CSSE 220

Interfaces and Polymorphism

Import *Interfaces* from the repo

Object-Oriented Programming

- The **three pillars** of **Object-Oriented Programming**
  - Encapsulation (already covered)
  - Polymorphism (start idea today)
  - Inheritance (next week)

Interfaces – What, When, Why, How?

- What:
  - Code structure that looks like a class
  - Used to express operations that multiple classes have in common
- Differences from classes:
  - No fields.
  - Methods contain **no code**.
- When:
  - When abstracting an idea that has multiple, different implementations

Generic Notation: In Code

```
public interface InterfaceName{
        /**
         *   regular javadocs
         */
        void methodName(int x, int y);

        /**
         *   regular javadocs here
         */
        int doSomething(Graphics2D g);
    }

public class SomeClass implements InterfaceName {

}
```

Automatically public, so we don't specify it

No method body, just a semi-colon

SomeClass promises to implement all the methods declared in the **InterfaceName** interface

Interface Types: Key Idea

- Interface types are like **contracts**
- A class can promise to **implement** an interface
- Any code that **uses** the interface can automatically use new classes that implement the interface!

Why?

- Interfaces help to **reduce coupling** by tying your code to the interface, not the class implementation.

Q1

Principles of Design (for CSSE220)

- Make sure your design **allows proper functionality**
  - Must be able to **store required information** (one/many to one/many relationships)
  - Must be able to **access the required information** to accomplish tasks
  - Data should **not be duplicated** (id/identifiers are OK!)
- Structure design **around the data** to be stored
  - **Nouns should become classes**
  - **Classes should have intelligent behaviors** (methods) **that may operate on their data**
- Functionality should be **distributed efficiently**
  - **No class/part should get too large**
  - **Each class should have a single responsibility** it accomplishes
- **Minimize dependencies** between objects when it does not disrupt usability or extendability
  - Tell don't ask
  - Don't have message chains
- **Don't duplicate** code
  - Similar "chunks" of code should be **unified into functions**
  - Classes with similar features should be given **common interfaces**
  - Classes with similar internals should be simplified using **inheritance**

Open simpleExample

- Live-coding
  - will arrive at solution in InterfaceSolution
  - feel free to just watch if you prefer
  - we will remove duplicate CODE (not data)

NumberSequence Example

- Your turn to implement an existing interface
  - Study code
  - Complete TODOs in order
  - Let me know if you complete them all

Interface Types can replace class types

- If Dog & Cat implement the Pet interface:
  1. Variable Declaration:
     - Pet d = new Dog(); Pet c = new Cat();
  2. Parameters:
     - public static void feedPet(Pet p) {…}
       Can call with any object of type Pet:
     - feedPet(new Dog()); feedPet(c); // from above
  3. Fields:
     - private Pet pet;
  4. Generic Type Parameters:
     - ArrayList<Pet> pets = new ArrayList<Pet>();
     - pets.add(new Dog()); pets.add(new Cat());

Check your understanding…

```
public interface Pet{
    private String name;

    public Pet(String name){
        this.name = name;
    }

    public void speak(){
        System.out.println(name);
    }
}
```

Is this interface valid? Why or why not?

Valid interface

```
public interface Pet{
    public void speak();
}
```

What happened to name?

A valid Pet with a name

```
public class Cat implements Pet {
    private String name;

    public Cat(String name){
        this.name = name;
    }

    public void speak(){
        System.out.println(name);
    }
}
```

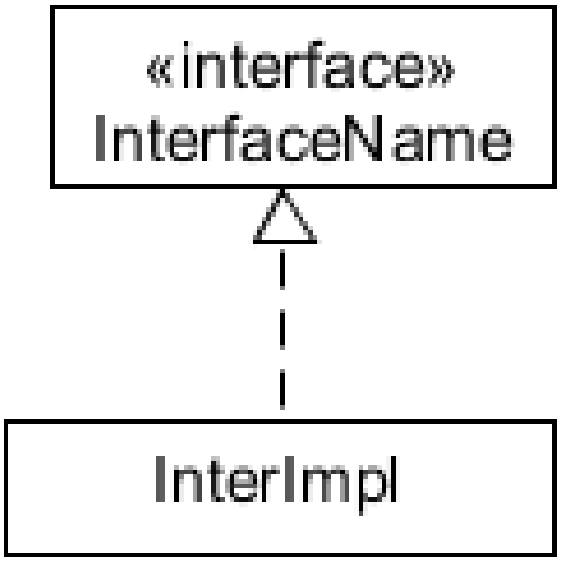
Why is this OK?

```
Pet p = new Dog();
p.feed();
p = new Cat();
p.feed();
p = new Pet(); // NO!
```

- Any child type may be stored into a variable of a parent type, but not the other way around.
  - A Dog is a Pet, and a Cat is a Pet, but a Pet is not **required** to be a Dog or a Cat.
  - And how could you ***construct*** a Pet?

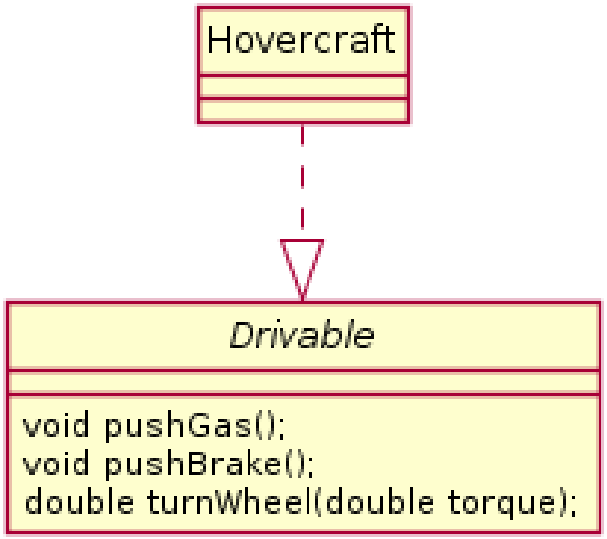
Q2

Notation: In UML



- Closed triangle with a dashed line in UML is an “is-a” relationship
- Read this as:

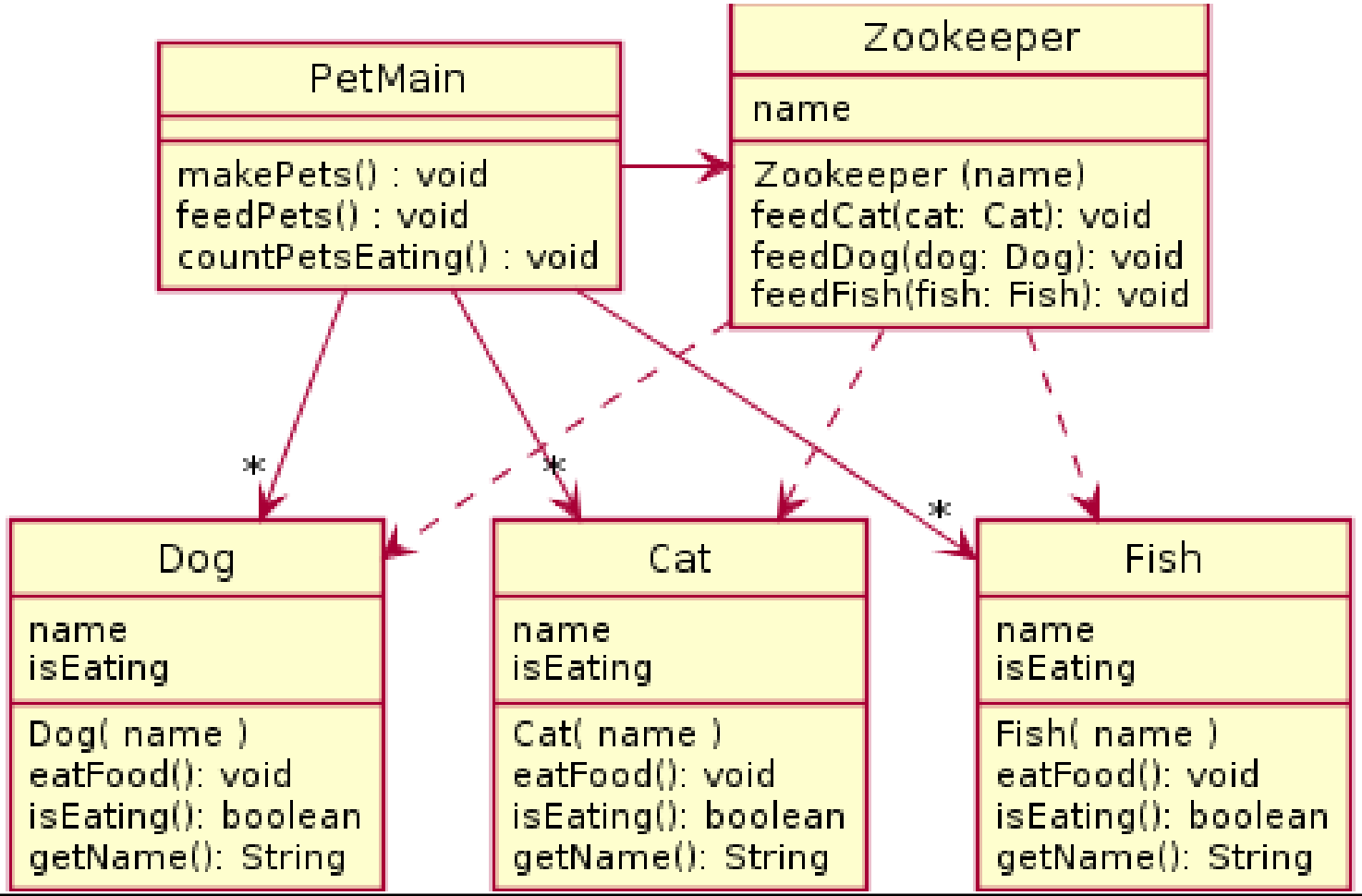
  InterImpl is-an InterfaceName



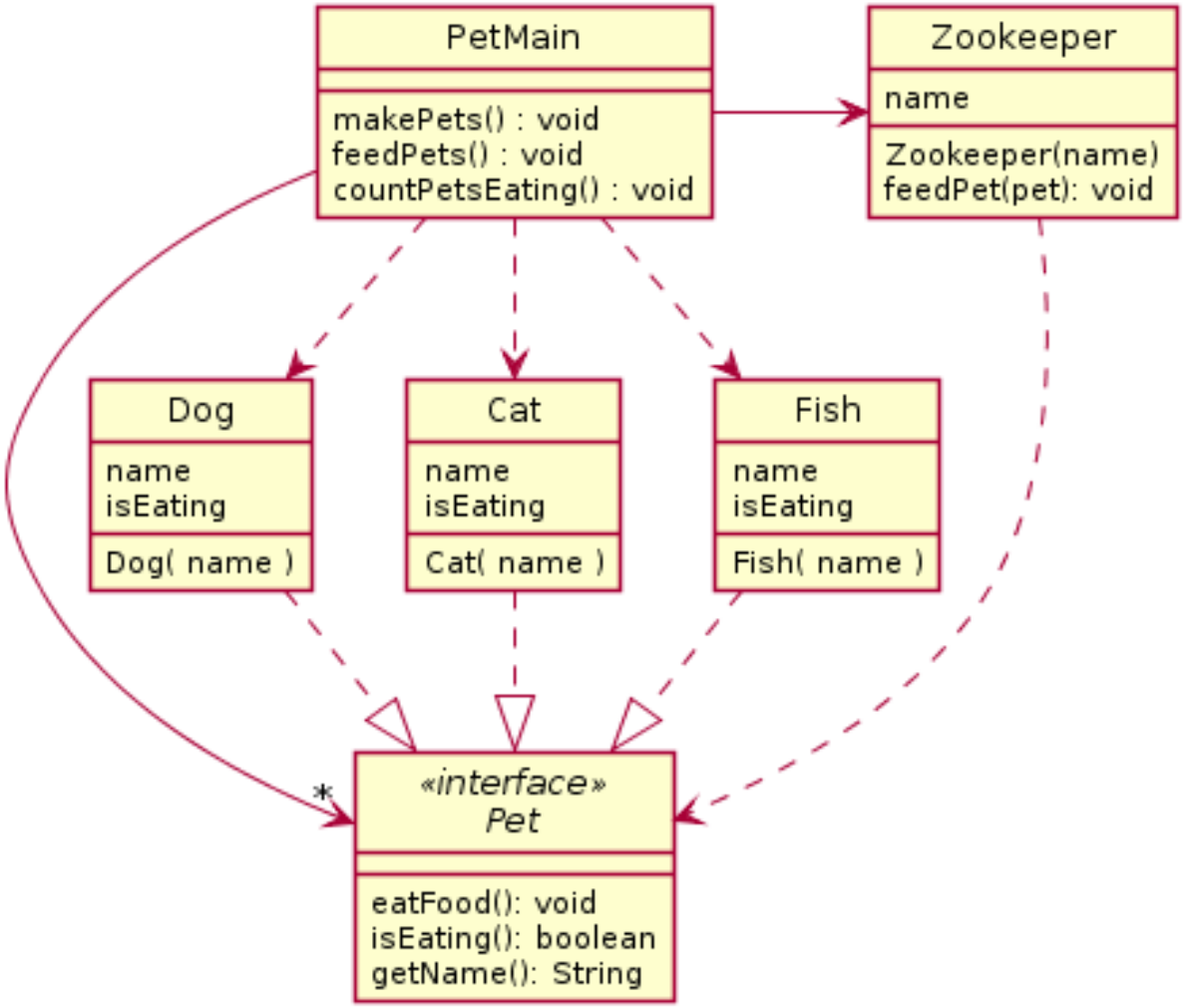
No need to repeat method names in classes that implement them, it is implies by the arrow.

Q3

In the following scenario we have a Pet Zoo, with a Zookeeper who is in charge of feeding different types of animals. When the simulator runs, various pets are made and fed. Also, there is a way to count the number of pets that are eating. The animals include cats, dogs, and fish. All the animals have names, and can be told to eat food, as well as report that they are eating (once fed they always report eating). Show how an improved approach using interfaces can remove code duplication from the following design.

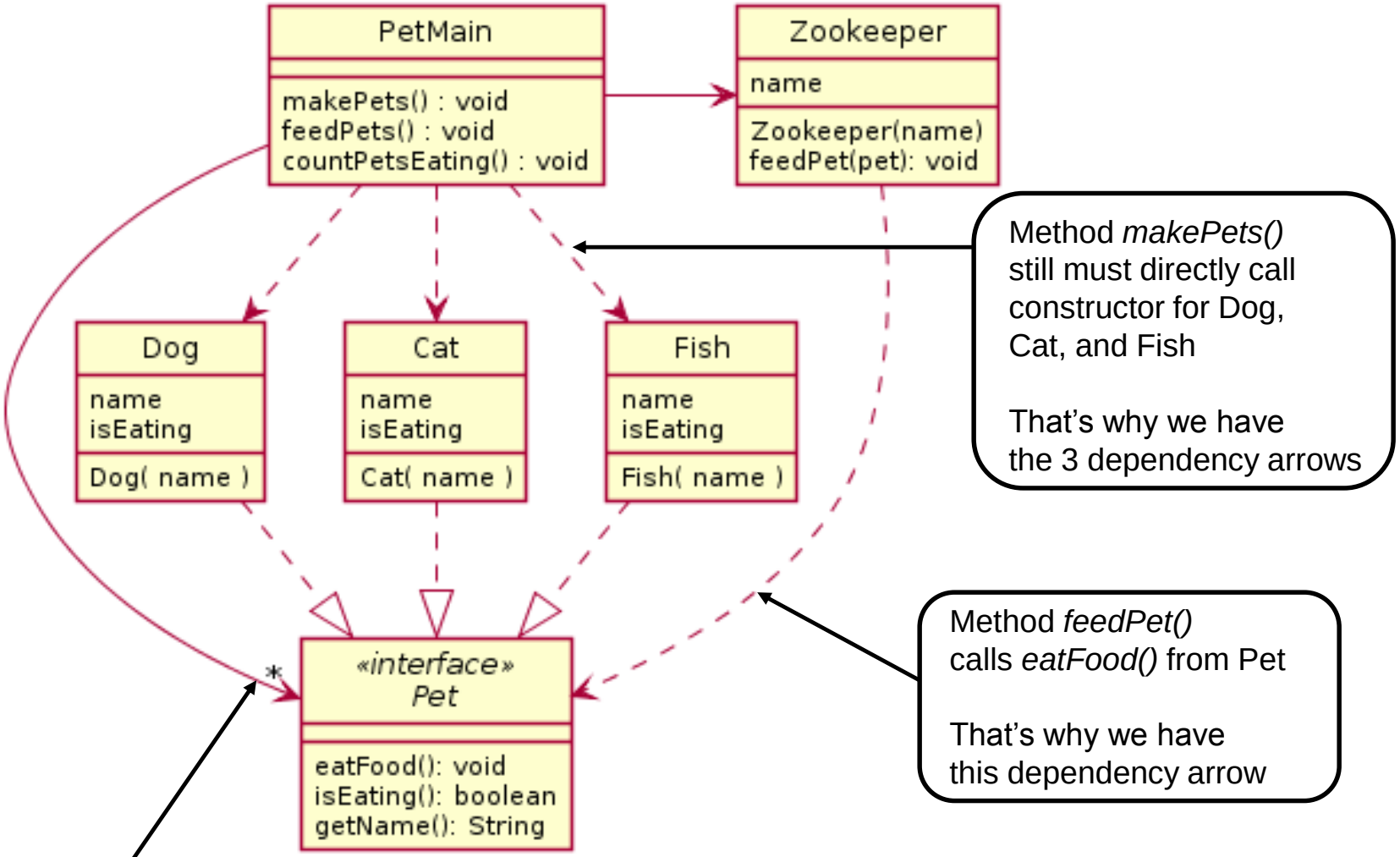


Solution


PetMain
makePets() : void
feedPets() : void
countPetsEating() : void
Zookeeper
name
Zookeeper(name)
feedPet(pet): void
Dog
name
isEating
Dog(name)
Cat
name
isEating
Cat(name)
Fish
name
isEating
Fish(name)
«interface»
Pet
eatFood(): void
isEating(): boolean
getName(): String
*

Q4

Solution



Code this in the pets project!

Polymorphism! (A quick intro)

- Etymology:
  - Poly → many
  - Morphism → shape

- Polymorphism means: An **Interface** can take **many shapes**.
  - A Pet variable could actually contain a Cat, Dog, or Fish

Polymorphic method calls

- pet.feed() **could** call:
  - Dog's feed()
  - Cat's feed()
  - Fish's feed()

- Your code is well designed if:
  - You **don't need to know** which implementation is used.
  - The end result is the same. ("pet is fed")

Q5

How does all this help reuse?

- Can pass an **instance** of a class where an interface type is expected
  - But only *if the class* ***implements*** *the interface*

- We could add new functions to a NumberSequence's abilities without changing the runner itself.
  - Sort of like application "plug-ins"

- We can use a new Pet interface without changing the method that uses the Pet instance. (When adding a Zebra class to PetMain, Zookeeper does not have to change!)

- **Use interface types** for field, method parameter, and return types whenever possible. Like Pet instead of Dog, and List for ArrayList.
  - **List**<Pet> pets= new ArrayList<Pet>();

- Next time: because of interfaces, we can add classes that listen for Button presses and mouse clicks, without changing the Button or window.