CPE207 Object Oriented Programming

Week 9

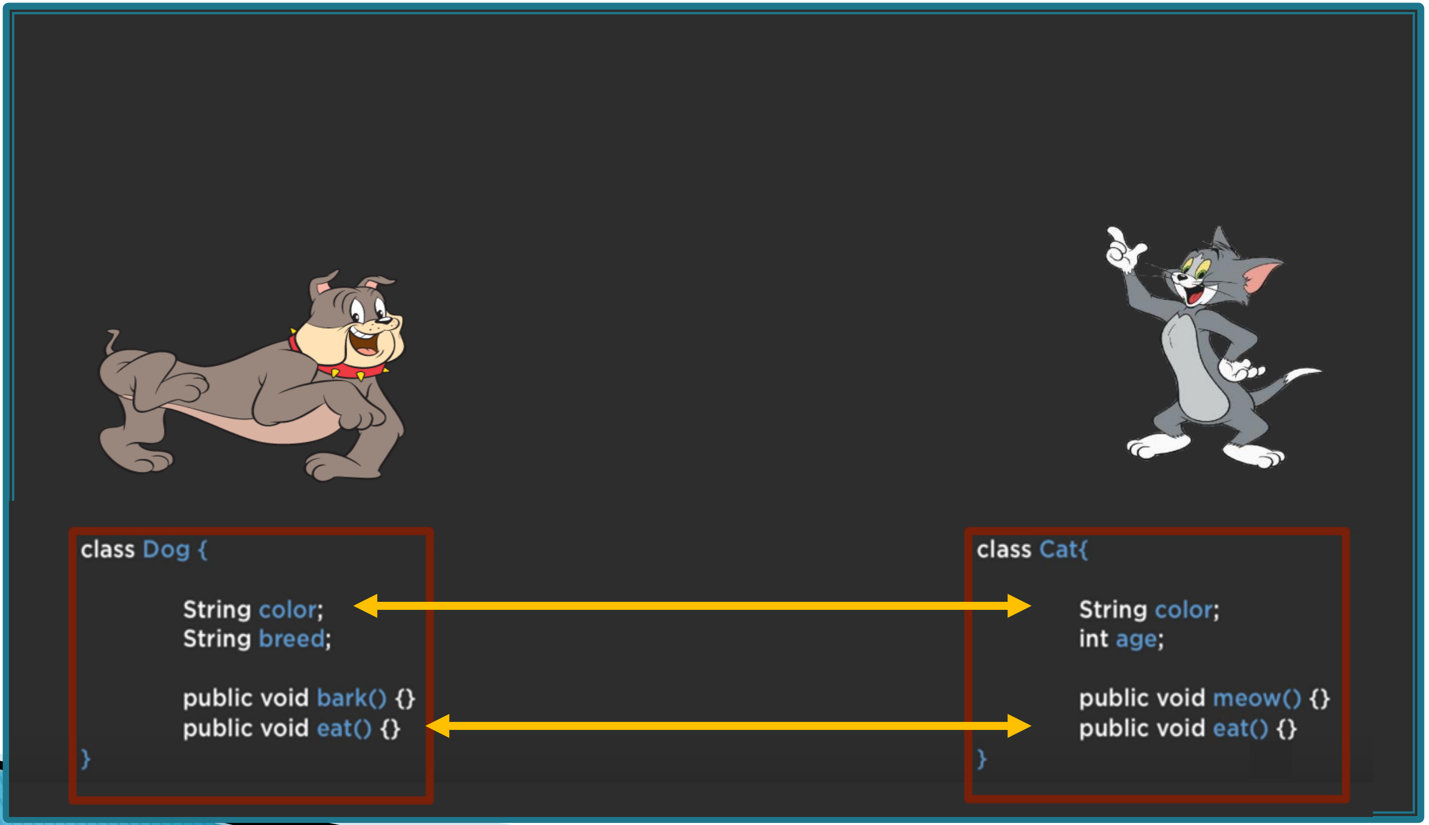
OOP Concepts: Inheritance

Dr. Nehad Ramaha,
Computer Engineering Department
Karabük Universities

These Slides mainly adopted from Assist. Prof. Dr. Ozacar Kasim lecture notes

Table of Contents

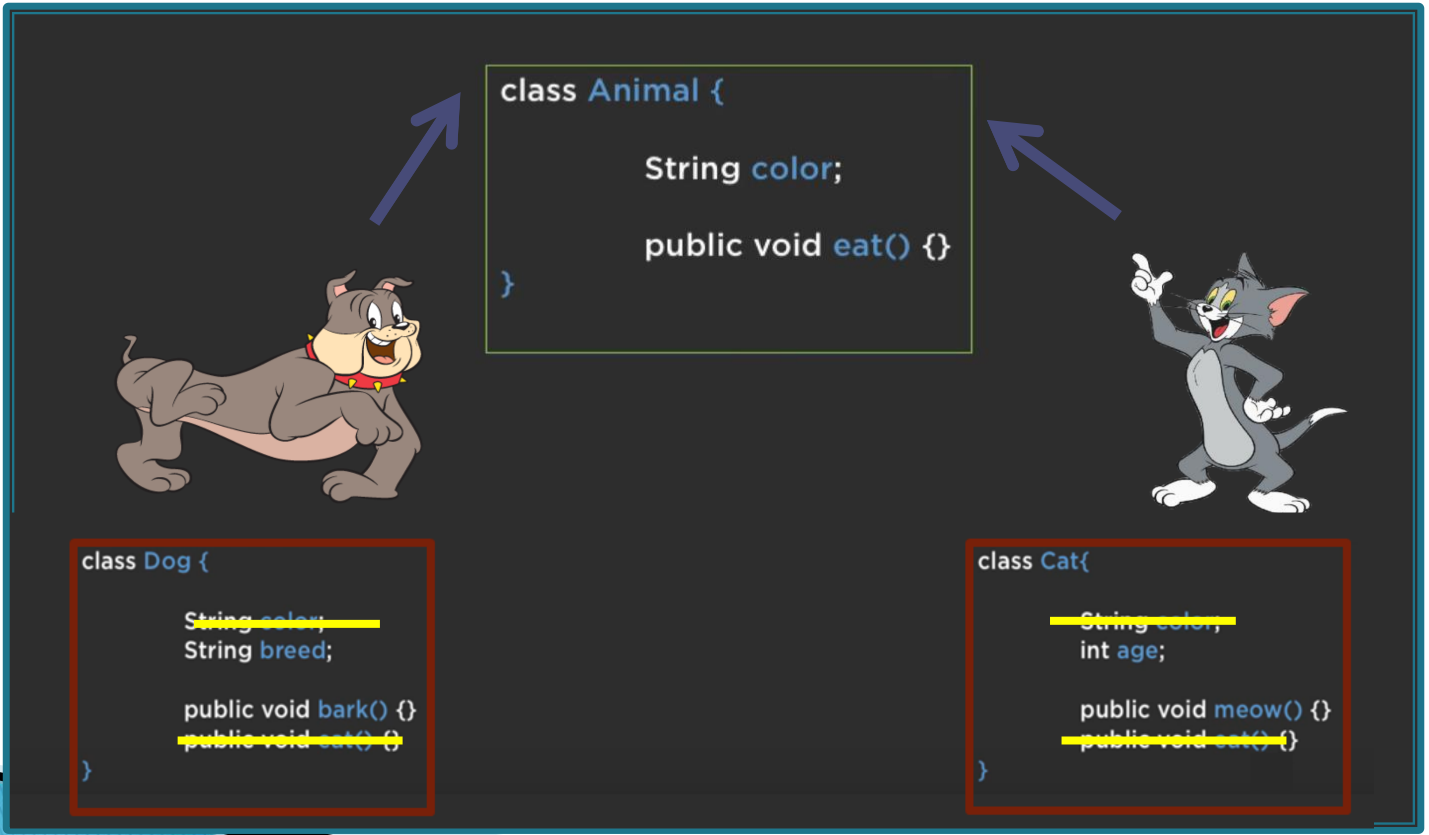
Need for Inheritance



```
class Dog {

        String color;
        String breed;

        public void bark() {}
        public void eat() {}
}
```

```
class Cat{

        String color;
        int age;

        public void meow() {}
        public void eat() {}
}
```

Need for Inheritance

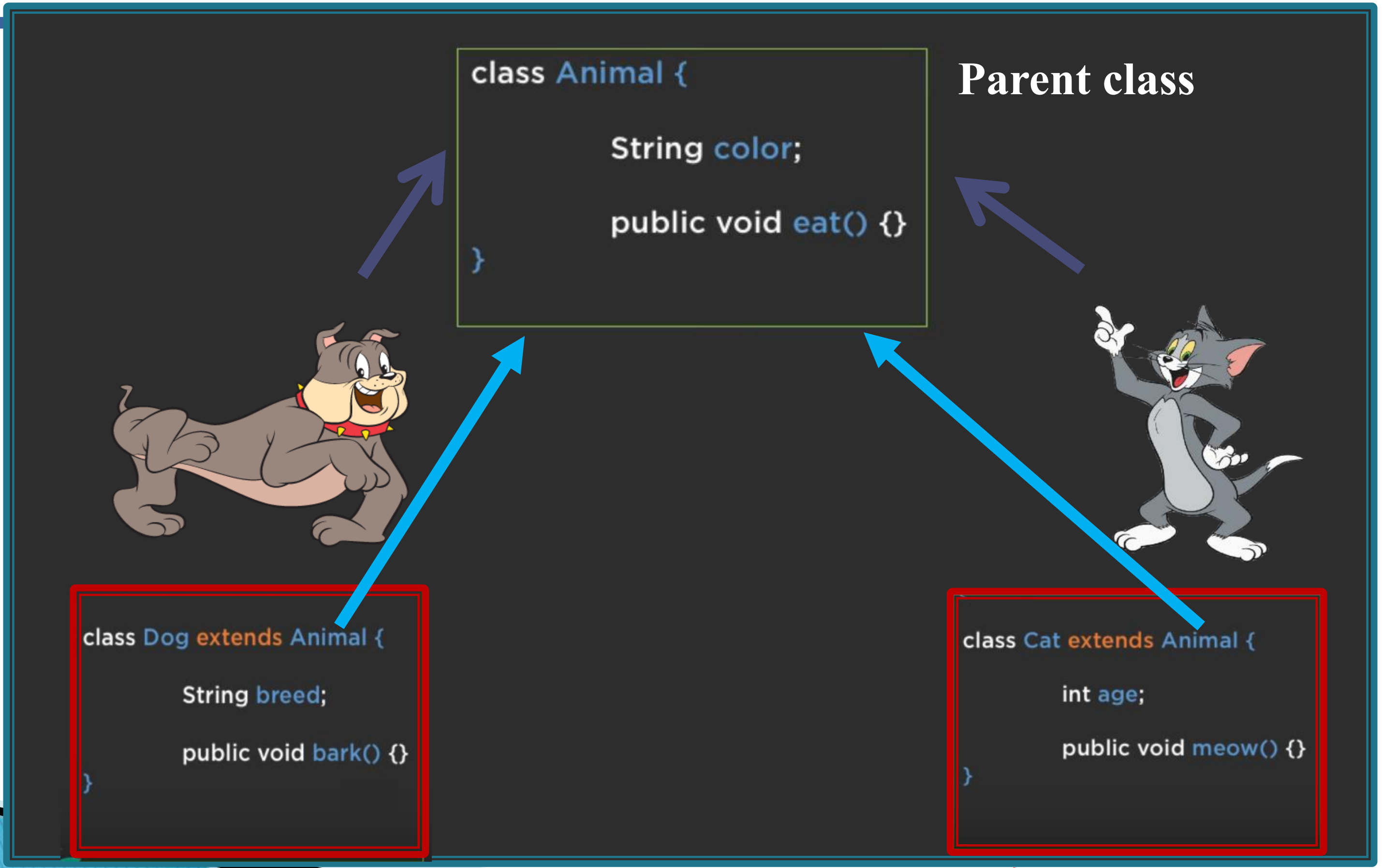


```
class Animal {

        String color;

        public void eat() {}
}
```

```
class Dog {

        ~~String color;~~
        String breed;

        public void bark() {}
        ~~public void eat() {}~~
}
```

```
class Cat{

        ~~String color;~~
        int age;

        public void meow() {}
        ~~public void eat() {}~~
}
```

Need for Inheritance



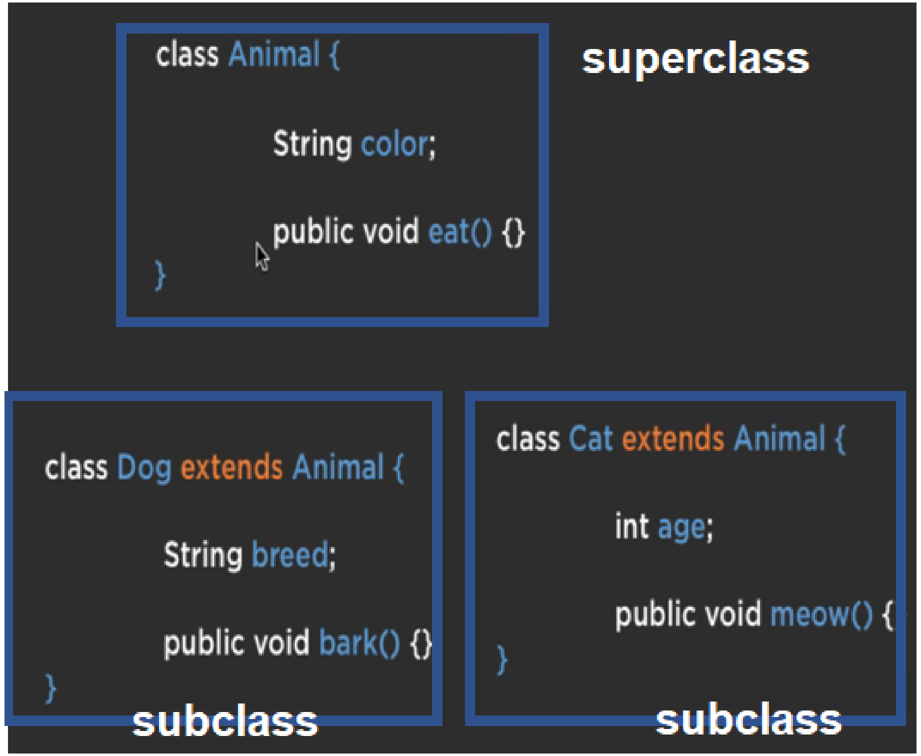
```
class Animal {

        String color;

        public void eat() {}
}
```

```
class Dog extends Animal {

        String breed;

        public void bark() {}
}
```

```
class Cat extends Animal {

        int age;

        public void meow() {}
}
```

Inheritance terms

- **superclass, base class, parent class**: terms to describe the parent in the relationship, which shares its functionality
- **subclass, derived class, child class**: terms to describe the child in the relationship, which accepts functionality from its parent
- **extend, inherit, derive**: become a subclass of another class

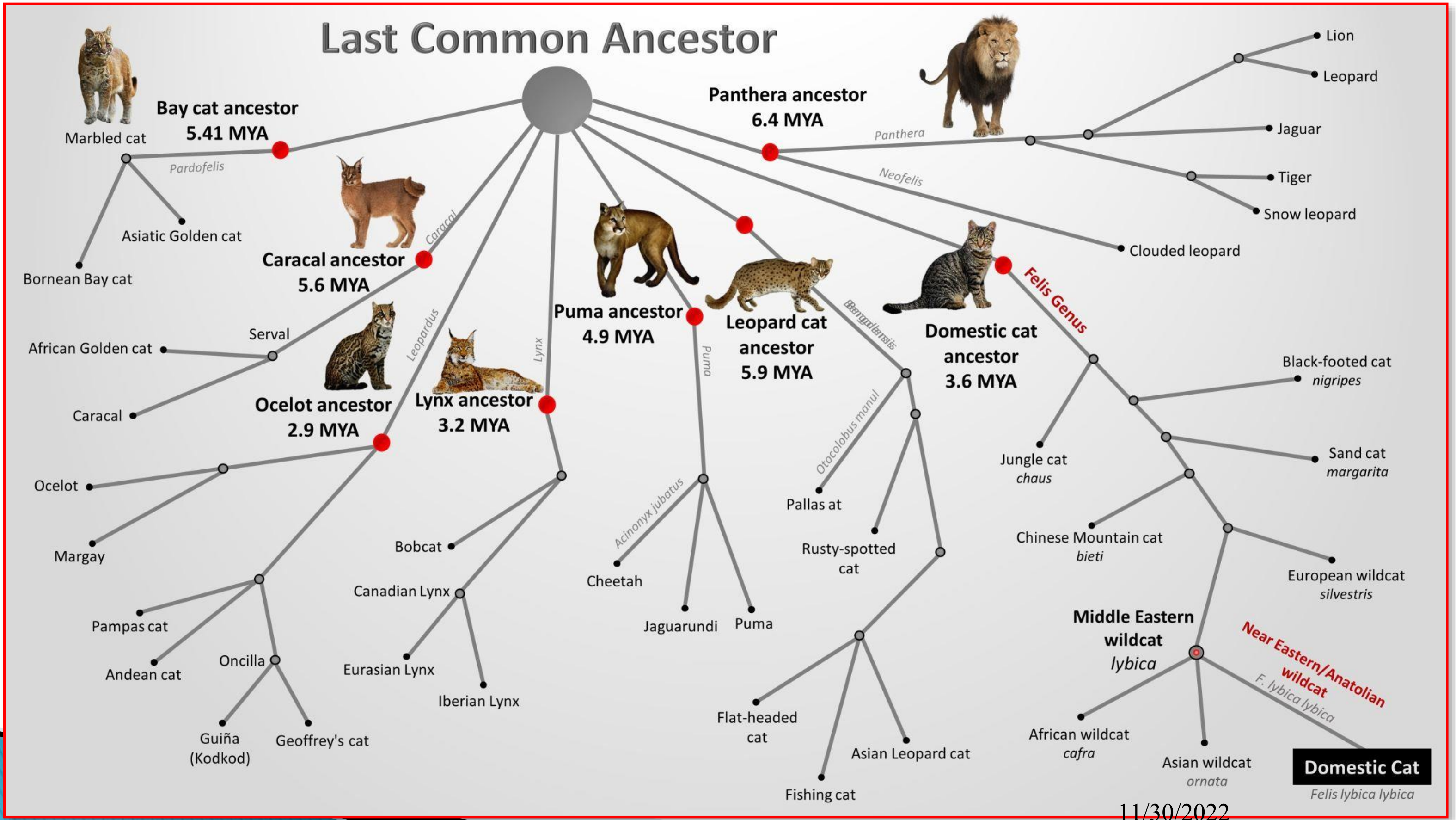


```
class Animal {
        String color;
        public void eat() {}
}
```

```
class Dog extends Animal {
        String breed;
        public void bark() {}
}
```

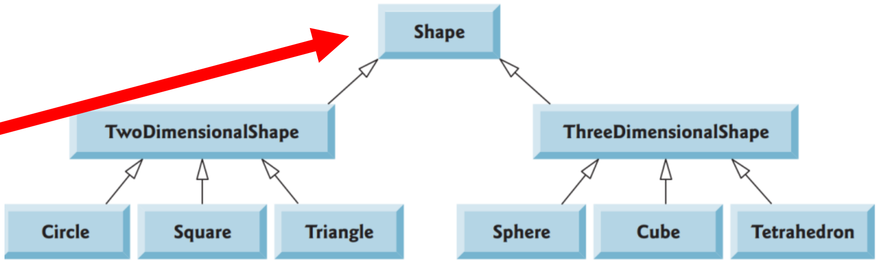
```
class Cat extends Animal {
        int age;
        public void meow() {
}
```

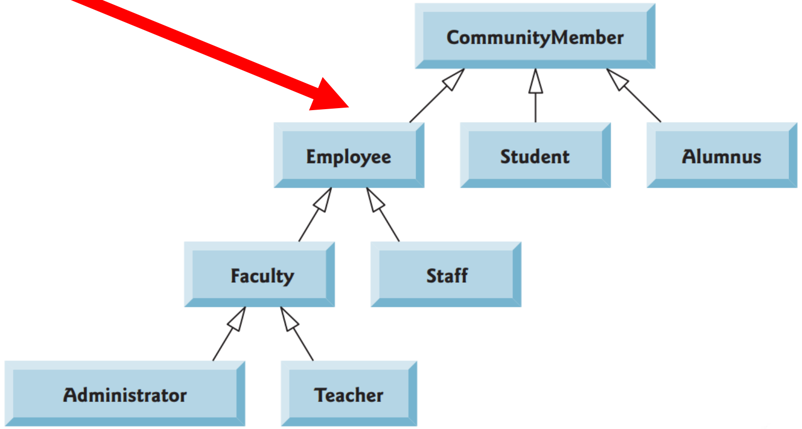
Inheritance: An example



More examples

Superclass	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount





Definition

inheritance: a parent-child relationship between classes

Why use inheritance in java?

1. For Code Reusability

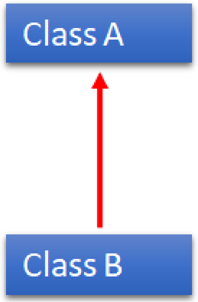
allows to reuse variables and methods of the existing class when you create a child from it.

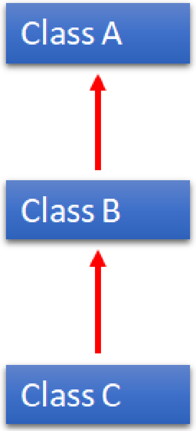
2.For Method Overriding

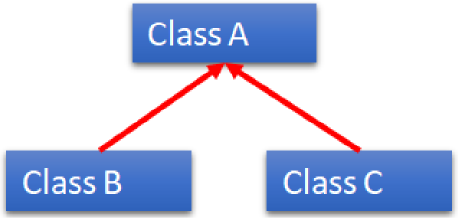
child class can **override** existing behavior from parent

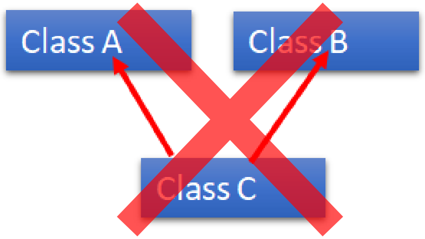
Table of Contents

Types of Inheritance









Not exists in Java

Single inheritance

Multilevel inheritance

Hierarchical inheritance

Inheritance in Java

- In Java, you specify a class as your parent **by using 'extends' keyword**

```
public class Cat extends Animal {
```

- A Java child class has exactly one parent
  - Some other languages (C++) allow multiple inheritance

- by default, a class's parent is `Object`

- constructors are not inherited
  - because they are not members of a class

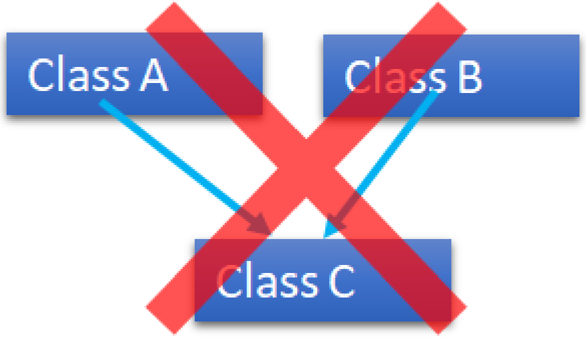


Table of Contents

Inheritance for Code Reusability

```
public class Animal {
    public String color;
    public void eat(){};
}

public class Cat extends Animal {
    public int age;
    public void meow(){};
}

public class Dog extends Animal {
    public String breed;
    public void bark(){};
}
```

A Cat object have

- color
- age
- meow()
- eat()

A Dog object have

- color
- breed
- bark()
- eat()

We reuse codes that in Parent class

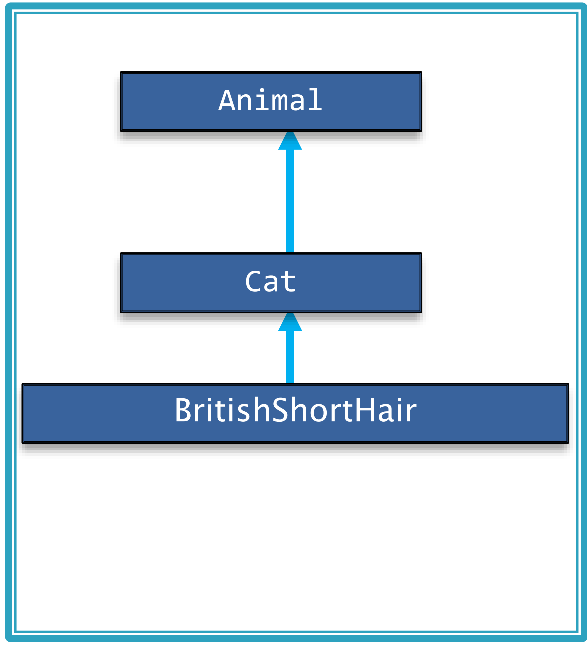
Let's move to Netbeans!

Multiple layers of inheritance

it is possible to extend a class that itself is a child class;

inheritance chains like this can be arbitrarily deep

```
public class BritishShortHair extends Cat {
    String someOtherAttribute;
}
```



MultiLevel inheritance

Table of Contents

Inheritance with Constructor

```java
public class Animal {
    public String color;

    public Animal(String color){
                this.color = color;
    }

    public void eat(){};
}
```

```java
public class Cat extends Animal {
    public int age;

    public Cat(int age, String color) {
        super(color);
        this.age = age;
    }

    public void meow(){};
}
```

```java
public class Dog extends Animal {
    public String breed;

    public Dog(String breed, String color) {
        super(color);
        this.breed = breed;
    }

    public void bark(){};
}
```

Remark: if the superclass has a constructor that requires any arguments (not `default constructor`),
you *must* put a constructor in the subclass and have it call the super-constructor
(call to super-constructor must be the first statement)

Using super keyword

```
class Animal{
    String color="white";
}

class Dog extends Animal{
String color="black";

    void printColor(){
        System.out.println(this.color);
        //prints color of Dog class



System.out.println(super.color);
        //prints color of Animal class
    }
}
```

```
public class Test {
public static void main(String[] args) {
        Dog d=new Dog();
        d.printColor();
    }
}
```

super keyword

- used to refer to superclass (parent) of current class
- can be used to refer to parent class's methods, variables, constructors to call them
  - needed when there is a name conflict with current class
- useful when overriding and you want to keep the old behavior but add new behavior to it (method overriding)

- syntax:

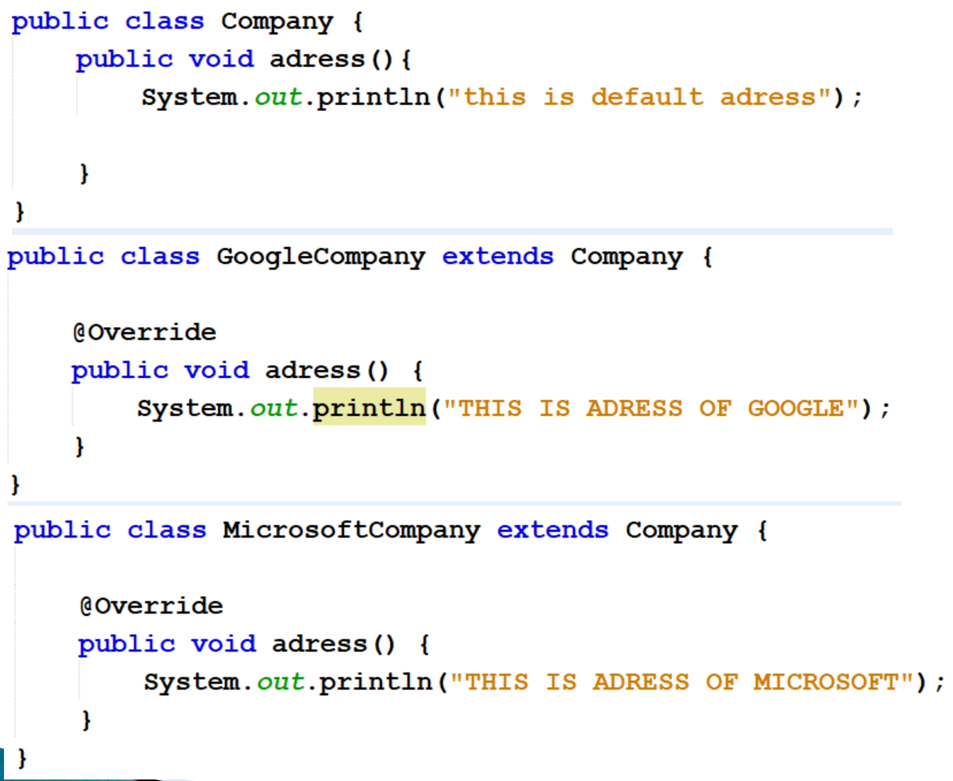
```
super(args);                // call parent’s constructor
super.attributeName          // access parent’s attribute
super.methodName(args);   // access parent’s method
```

Table of Contents

Inheritance for Method Overriding

- In OOP, overriding means to override the functionality of an existing method.
- Child class can replace behavior of its parent's methods by redefining them
- a subclass can implement a parent class method based on its requirement

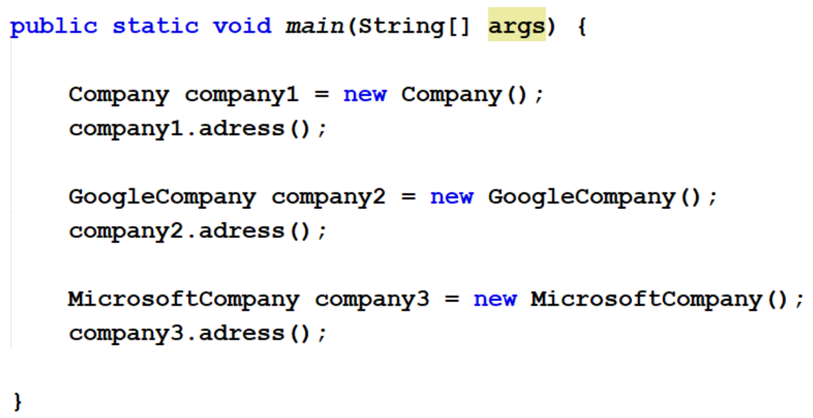
Method Overriding: Example



```
public class Company {
    public void adress(){
        System.out.println("this is default adress");

    }
}
```

```
public class GoogleCompany extends Company {

    @Override
    public void adress()  {
        System.out.println("THIS IS ADRESS OF GOOGLE");
    }
}
```

```
public class MicrosoftCompany extends Company {

    @Override
    public void adress()  {
        System.out.println("THIS IS ADRESS OF MICROSOFT");
    }
}
```



```
public static void main(String[] args)  {

    Company company1 = new Company();
    company1.adress();

    GoogleCompany company2 = new GoogleCompany();
    company2.adress();

    MicrosoftCompany company3 = new MicrosoftCompany();
    company3.adress();

}
```

you have already done this ...

where?

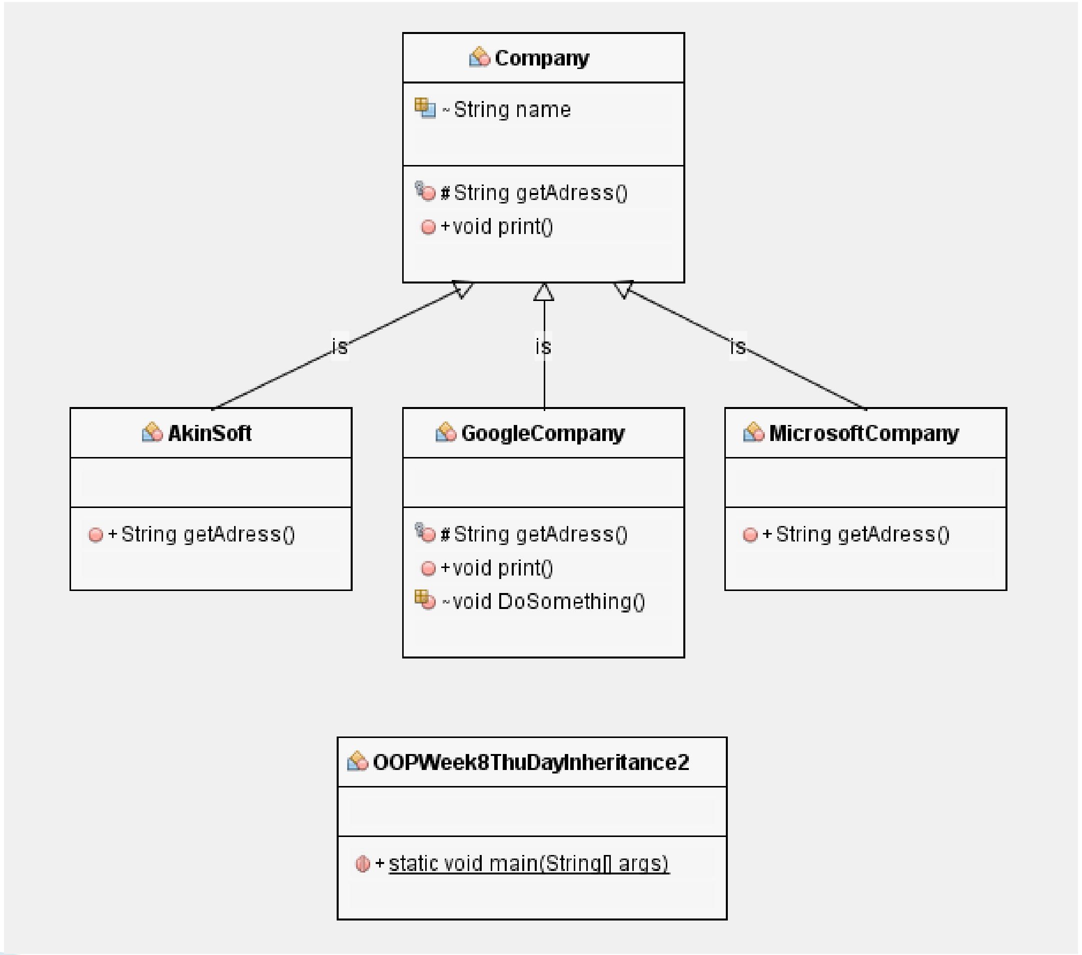
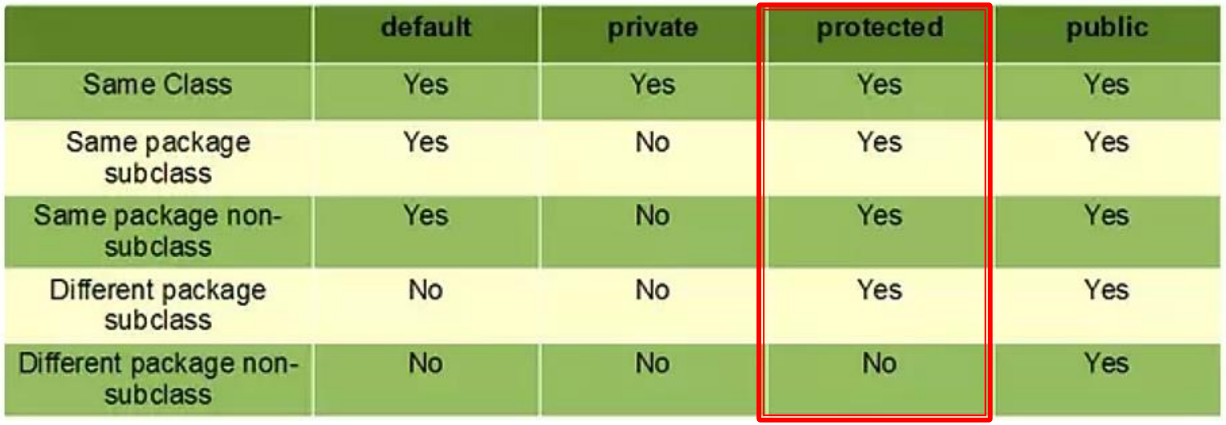

Company
~String name
#String getAdress()
+void print()
is
is
is
AkinSoft
+String getAdress()
GoogleCompany
#String getAdress()
+void print()
~void DoSomething()
MicrosoftCompany
+String getAdress()
OOPWeek8ThuDayInheritance2
+static void main(String[] args)

Table of Contents

Access modifiers : protected

- **public**: visible to all other classes
  public class Animal
- **private**: visible only to the current class, its methods, and every instance (object) of its class
  - a child class cannot refer to its parent's private members!
    private String name;
- **protected** (this one's new to us): visible to the current class, and all of its child classes
  protected int age;



	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Access Control and Inheritance

The following rules for inherited methods are enforced :

1. Public methods in a superclass also must be public in all subclasses.
2. Protected methods in a superclass must be protected or public in subclasses; **they cannot be private.**
3. Private methods are not inherited at all, so there is no rule for them.

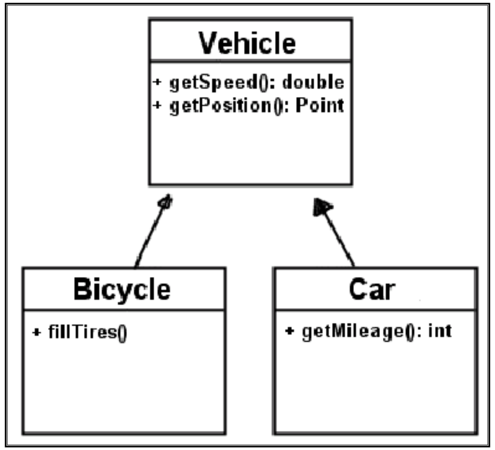
Example: An Access modifier problem

```
public class Parent {
   private int attribute1;
   protected int attribute2;
   public int attribute3;
   protected final static int
   attribute5=1;

  private void method1() {}
  public void method2() {}
  protected void setAttribute1(int
   value){
   this.attribute1 = value;
  }
}
```

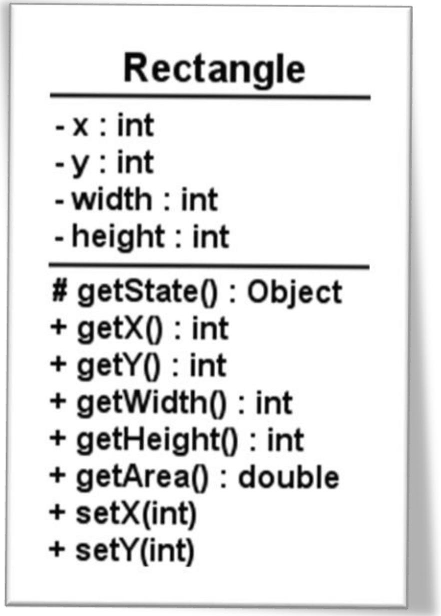
```
public class Child extends Parent {
  public int attribute4;

  public Child() {      // Which are legal?
    attribute4 = 0;          // _________
    attribute1++;          // _______
    attribute2++;          // _______
    attribute3++;          // _______
   attribute5++;          // _________

    super.method1();     // _________
   method2();            // _________

    setAttribute1(attribute4); // _____
  }
}
```

Class diagram: inheritance-1

- classes that have inheritance relationships are connected by **arrows**
- hierarchies drawn top-down with arrows from child to parent

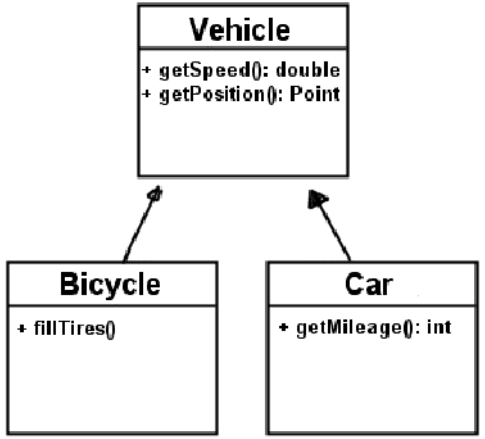


- Attributes
  accessModifier name* : *type
  - – for private
  - + for public
  - # for protected



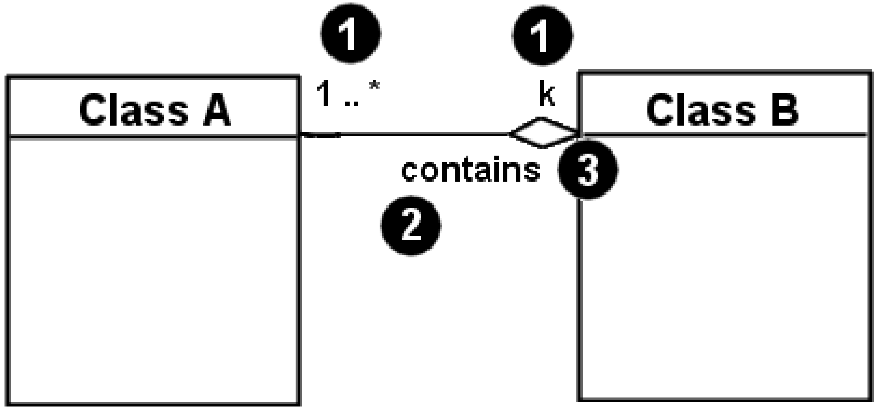
Class diagram: inheritance -2

- **inheritance relationships (is a relationship)**
  - hierarchies drawn top-down with arrows from child to parent

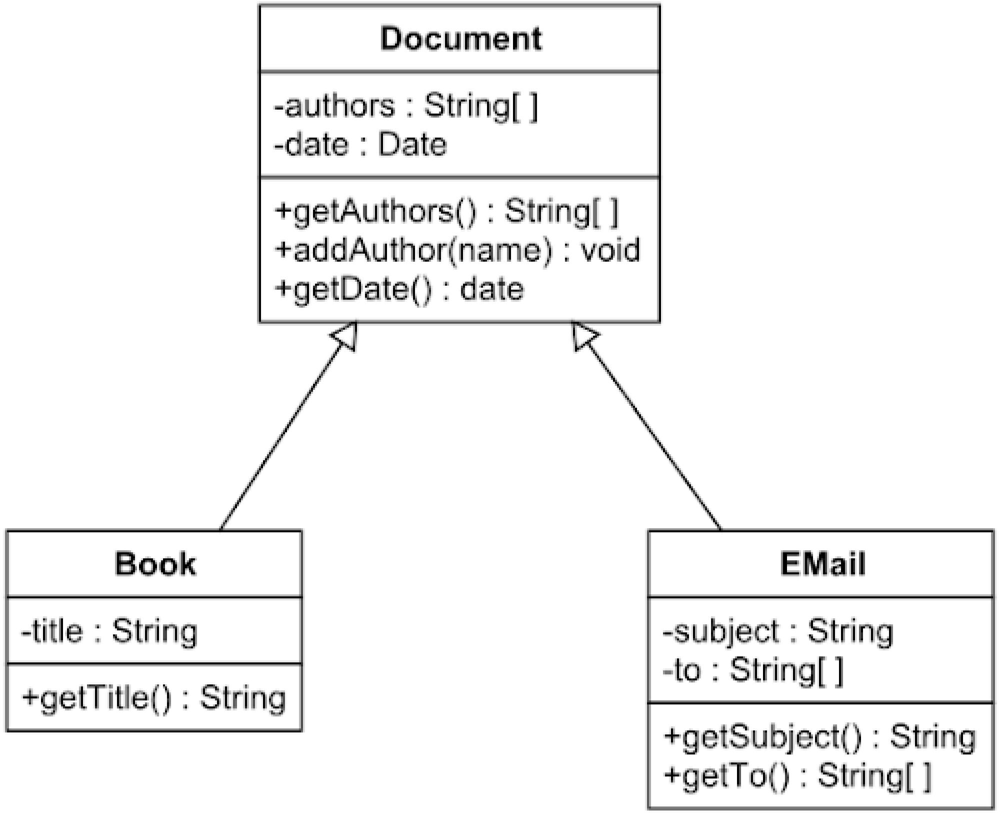


- **associational relationships (has a relationship)**

  1. multiplicity (how many)
  2. name (what relationship the objects have)
  3. navigability (who has relationship with whom)



Lab exercise



Thanks ☺