Object-Oriented Programming

Inheritance

Understanding the “is a” relationship

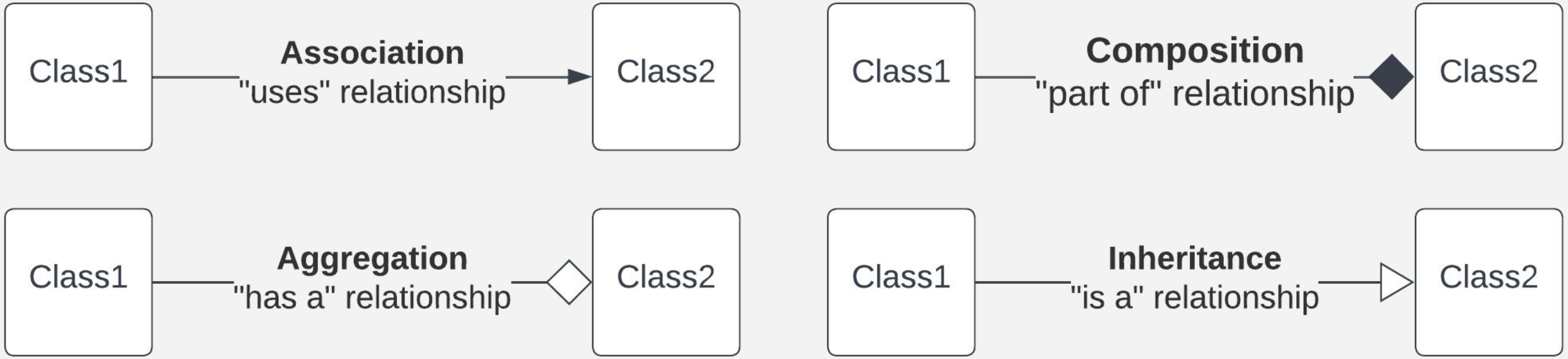
CCPROG3 – AY2223T3 – ETighe

Outline

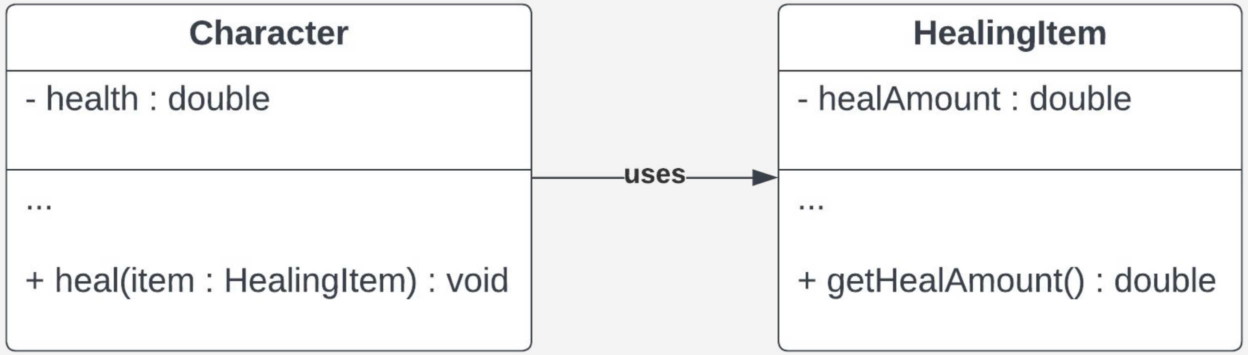
- Recall: Class Relationships
- Inheritance
  - Super & sub classes
  - Using `extend`
  - Inheritance in code
  - Protected access modifiers
- Polymorphism

Class Relationships

- Class relationships fall under four major types



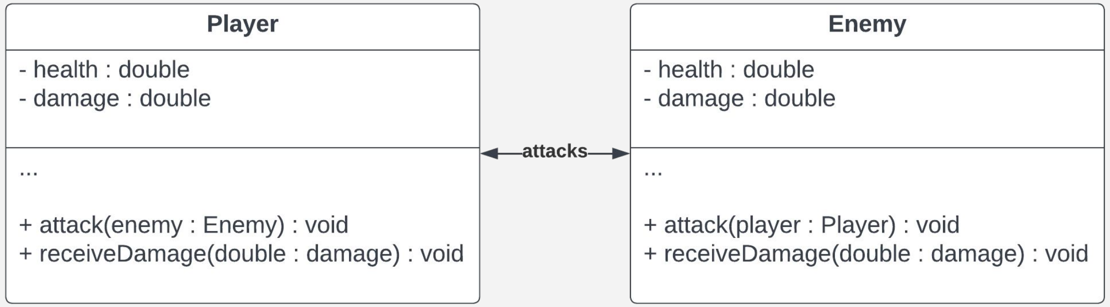
Association – Directed



```java
public class Character {
   private double health;

   // Other parts of the class here

   public void heal(HealingItem item) {
      this.health = this.health + item.getHealAmount();
   }
}
```

```java
public class HealingItem {
   private double healAmount;

   // Other parts of the class here

   public double getHealAmount() {
      return this.healAmount;
   }
}
```

Association – Bidirectional

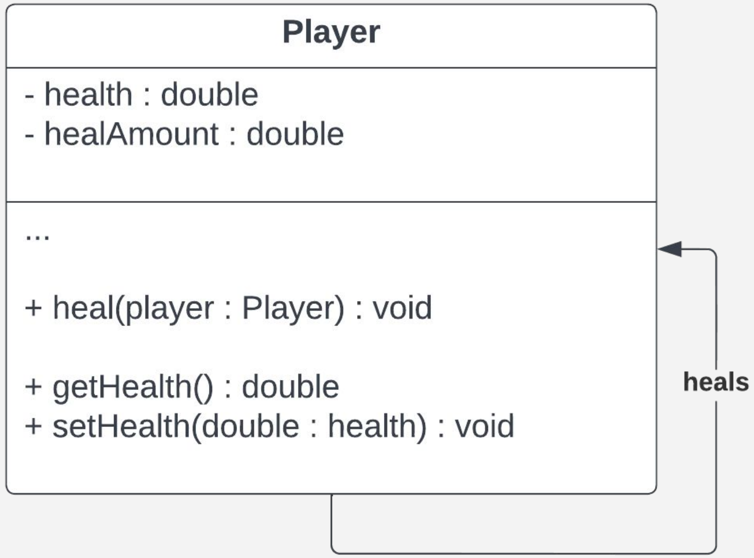

Player
- health : double
- damage : double
...
+ attack(enemy : Enemy) : void
+ receiveDamage(double : damage) : void
attacks
Enemy
- health : double
- damage : double
...
+ attack(player : Player) : void
+ receiveDamage(double : damage) : void

```java
public class Player {
   private double health;
   private double damage;

   // Other parts of the class here

   public void attack(Enemy enemy) {
      enemy.receiveDamage(this.damage);
   }

   public void receiveDamage(double damage) {
      this.health = this.health – damage;
   }
}
```

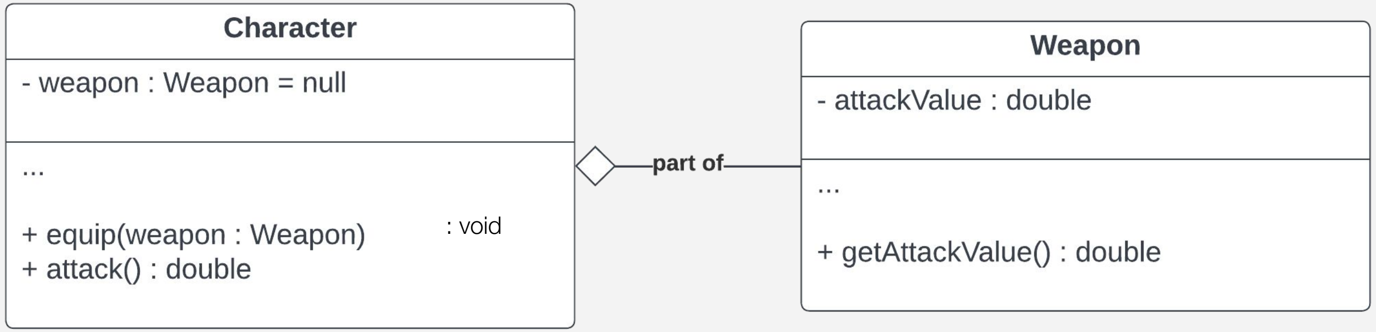
```java
public class Enemy {
   private double health;
   private double damage;

   // Other parts of the class here

   public void attack(Player player) {
      player.receiveDamage(this.damage);
   }

   public void receiveDamage(double damage) {
      this.health = this.health – damage;
   }
}
```

Association – Reflexive

```
public class Player {
   private double health;
   private double healAmount;

   // Other parts of the class here

   public void heal(Player player) {
      player.setHealth(player.getHealth() + this.healAmount);
   }

   public double getHealth() {
      return this.health;
   }

   public void setHealth(double health) {
      this.health = health;
   }
}
```



Aggregation



Character

- weapon : Weapon = null

...

+ equip(weapon : Weapon) : void
+ attack() : double

part of

Weapon

- attackValue : double

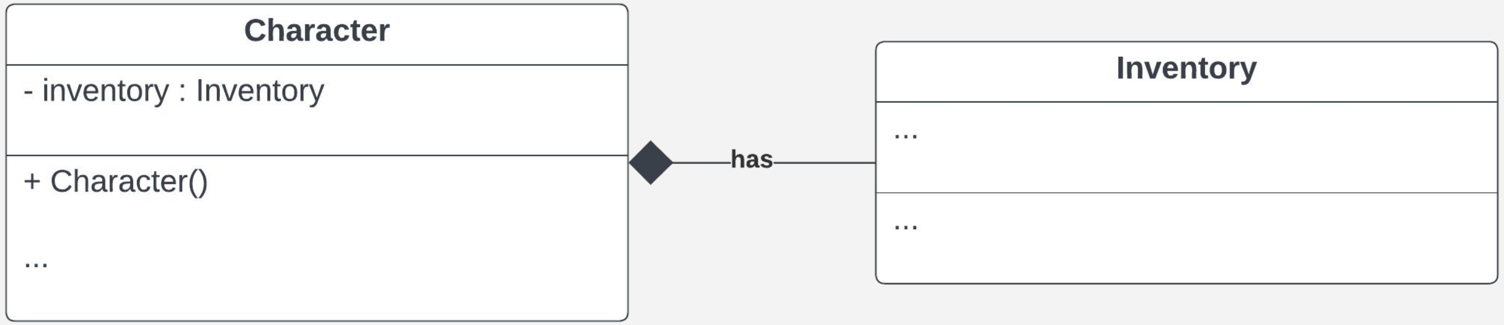
...

+ getAttackValue() : double

```
public class Character {
   private Weapon weapon = null;

   // Other parts of the class here

   public void equip(Weapon weapon) {
      this.weapon = weapon;
   }

   public double attack() {
      double damage = 0;
      if(this.weapon != null) {
         damage = this.weapon.getAttackValue();
      }
      return damage;
   }
}
```

```
public class Weapon {
   private double attackValue;

   // Other parts of the class here

   public double getAttackValue() {
      return this.attackValue;
   }
}
```

Composition



Character

- inventory : Inventory

+ Character()

...

has

Inventory

...

...

```
public class Character {
   private Inventory inventory;

   public Character() {
      this.inventory = new Inventory();
   }

   // Other parts of the class here
}
```

```
public class Inventory {
   // Other parts of the class here
}
```

In this example, an Inventory object is centered in the Character class. Assuming the Inventory object can't be passed out, destroying a Character object would result in the destruction of the Inventory instance.

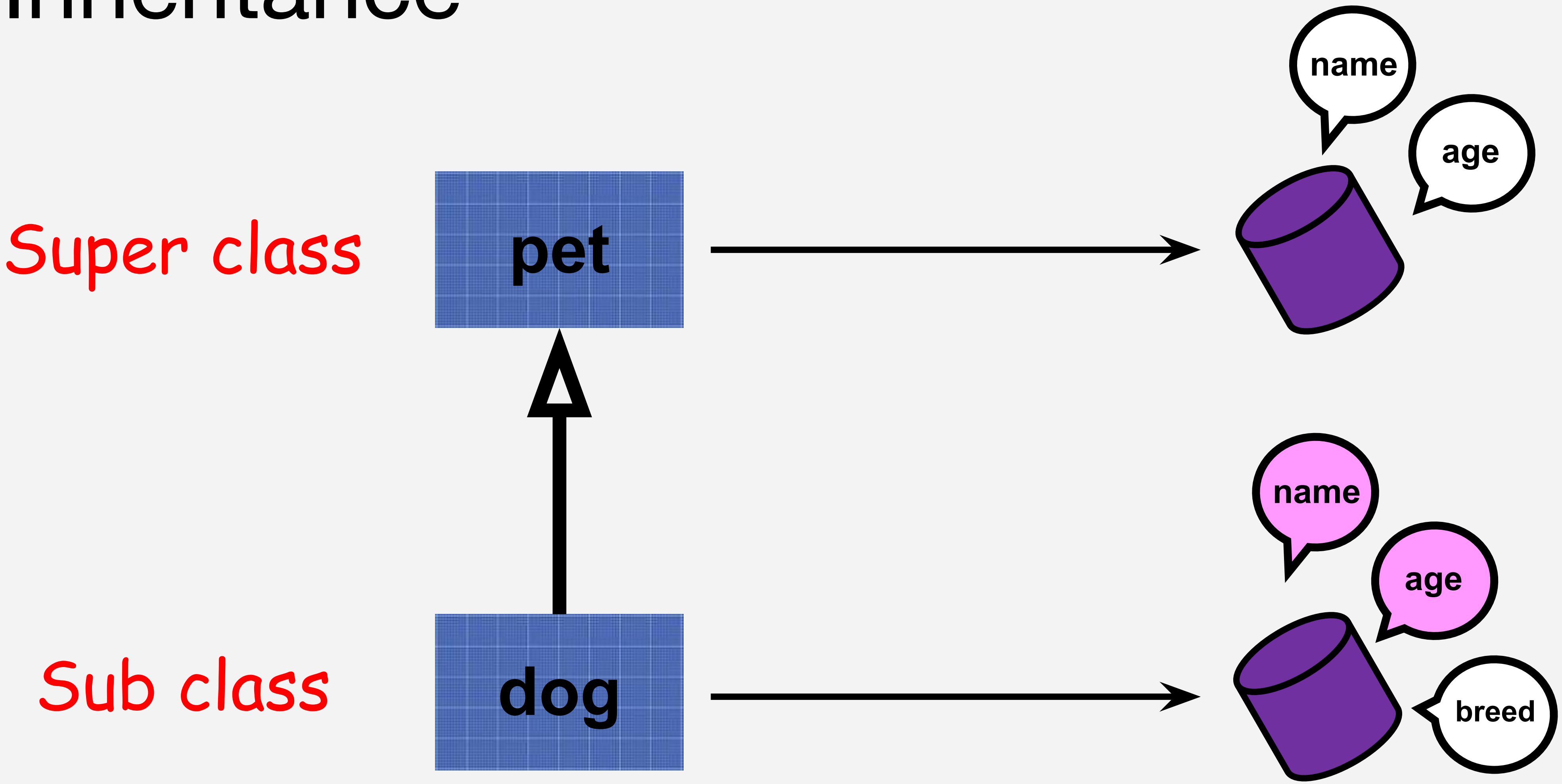
Questions?

What comes to mind when you hear Inheritance?

Inheritance in OOP

- Formalizes the “is a” relationship
    - A *subclass* inherits all attributes and methods from a *superclass*
        - A subclass is the class that is derived from another class
        - A superclass is the class that is inherited from
            - Except `Object`, as this is the super class of all classes

Inheritance



Super class

pet

Sub class

dog

In Code...

```
public class Pet {
    private String name;
    private int age;

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

    // getters, setters, other methods
}
```

```
public class Dog extends Pet {


}
```

If we were to compile these, would this work out?

In Code…

```java
public class Pet {
    private String name;
    private int age;

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

    // getters, setters, other methods
}
```

```java
public class Dog extends Pet {


}
```

- This would result in an error in the Dog class
- Recall that whenever no constructor is provided, Java automatically creates a default constructor
    - For our example, the Dog class would actually look like…

```java
public class Dog extends Pet {
    public Dog() {

    }
}
```

In Code...

```
public class Pet {
    private String name;
    private int age;

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

    // getters, setters, other methods
}
```

```
public class Dog extends Pet {
    public Dog() {
        super();
    }
}
```

But this does not exist in class Pet – hence, the error!

```
public class Dog extends Pet {


}
```

- This would result in an error in the Dog class
- Recall that whenever no constructor is provided, Java automatically creates a default constructor
    - For our example, the Dog class would actually look like…
- However, when a class inherits from another class, it must call the superclass' constructor
    - By default, Java makes it such that a subclass calls the default of the superclass
    - We refer to a superclass by using the keyword super

In Code...

```
public class Pet {
    private String name;
    private int age;

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

    // getters, setters, other methods
}
```

```
public class Dog extends Pet {
    public Dog(String name, int age) {
        super(name, age);
    }
}
```

- This change to the Dog class should fix the error
- Notice how the name and **age** attributes are not stored in the Dog class (at least by design)
    - Technically, a Dog object/instance would contain its own variables
    - By design, however, the Pet class has control over the attributes
    - The Dog class simply “borrows” / inherits characteristics of the Pet class

Inheritance

- Inheritance allows for subclasses to utilize superclasses as a foundation to build more complex logic upon

```
public class Pet {
    private String name;
    private int age;

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

    // getters, setters, other methods
}
```

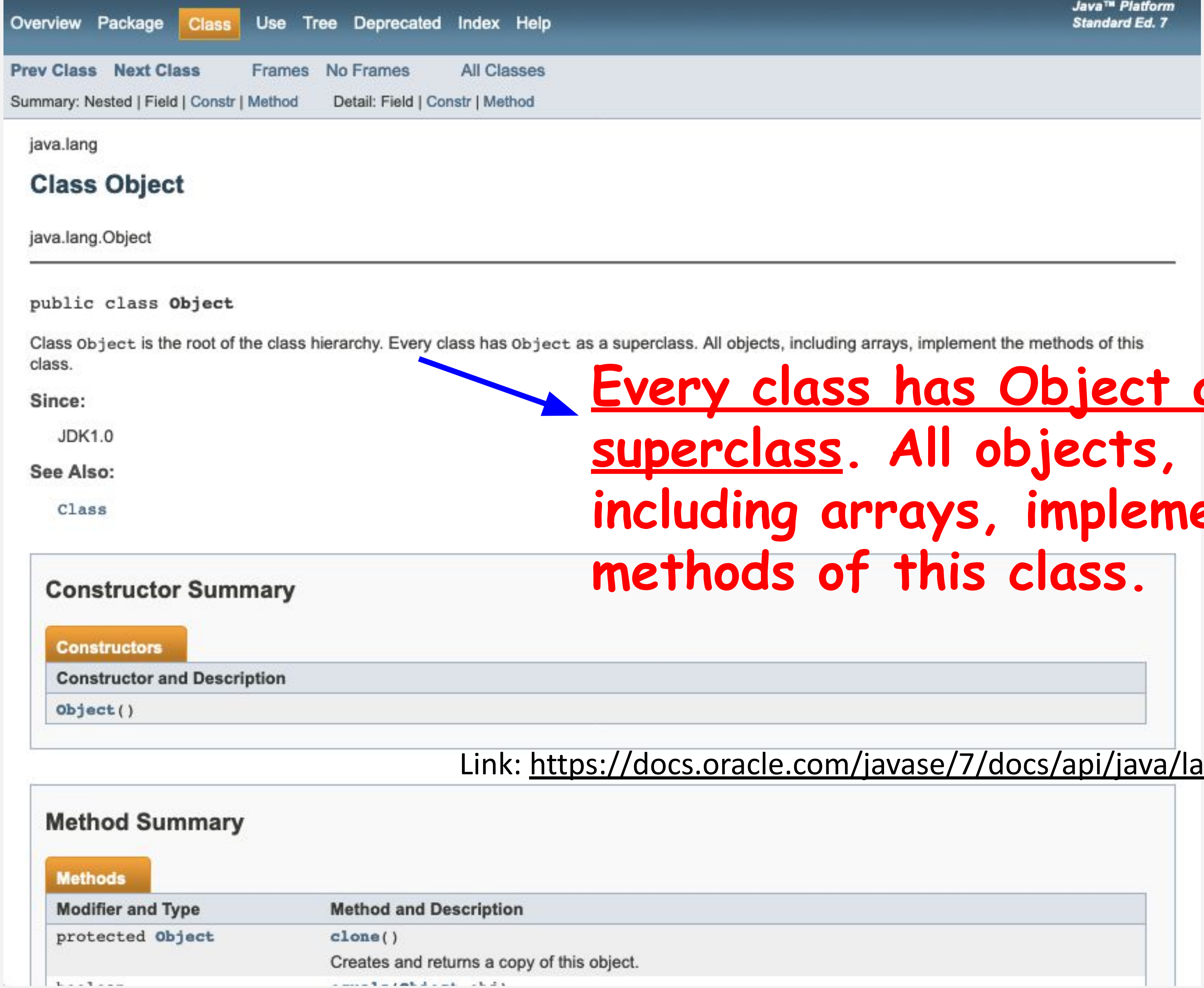
```
public class Dog extends Pet {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age);
        this.breed = breed;
    }
}
```

Question!

- Can we do this?
- Yes, this works!
  - All classes are subclasses of Object
  - Object has a default constructor, so this automation doesn't affect our code

```
public class Pet {
    private String name;
    private int age;

    public Pet(String name, int age){
        super();
        this.name = name;
        this.age = age;
    }

}
```

```
public class Dog extends Pet {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age);
        this.breed = breed;
    }
}
```



java.lang

Class Object

java.lang.Object

```
public class Object
```

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:

JDK1.0

See Also:

`Class`

Constructor Summary

Constructors

Constructor and Description
`Object()`

Method Summary

Methods

Modifier and Type	Method and Description
`protected Object`	`clone()` Creates and returns a copy of this object.
`boolean`	`equals(Object obj)`

Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Link: https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

Inheritance

- Subclasses inherit all members from its superclass
    - Members: Fields, methods, nested classes
- Constructors are not members; hence, not inherited
    - They are invoked / called

Inheritance

- Subclasses inherit all of the public and protected members of its parent class

- But happens with private variables?

```
public class Pet {
    private String name;
    private int age;

    public Pet(String name, int age){
        this.name = name;
        this.age = age;
    }
}
```

```
public class Dog extends Pet {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age);
        this.breed = breed;
    }

    public int getRealAge() {
        return age * 7;
    }
}
```

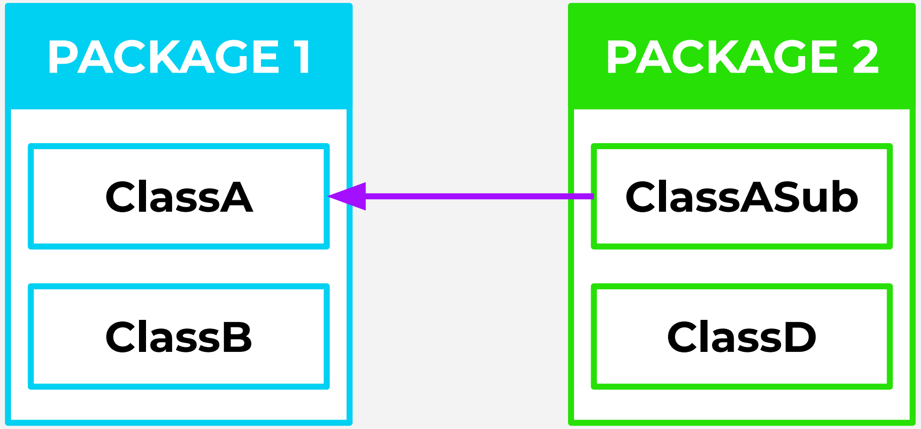
Field is not visible (private) and will result in an error

[Recall] Access Modifiers – Protected

- Methods or variables declared as protected are accessible by subclasses within any package
- Classes can't be declared protected. This access modifier is generally used in a parent-child relationship

We're reached the point where we're discussing this! ☺️

A **PROTECTED** METHOD OR PROPERTY IN **ClassA...**



Can be accessed by **subclasses** from **any package**

```
public class Pet {
    protected String name;
    protected int age;

    public Pet(String name, int age){
        this.name = name;
        this.age = age;
    }
}
```

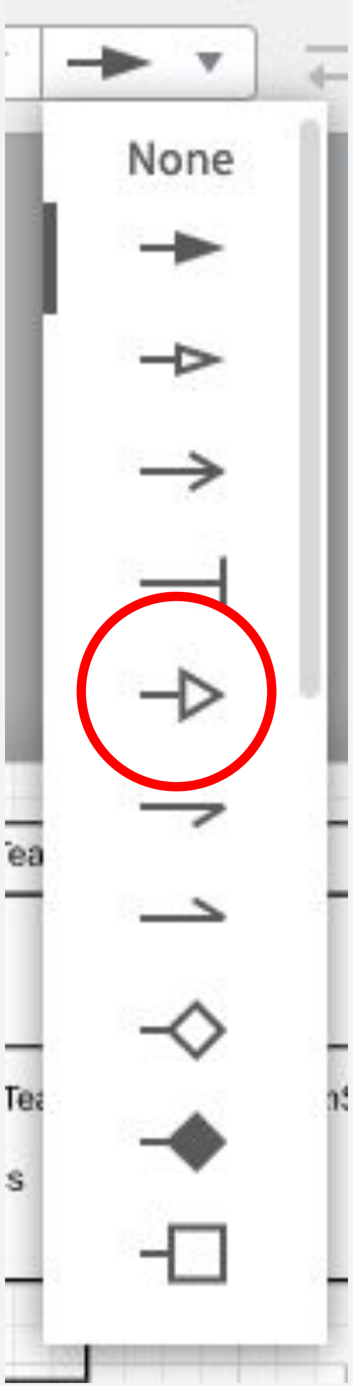
```
public class Dog extends Pet {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age);
        this.breed = breed;
    }

    public int getRealAge() {
        return age * 7;
    }
}
```

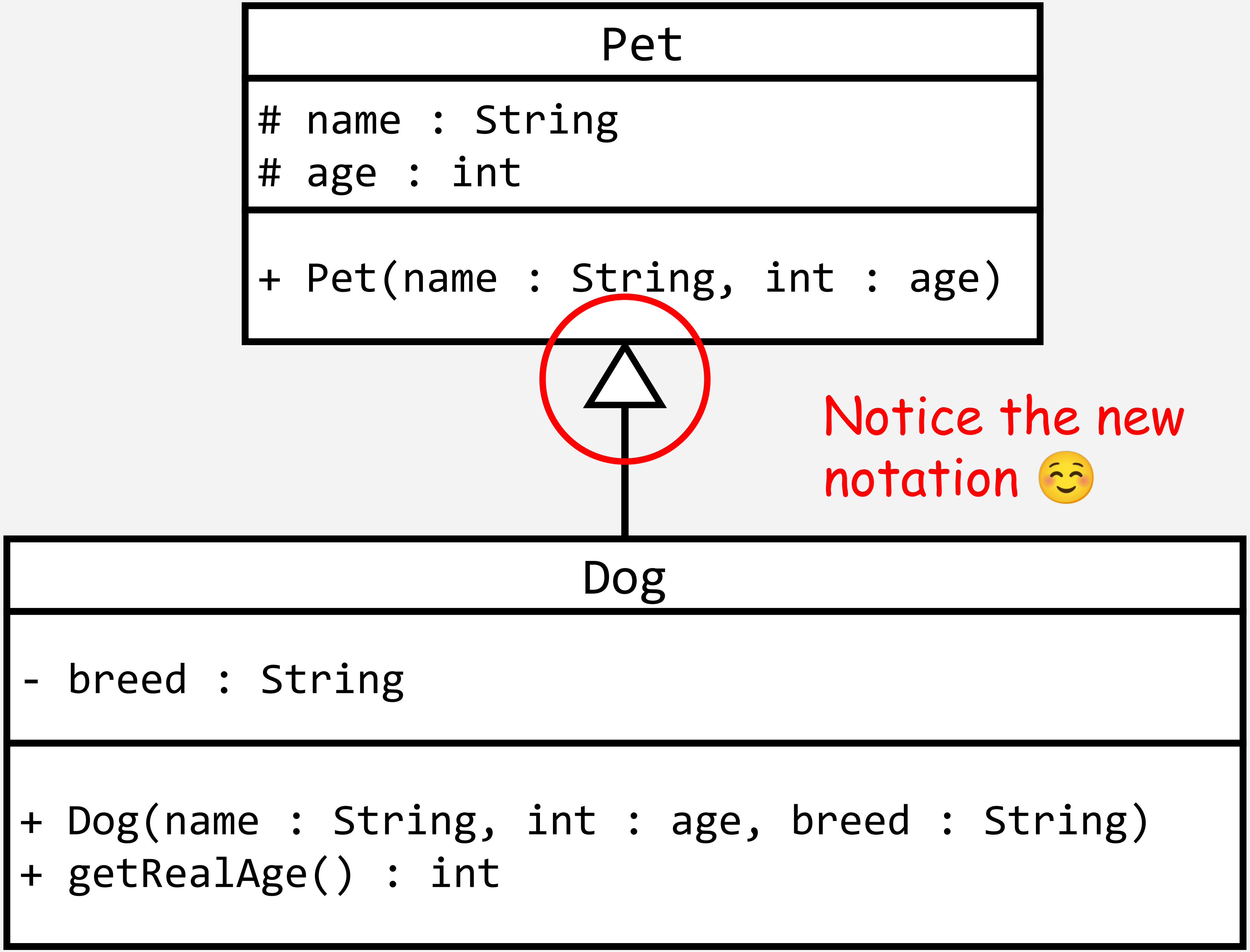
You can even write…
`super.age`
to distinguish between super and sub class members

Field is visible (protected) and is allowed

Disclaimer on using Protected

- Use the access modifier only if there truly is a need for subclasses to have access to the superclass
- Another way around keeping classes responsible for themselves (i.e. encapsulation) would be to have private variables in the superclass and utilize getters to pass the values needed
    - super.getterMethod();





```
Pet
# name : String
# age : int

+ Pet(name : String, int : age)

Dog

- breed : String

+ Dog(name : String, int : age, breed : String)
+ getRealAge() : int
```

Notice the new notation ☺

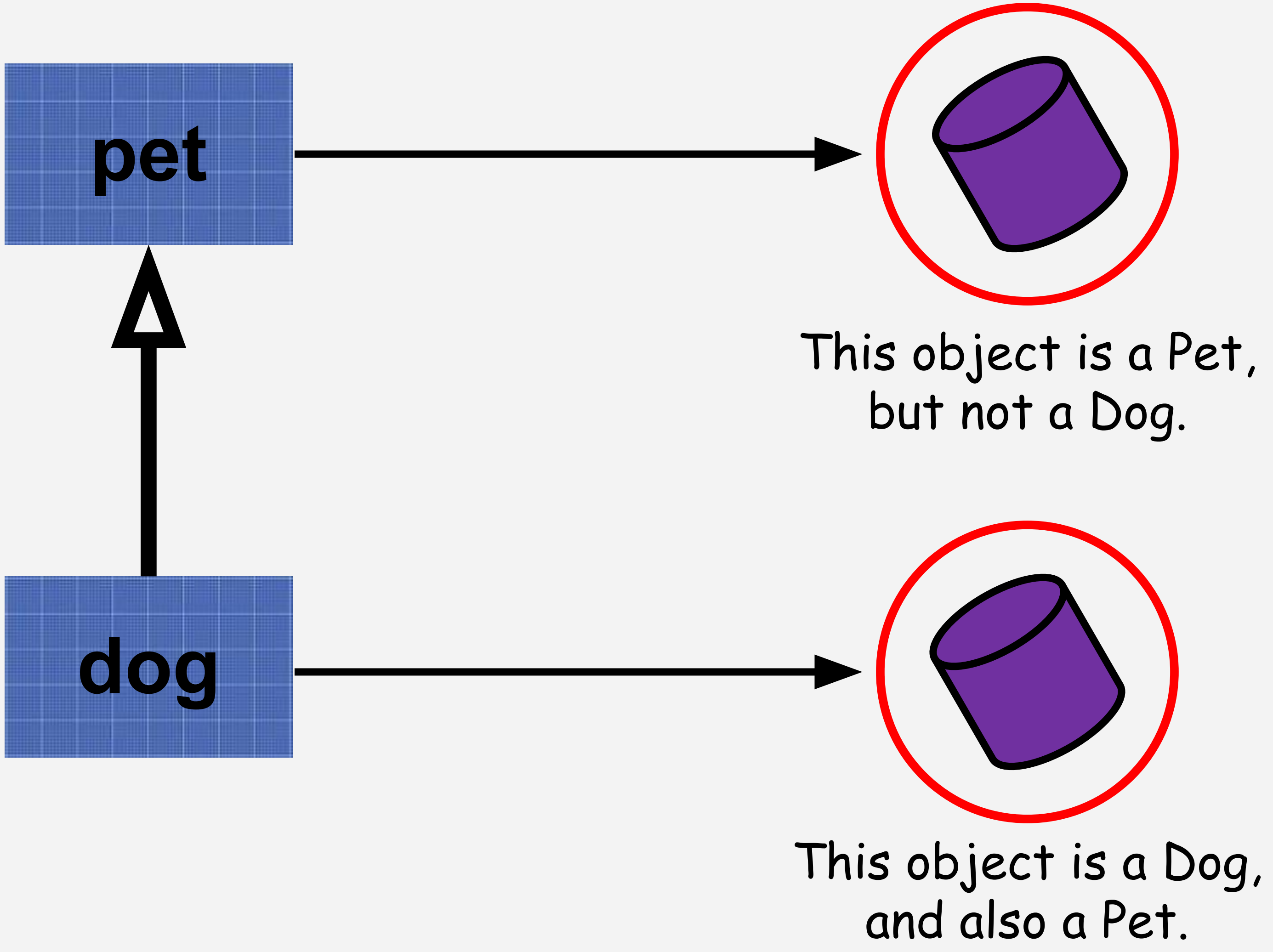
Why use Inheritance?

- OOP highly values reusability without redundancy
  - If multiple classes share the responsibility to maintain characteristics, then there might be merit to elevating the responsibility to a superclass
- Can help in decluttering code
- HOWEVER, do not force inheritance as this can lead to design issues

Questions?

Polymorphism

- Permits a reference to a subclass to be stored in a variable defined to be a reference to an object of the superclass
- Think of typecasting



Question: Which among the following works?

```
public class Main {
   public static void main(String[] args) {
    ✅Pet pet1 = new Pet("Rocky", 3);
    ✅Pet pet2 = new Dog("Munch", 5, "Lhasa Apso");
    ❌Dog dog1 = new Pet("Pew", 1);
    ✅Dog dog2 = new Dog("Guts", 6, "Bulldog");
   }
}
```

```
public class Pet {
    protected String name;
    protected int age;

    public Pet(String name, int age){
        this.name = name;
        this.age = age;
    }
}
```

```
public class Dog extends Pet {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age);
        this.breed = breed;
    }

    public int getRealAge() {
        return age * 7;
    }
}
```

Question: Which among the following works?

```
public class Main {
   public static void main(String[] args) {
      Pet pet = new Dog("Munch", 5, "Lhasa Apso");
      Dog dog = new Dog("Guts", 6, "Bulldog");
```

This won't work because class Pet does not have getRealAge()

```
    ❌System.out.println("" + pet.getRealAge());
    ✅System.out.println("" + dog.getRealAge());
   }
}
```

```
public class Pet {
    protected String name;
    protected int age;

    public Pet(String name, int age){
        this.name = name;
        this.age = age;
    }
}
```

```
public class Dog extends Pet {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age);
        this.breed = breed;
    }

    public int getRealAge() {
        return age * 7;
    }
}
```

To solve this, we can **typecast**...

```
public class Main {
   public static void main(String[] args) {
      Pet pet = new Dog("Munch", 5, "Lhasa Apso");
      Dog dog = new Dog("Guts", 6, "Bulldog");

    ✅System.out.println("" + ((Dog) pet).getRealAge());
    ✅System.out.println("" + dog.getRealAge());
   }
}
```

Typecasting can be useful when you have a list of subclass instances
and need to access class specific members

We can also use the
`instanceof` operator
to check for related instances

instanceof Operator

```
Dog d = new Dog();
Pet p = new Pet();

System.out.println(d instanceof Dog);
System.out.println(d instanceof Pet);
System.out.println(p instanceof Dog);
System.out.println(p instanceof Pet);
```

instance

class

instanceof Operator

```
Pet p = new Dog();

System.out.println(p instanceof Pet);
System.out.println(p instanceof Dog);
```

Practice Exercise 8

- Create a UML following the idea of Inheritance
- We will discuss your answers next, next meeting

Next meeting…

- Design Quiz (1.5 hrs)
    - *Inheritance not yet included*
    - Use Practice Exercise 6 as your review

Keep learning…