

复旦大学计算机科学技术学院



# 编程方法与技术

## 6.1. 面上对象复习

周扬帆

2021-2022第一学期

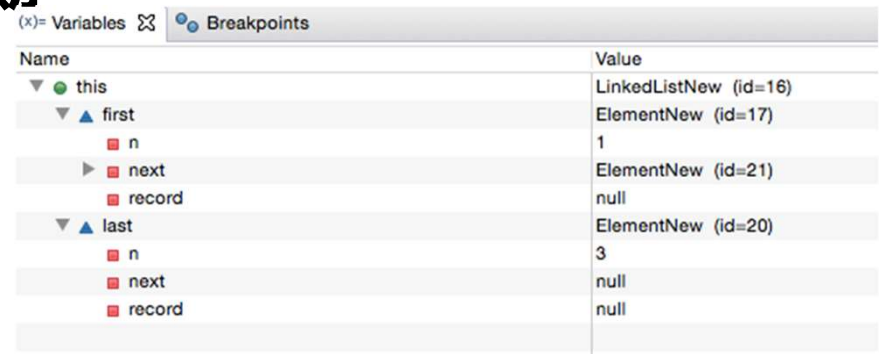
# 课堂练习：程序的调试

## □ 掌握简单测试用例的设计

- Add 1 2, Remove 2 1, Add 3 (Bug 1)
- Add 1 2 2 2 3 4, Remove 2 (Bug 2)

## □ 基本的调试方法

- **单步执行** (step into, step over)
  - `int a = b.getNum();`
- 执行到**断点**
- **看变量的值**是否符合预期



The screenshot shows a debugger's 'Variables' window. It displays the state of a linked list. The 'this' variable points to a 'LinkedListNew' object (id=16). This object has two main fields: 'first' and 'last', both of type 'ElementNew'. The 'first' field points to an 'ElementNew' object (id=17) with 'n' value 1 and 'next' pointing to another 'ElementNew' object (id=21). The 'last' field points to an 'ElementNew' object (id=20) with 'n' value 3 and 'next' set to null. The 'record' field in both 'ElementNew' objects is null.

Name	Value
this	LinkedListNew (id=16)
first	ElementNew (id=17)
n	1
next	ElementNew (id=21)
record	null
last	ElementNew (id=20)
n	3
next	null
record	null

# 类里方法的重载:总结

## □ 重载目的

- 简洁命名、方便理解
- 方便一个类实现多种构造方法

## □ 重载方法

- 一个类里两个方法的名字可以一样
- **参数表不一样**
- 返回值可以一样或者不一样
- 调用时, 根据参数不同, 找对应的方法
- **不能**仅靠返回值不一样 (参数表一样) 重载

# 类的继承

- ❑ 目的：数据结构和操作方法的复用
- ❑ 在类的定义时，使用extends关键字，指定复用的类

子类、派生类、扩展类、导出类

父类、基类、超类

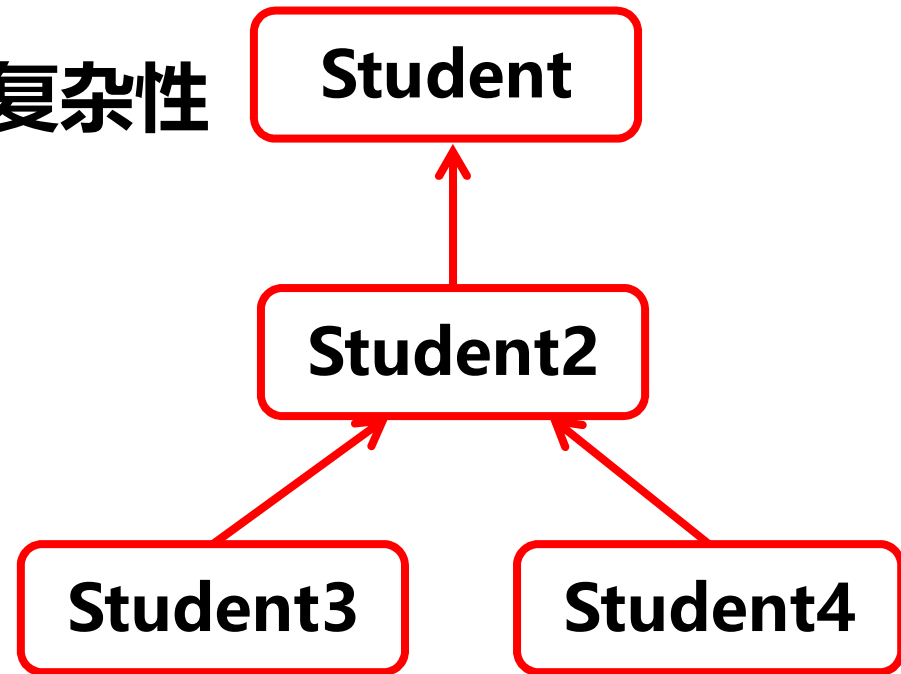
```
class Student2 extends Student
{
    String address;
    void setAddress(String studentAddress)
    {
        address = studentAddress;
    }
};
```

可以新增数据

可以新增  
方法

# 类的继承

- ❑ 子类可以访问父类的
  - **public**方法和变量
  - **protected**方法和变量
- ❑ 不能继承多个父类
  - 避免处理钻石问题的复杂性
- ❑ 可以多次继承



# 不想被继承的类/方法

- **final**关键字
  - **final** class XXX
  - **final** 方法定义
    - **final** String talk()

# 类的继承

- 子类可以对父类进行**方法覆盖及方法重载**
  - 适应子类新定义
- 子类可以对父类的变量进行**覆盖定义**
- 父类的同名变量和同名方法用**super**关键字访问
  - `super.name`
  - `super.foo()`
  - `super`是针对普通方法和变量的（实例）
  - 子类的static方法只能访问父类的static方法和变量(? )
  - 访问父类的static方法和变量**用类名**.访问
  - `super(参数)`显式调用父类构造函数: 位置?

复旦大学计算机科学技术学院



# 编程方法与技术

## 6.2. JavaScript继承

周扬帆

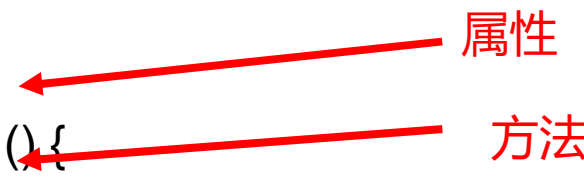
2021-2022第一学期



# JavaScript对象

## ■ 对象的属性和方法可以动态增删

```
var lecturer = {  
  name: 'YZ',  
  teach: function () {  
    alert(this.name + ' teaches nothing useful');  
  }  
}
```



```
lecturer.teach();  
lecturer.title = 'Dr. ';  
lecturer.quit = function () {  
  alert(this.title + this.name + ' quits.');
```

```
}  
lecturer.quit();  
  
delete lecturer.quit;  
lecturer.quit();
```

# JavaScript对象的创建

- **new func(...)**就是以func为构造函数，构造了一个对象，并返回
  - 函数内部的this指向新构建的对象

- 例子

实例属性

实例方法

```
function Lecturer (name) {  
    this.name = name; //var name = name  
    alert(name + ' teaches nothing useful');  
    this.getName = function () {  
        return this.name;  
    }  
}  
var Y = new Lecturer ('Y');  
alert(Y.getName());  
  
var Z = new Lecturer ('Z');  
alert(Z.getName());
```

# JS的对象继承方法

## ❑ 方法1: 通过构造函数的prototype实现继承

```
function Animal() {  
  this.gender = 'male';  
}  
function Dog(){  
  this.talk = function () {  
    console.log('I am a dog');  
  };  
}
```

```
Dog.prototype = new Animal();
```

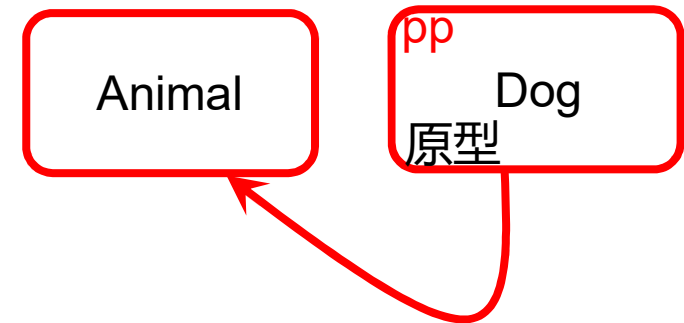
```
var pp = new Dog();
```

```
console.log(pp.gender);
```

创建原型对象

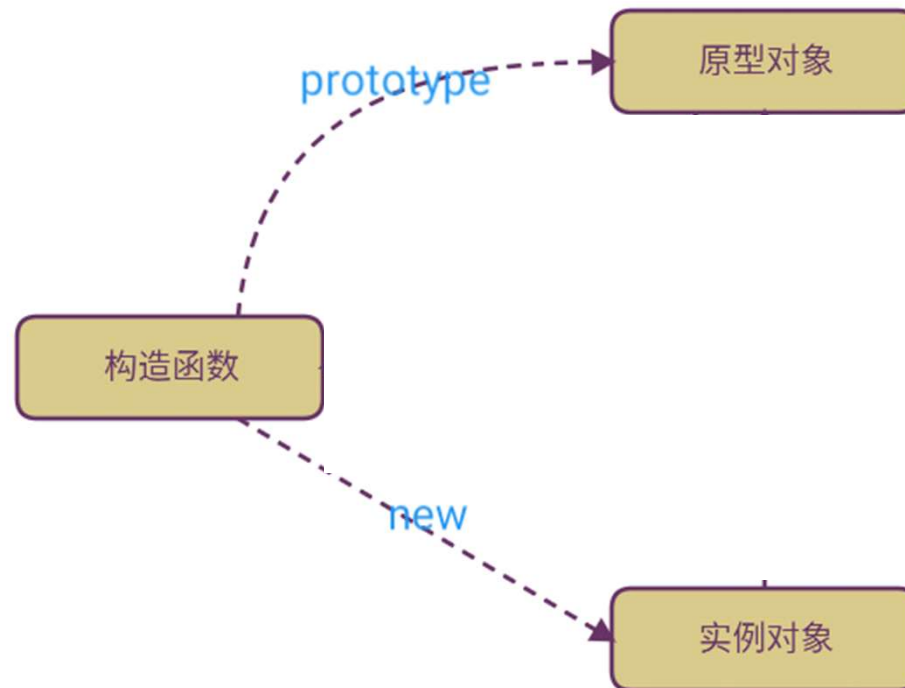
构造对象

引用在原型链上的对象属性



# JS的对象继承方法

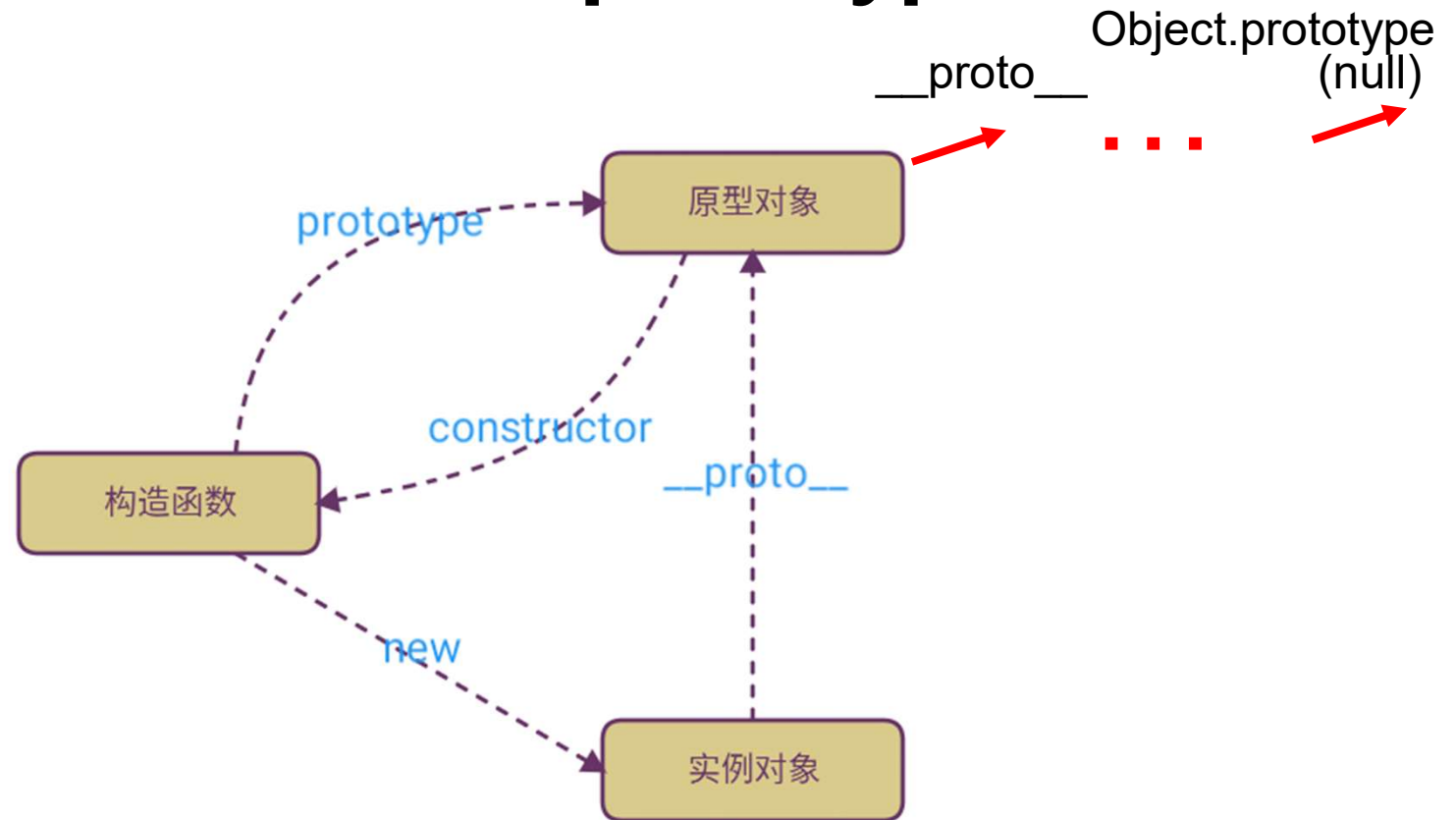
## □ 方法1:通过构造函数的prototype实现继承



```
Dog.prototype = new Animal();  
var pp = new Dog();
```

# JS的对象继承方法

## ❑ 方法1:通过构造函数的prototype实现继承

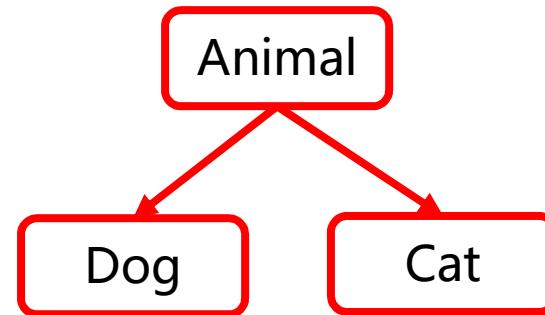


```
Dog.prototype = new Animal();  
var pp = new Dog();
```

# JS的对象继承方法

## ❑ 方法1:通过构造函数的prototype实现继承

```
function Animal(){  
    this.gender = 'male';  
}  
function Dog(){  
    this.talk = function () {  
        console.log('I am a dog');  
    };  
}  
function Cat(){  
    this.talk = function () {  
        console.log('I am a cat');  
    };  
}
```



➡

```
Dog.prototype = new Animal();  
var dog = new Dog();  
Cat.prototype = new Animal();  
var cat = new Cat();  
dog.talk();  
cat.talk();
```

# JS的对象继承方法

## □ 方法1:通过构造函数的prototype实现继承

### ■ 思考/研究

```
function Animal(){  
  this.gender = 'male';  
}  
function Dog(){  
  this.talk = function () {  
    console.log('I am a dog');  
  };  
}  
Dog.prototype = new Animal();  
var pp1 = new Dog();  
var pp2 = new Dog();
```

类似于COW: Copy on Write  
a.b = ... 当前对象a没有b, 创建一个

```
pp1.gender = "female";  
console.log(pp2.gender)
```

pp1和pp2是否共享  
同一个gender变量?

```
▼ pp1 = Dog {talk: ,gender: "female"}  
  gender = "female"  
  > talk = function () {  
  > __proto__ = Animal {gender: "male"}  
▼ pp2 = Dog {talk: }  
  > talk = function () {  
  > __proto__ = Animal {gender: "male"}
```

# JS的对象继承方法

- ❑ 为什么需要继承?
- ❑ 方法1:通过构造函数的prototype实现继承
  - 思考/研究

```
function Animal(){  
  this.gender = new Array();  
}  
function Dog(){  
  this.talk = function () {  
    console.log('I am a dog');  
  };  
}  
Dog.prototype = new Animal();  
var pp1 = new Dog();  
var pp2 = new Dog();  
pp1.gender.push('female');  
pp2.gender.push('male');
```

pp1和pp2是否共享  
同一个array? 为什么

console.log(pp1.gender)



['female', 'male' ]



# JS的对象继承方法

## □ 方法2: 通过构造函数实现继承

```
function Animal(gender){
    this.gender = gender;
}
function Dog(){
    this.talk = function () {
        console.log('I am a dog');
    };
}
Dog.prototype = new Animal(null);
var pp1 = new Dog('male');
var pp2 = new Dog('female');
console.log(pp1.gender);
```

无法把构造参数传给原型构造

利用call/apply!

# JS的对象继承方法

## □ 方法2: 通过构造函数实现继承

```
function Animal(gender){
    this.gender = gender;
}
function Dog(gender){
    Animal.call(this, gender);
    this.talk = function () {
        console.log('I am a ' + this.gender + ' dog');
    };
}
Dog.prototype = new Animal(null);
var pp1 = new Dog('male');
var pp2 = new Dog('female');
pp1.talk();
pp2.talk();
```

Dog函数借助Animal函数，制造了Dog函数构造的对象的属性

# JS的对象继承方法

## □ 方法2: 通过构造函数实现继承

```
function Animal(gender){  
  this.gender = gender;  
  this.getName = function () {  
    console.log('Root');  
  }  
}  
function Dog(gender){  
  Animal.call(this, gender);  
  this.talk = function () {  
    console.log('I am a ' + this.gender + ' dog');  
  };  
}
```

```
Dog.prototype = new Animal(null);  
var pp1 = new Dog('male');  
var pp2 = new Dog('female');  
pp1.talk();  
pp2.talk();  
pp1.getName();
```

几份?

# JS的对象继承方法

## □ 方法2: 通过构造函数实现继承

```
function Root() {  
  this.getName = function () {  
    console.log('Root');  
  }  
}  
  
function Animal(gender){  
  this.gender = gender;  
}  
  
Animal.prototype = new Root();  
function Dog(gender){  
  Animal.call(this, gender);  
  this.talk = function () {  
    console.log('I am a ' + this.gender + ' dog');  
  };  
}
```

```
Dog.prototype = new Animal(null);  
var pp1 = new Dog('male');  
var pp2 = new Dog('female');  
pp1.talk();  
pp2.talk();  
pp1.getName();
```

逻辑（函数）只有一份

Dog构造pp1/pp2  原型 Animal构造  原型 Root构造

# JS的对象继承方法

## □ 方法2: 通过构造函数实现继承

匿名

```
1 function Animal(name, sex) {  
2     this.name = name;  
3     this.sex = sex;  
4 }  
5  
6 Animal.prototype.getName = function () {  
7     return this.name;  
8 }  
9  
10 var cat = new Animal('white', 'male');  
11 cat.getName(); // white  
12  
13 function People(name, sex) {  
14     Animal.call(this, name, sex);  
15 }  
16  
17 People.prototype = new Animal('people', null);  
18  
19 var Chris = new People('Chris', 'male');  
20 Chris.getName(); // Chris
```

# JS的对象继承方法

## ❑ 方法2: 通过构造函数实现继承

```
1 function Animal(name, sex) {  
2     this.name = name;  
3     this.sex = sex;  
4 }  
5  
6 Animal.prototype.getName = function () {  
7     return this.name;  
8 }  
9  
10 var cat = new Animal('white', 'male');  
11 cat.getName(); // white  
12  
13 function People(name, sex) {  
14     Animal.call(this, name, sex);  
15 }  
16  
17 People.prototype = new Animal('people', null);  
18  
19 var Chris = new People('Chris', 'male');  
20 Chris.getName(); // Chris
```

两次调用  
浪费空间



# JS的对象继承方法

## □ 方法3: 寄生组合继承

```
function derive(o) { //Object.create()
  function F() {
  }
  F.prototype = o;
  return new F();
}
```

```
var pp = new Dog('male');
```

```
console.log(pp1.gender);
console.log(pp1.getName());
```

```
function Animal(gender){
  this.gender = gender;
}
Animal.prototype.getName = function(){ return 'Animal';};
```

方法和数据剥离

```
function Dog(gender){
  Animal.call(this, gender);
  // ...
}
```

拷贝数据

```
var proto = derive(Animal.prototype);
proto.constructor = Dog;
Dog.prototype = proto;
```

“虚” 的原型，只负责连接方法



# ES6 Class关键字

## □ 语法糖

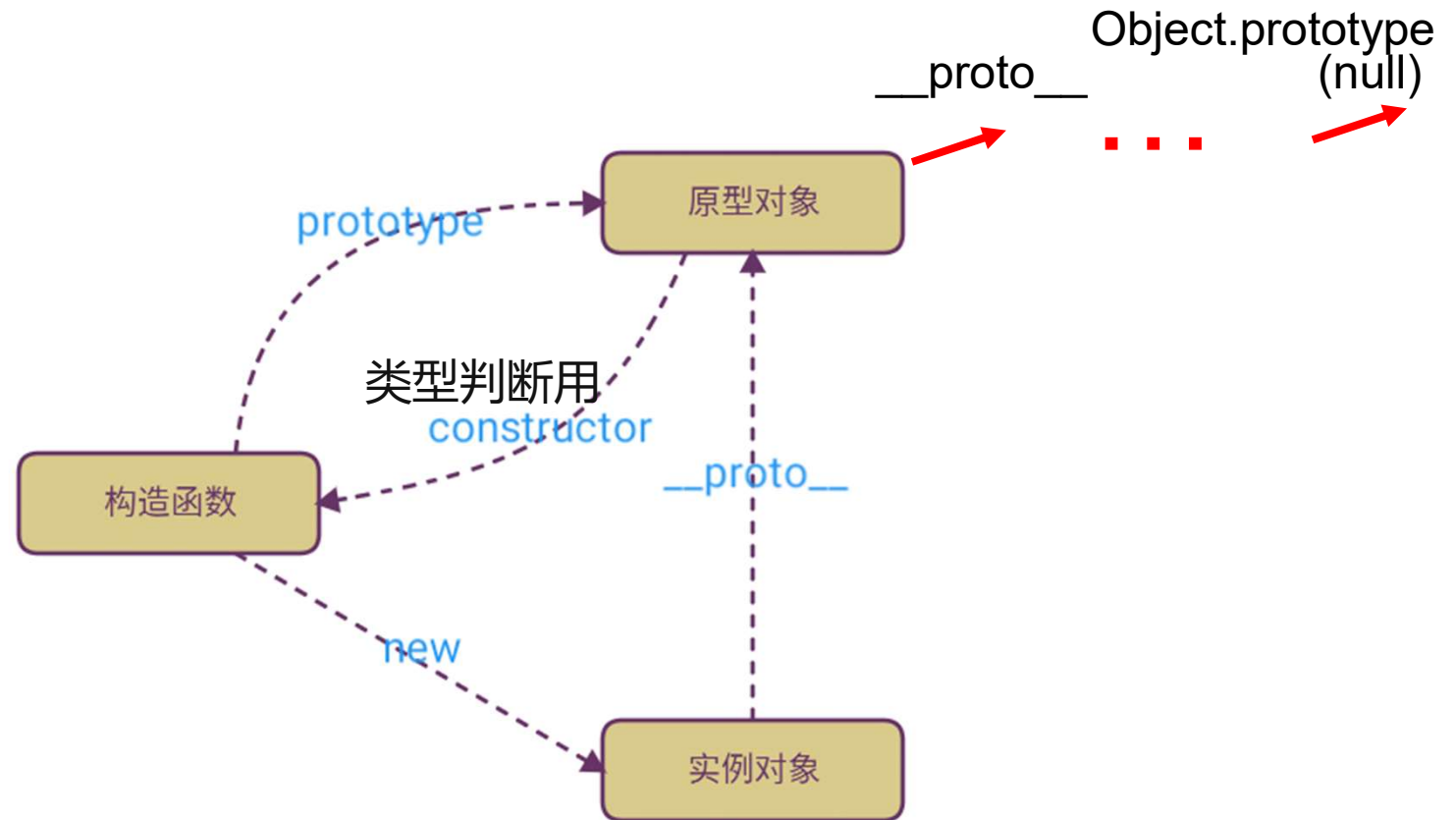
- 简化操作
- 易读

```
1  class Animal {  
2      constructor(name, sex) {  
3          this.name = name;  
4          this.sex = sex;  
5      }  
6  
7      getName() {  
8          return this.name;  
9      }  
10 }  
11  
12 class People extends Animal {  
13     constructor(name, sex) {  
14         super(name, sex);  
15     }  
16 }  
17  
18 var Chris = new People('Chris', 'male');  
19 Chris.getName(); // Chris  
20
```



# JS的对象继承方法

## □ 原型链



原型链查找属性很慢，只要查自己的？ **hasOwnProperty**

复旦大学计算机科学技术学院



# 编程方法与技术

## 6.3. 多态 Polymorphism

周扬帆

2021-2022第一学期

# 更多继承的例子：LinkedList

- 之前我们编写过一个简单的单向链表
- 每一个链表元素定义如下

```
class Element {  
    private int n;  
    private Element next;  
    public void setNext(Element nextElement) {  
        next = nextElement;  
    }  
    public void setNum(int num) {  
        n = num;  
    }  
    public int getNum() {  
        return n;  
    }  
    public Element getNext() {  
        return next;  
    }  
}
```

# 更多继承的例子：LinkedList

- ❑ 新需求：整形变量不够了，每个元素需存一个Student引用
- ❑ 采用继承的方法复用Element类

```
class NewElement extends Element {  
    private Student student;  
    public void setRecord (Student rec) {  
        student = rec;  
    }  
    public Student getRecord () {  
        return rec;  
    }  
    ...  
}
```

针对新数据student的  
新操作方法

# 更多继承的例子：LinkedList

## □ Element 型引用怎么处理？

```
class Element {  
    private int n;  
    private Element next;  
    public void setNext(Element nextElement) {  
        next = nextElement;  
    }  
    public void setNum(int num) {  
        n = num;  
    }  
    public int getNum() {  
        return n;  
    }  
    public Element getNext() {  
        return next;  
    }  
}
```

**相关的代码全部重写？ 太费事，说好的复用呢？**

# 多态Polymorphism

□ 继承: “是一种” (is a)关系

□ Java支持引用的动态绑定

```
class Student2 extends Student {
```

```
...
```

```
};
```

```
Student student = new Student2();  
Student2 student2 = new Student2();  
NewElement ele = new NewElement();  
ele.setRecord(student);
```

```
ele.setRecord(student2);
```

```
public void setRecord (Student rec) {  
    student = rec;  
}
```

■ Student2是一种Student数据, 因此Student2对象可以作为Student对象

■ 把Student2的对象当做其父类Student的对象来使用

# 单向链表的例子

```
class Element {  
    private int n;  
    private Element next;  
    public void setNext(Element nextElement) {  
        next = nextElement;  
    }  
    public void setNum(int num) {  
        n = num;  
    }  
    public int getNum() {  
        return n;  
    }  
    public Element getNext() {  
        return next;  
    }  
}  
class NewElement extends Element {  
    ...  
}
```

**可复用，在NewElement中不需重写**

# 单向链表的例子

```
public class LinkedList {  
    Element first = null;  
    Element last = null;  
    public boolean removeFirst() { ... }  
    public int getFirst() { ... }  
    public int getSize() { ... }  
    public void add(int i) { ... }  
    public boolean delete(int i) { ... }  
    public boolean isExist(int i) { ... }  
    public void print() { ... }  
    ...  
}
```

哪些方法需要重写?

原始设计有什么不好?  
一会回来讲

```
public class NewLinkedList extends LinkedList{  
    public void add(NewElement e) { ... }  
}
```

**add是涉及到new 一个元素的方法，它需要重写。  
否则就new出来Element对象，不是NewElement对象。**

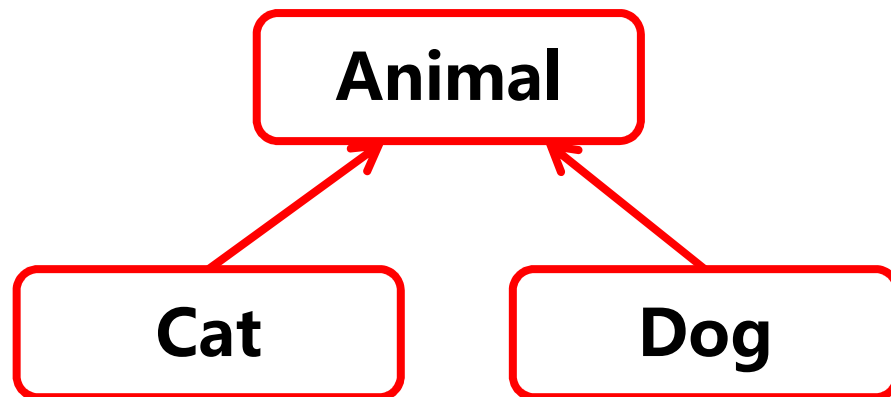


# 多态Polymorphism

## □ 引用的类型转换

- 子类可以自动视为父类
- 父类变成子类需要显式的类型转换 ( ? )
- 除了继承关系，否则不允许类型转换

```
NewElement a = new NewElement();  
Element b = a;  
(NewElement)b.setRecord(...);
```



# 多态Polymorphism

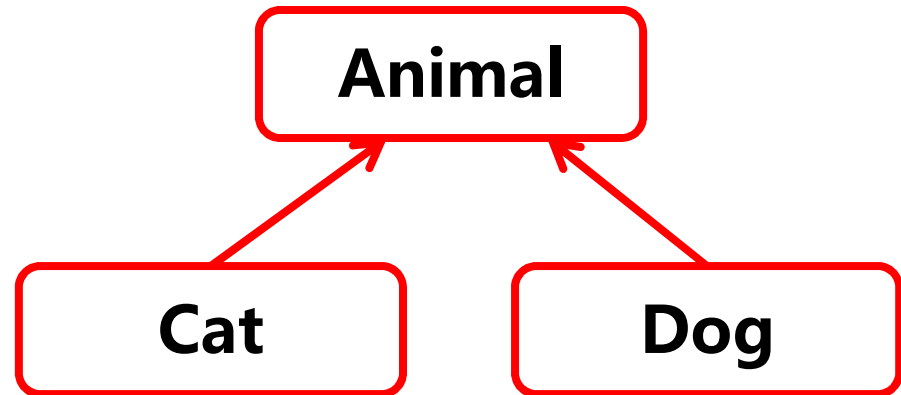
```
class A {  
    void print() {  
        System.out.println("A");  
    }  
}  
class B extends A {  
    void print() {  
        System.out.println("B");  
    }  
}  
class C extends A {  
    void print() {  
        System.out.println("C");  
    }  
}  
  
public static void main(String[] args) {  
    A b = new B();  
    ((C)b).print(); // ((A)b).print();  
}
```

报错!

# 多态Polymorphism

## □ 更有趣的例子：真正的动态绑定

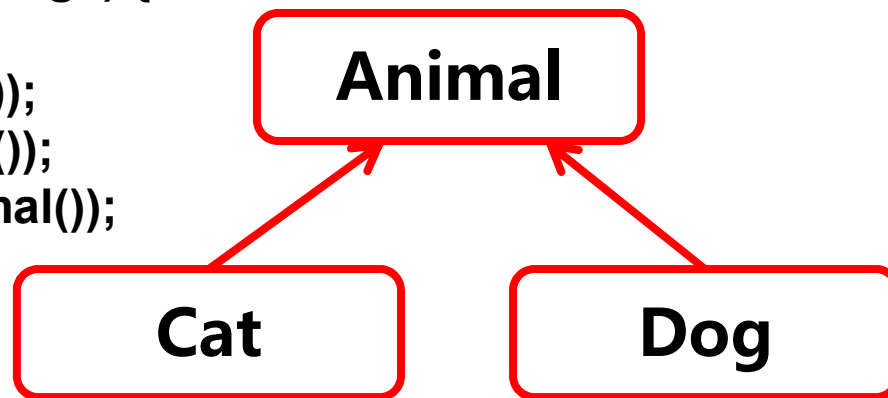
```
class Animal {  
    String talk() {  
        return "Error: I am undefined. I don't know how to talk";  
    }  
};  
class Cat extends Animal {  
    String talk() {  
        return "Meow!";  
    }  
};  
class Dog extends Animal {  
    String talk() {  
        return "Woof!";  
    }  
};
```



# 多态Polymorphism

## □ 更有趣的例子：真正的动态绑定

```
public class Poly {  
    void letsHear(Animal a) {  
        System.out.println(a.talk());  
    }  
    public static void main(String[] args) {  
        Poly poly = new Poly();  
        poly.letsHear(new Cat());  
        poly.letsHear(new Dog());  
        poly.letsHear(new Animal());  
    }  
}
```



Meow!

Woof!

Error: I am undefined. I don't know how to talk animal

# 多态Polymorphism

- ❑ 运行时灵活绑定子类，执行相应的子类方法
- ❑ 如果没有多态？
  - 无法通过父类的方法名调用实际的子类方法
  - 需要为每种子类创造相应的引用变量
    - 不简洁、扩展性受限

```
public class TestPoly {  
    void letsHear(Animal a) {  
        System.out.println(a.talk());  
    }  
    public static void main(String[] args) {  
        Poly poly = new Poly();  
        poly.letsHear(new Cat());  
        poly.letsHear(new Dog());  
    }  
}
```

# 里氏替换法则

*Let  $q(x)$  be a property provable about objects  $x$  of type  $T$ . Then  $q(y)$  should be true for objects  $y$  of type  $S$  where  $S$  is a subtype of  $T$ .*

- ❑ 派生类（子类）对象能够替换其基类（父类）对象被使用
- ❑ Liskov substitution principle
  - Barbara Liskov 1987
- ❑ Barbara Liskov
  - 2004年冯诺依曼奖
  - 2008年图灵奖
  - 导师John McCarthy
    - Raj Reddy



# 思考

- 思考、理解面向对象语言多态的重要意义
- 回忆、思考、理解继承、封装、多态三个面向对象语言的重要思想
- 比较Java与其他语言实现多态的异同点

复旦大学计算机科学技术学院



# 编程方法与技术

## 6.4. 多态 Polymorphism

周扬帆

2021-2022第一学期



# Internet协议的回顾

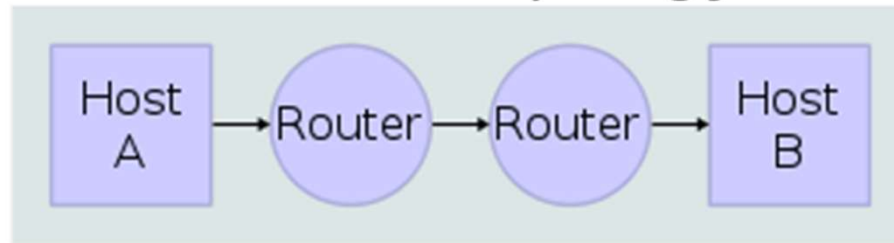
## □ 分层

- 链接层 (Link Layer)
- 网络层 (Internet Layer)
- 传输层 (Transport Layer)
- 应用层 (Application Layer)

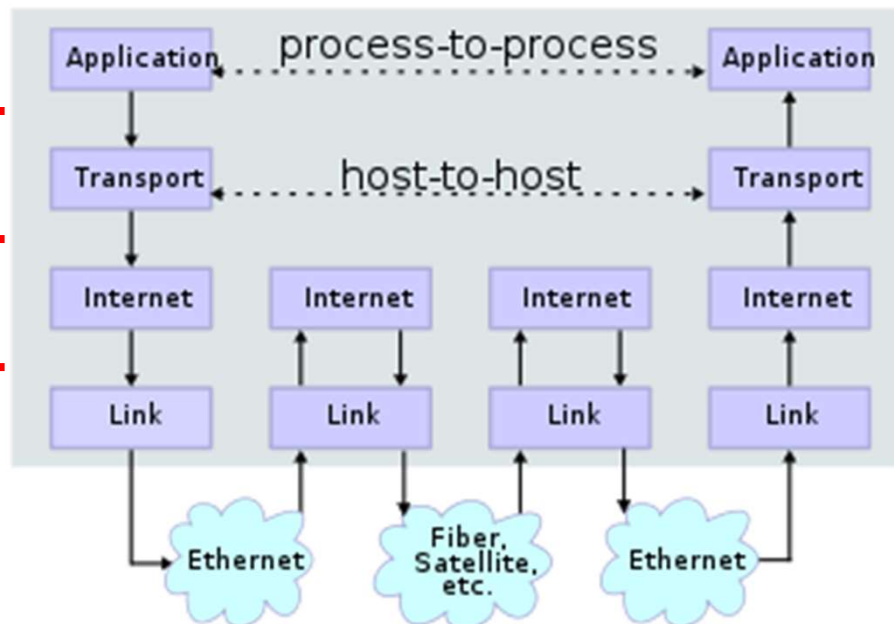
## □ Vinton Cerf & Bob Kahn (2004)

# 互联网架构

Network Topology



Data Flow



HTTP, FTP, ...

TCP, UDP

IP

有线/无线...

# 数据包Packet结构

## □ Packet数据结构



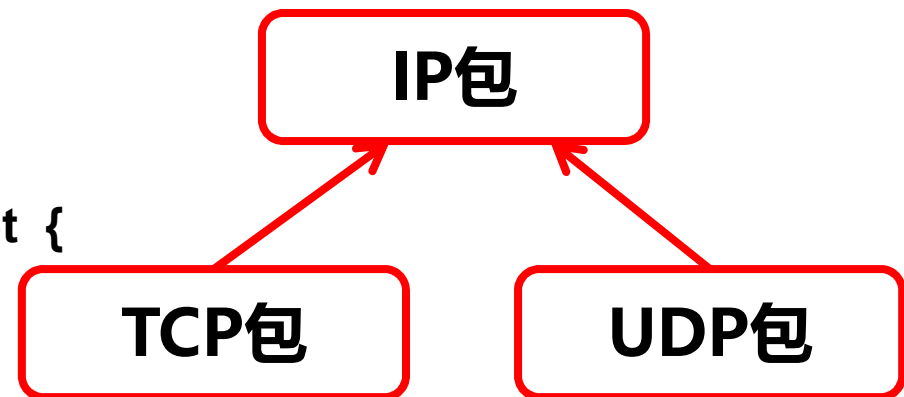
- TCP包是一个IP包
- UDP包是一个IP包

# Packet的实现示例

```
public class IPacket {  
    ... //数据结构定义, 包括IP源地址, 目的地址等  
    void getIPHeader { ... }  
    void parsePacket(); //解包  
}
```

```
public class TCPPacket extends IPacket {  
    int srcPort;  
    int dstPort;  
    ... //其他数据, 如序号等  
    void getTCPHeader { ... }  
    void parsePacket(); //解包  
}
```

```
public class UDPPacket extends IPacket {  
    int srcPort;  
    int dstPort;  
    ... //其他数据, 如长度, 校验和等  
    void getUDPHeader { ... }  
    void parsePacket(); //解包  
}
```



# Packet的实现示例

```
public class Node {  
    ... //数据结构定义, 包括本节点IP地址等  
    void recv(Packet p) {  
        ...  
        p.parsePacket();  
        ...  
    }  
    void send(Packet p) {...}  
    ...  
}
```

**可以使用多态**

```
Node node = ...;  
TCPPacket p1 = ...;  
node.recv(p1);  
UDPPacket p2 = ...;  
node.recv(p2);
```

**针对不同包类型, 进行解包**

**设想后续需要加入  
新包类型SCTP**

# 里氏替换法则

*Let  $q(x)$  be a property provable about objects  $x$  of type  $T$ . Then  $q(y)$  should be true for objects  $y$  of type  $S$  where  $S$  is a subtype of  $T$ .*

- ❑ 派生类（子类）对象能够替换其基类（父类）对象被使用
- ❑ Liskov substitution principle
  - Barbara Liskov 1987
- ❑ OCP (Open-Close Principle):  
Software entities should be open for extension, but closed for modification

# 单向链表的例子

```
public class LinkedList {  
    Element first = null;  
    Element last = null;  
    public boolean removeFirst() { ... }  
    public int getFirst() { ... }  
    public int getSize() { ... }  
    public void add(int i) { ... }  
    public boolean delete(int i) { ... }  
    public boolean isExist(int i) { ... }  
    public void print() { ... }  
    ...  
}
```

原始设计有什么不好?  
一会回来讲



```
public void add (Element e) { ... }
```

使用e (比如调用e.XXX) 来完成某些操作  
扩展Element数据结构之后, add也不用改

复旦大学计算机科学技术学院



# 编程方法与技术

## 6.5. 抽象类

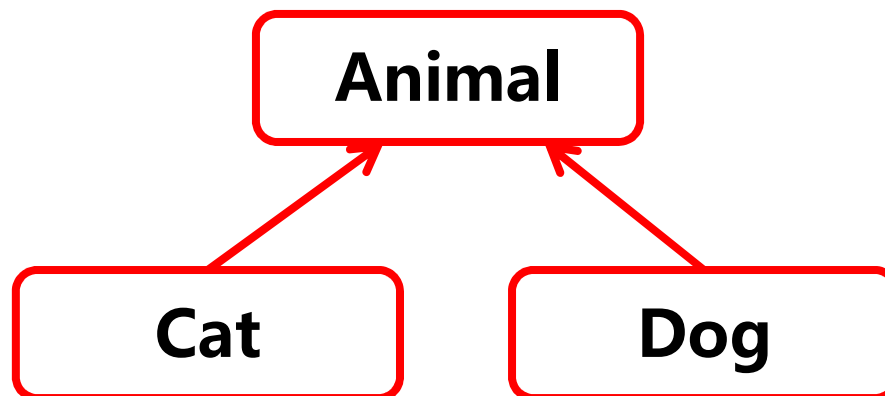
周扬帆

2021-2022第一学期



# Animal/Cat/Dog的例子

```
class Animal {  
    private double weight;  
    public double getWeight();  
    public String talk() {  
        return "Error: I am undefined. I don't know how to talk";  
    }  
    ...  
};  
class Cat extends Animal {  
    public String talk() {  
        return "Meow!";  
    }  
};  
class Dog extends Animal {  
    public String talk() {  
        return "Woof!";  
    }  
};
```



□ 抽出子类的共性特性，实现父类

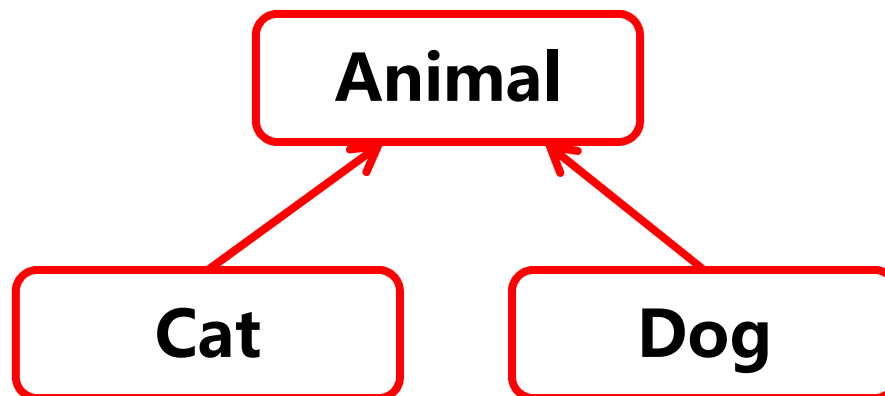
■ 代码复用、代码可读性

# Animal/Cat/Dog的例子

```
class Animal {  
    private double weight;  
    public double getWeight();  
    public String talk() {  
        return "Error: I am undefined. I don't know how to talk";  
    }  
    ...  
};  
class Cat extends Animal {  
    public String talk() {  
        return "Meow!";  
    }  
};  
class Dog extends Animal {  
    public String talk() {  
        return "Woof!";  
    }  
};
```

❑ 父类有时不能成为具体的对象

- 不完整



# 抽象父类

## ❑ 不能具体化的父类

### ■ 需避免被错误实例化

```
class Animal {  
    private double weight;  
    public double getWeight();  
    public String talk() {  
        return "Error: I am undefined. I don't know how to talk";  
    }  
    ...  
};  
...  
Animal a = new Animal();  
a.talk();
```

### ■ 强制其不能实例化: 加上abstract关键字

```
abstract class Animal {  
    ...  
}
```

# 抽象类的抽象方法

## ❑ 不能具体化的抽象类

```
abstract class Animal {  
    private double weight;  
    public double getWeight();  
    public String talk() {  
        return "Error: I am undefined. I don't know how to talk";  
    }  
    ...  
};
```

- 错误处理没必要了
- 但是，如何保证子类一定会实现这个方法？
  - 声明abstract方法，子类必须实现（除非...）

```
abstract class Animal {  
    abstract String talk();  
    ...  
};
```

# 思考

- ❑ Java抽象类可以有构造函数吗?
- ❑ Java抽象类可以是final的吗?
- ❑ Java抽象类可以有static方法吗?
- ❑ Java抽象类必须有抽象方法吗?
- ❑ Java抽象类中可以包含程序入口的main方法吗?
- ❑ 抽象类之间可以有继承关系吗?
- ❑ 思考、举例适合使用抽象类的应用场景

复旦大学计算机科学技术学院



# 编程方法与技术

## 6.6. 内部类

周扬帆

2021-2022第一学期

# 类的定义

## □ public的类

- 定义在独立文件中
- 类名和文件名相同

## □ default的类

- 可以定义在独立的文件中
- 也可以和别的类一起并列定义在文件中

## □ 内部类

# 内部类的定义

## □ 内部类

- 定义在其他类定义内部的类
- 定义方法
  - 与类成员变量、方法并列
  - 或代码块中
- 作用域？

```
class Log {  
    class LogWriter{  
        ...  
    }  
    private String fileName;  
    void WriteLog {  
        ...  
    }  
}
```

```
class Log {  
    void WriteLog {  
        ...  
        for( ... ) {  
            class LogWriter{  
                ...  
            }  
        }  
    }  
}
```



# 访问类的私有数据

```
public class Log {  
    private String fileName;  
    void WriteLog {  
        //to a database  
        //or to a file  
    }  
    public String GetFileName() {  
        String a = filename;  
    }  
    ...  
}
```

- 如果需要实现一个负责向文件系统写Log的类
- 要用到fileName, 怎么办?
- 可是, 又不想别的类用  
→ 内部类

# 内部类的目的

- ❑ 类需要访问另一个类的私有数据
  - `getValue` ?
  - 太麻烦 + 没必要公开
- ❑ 只在类内部使用的类，没必要和任何类分享
  - 包括只用一次的类
- ❑ 类和类有非常明确的从属关系
  - 提高可读性

```
public class Vehicle{  
    public class Wheel{  
        ...  
    }  
}
```

# 内部类的种类

- ❑ 普通内部类
- ❑ 静态内部类
- ❑ 局部内部类
- ❑ 匿名内部类

# 普通内部类

- 内部类可使用外部类的变量和方法
- 外部类可以创建内部类实例
- 作用域关键字用法一样
- 需要外部类实例创建内部类对象

```
Log log = new Log();  
Log.FileLogger a = log.new FileLogger();
```

```
public class Log {  
    private String fileName;  
    public class FileLogger {  
        String a = filename;  
        ...  
    }  
    void WriteLog {  
        FileLogger fl = new FileLogger();  
        ...  
    }  
}
```

# 普通内部类

- 内部类可使用外部类的变量和方法
- 外部类可以创建内部类实例
- 作用域关键字用法一样
- 需要外部类实例创建内部类对象 (?)
- 不能定义静态变量 ( ? )
- 可以定义常量

```
public class Log {  
    private String fileName;  
    public class FileLogger {  
        String a = filename;  
        final static int i = 0;  
  
        ...  
    }  
}
```

# 静态内部类

- ❑ 可用static修饰内部类 → 静态内部类
- ❑ 只能访问外部类的static变量和方法
  - 不能直接访问外部类的非static变量和方法
- ❑ 可直接创建，不需要外部类引用
- ❑ 用处
  - 层级非常明显的两个类：提升代码可读性

```
public class Log {  
    private String fileName;  
    public static class FileLogger {  
        ...  
    }  
}  
  
Log.FileLogger a = new Log.FileLogger();
```

# 静态内部类

- ❑ 可用static修饰内部类 → 静态内部类
- ❑ 只能访问外部类的static变量和方法
  - 不能直接访问外部类的非static变量和方法
- ❑ 可直接创建，不需要外部类引用
- ❑ 用处
  - 层级非常明显的两个类：提升代码可读性
  - 方便写测试代码

```
public class Log {  
    private String fileName;  
    public static class Test1 {  
        public static void main(String args[]) {  
            Log testLog = new Log();  
            testLog.XXX(); //开始测试  
        }  
    }  
    public static class Test2 {  
        public static void main(String args[]) {  
        }  
    }  
}
```

# 局部内部类

- ❑ 定义在程序块中，只在块内有效
  - 块外不能用到：创建，引用
- ❑ 不加任何访问修饰符，不能加static
  - 但是可以用abstract和final修饰
- ❑ 可访问外部类成员
  - static函数中的呢？
- ❑ 可访问块中的**final**局部变量

1.8之后，可以不写final，但默认是final，类里不能改，类外也不能改

```
public class Log {  
    private String fileName;  
    public void writeLog(String a) {  
        final String b = a;  
        class Test {  
            public void show() {  
                System.out.println(fileName);  
                System.out.println(b);  
            }  
        }  
        Test c = new Test();  
        c.show();  
    }  
}
```



# 匿名内部类

## □ 没有引用名的对象

```
class Test {  
    public void show() {  
        System.out.println("Hello");  
    }  
}
```

```
new Test().show();
```



等价于: (new Test()).show()  
new Test()创建了一个Test对象  
.show()调用了这个对象的show方法

```
new Test() {
```

```
    public void show() {  
        super.show();  
        System.out.println("Hello2");  
    }  
}
```

```
}.show();
```

覆盖了父类的show()方法

## □ 匿名类

- 继承父类
- 或实现接口

# 思考

- 根据JVM的内存管理，分析理解普通内部类不能定义静态变量
- 根据JVM的内存管理，分析理解静态内部类不能直接访问外部类的非static变量和方法

复旦大学计算机科学技术学院



# 编程方法与技术

## 6.7. 课堂练习

周扬帆

2021-2022第一学期

# 构造函数的执行顺序

- 子类与父类、类与各种内部类
- 静态成员变量、普通成员变量
- 静态块

```
static{  
    System.out.println( "内容" );  
}  
...
```

设计你的project, 对各种情况的构造函数/静态块的执行顺序分析排序