CSSE 220

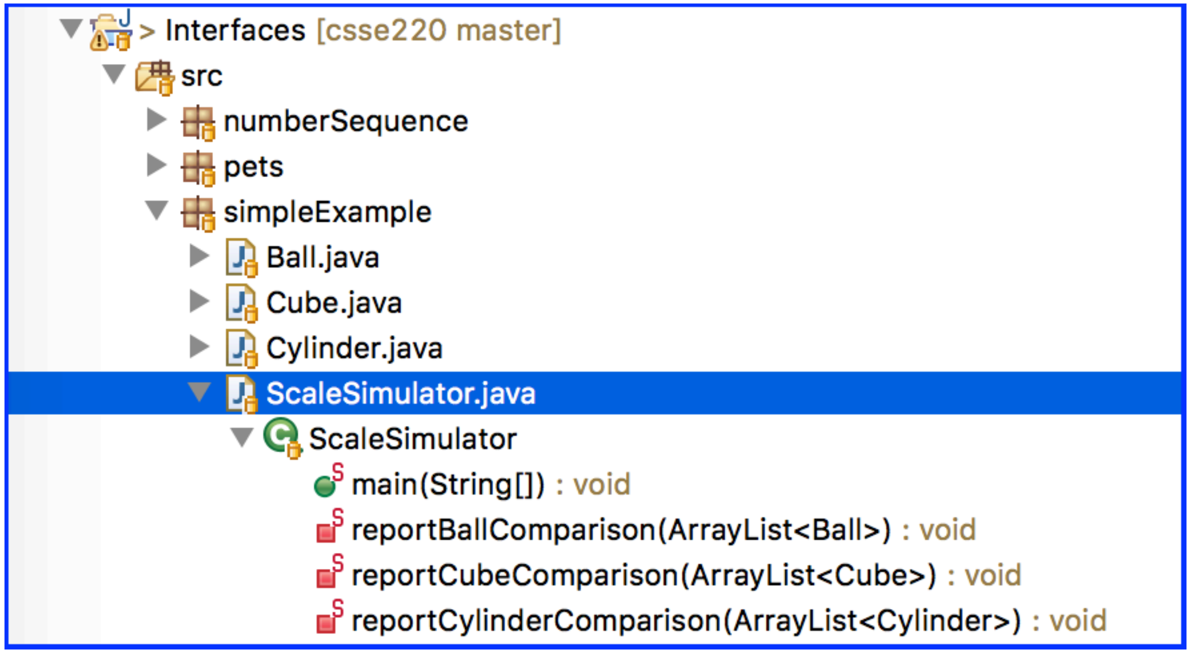
Interfaces and Polymorphism

Check out *Interfaces* from repo

Object-Oriented Programming

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

Open: Interface | simpleExample |scaleSimulator.java



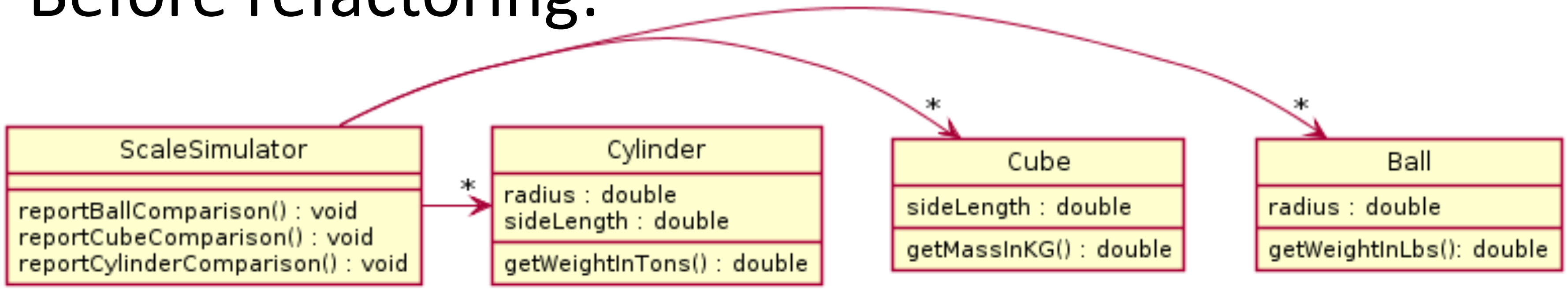
```
▼ > Interfaces [csse220 master]
  ▼ src
    ▶ numberSequence
    ▶ pets
    ▼ simpleExample
      ▶ Ball.java
      ▶ Cube.java
      ▶ Cylinder.java
      ▼ ScaleSimulator.java
        ▼ ScaleSimulator
            main(String[]) : void
            reportBallComparison(ArrayList<Ball>) : void
            reportCubeComparison(ArrayList<Cube>) : void
            reportCylinderComparison(ArrayList<Cylinder>) : void
```

Note: Three ‘report’ methods almost identical

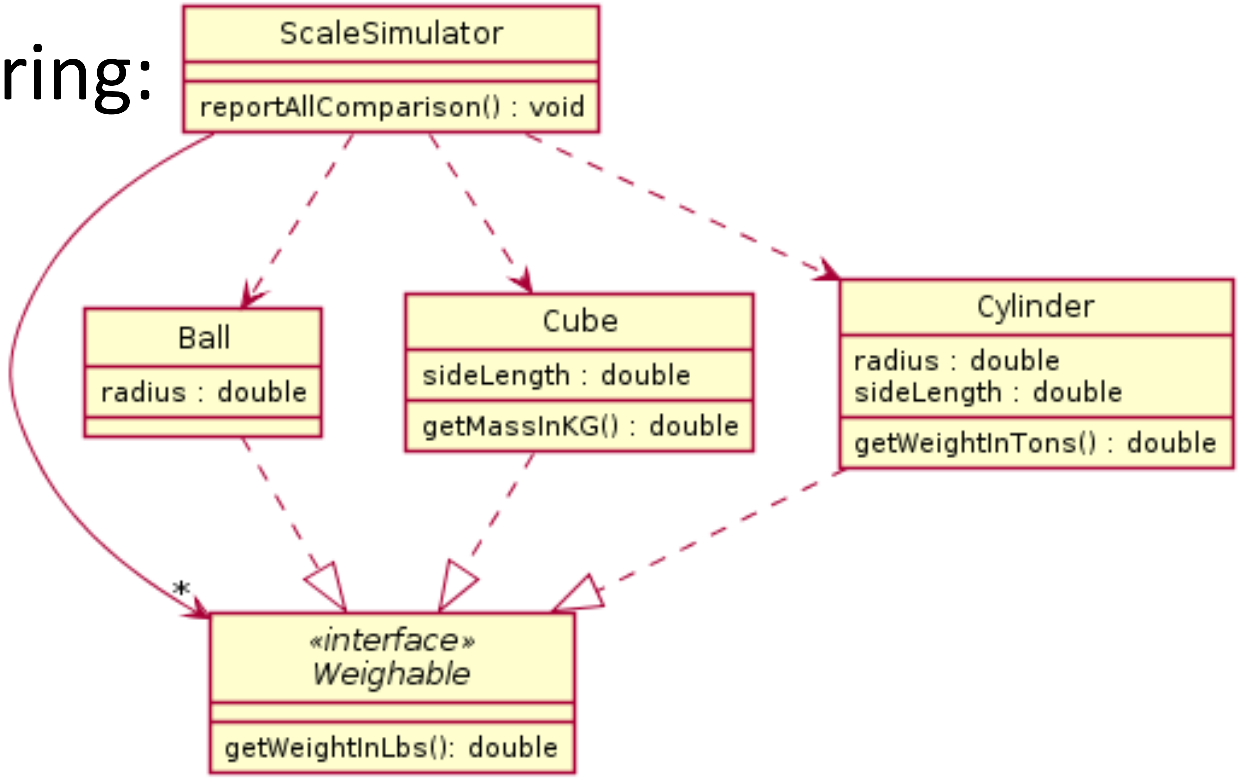
- Code duplication
- Use Java Interface to help eliminate duplication
- Parameter types are different, but similar
- Method being called is slightly different: pounds, kilograms, tons

Code Refactoring

Before refactoring:



After refactoring:



Use Two Steps of *Code Refactoring*

1. Make three operations the same
2. Then eliminate duplication

Recommendation:

- Make incremental changes, i.e., in small steps
- Between increments, validate that program works the same as it did before change

Step One – Make Similar

Live coding:

1. Create getWeight method in all three classes: Ball, Cube, Cylinder
2. Go to client operations and make them call getWeight
3. Run code and visually validate it runs the same as before – better method would be to have Junit tests

Next Step – Make an Interface

Live coding:

1. Convention – Interface name ends in *able*
2. Create Interface
3. Then go to 3 classes and add “implements Weighable”

Note:

- By making a class C1 “implements *Interface-name*”, Java language requires that class C1 must implement (provide operation body) for the methods listed in the Interface

Next Step – Modify Main Part of Client

Live coding:

1. Go to *main* operation and change variable declarations to use Weighable
2. Eliminate duplicated ‘report’ operations

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 – except for “static final” for constants
  - Methods contain **no code, no constructors**

- When:
  - When abstracting an idea that has multiple, different implementations

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 have to 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**
  it is useful for capturing the "what it does"

- A class can promise to **implement** an interface
  - Class that implements the Interface must implement all the methods in the Interface

- Any client that **uses** the interface can automatically use new classes that implement the interface!

Why?

- Interfaces help to **reduce coupling** by tying your client code to the interface, not the class implementation.
  - The client declares variables of the Interface instead of the class(es) that implement the Interface
  - The client depends on "the what it does" of the Interface

Q1

Interface Types can replace class types

- If Dog & Cat implement the Pet interface:
  1. Variable Declaration & constructor call:
     - Pet d = new Dog(); Pet c = new Cat();
  2. Parameters:
     - public static void feedPet(Pet p) {…}
       Client can call with any object of type Pet:
     - feedPet(new Dog()); feedPet(c); // from above
  3. Use Interface as 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();
} // Pet
```

What happened to field *name*?
Ans: it appears in the class that Implements Pet

3 Steps to Implementing an Interface

```
public class Cat implements Pet {
   // 1. Declaration of data members
   private String name;

   // 2. Declaration of constructor(s)
   public Cat(String name){
       this.name = name;
   } // Cat

   // 3. Implementation of all Interface operations
   public void speak(){
       System.out.println(name);
   } // speak
} // Cat
```

Is-a – Relationship - Why is this OK?

```
Pet p = new Dog();
p.feed();
p = new Cat();
p.feed();
```

Any *child* type may be stored into a variable of a *parent* type, but not the other way around.

- *parent*: Pet
- *child*: Cat, Dog
- A Dog *is a* Pet, and a Cat *is a* Pet, but a Pet is not **required** to be a Dog or a Cat

Q2

Why is this not OK?

```
p = new Pet;
```

An Interface does not (is not allowed) to have a constructor

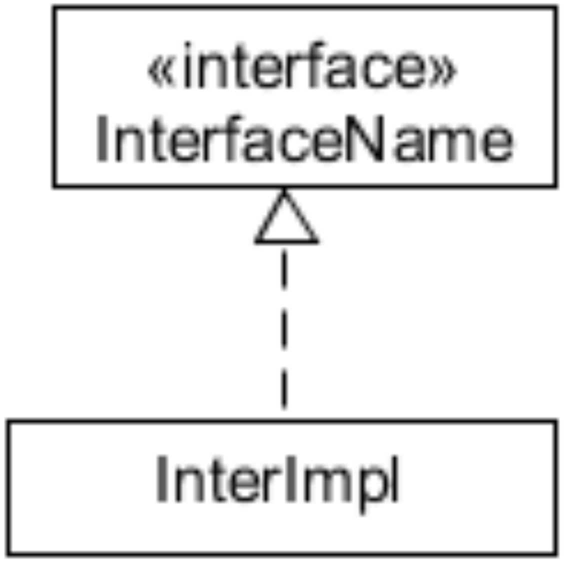
So this is not OK, because there is no way to construct an object of type Pet

Q2

NumberSequence Example

- Your turn to work with Interface

Notation: In UML



- Interface rectangle has two things:
    1. <<interface>>
    2. Interface Name
- "Closed triangle with a dashed line" arrow in UML is an "is-a" relationship

- Read this as:

  InterImpl is-an InterfaceName
  InterImpl

  stands for "implements Interface"
  I.e., class that implements the Interface

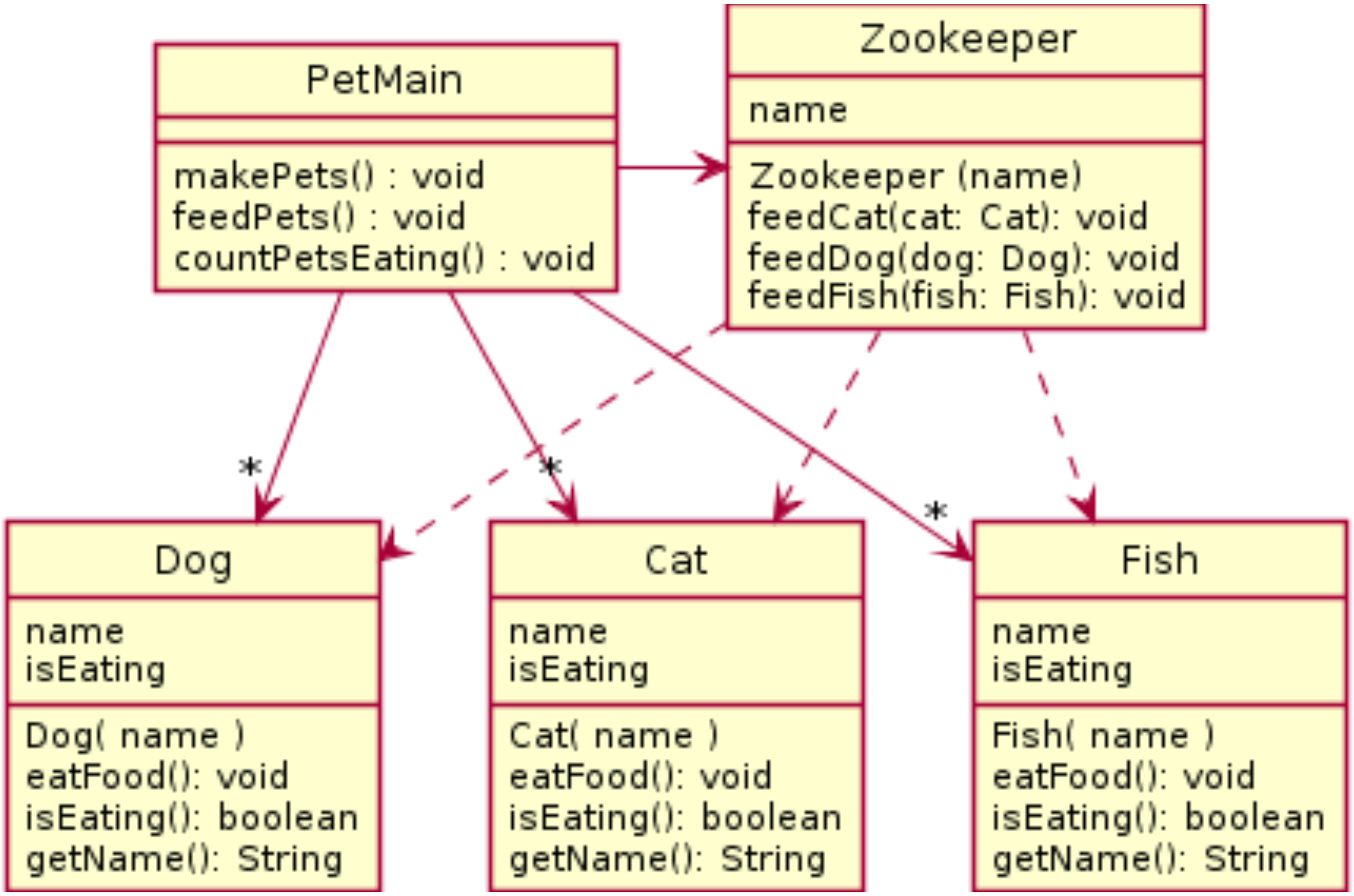
Q3

In-Class Quiz Question #4

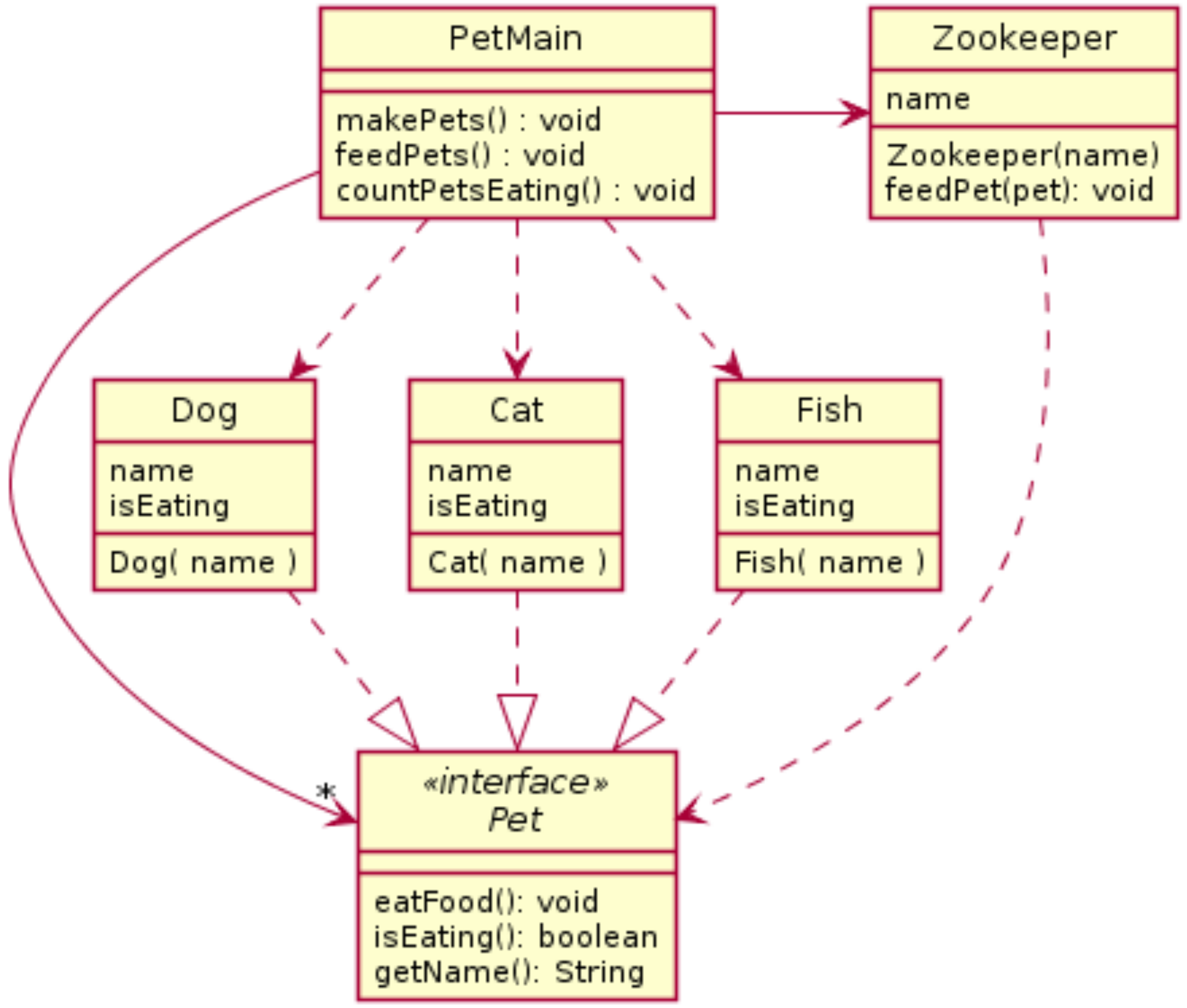
- Refactor UML diagram on next slide using Interface
- Work in groups of 2 or 3
- Recommended: use a pencil
- About 10 minutes

Q4

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.



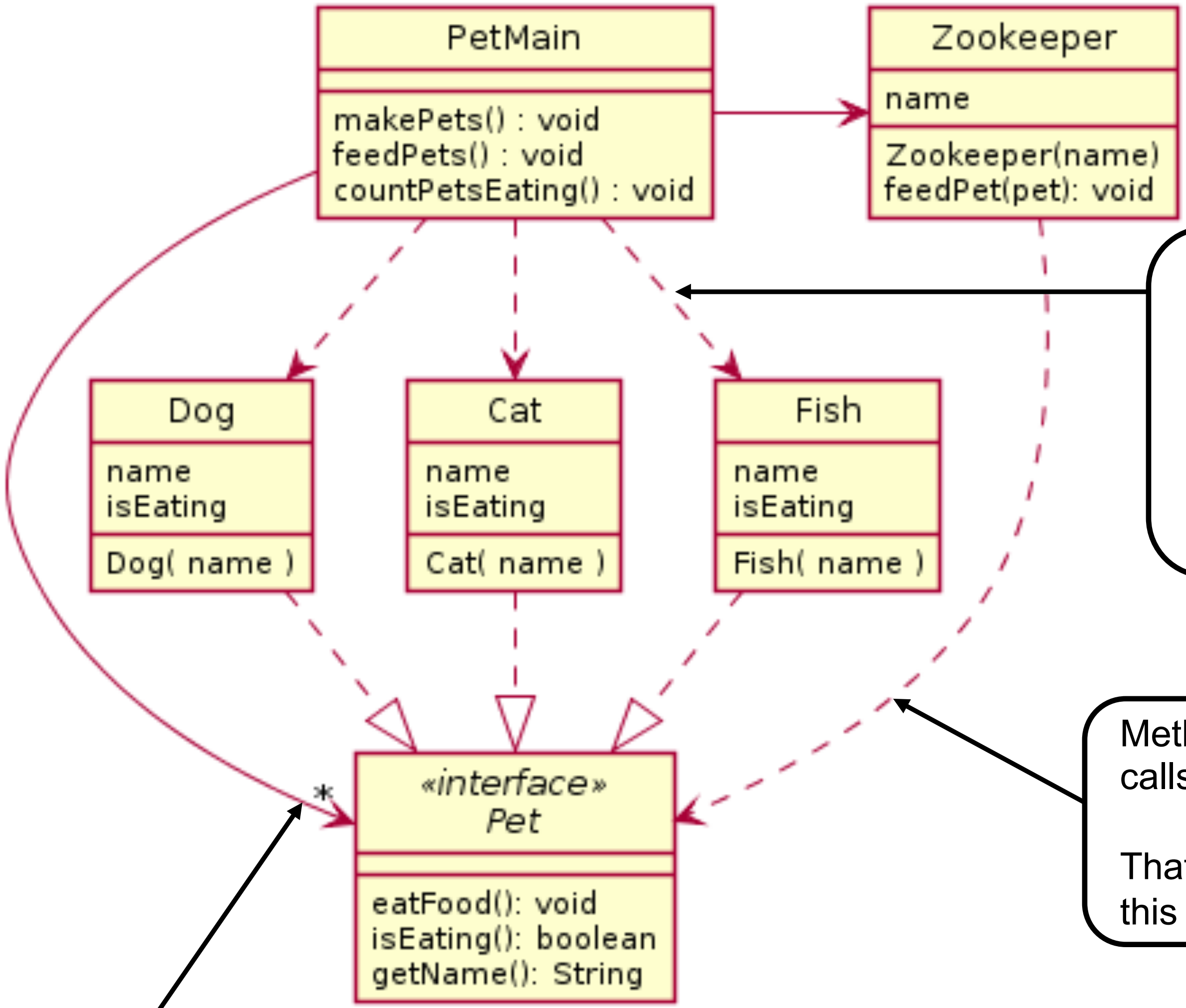
Solution



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



Method *makePets()* still must directly call constructor for Dog, Cat, and Fish

That's why we have the 3 dependency arrows

Method *feedPet()* calls *eatFood()* from Pet

That's why we have this dependency arrow

1 of List<Pet> in PetMain

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

Code Example

```
// PetMain's data members
ZooKeeper z1 = new ZooKeeper();
ArrayList<Pet> aP1 = new ArrayList<Pet>();

// PetMain's feedPets operation
void feedPets()
   for (Pet p : this.aP1) {
      this.z1.feedPet(p);
   } // end for
} // feedPets
```

```
// ZooKeepers's feedPet operation
void feedPet(Pet p)
   p.eatFood();
} // feedPet
```

Q5

Polymorphic method calls

- p.eatFood() **could** call:
  - Dog’s eatFood()
  - Cat’s eatFood()
  - Fish’s eatFood()

- Your code is well designed if:
  - You **don’t need to know** which operation is called
  - The end result is the same – the pet eats

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.