Software Design Patterns

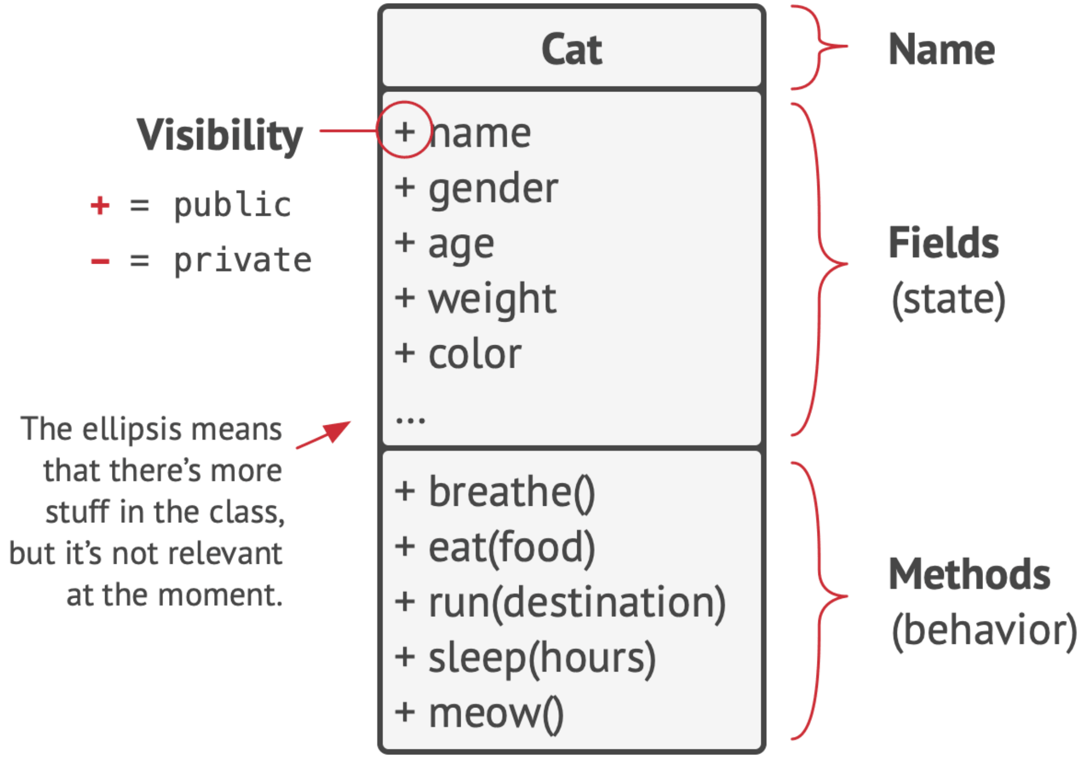
Lecture 1

OOP and UML Class Diagrams
OOP with Java

Dr. Fan Hongfei
5 September 2024

Object-Oriented Programming

- A programming paradigm
- Classes and objects



Cat

Visibility

+ = public

– = private

+ name
+ gender
+ age
+ weight
+ color
...

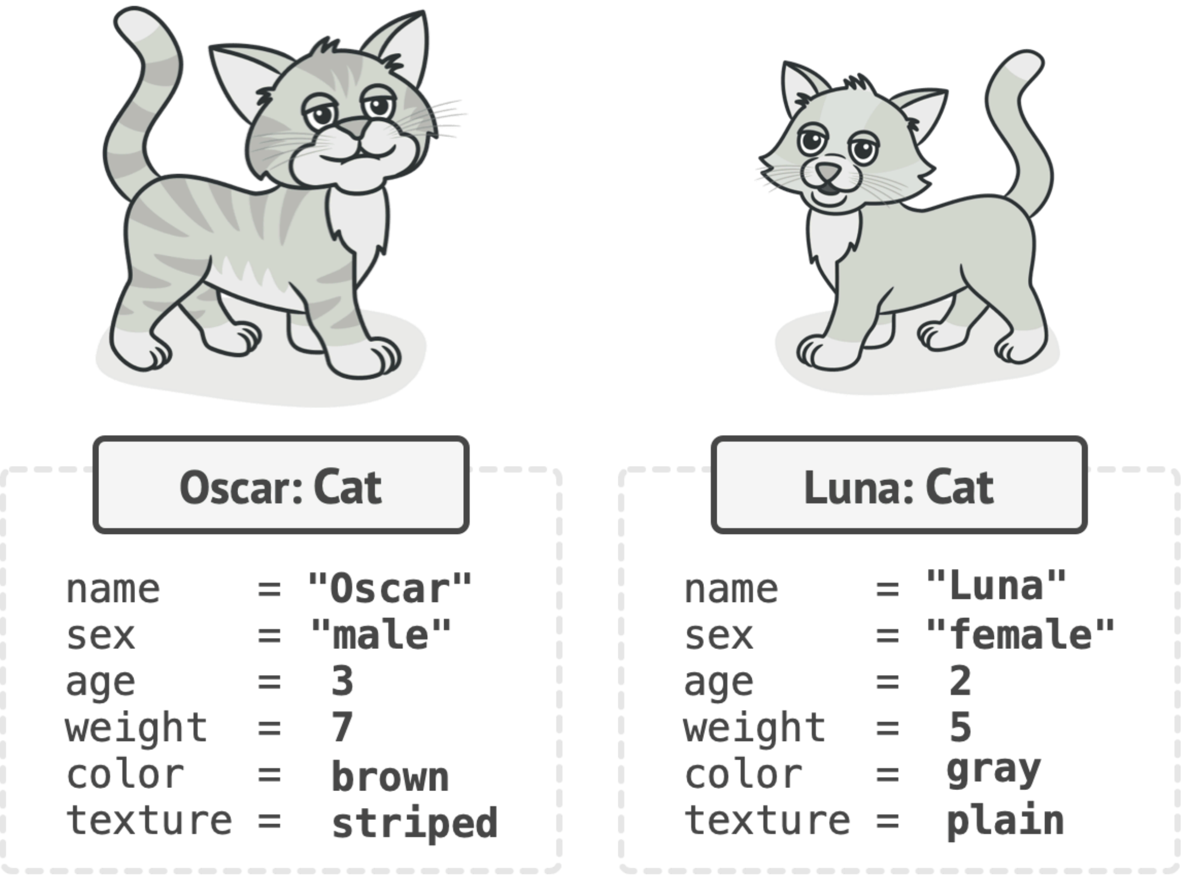
The ellipsis means that there's more stuff in the class, but it's not relevant at the moment.

+ breathe()
+ eat(food)
+ run(destination)
+ sleep(hours)
+ meow()

Name

Fields (state)

Methods (behavior)



Oscar: Cat

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

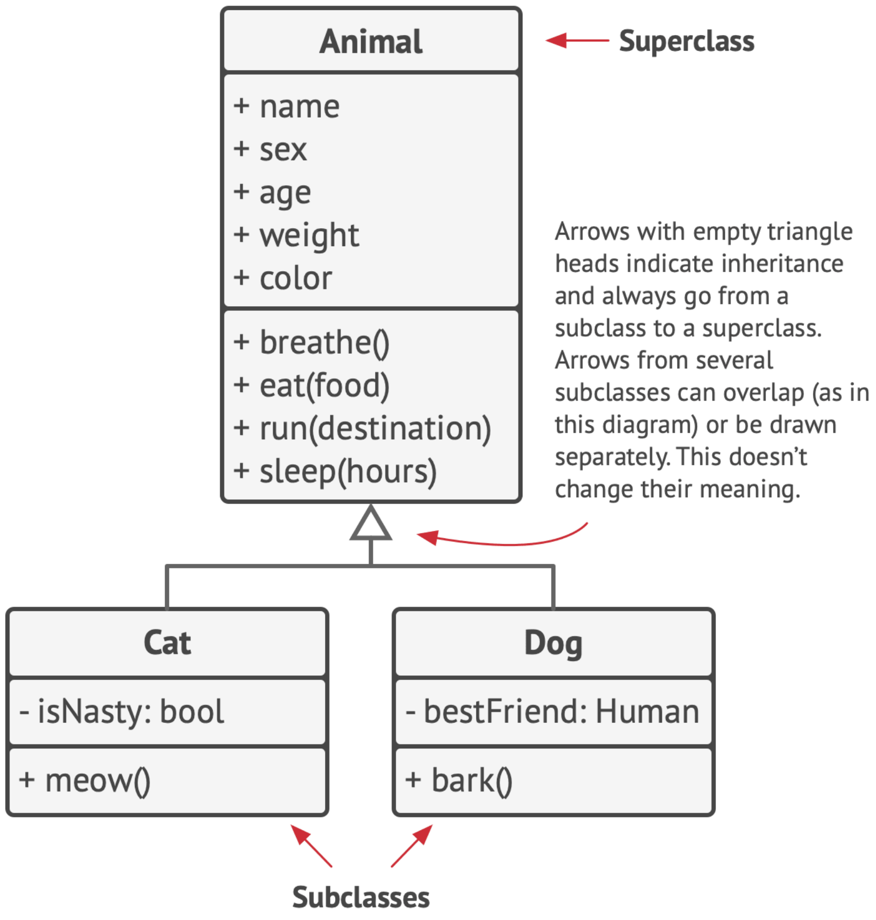
Luna: Cat

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

[Figures in this slide are extracted from "Dive Into Design Patterns" by Alexander Shvets]

Class Hierarchies

- Superclass and subclass



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

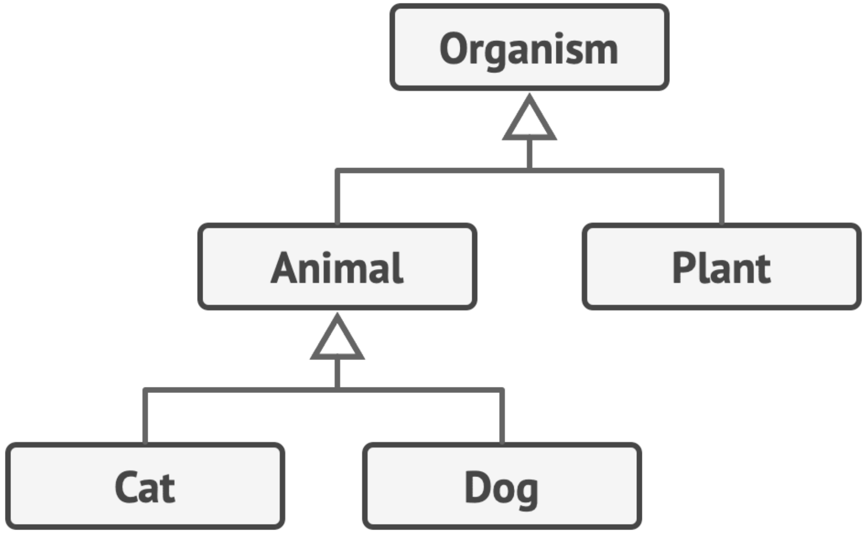
+ meow()

Dog

- bestFriend: Human

+ bark()

Subclasses



Organism

Animal

Plant

Cat

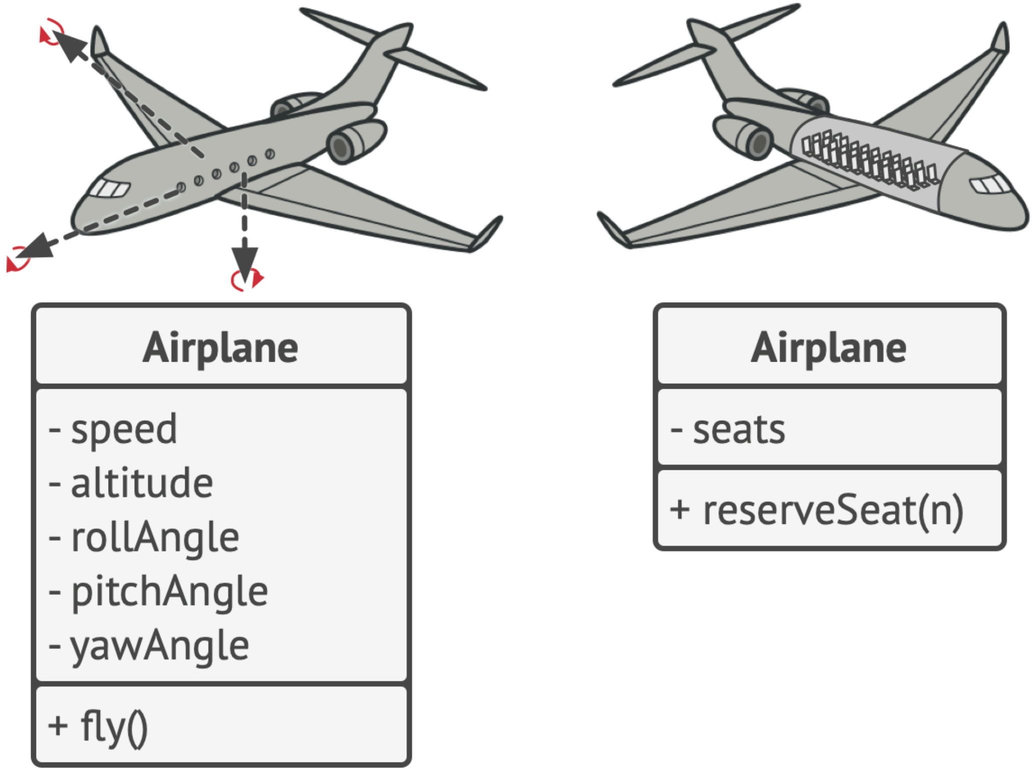
Dog

[Figures in this slide are extracted from “Dive Into Design Patterns” by Alexander Shvets]

Pillars of Object-Oriented Programming

1) Abstraction

– Modelling attributes and behaviors of real objects, in specific contexts



Airplane

- speed
- altitude
- rollAngle
- pitchAngle
- yawAngle

+ fly()

Airplane

- seats

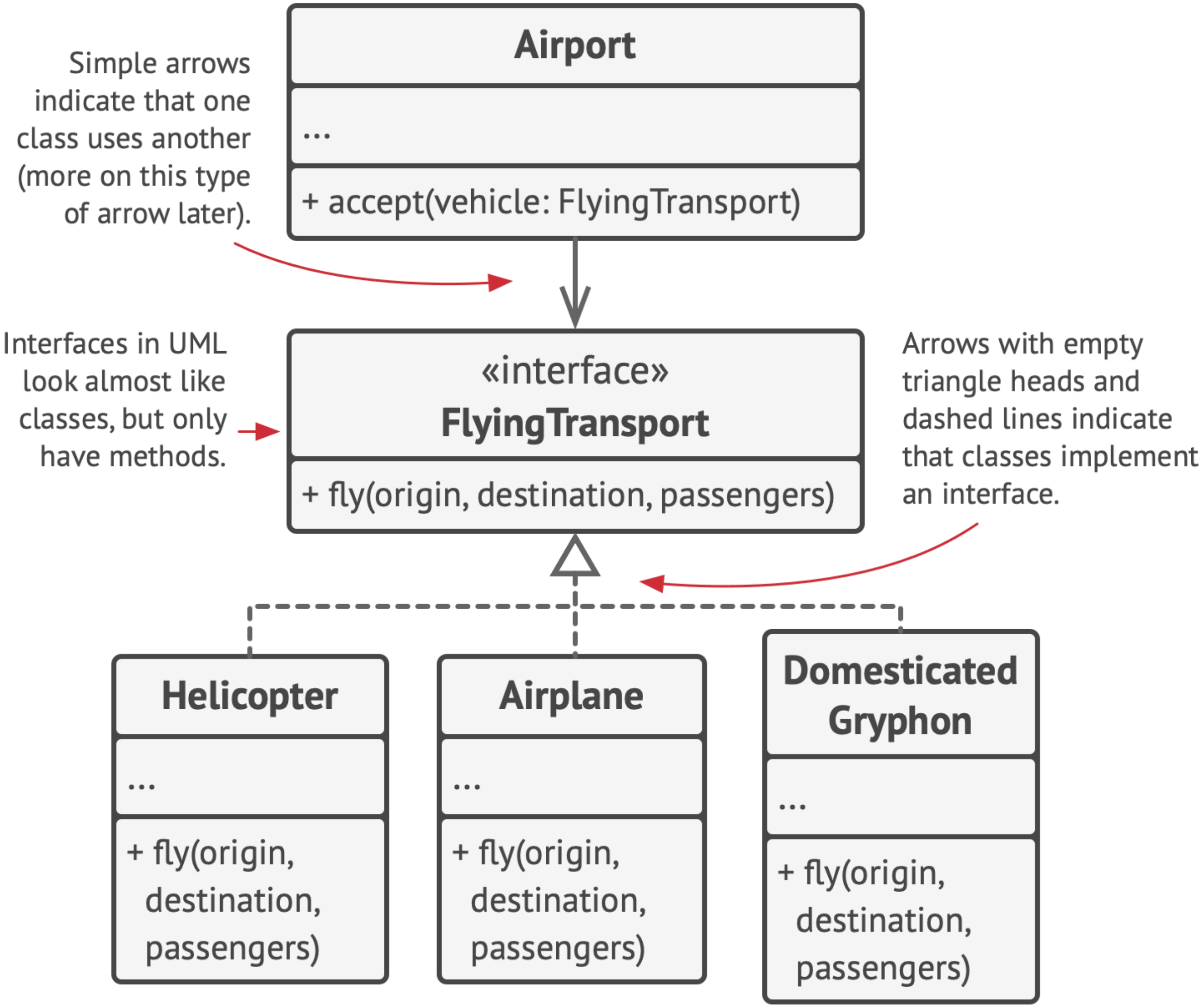
+ reserveSeat(n)

[Figures in this slide are extracted from “Dive Into Design Patterns” by Alexander Shvets]

Pillars of Object-Oriented Programming (cont.)

2) Encapsulation

- Hiding parts of an object’s states and behaviors from others, and exposing a limited set of interfaces
- public, private, and protected
- Interfaces and abstract classes



Simple arrows indicate that one class uses another (more on this type of arrow later).

Interfaces in UML look almost like classes, but only have methods.

Arrows with empty triangle heads and dashed lines indicate that classes implement an interface.

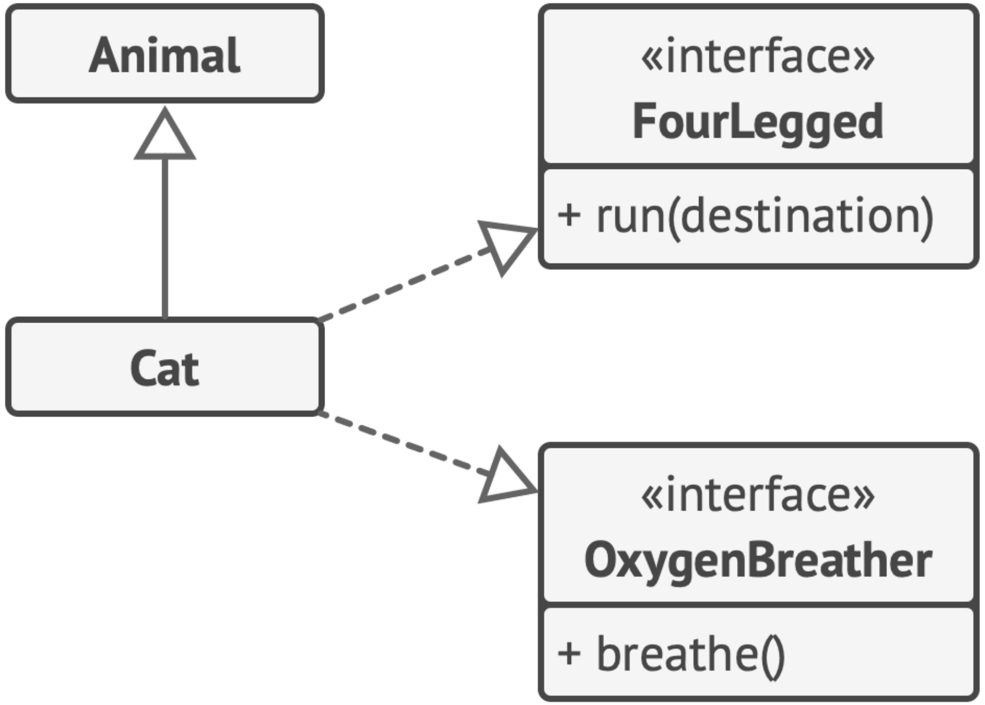
Airport
...
+ accept(vehicle: FlyingTransport)
«interface»
FlyingTransport
+ fly(origin, destination, passengers)
Helicopter
...
+ fly(origin, destination, passengers)
Airplane
...
+ fly(origin, destination, passengers)
Domesticated Gryphon
...
+ fly(origin, destination, passengers)

[Figures in this slide are extracted from “Dive Into Design Patterns” by Alexander Shvets]

Pillars of Object-Oriented Programming (cont.)

3) Inheritance

- Main benefit: code reuse

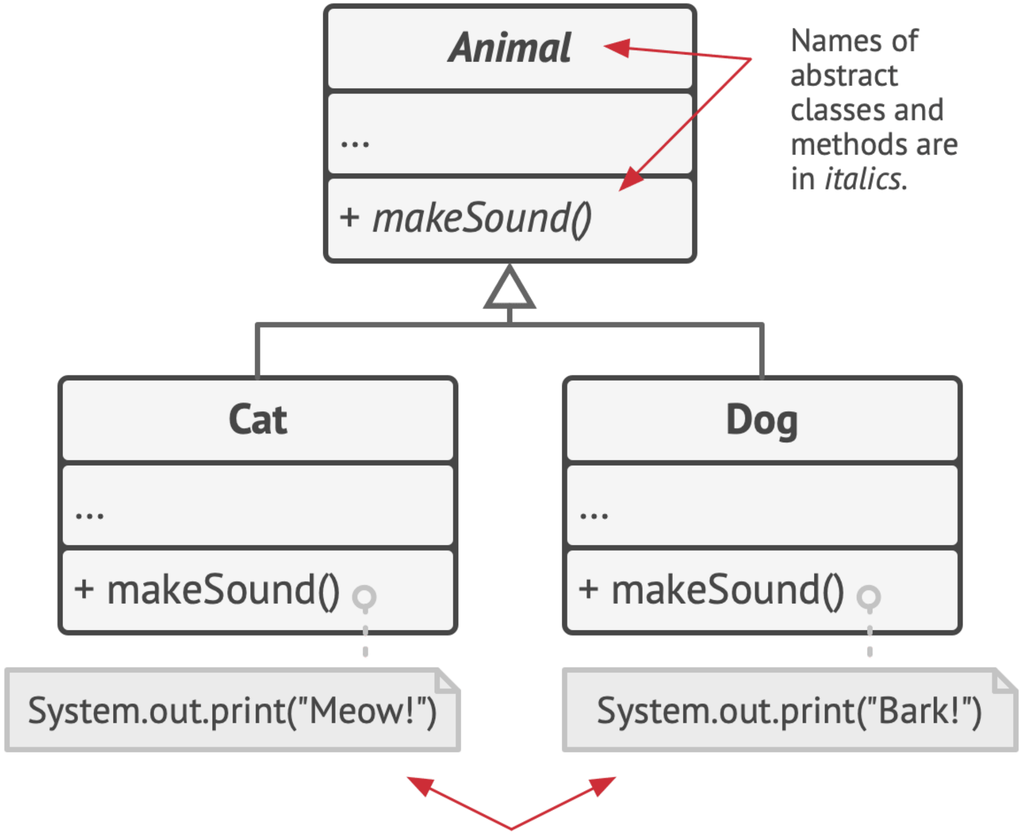


[Figures in this slide are extracted from “Dive Into Design Patterns” by Alexander Shvets]

Pillars of Object-Oriented Programming (cont.)

4) Polymorphism

- Performing an action in many forms
- A mechanism for detecting the real class of an object and call its implementation



```
System.out.print("Meow!")
```

```
System.out.print("Bark!")
```

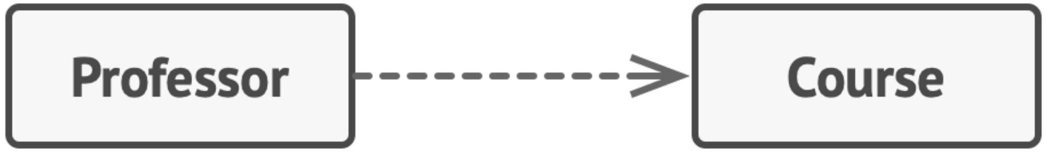
Names of abstract classes and methods are in *italics*.

These are UML comments. Usually they explain implementation details of the given classes or methods.

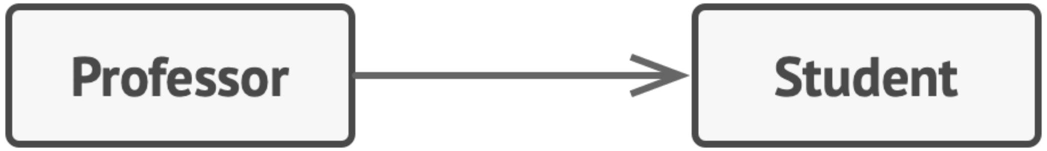
[Figures in this slide are extracted from "Dive Into Design Patterns" by Alexander Shvets]

More Relations Between Objects

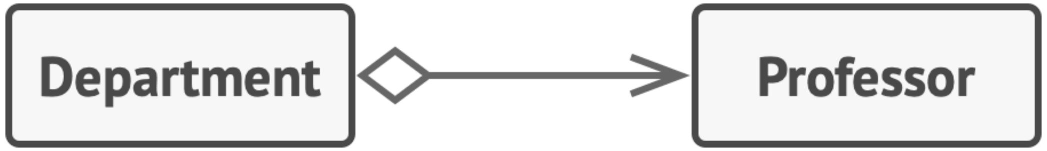
- **Dependency**



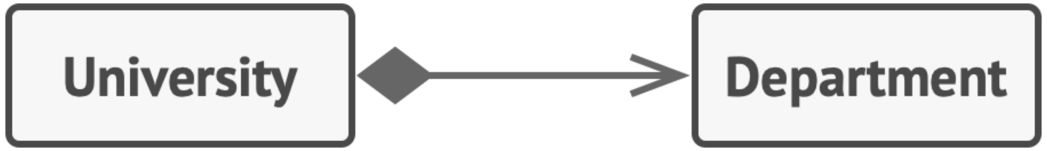
- **Association**

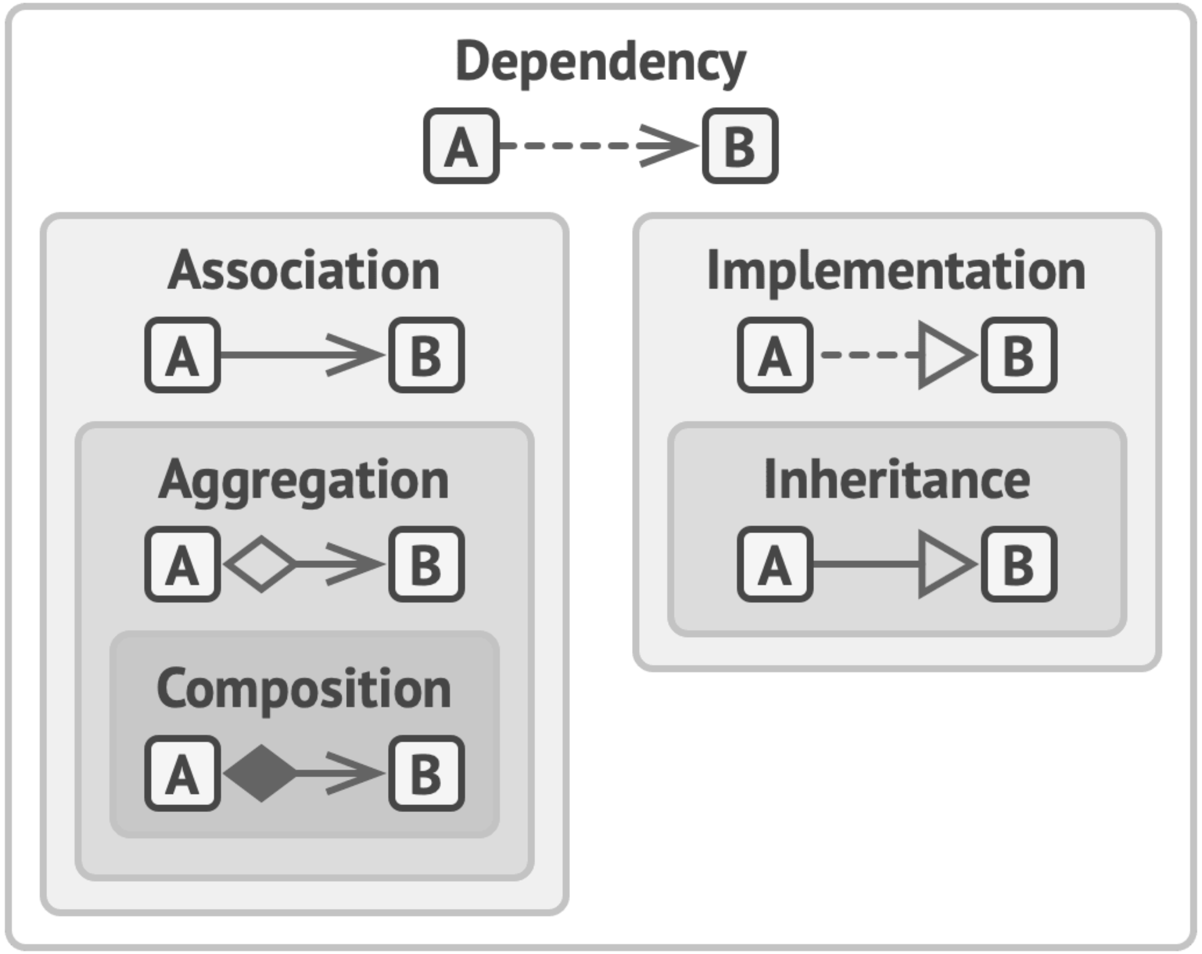


- **Aggregation**



- **Composition**





[Figures in this slide are extracted from "Dive Into Design Patterns" by Alexander Shvets]

OOP with Java: Declaring Classes and Creating Objects

- Class declaration

```
class MyClass extends MySuperClass implements YourInterface {
     // field, constructor, and
     // method declarations
}
```

- Creating objects
  - Declaration, instantiation, initialization

```
Account a = new Account("Garfield", 8);
```

  - The reference returned by the new operator does not have to be assigned to a variable
    - Example: `new Rectangle(100, 50).getArea()`

OOP with Java: Access Control

- At the top level
  - public, or package-private (no explicit modifier)
- At the member level
  - public, private, protected, or package-private (no explicit modifier)

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

OOP with Java: Inheritance

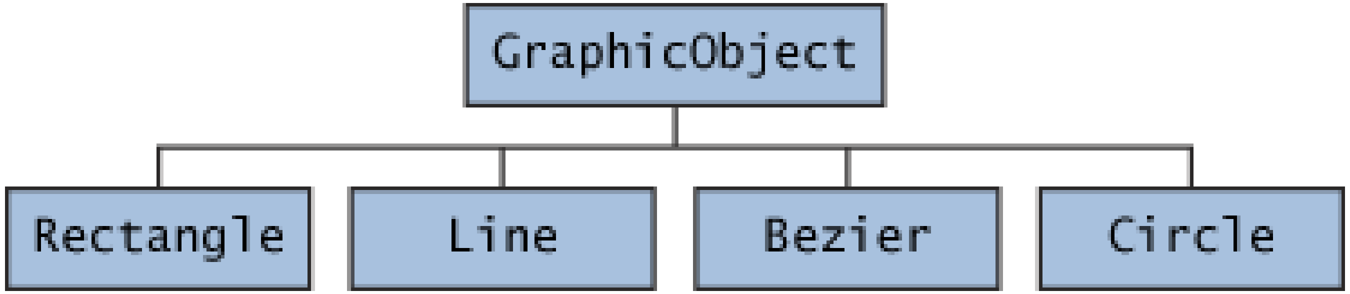
- Classes can be derived from other classes, inheriting fields and methods
- Definitions
  - **Subclass** (derived class/extended class/child class)
  - **Superclass** (base class/parent class)
- Every class has one and only one direct superclass (**single inheritance**)
  - Excepting *Object*, which has no superclass
- A subclass inherits **all the members** (fields, methods, and nested classes) from its superclass

OOP with Java: What You Can Do in a Subclass

- Use the inherited members as is, replace them, hide them, or supplement them
  - Declare a field in the subclass with the same name as the one in the superclass, thus **hiding** it (NOT recommended)
  - Write a new instance method in the subclass that has the same signature as the one in the superclass, thus **overriding** it
  - Write a new static method in the subclass that has the same signature as the one in the superclass, thus **hiding** it
  - Write a subclass constructor that **invokes** the constructor of the superclass
- How about private members in a superclass?

OOP with Java: Abstract and Final Methods/Classes

- An abstract class is a class declared abstract: it may or may not include abstract methods
- An abstract method is a method declared without an implementation
- Example



- Final methods and classes
  - Methods called from constructors should generally be declared final

OOP with Java: Interfaces

- Interfaces are **contracts**
- A reference type, containing only constants, method signatures, default methods, static methods, and nested types
- Cannot be instantiated
    - They can only be implemented by classes or extended by other interfaces
- Consisting of modifiers, keyword, interface name, a comma-separated list of parent interfaces (if any), and the interface body
- Interface body can contain abstract methods, default methods, and static methods

OOP with Java: Implementing and Using Interfaces

- Include an **`implements`** clause in the class declaration
  - Your class can implement more than one interface
- If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface

OOP with Java: Abstract Classes vs. Interfaces

- Similarities and differences
- Consider using abstract classes when
  - You want to **share code** among several closely related classes
  - You expect that classes extending the abstract class have **many common methods or fields**, or require **access modifiers other than public**
  - You want to declare **non-static or non-final** fields
- Consider using interfaces when
  - You expect that **unrelated classes** would implement your interface
  - You want to specify the **behavior** of a particular data type, but not concerned about who implements its behavior
  - You want to take advantage of **multiple inheritance**