Java 的对象如何算相同

举出一个场景，你必须改写现有类库的 equals 方法

答：如果两个对象指向同一个内存地址，则直接认为相同。否则继续判断对象“内部”是否相同，具体的，如果是基本类型，则判断基本类型字面量是否相等，如果仍然是对象，则通过该对象自身的相同方法进行判断，严格的判断相同方法最终都会比较到基本类型上。

场景：

假设实现一个 LinkedMap，其中 Class Student 作为 Key，Class Course 作为 Value，Student 的定义如下:

```
class Student {
    String name;
    String stuId;
    int age;
}
```

实现的业务是，存放一个<student, course>到该 map 中，如果 map 中并没有该学生，则增加信息，如果存在过该学生，则进行课程信息替换。这里就需要比较学生对象是否相同。比较的代码如下：

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Student student = (Student) o;
    return age == student.age &&
            Objects.equals(name, student.name) &&
            Objects.equals(stuId, student.stuId);
}
```

比较方法中，限定了子类与父类不相同，因此采用 getClass()方法，如果允许子类判断，则改为，instanceof，之后判断基本类型 age 是否相同，String 对象是否相同。

总结 JavaScript 语言的面向对象特征，你认为 JavaScript(是/否)归属于面向对象语言的理由是什么？

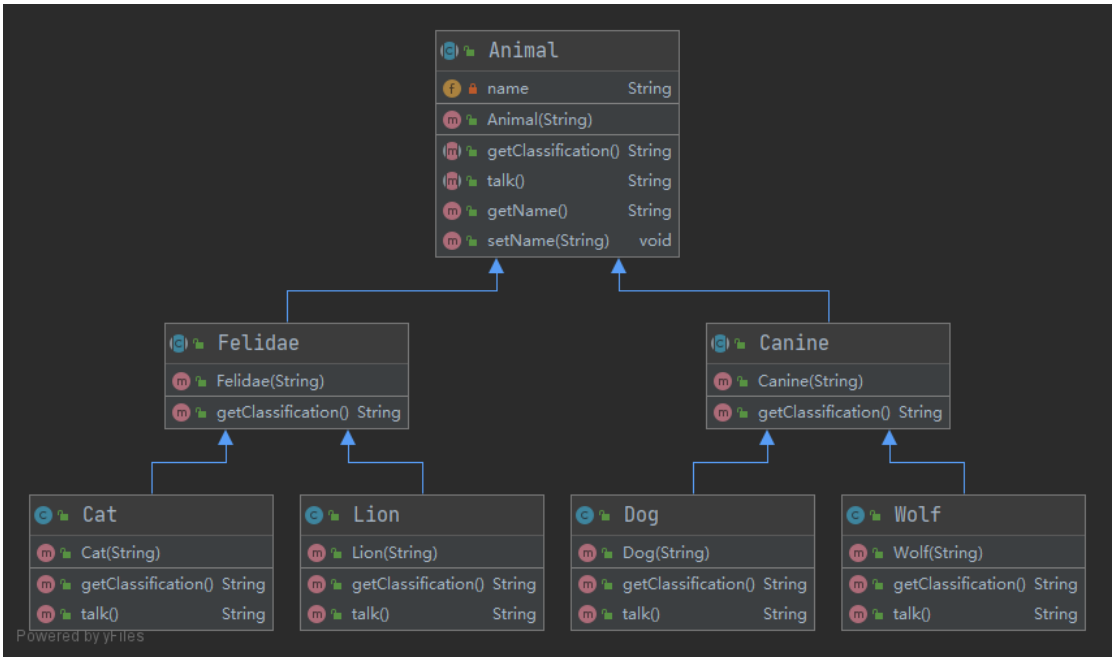
答：是。虽然 JavaScript 并没有提供类似于 Java 和 C#的对象创建“模板”，也没有明确的定义继承方式和多态使用方式，但本质上讲，封装、继承、多态这些特性 js 也都拥有，更合何况从 ES6 开始，js 就已经提供了 class 关键字用于定义类，js 中通过构造器创建对象，并且状态属性和行为都被抽象成了属性，且拥有高度的动态性，可以在运行时动态的为对象添加属性，和修改/获取属性的属性（property）。继承方面，采用原型链继承，通过查找原型链访问父类的属性，与主流面向对象不同的是，原型上的属性是共享的，一个实例修改了原型的属性，则另一个实例的原型属性也会被修改。多态方面，只要是通过 isintance of 判断的，都可以调用其方法，符合动态调用思想。

因此，虽然与主流的面向对象语言设计不同，但本质是一样的，判断一个语言是否属于某一类语言，不应被其他该类语言的特征所左右，而是应回归语言范式的本质。

class TalkingClock 是一个类，class TimePrinter 是一个类，为什么 TimePrinter 可以使用 TalkingClock 的私有变量，请分析这么使用的潜在安全风险。

答：内部类是类之前的嵌套关系，而并不是类实例间的嵌套关系，使用内部类仅是为了命名控制和访问控制，然而内部类可以使用外部类的数据空间，是因为在编译过程中，编译器自动的将内部类翻译为了用$分割外部类和内部类名的常规类文件，使得相当于在构造内部类时，内部类对象创建了一个对外部类对象的引用，通过该引用可以访问到外部类的属性。此时，如果通过反射机制创建了该类对象，就可以访问到原本访问不到的私有变量了。

多态作业



类及类关系：

abstract class Animal：提供公共方法，诸如 talk(),getClassification(),构造方法，getname()和 setname。

abstract class Felidae：实现父类 getClassification()方法。

abstract class Canine：实现父类 getClassification()方法。

class Cat：重写 Felidae 类 getClassification()方法（实现更细节的动物分类），实现父类 talk()方法。

class Lion：重写 Felidae 类 getClassification()方法（实现更细节的动物分类），实现父类 talk()方法。

class Cat：重写 Canine 类 getClassification()方法（实现更细节的动物分类），实现父类 talk()方法。

class Lion：重写 Canine 类 getClassification()方法（实现更细节的动物分类），实现父类 talk()方法。

运行结果：

```
class: cat of Felidae   name: catty     talk: meow
class: lion of Felidae  name: xinba     talk: roar
class: dog of Canine    name: doggy     talk: woof
class: wolf of Canine   name: lang      talk: howl
class: null name: animal        talk: null
class: Felidae  name: felidae       talk: null
```

代码：

Animal.java

```java
package lec04;

public abstract class Animal {
    private String name;

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

    public abstract String getClassification();

    public abstract String talk();

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Canine.java

```java
package lec04;

public abstract class Canine extends Animal{
    public Canine(String name) {
        super(name);
    }

    @Override
    public String getClassification() {
        return "Canine";
    }
}
```

Felidae.java

```
package lec04;

public abstract class Felidae extends Animal {
    public Felidae(String name) {
        super(name);
    }

    @Override
    public String getClassification() {
        return "Felidae";
    }
}
```

Cat.java

```
package lec04;

public class Cat extends Felidae {
    @Override
    public String getClassification() {
        return "cat of "+super.getClassification();
    }

    public Cat(String name) {
        super(name);
    }

    @Override
    public String talk() {
        return "meow";
    }
}
```

Dog.java

```
package lec04;

public class Dog extends Canine{
    @Override
    public String getClassification() {
        return "dog of "+super.getClassification();
    }

    public Dog(String name) {
        super(name);
    }
```

```
    @Override
    public String talk() {
        return "woof";
    }
}
```

Lion.java

```
package lec04;

public class Lion extends Felidae{
    @Override
    public String getClassification() {
        return "lion of "+super.getClassification();
    }

    public Lion(String name) {
        super(name);
    }

    @Override
    public String talk() {
        return "roar";
    }
}
```

Wolf.java

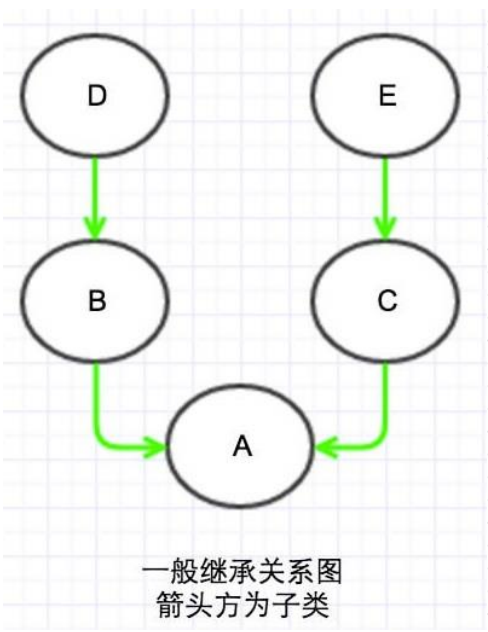
```
package lec04;

public class Wolf extends Canine {
    @Override
    public String getClassification() {
        return "wolf of "+super.getClassification();
    }

    public Wolf(String name) {
        super(name);
    }

    @Override
    public String talk() {
        return "howl";
    }
}
```

Test.java

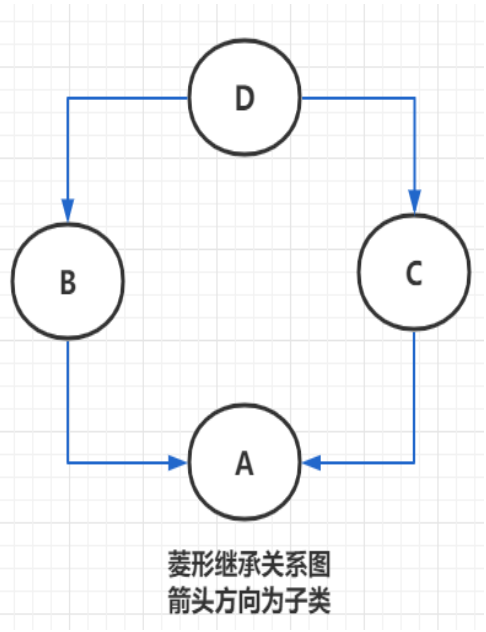
```
package lec04;
```

```
import java.util.ArrayList;

public class test {
    public static void main(String[] args) {
        Animal cat = new Cat("catty");
        Animal lion = new Lion("xinba");
        Animal dog = new Dog("doggy");
        Animal wolf = new Wolf("lang");
        Animal aanimal=new Animal("animal") {
            @Override
            public String getClassification() {
                return null;
            }

            @Override
            public String talk() {
                return null;
            }
        };
        Animal felidae= new Felidae("felidae") {
            @Override
            public String talk() {
                return null;
            }
        };
        ArrayList<Animal> animals = new ArrayList<>();
        animals.add(cat);
        animals.add(lion);
        animals.add(dog);
        animals.add(wolf);
        animals.add(aanimal);
        animals.add(felidae);

        for (Animal animal : animals) {
            System.out.println("class: " + animal.getClassification() +
"\tname: " + animal.getName() + "\t\ttalk: " + animal.talk());
        }
    }
}
```

查阅 Python 中 MRO 生成算法(DFS、BFS 和 C3 算法)，并根据 C3 算法写出如下两幅图的 MRO 列表



一般继承关系图
箭头方为子类



菱形继承关系图
箭头方向为子类

答：

(<class '__main__.A'>, <class '__main__.B'>, <class '__main__.D'>, <class '__main__.C'>, <class '__main__.E'>)

(<class '__main__.A'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.D'>)