Lecture 2

•••

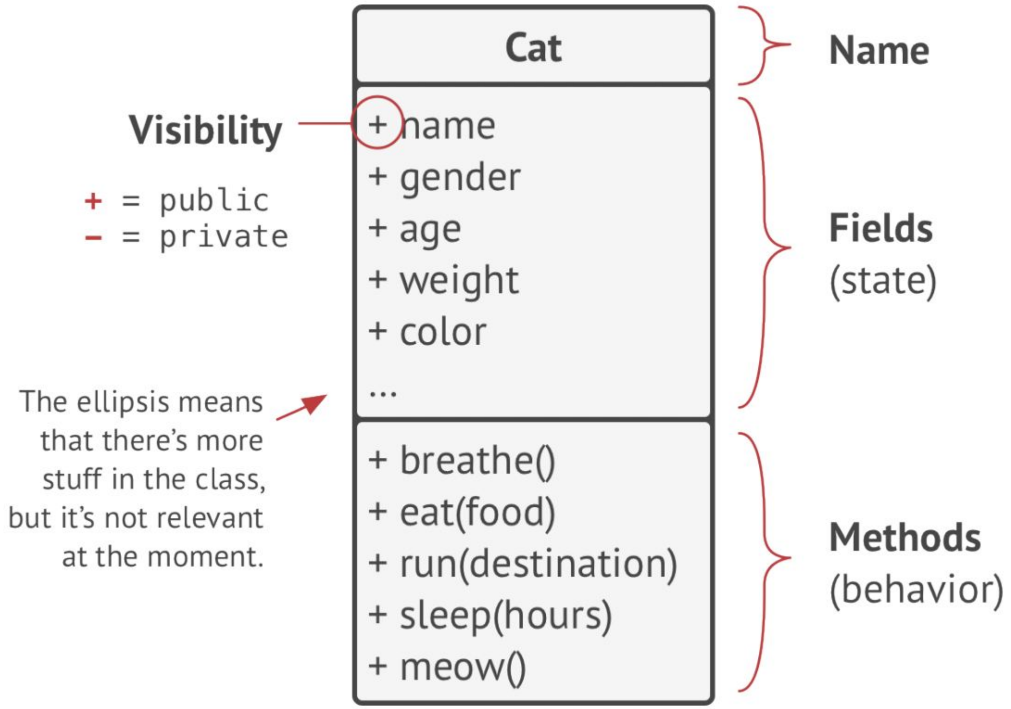
Object-Oriented Programming

Object Oriented Programming

- Object-oriented programming is a paradigm for organizing code
- Idea: wrapping pieces of data, and behavior related to that data, into special bundles called **objects**.
    - In python, these are usually classes
- Objects are constructed from a set of “blueprints”, defined by a programmer, called **classes**.
- Object - Oriented - Programming is usually abbreviated as “OOP”

OOP: Cat example

OOP: Cat example

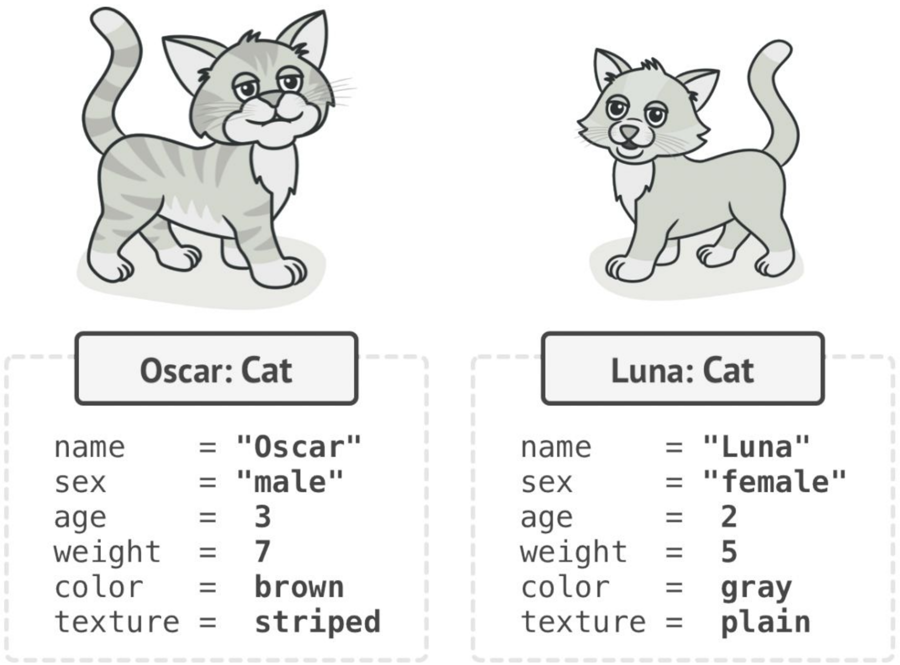


OOP: Cat example

- Say you have a cat named Oscar. Oscar is an **object**, an **instance** of the Cat class.
- Every cat has a lot of standard attributes:
  - name, sex, age, weight, color, favorite food, etc.
  - These are the class's **fields**.
- All cats also behave similarly:
  - breathe, eat, run, sleep, meow.
  - These are the class's **methods**.
- Collectively, fields and methods can be referenced as the members of their class.
- Data stored inside the object's fields is often referenced as **state**, and all the object's **methods** define its **behavior**.

OOP: Cats

- Oscar and Luna are both Instances of the cat class.
- So a class is like a blueprint:
  - defines the structure for **objects**, which are concrete **instances** of that class.



```
name     = "Oscar"
sex      = "male"
age      =  3
weight   =  7
color    =  brown
texture  =  striped
```

```
name     = "Luna"
sex      = "female"
age      =  2
weight   =  5
color    =  gray
texture  =  plain
```

Objects are instances of classes.

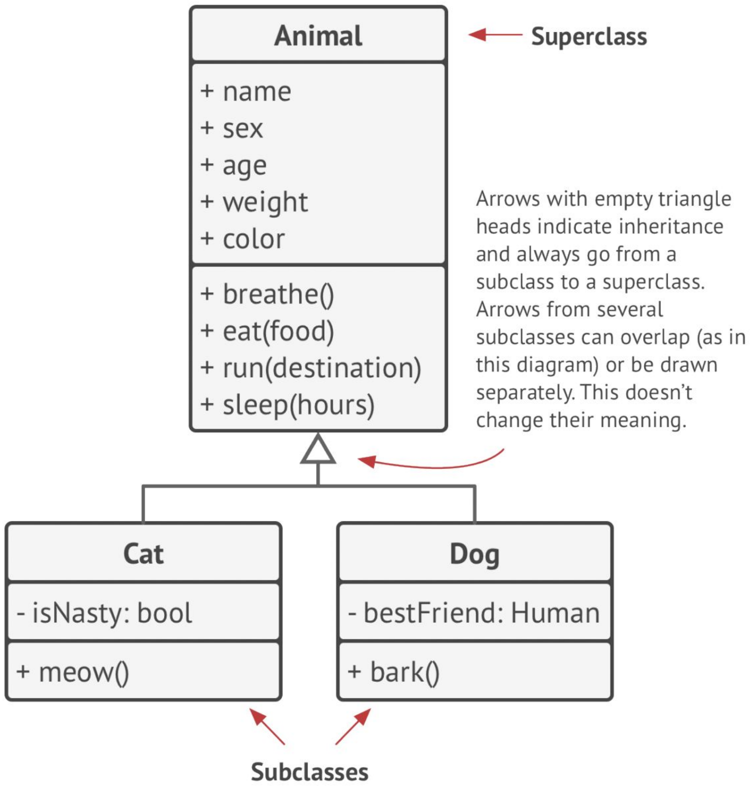
OOP: Multiple Classes and hierarchy

- A real program contains more than a single class.
- Some of these classes might be organized into **class hierarchies.**
- Let’s find out what that means.
- Imagine we want to model a household with Dogs for pets, as well as Cats…

OOP: Multiple Classes and hierarchy

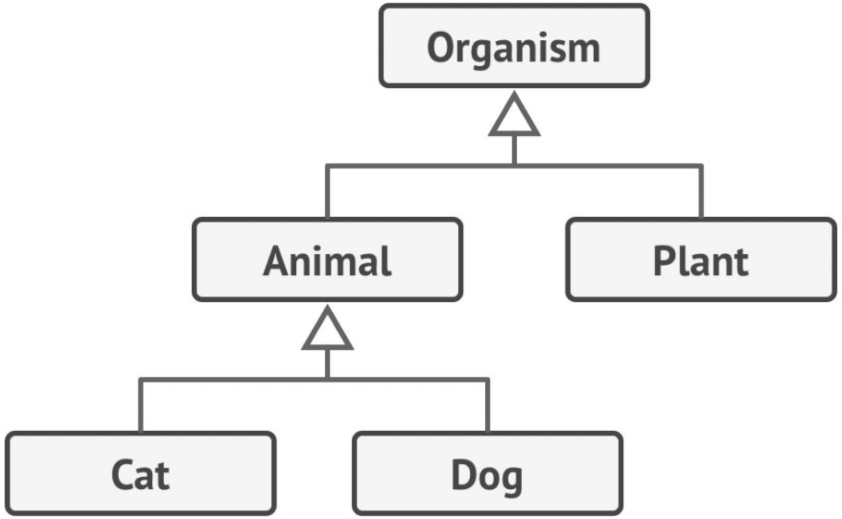
- Dogs and Cats have a lot in common:
    - name, sex, age, and color are attributes of both dogs and cats.
    - Breathing, sleeping, running are common behaviours between both
- Define the base Animal class that lists the common attributes and behaviors.
- A parent class, like the one we've just defined, is called a **superclass**.
- Its children are **subclasses**.
- Subclasses inherit state and behavior from their parent, **defining only attributes or behaviors that differ.**
    - Thus, the Cat class would have the **meow** method, and the Dog class the **bark** method.

OOP: Multiple Classes and hierarchy


Animal
+ name
+ sex
+ age
+ weight
+ color
+ breathe()
+ eat(food)
+ run(destination)
+ sleep(hours)
Superclass
Arrows with empty triangle heads indicate inheritance and always go from a subclass to a superclass. Arrows from several subclasses can overlap (as in this diagram) or be drawn separately. This doesn't change their meaning.
Cat
- isNasty: bool
+ meow()
Dog
- bestFriend: Human
+ bark()
Subclasses

OOP: Hierarchy

- We can go even further!
- Extract a more general class for all living Organisms
  - Superclass for Animals and Plants
- Such a pyramid of classes is a **hierarchy**.
- The Cat class inherits everything from **both** the Animal and Organism classes.



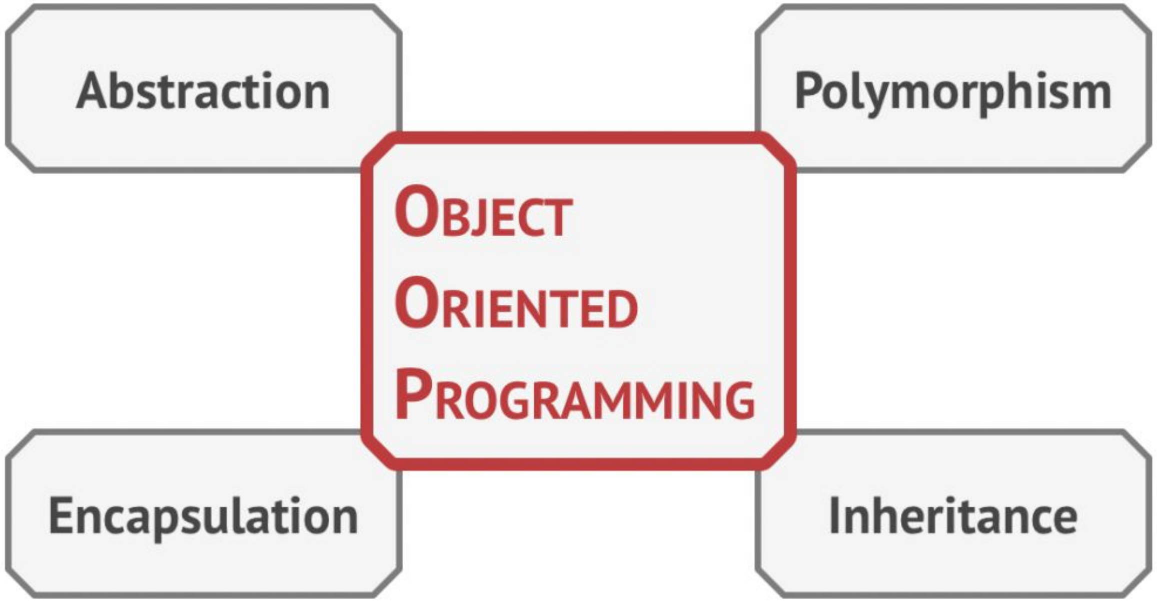
OOP: Hierarchy

- Subclasses can **override** the behavior of methods that they inherit from parent classes.
- A subclass can either completely replace the default behavior or just enhance it with some extra stuff.
- This allows you to have a default which makes sense in the general case, which can be overridden in some subclasses

OOP: What to model

- In general, you only model out the classes that you need for the project's functionality.
- YAGNI principle: You Ain't Gonna Need It:
  - only build the things which you're **sure** you have a business need for!

Pillars of OOP

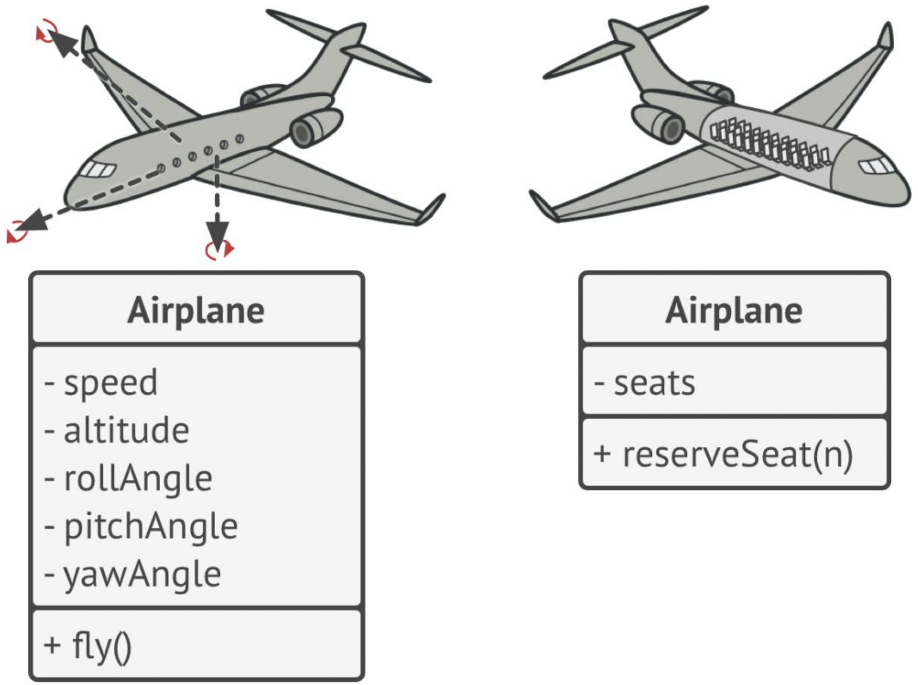


Abstraction

- Abstraction is a model of a real-world object or phenomenon
  - Limited to a specific context
  - Represents all details relevant to this context with high accuracy
  - Omits all the rest
- Sometimes: shape objects of the program based on real-world objects.
  - However, objects of the program don't represent the originals with 100% accuracy (and it's rarely required that they do).
- Instead, your objects only model attributes and behaviors of objects in a **specific context**, ignoring the rest.

Abstraction

- Example: **Airplane** class
- Flight simulator vs flight booking
  - Flight simulator app:
    - details related to the actual flight physics
  - Flight booking app:
    - seat map
    - which seats are available



Different models of the same real-world object.

Encapsulation

- To start a car engine, you only need to turn a key or press a button.
    - Don’t need to connect wires under the hood, rotate the crankshaft and cylinders, and initiate the power cycle of the engine.
- These details are **hidden under the hood of the car.**
- You have only a **simple interface**
    - a start switch, a steering wheel, and some pedals.
- This illustrates how each **object** has an **interface**
    - a **public part** of an object, **open to interactions** with other objects.

Encapsulation and Interfaces

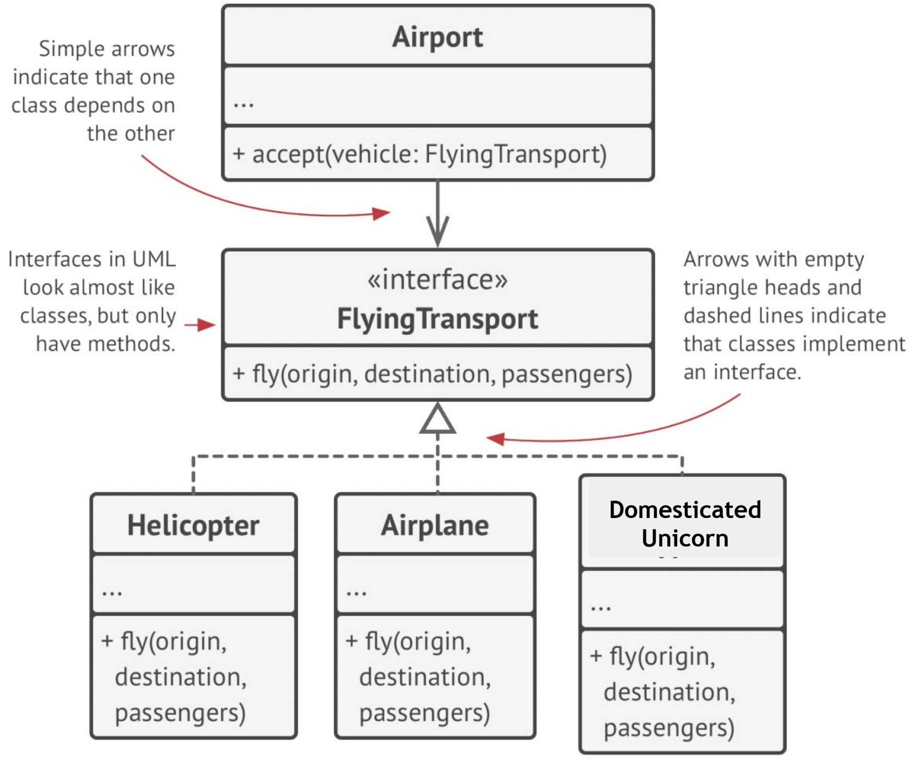
- **Encapsulation**: the ability to hide parts of its state and implementation from other objects, exposing only limited parts of the object
- **Interface**: the exposed parts. An interface defines the contract between this object and other objects it interacts with.
- Multiple classes can implement the same interface – that is, multiple classes can expose the same, standard set of functions and properties.
  - Example coming!

Encapsulation and Interfaces

Imagine a `FlyingTransport` **interface** with a method `fly(origin, destination, passengers)`

When designing an `Airport` class, restrict it to only work with objects that implement the `FlyingTransport` interface.

This ensures that any object passed to an Airport object, whether it's an `Airplane`, a `Helicopter` or a `DomesticatedUnicorn` would be able to arrive or depart from this type of airport.


Simple arrows indicate that one class depends on the other
Airport
...
+ accept(vehicle: FlyingTransport)
Interfaces in UML look almost like classes, but only have methods.
«interface»
FlyingTransport
+ fly(origin, destination, passengers)
Arrows with empty triangle heads and dashed lines indicate that classes implement an interface.
Helicopter
...
+ fly(origin, destination, passengers)
Airplane
...
+ fly(origin, destination, passengers)
Domesticated Unicorn
...
+ fly(origin, destination, passengers)

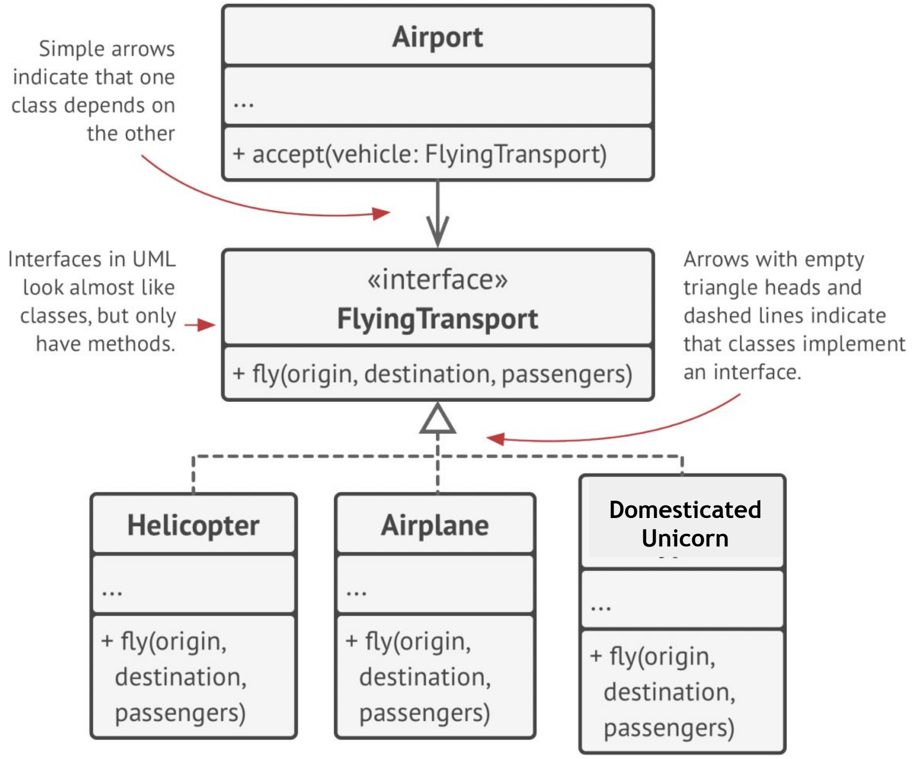
Encapsulation and Interfaces

Benefit of Interface:

You could change the implementation of the `fly` method in these classes in any way you want.

As long as the **signature** of the method remains the same as declared in the **interface**, all instances of the Airport class can work with your flying objects just fine.

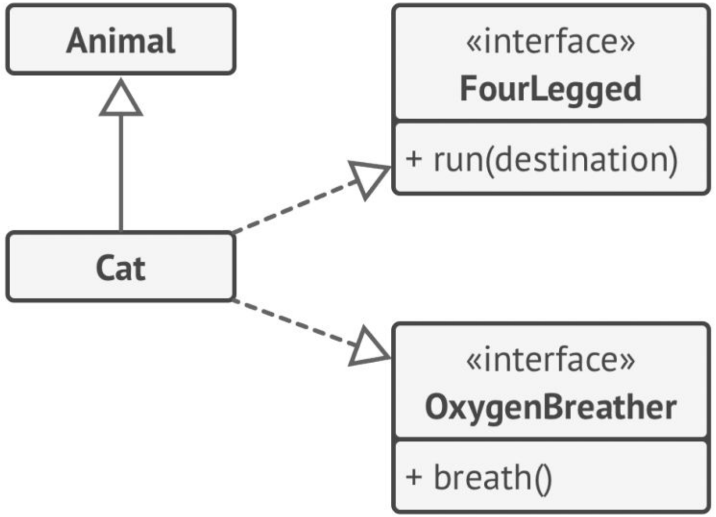
The "contract" that the interface provides gives you structured flexibility.



Simple arrows indicate that one class depends on the other
Airport
...
+ accept(vehicle: FlyingTransport)
Interfaces in UML look almost like classes, but only have methods.
«interface»
FlyingTransport
+ fly(origin, destination, passengers)
Arrows with empty triangle heads and dashed lines indicate that classes implement an interface.
Helicopter
...
+ fly(origin, destination, passengers)
Airplane
...
+ fly(origin, destination, passengers)
Domesticated Unicorn
...
+ fly(origin, destination, passengers)

Encapsulation and Interfaces

- A single class can implement multiple interfaces.
- Subclasses always implement their Superclass's interface
- But not vice versa!

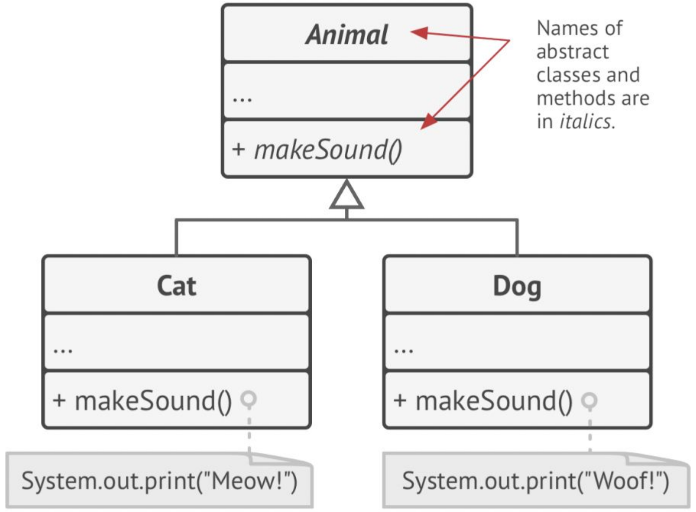


Inheritance

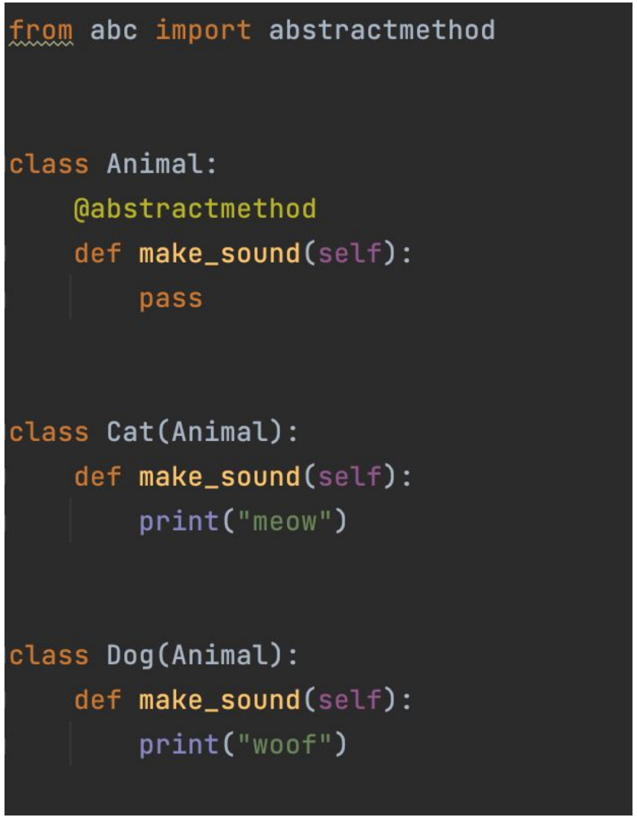
- Inheritance is the ability to build new classes on top of existing ones.
- The main benefit of inheritance is code reuse.
- If you want to create a class that's slightly different from an existing one, there's no need to duplicate code.
    - Put the extra functionality into a resulting subclass, which inherits fields and methods of the superclass.
    - Animal, Dog, and Cat classes are a good example of this

Abstract Methods

- Most `Animals` can make sounds.
- All subclasses will need to override the base `makeSound` method so each subclass can emit the correct sound
- Declare the method **abstract**
  - Lets us omit any default implementation of the method in the superclass
  - Forces all subclasses to come up with their own.


Animal
...
+ *makeSound()*
Names of abstract classes and methods are in *italics*.
Cat
...
+ makeSound()
System.out.print("Meow!")
Dog
...
+ makeSound()
System.out.print("Woof!")

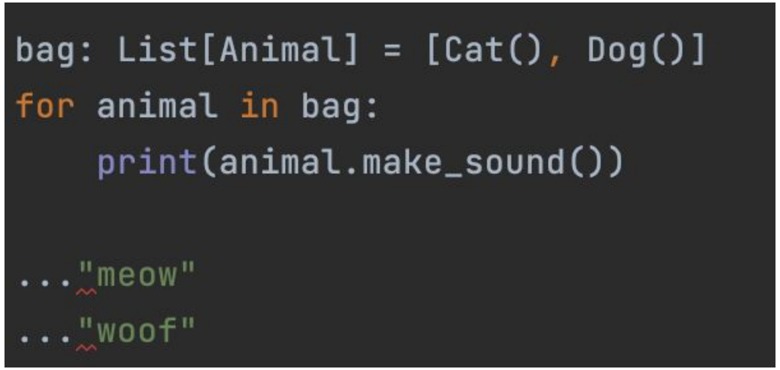
Abstract Methods in Python



```python
from abc import abstractmethod


class Animal:
    @abstractmethod
    def make_sound(self):
        pass


class Cat(Animal):
    def make_sound(self):
        print("meow")


class Dog(Animal):
    def make_sound(self):
        print("woof")
```

Polymorphism

- Polymorphism is the ability of a program to:
  - Detect the real class of an object
  - Call its implementation
    - Even when the actual type is unknown in the current context.
- The program can trace down the subclass of the object whose method is being executed and run the appropriate behavior.
- You can also think of polymorphism as the ability of an object to “pretend” to be something else
  - Pretend to be a **class** it extends or an **interface** it implements.
  - In our example, the dogs and cats in the bag are “pretending” to be generic animals.



```
bag: List[Animal] = [Cat(), Dog()]
for animal in bag:
    print(animal.make_sound())


..."meow"
..."woof"
```

How can classes interact with each other?

- The first way is **inheritance**, which we have seen a lot of this lecture.
- There’s an alternative to inheritance called **composition**.
  - **inheritance** represents the “is a” relationship between classes (a car is a transport),
  - **composition** represents the “has a” relationship (a car has an engine).

Why not just use inheritance?

- Inheritance is probably the most obvious and easy way of reusing code between classes.
    - You have two classes with the same code.
    - Create a common base class for these two and move the similar code into it.
- Unfortunately, inheritance comes with caveats that often become apparent only after your program already has tons of classes and changing anything is pretty hard.
- Here’s a list of those problems:

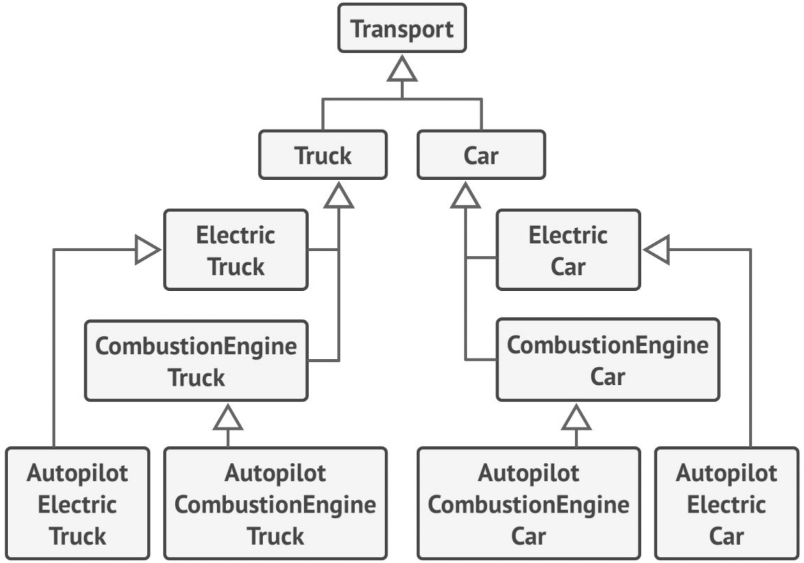
Why not use inheritance

- A subclass can't **reduce** the interface of the superclass.
  - You have to implement all abstract methods of the parent class even if you won't be using them.
- When overriding methods, you need to make sure that the new behavior is compatible with the base one.
  - Objects of the subclass may be passed to **any code** that expects objects of the superclass
  - And you don't want that code to break!
- Trying to reuse code through inheritance can lead to creating parallel inheritance hierarchies.
  - Inheritance usually takes place in a single dimension.
  - But whenever there are two or more dimensions, you have to create lots of class combinations
  - This bloats the class hierarchy to a ridiculous size.
  - Example:

Inheritance: Bloating

- Imagine that you need to create a catalog app for a car manufacturer.
    - The company makes both cars and trucks
    - Can be either electric or gas
    - Can have either manual controls or an autopilot.

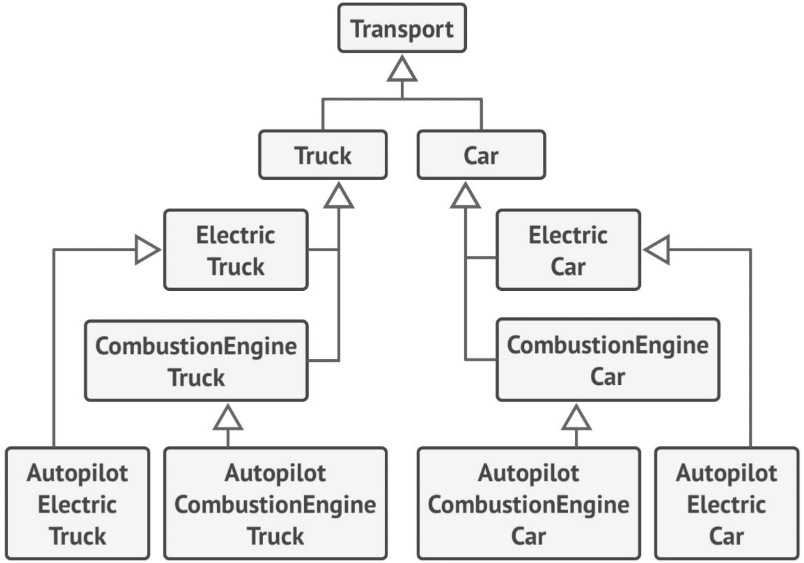
Inheritance: Bloating



INHERITANCE: extending a class in several dimensions (cargo type × engine type × navigation type) may lead to a combinatorial explosion of subclasses.

Inheritance: Bloating

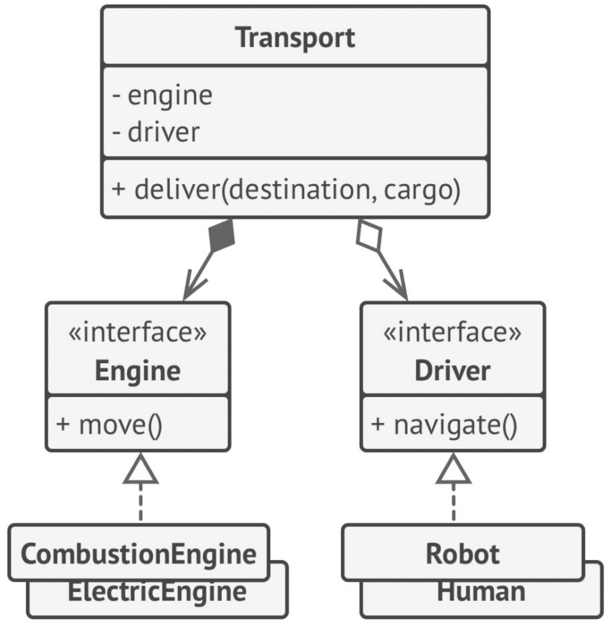
- As you see, each additional parameter results in multiplying the number of subclasses.
    - There's a lot of duplicate code between subclasses because a subclass can't extend two classes at the same time.
- You can solve this problem with **composition**.
    - Instead of car objects implementing a behavior on their own, they can delegate it to other objects.



INHERITANCE: extending a class in several dimensions (cargo type × engine type × navigation type) may lead to a combinatorial explosion of subclasses.

Composition (with interfaces)

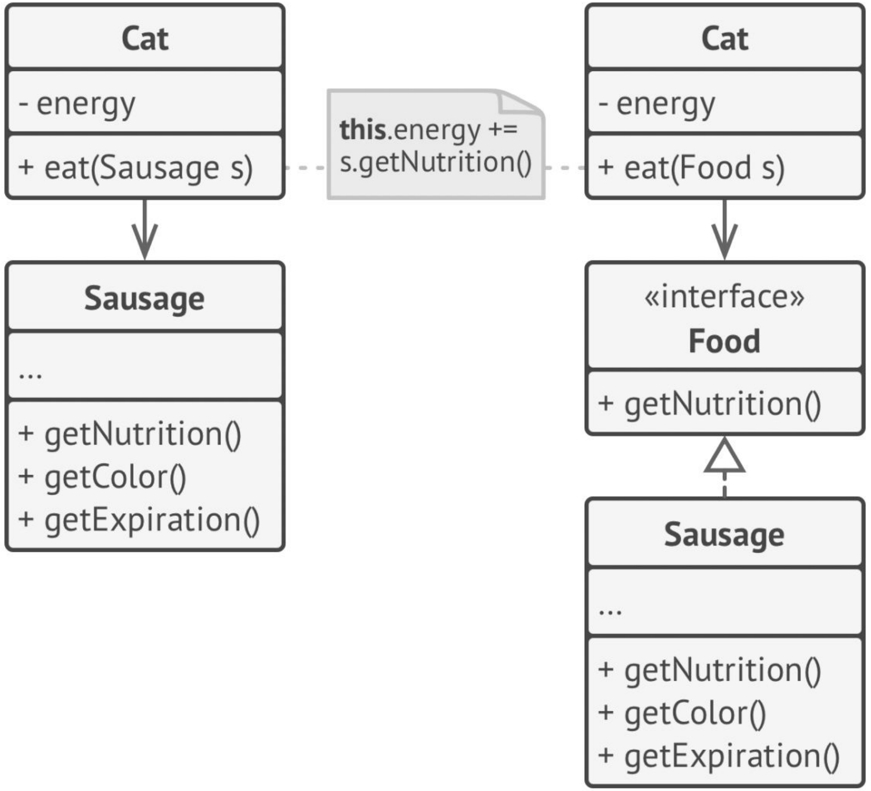
- Composition of classes allows you to separate Engine and Driver into separate classes
- The added benefit is that you can replace or add a behavior easily.
  - You can replace an engine object linked to a car object just by assigning a different engine object to the car.
- Key idea: extract different "dimensions" to their own class hierarchies



COMPOSITION: different "dimensions" of functionality extracted to their own class hierarchies.

When to use Interfaces?

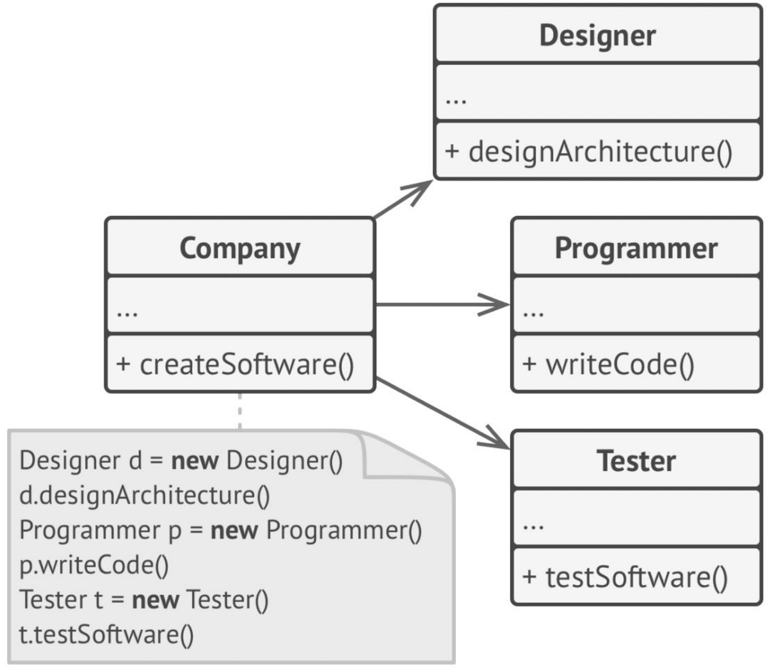
- The code on the left uses direct dependency.
  - Less flexible
  - Less complicated
- The code on the right uses an interface.
  - More easily extensible
  - More complicated



When to use Interfaces?

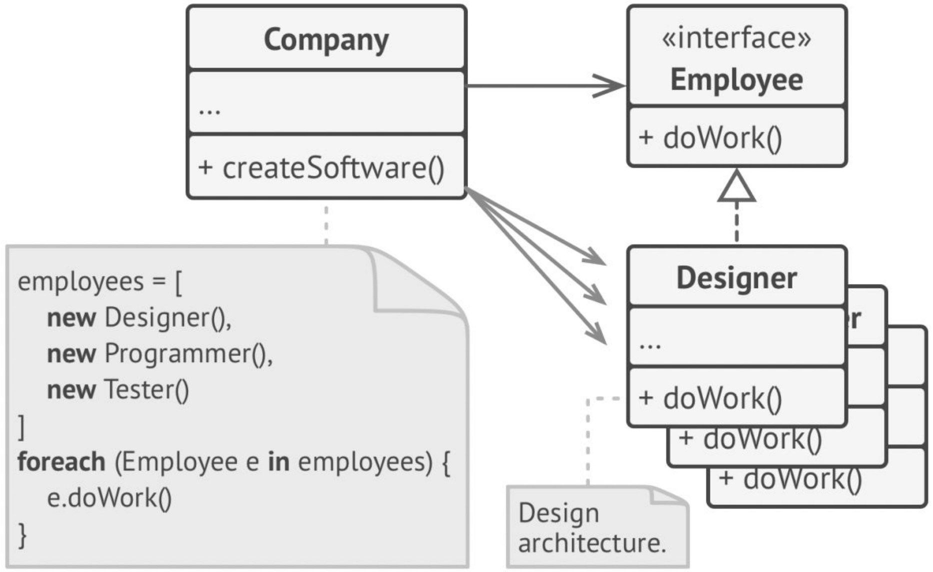
- When you want to make two classes collaborate, start by simply making one of them dependent on the other.
- Change to an interface when:
    - You need to add another extension
        - the cat starts eating something other than Sausage
    - You know already that this will need extension
        - you plan, in the near future, to feed the cat something other than Sausage
    - You want to encourage some other people to extend in a specific way
        - You aren’t going to feed the cat anything other than Sausage, but you want others who contribute to the codebase to feel empowered to feed the cat other things.

Interfaces can make code more readable



```
Designer d = new Designer()
d.designArchitecture()
Programmer p = new Programmer()
p.writeCode()
Tester t = new Tester()
t.testSoftware()
```

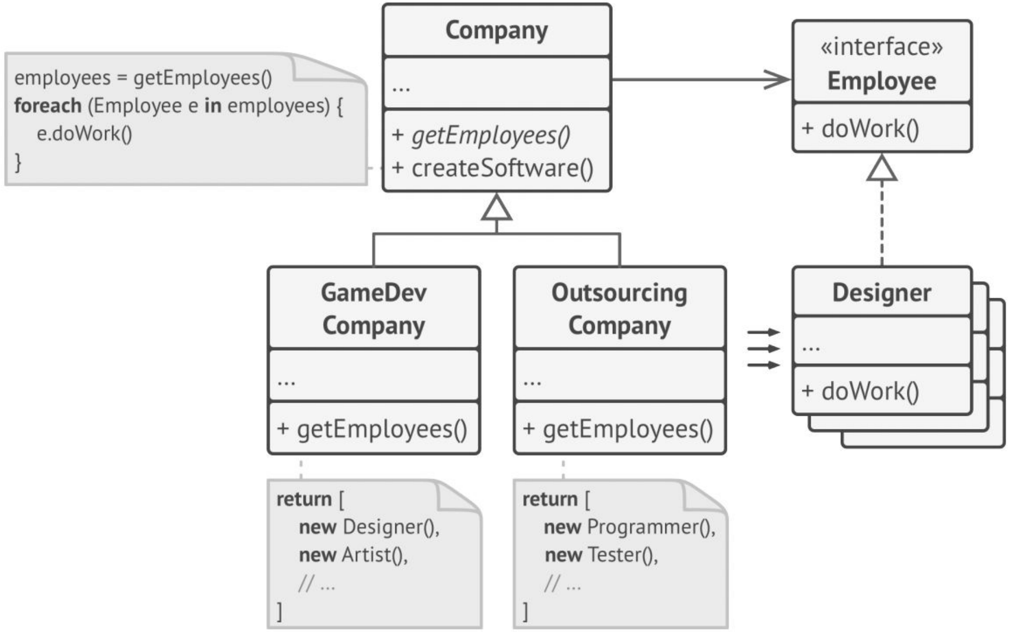
Interfaces can make code more readable



```
employees = [
   new Designer(),
   new Programmer(),
   new Tester()
]
foreach (Employee e in employees) {
   e.doWork()
}
```

Interfaces can make code more scalable

- Company class now independent from various employee classes.
- Can introduce new types of companies and employees
  - while still reusing a portion of the base company class.
- Extending the base company class doesn't break any existing code that already relies on it.



```
employees = getEmployees()
foreach (Employee e in employees) {
  e.doWork()
}
```

```
return [
  new Designer(),
  new Artist(),
  // ...
]
```

```
return [
  new Programmer(),
  new Tester(),
  // ...
]
```

Python: Interfaces

```
class EmployeeInterface:

  def do_work(self):

    """Does a unit of work."""

    pass


class Designer(Employee):

  def do_work(self):

    """Designs one webpage mockup"""

     return self._design_webpage()

  def _design_webpage(self):

   ...
```

OOP In-Class Activity

- Use Excalidraw to make a system diagram, similar to those shown today, of a Car.
- Your diagram should include at least three classes (more is fine).
  - The diagram should show Classes and their methods or instance variables, similar to those we saw today in lecture.
  - Feel free to ask me for help or guidance – this is for learning, not testing :)
  - Also ok to discuss with a classmate!
- The car should be able to:
  - Check gas
  - Check tire pressure
  - Fill gas
  - Change tire
  - Check engine
  - Repair engine
- When done: show me!