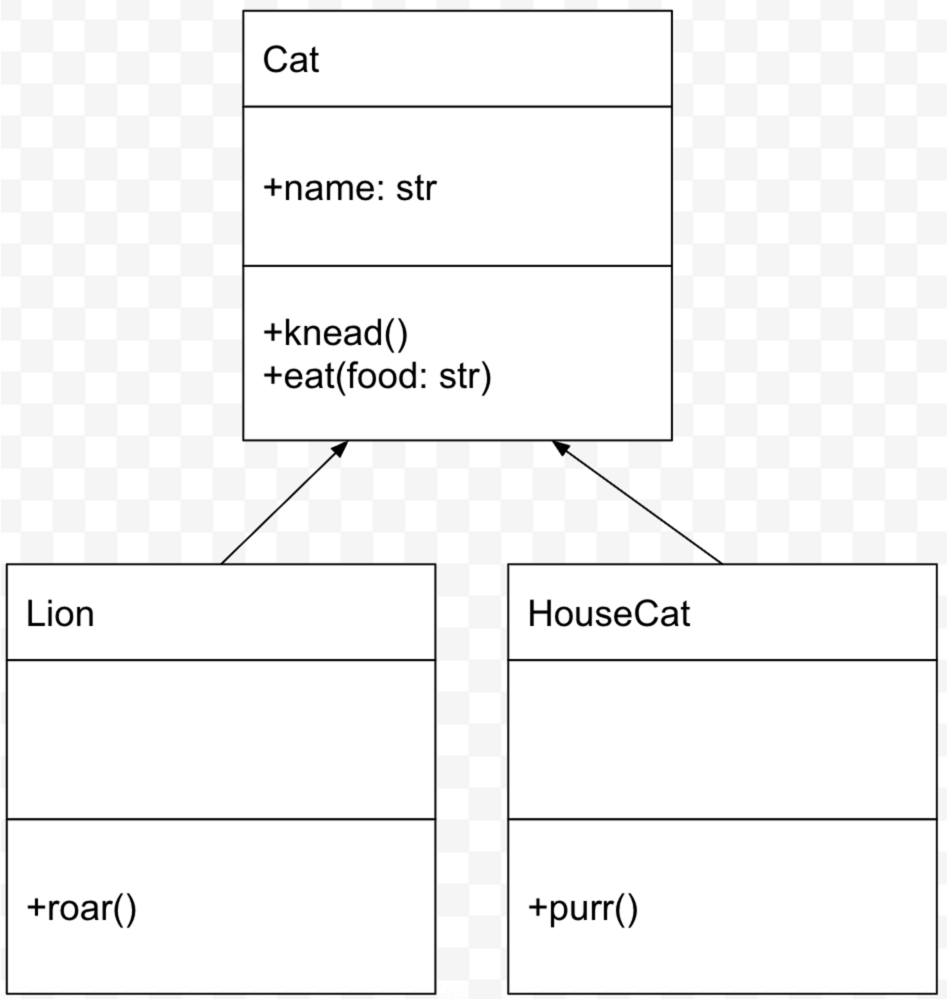
- class name: `Cat`
- `str` attribute called `name`
- method called `knead()`
- method called `eat(food: str)`

`+` indicates that a method or attribute is intended to be publicly available.

`-` (minus) indicates that it is not (two underscores `__`).

UML Diagram: Subclass / Superclass Relationship

Draw an arrow from the subclass to the superclass:



The Liskov Subsitution Principle

"If S is a subtype of T, then objects of T can be substituted with objects of S without altering any of expected functionality."

In other words:

A member of the subclass can be used wherever a member of the superclass is required.

Example:

- you want coffee
- you receive an espresso
- espresso is a subclass of coffee
- you are satisfied because the coffee hierarchy follows the Liskov Substitution Principle

May 7, 2025: TSA requires "Real ID"

Bear + star in top right corner = REAL ID

- Federal compliant? ☑ **YES**
- Use to board domestic flights after the new May 7, 2025 deadline? ☑ **YES**
- Use to enter secure federal facilities after the new May 7, 2025 deadline? ☑ **YES**

Federal Limits Apply = Not REAL ID

- Federal compliant? ☒ **NO**
- Use to board domestic flights after the new May 7, 2025 deadline? ☒ **NO**
- Use to enter secure federal facilities after the new May 7, 2025 deadline? ☒ **NO**
- * May be used as photo identification, but not as evidence of legal presence in U.S. Additional documentation may be required.

https://www.dmv.ca.gov/portal/driver-licenses-identification-cards/real-id/what-is-real-id/

Poll: Which are true?

1. The Real ID requirements follow the Single Responsibility Principle because one object covers multiple uses (TSA and driving)
2. The Real ID requirements break the Single Responsibility Principle because the same object is used for multiple unrelated activities (TSA and driving)
3. The Real ID requirements follow the Liskov Substitution Principle because anywhere that the old ID is used, the new (more specific one) can be used instead
4. The Real ID requirements break the Liskov Substitution Principle because there are things that the Real ID can do that the old ID (less specific one) cannot

Poll:

1. What is your main takeaway from today?
2. What would you like to revisit next time?