

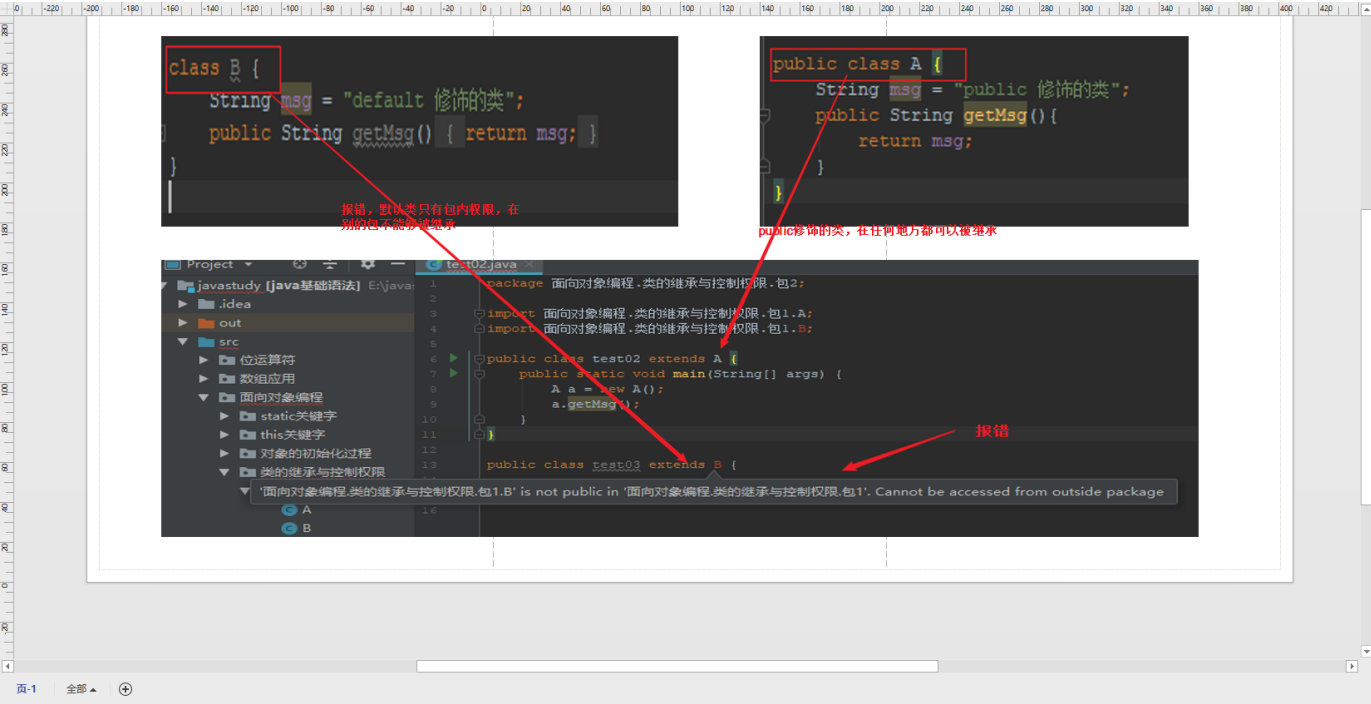
类的继承与权限控制

类的继承与权限控制

- java中使用extends关键字实现类的继承机制
- 通过继承，子类自动用于基类的所以成员（包括成员变量与方法）
- java只支持单继承，不允许多继承（一个子类只能有一个基类，一个基类可以派生多个子类）

访问控制权限

- java权限修饰符public、protected、private置于类的成员定义之前，用来限定其他对象对该对象成员的访问限制。
- 对与class的权限修饰只能用public和default
 - 其中public类可以在任意位置被访问
 - default类只可以被同一包内部的类访问



成员修饰符	类内部	同一个包	子类	任何地方
private	可以			
default	可以	可以		
protected	可以	可以	可以	

public	可以	可以	可以	可以
--------	----	----	----	----

访问控制——无访问控制符（包内友好）

package 面向对象编程.类的继承与控制权限.包1;

```
public class A extends B {
    public String pmsg = "public 修饰的成员";
    String dmsg = "default 修饰的成员";
    protected String prmsg = "protected 修饰的成员";
    private String primsg = "private 修饰的成员";

    public String getMsg(){
        return msg;
    }
}
```

package 面向对象编程.类的继承与控制权限.包1;

```
class B {
    String msg = "default 修饰的类";
    public String getMsg(){
        return msg;
    }
}
```

package 面向对象编程.类的继承与控制权限.包2;

import 面向对象编程.类的继承与控制权限.包1.A;

```
public class test02 extends A {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.pmsg);

        //    以下访问都有问题
        //    System.out.println(a.dmsg);
        //    System.out.println(a.prmsg);
        //    System.out.println(a.primsg);
    }
}
```

方法的重写

- 在子类中可以根据需要对从基类中继承的方法进行重写
- 重写方法必须和被重写的方法具有相同的方法名称、参数列表、返回类型。
 - 方法重载是（相同的方法名，参数列表不同即可）
- 重写方法不能使用比被重写方法更严格的访问权限

```
package 面向对象编程.方法的重写;

import 面向对象编程.对象的初始化过程.Test;

public class Person {
    private String name;
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void display(){
        System.out.println(name);
    }
}

class Teacher extends Person{
    private Integer id;
    public void setId(int id){
        this.id = id;
    }
    // public void display(){
    //     System.out.println(getName() + " " + id);
    // }

    public static void main(String[] args) {
        Person person = new Person();
        person.setName("汤家平");
        person.display();

        Teacher teacher = new Teacher();
        teacher.setName("张三");
        teacher.setId(11111);
        teacher.display();
    }
}
```

没重写前输出：

汤家平
张三

重写后输出：

汤家平

张三 11111

super关键字

package 面向对象编程.super关键字;

```
class Father{
    public int value;
    public void f(){
        value = 100;
        System.out.println("father.value" + value);
    }
}
class Child extends Father{
    public int value;
    public void f(){
        super.f();
        value = 200;
        System.out.println("child.value" + value);
        System.out.println(value);
        System.out.println(super.value);
    }
}
public class test01 {
    public static void main(String[] args) {
        Child child = new Child();
        child.f();
    }
}
```

输出：

father.value100

child.value200

200

100

继承中的构造方法（重点）

- 子类的构造过程中必须调用其基类的构造方法
- 子类可以在自己的构造方法中使用super（）调用基类的构造方法
 - 使用this（）调用本类的构造方法
 - 如果使用super（），必须写在子类构造方法的第一行
 - 在一个构造方法中，this和super最多只能调用其中一个，即不可以同时调用（因为不管是this还是super，都必须写在构造方法的第一行）

- 如果子类的构造方法中**没有显示的调用基类的构造方法**，系统默认调用基类的无参构造方法
- 如果子类构造方法即没有显示调用基类构造方法，而基类中又没有无参的构造方法，则编译出错
- 构造方法可以重载，但是不能继承
- **子类对象的初始化顺序：**
 - static变量初始化：基类——>子类
 - 基类构造器被调用：初始化非静态变量——>其他语句执行
 - 子类构造器被调用：初始化非静态变量——>其他语句执行

分析图：

```

17 class Child extends Super{
18     private int n;
19
20     Child(){
21         super(n: 10);
22         System.out.println("调用子类的无参构造方法");
23     }
24     Child(int n){
25         this.n = n;
26         System.out.println("调用子类的有参构造方法");
27         super();
28     }
29
30 public class test01 {
31
32 }
33
  
```

Call to 'super()' must be first statement in constructor body

子类调用父类的构造方法 由于没放在第一行，所以报错

```

class Child extends Super{
    private int n;

    Child(){
        super(n: 10);
        System.out.println("调用子类的无参构造方法");
    }
    Child(int n){
        super();
        this();
        this.n = n;
        System.out.println("调用子类的有参构造方法");
    }
}

public class test01 {
}
  
```

同时出现this和super，报错，这与this和super只能在构造方法的第一行一个意思

```
class Super{
    private int n;

    // Super(){
    //     System.out.println("调用基类的无参构造方法");
    // }
    Super(int n){
        this.n = n;
        System.out.println("调用基类的有参构造方法");
    }
}

class Child extends Super{
    private int n;

    Child(){
        super(n:10);
        System.out.println("调用子类的无参构造方法");
    }
    Child(int n){
        // super();
        this.n = n;
        System.out.println("调用子类的有参构造方法");
    }
}

public class test01 {
}
```

该父类没有无参构造方法

该子类的构造方法指明了调用基类的有参构造方法，故该构造方法正确

该子类的构造方法没有指明调用父类的构造方法，故系统会默认调用基类的无参构造方法，而该基类又没有无参构造方法，故报错。

```
class Super{
    private int n;

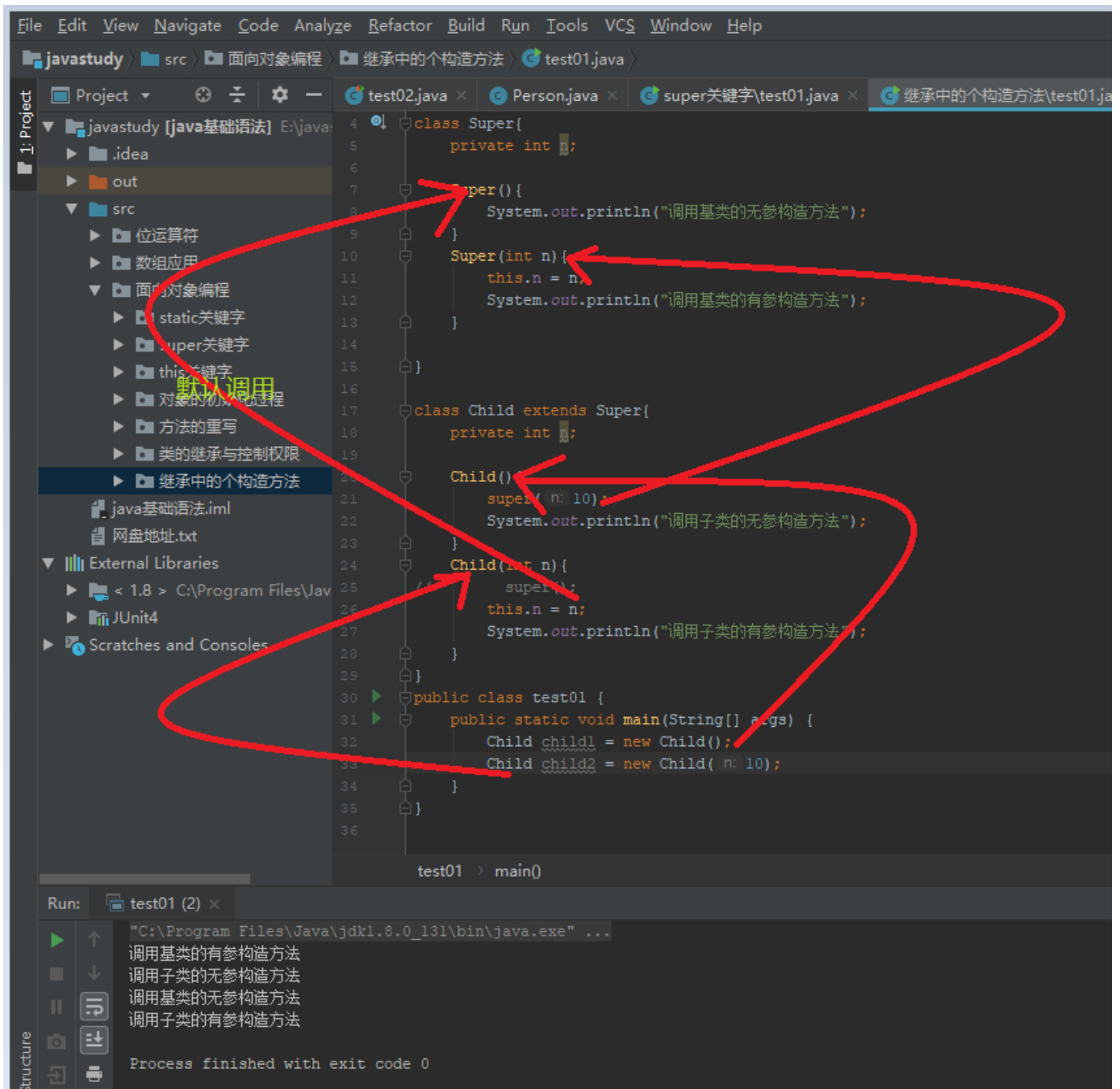
    Super(){
        System.out.println("调用基类的无参构造方法");
    }
    Super(int n){
        this.n = n;
        System.out.println("调用基类的有参构造方法");
    }
}

class Child extends Super{
    private int n;

    Child(){
        super(n:10);
        System.out.println("调用子类的无参构造方法");
    }
    Child(int n){
        // super();
        this.n = n;
        System.out.println("调用子类的有参构造方法");
    }
}

public class test01 {
}
```

子类的构造方法都正确



package 面向对象编程.继承中的个构造方法;

```
class Super{
    private int n;

    Super(){
        System.out.println("调用基类的无参构造方法");
    }
    Super(int n){
        this.n = n;
        System.out.println("调用基类的有参构造方法");
    }
}
```

```

class Child extends Super{
    private int n;

    Child(){
        super(10);
        System.out.println("调用子类的无参构造方法");
    }
    Child(int n){
        // super();
        this.n = n;
        System.out.println("调用子类的有参构造方法");
    }
}

public class test01 {
    public static void main(String[] args) {
        Child child1 = new Child();
        Child child2 = new Child(10);
    }
}

```

结果：

调用基类的有参构造方法
调用子类的无参构造方法
调用基类的无参构造方法
调用子类的有参构造方法

内存分配

父类

子类

两者除了名字一样
其他无关

```

class Child extends Super{
    private int n;

    Child(){
        super(10);
        System.out.println("调用子类的无参构造方法");
    }
    Child(int n){
        // super();
        this.n = n;
        System.out.println("调用子类的有参构造方法");
    }
}

public class test01 {
    public static void main(String[] args) {
        Child child1 = new Child();
        Child child2 = new Child(10);
    }
}

```

test01 (2) ×

"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...

调用基类的有参构造方法

调用子类的无参构造方法 10 0

调用基类的无参构造方法

调用子类的有参构造方法

Process finished with exit code 0

继承中的构造方法的初始化顺序


```

package 面向对象编程.继承中的个构造方法;
class Insect{
    private int i = 9;
    protected int j;

    Insect(){
        System.out.println("i = " + i + "j = " + j);
        j = 39;
    }
    private static int x1 = println("static Insect.x1 initialized");

    static int println(String s){
        System.out.println(s);
        return 47;
    }
}
class Beetle extends Insect{
    private int k = println("Beetle.k initialized");
    public Beetle(){
        System.out.println("k = " + k);
        System.out.println("j = " + j);
    }
    private static int x2 = println("static Beetle.x2 initialized");

    public static void main(String[] args) {
        System.out.println("开始执行main函数");
        Beetle b = new Beetle();
    }
}
public class test02 {
    public static void main(String[] args) {
        System.out.println("开始执行main函数");
        Beetle b = new Beetle();
    }
}

```

第一个main结果:

```

static Insect.x1 initialized
static Beetle.x2 initialized
开始执行main函数
i = 9j = 0
Beetle.k initialized
k = 47
j = 39

```

第二个main结果:

```

开始执行main函数
static Insect.x1 initialized
static Beetle.x2 initialized
i = 9j = 0

```

Beetle.k initialized

k = 47

j = 39

两个结果情况分析：

```
1 package 面向对象编程.继承中的个构造方法;
2 class Insect{
3     private int i = 9;
4     protected int j;
5
6     Insect(){
7         System.out.println("i = " + i + "j = " + j);
8         j = 39;
9     }
10    private static int x1 = printInit("static Insect.x1 initialized");
11
12    static int printInit(String s){
13        System.out.println(s);
14        return 47;
15    }
16 }
17 class Beetle extends Insect{
18     private int k = printInit("Beetle.k initialized");
19     public Beetle(){
20         System.out.println("k = " + k);
21         System.out.println("j = " + j);
22     }
23     private static int x2 = printInit("static Beetle.x2 initialized");
24
25     public static void main(String[] args) {
26         System.out.println("开始执行main函数");
27         Beetle b = new Beetle();
28     }
29 }
30 public class test02 {
31     public static void main(String[] args) {
32         System.out.println("开始执行main函数");
33         Beetle b = new Beetle();
34     }
35 }
36
```

```
static Insect.x1 initialized
static Beetle.x2 initialized
开始执行main函数
i = 9j = 0
Beetle.k initialized
k = 47
j = 39
```

```
开始执行main函数
static Insect.x1 initialized
static Beetle.x2 initialized
i = 9j = 0
Beetle.k initialized
k = 47
j = 39
```

分析两个main函数的执行结果为什么有区别???

提示: static成员及static代码块只是先于本类对象构造前执行。

对象转型 (casting)

package 面向对象编程.对象转型;

import 面向对象编程.类的继承与控制权限.包1.A;

```
class Animal{
    public String name;
    Animal(String name){
        this.name = name;
    }
    public void run(){
        System.out.println("run");
    }
}
```

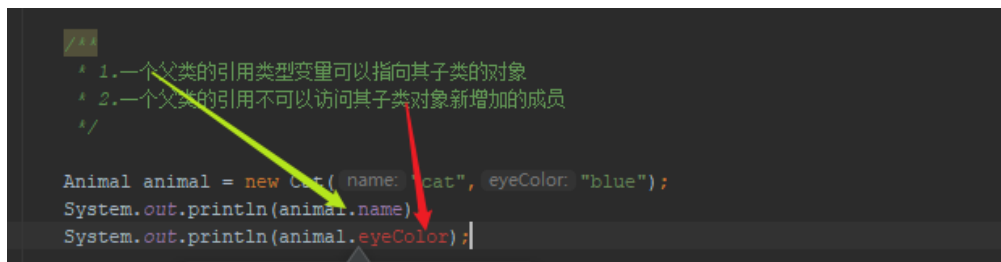
```
class Cat extends Animal{
    public String eyeColor;
    Cat(String name,String eyeColor){
```

```

    super(name);
    this.eyeColor = eyeColor;
}
public void catchMouse(){
    System.out.println("cat catch mouse");
}
}
class Dog extends Animal{
    public String furColor;
    Dog(String name,String furColor){
        super(name);
        this.furColor = furColor;
    }
    public void swim(){
        System.out.println("dog can swim");
    }
}

```

- 一个父类的引用类型变量可以指向其子类的对象
- 一个父类的引用不可以访问其子类对象新增加的成员（属性和方法）
 - 目标引用类型变量允许直接访问的属性和方法有目标引用类型变量的申明类型决定



```

/**
 * 1.一个父类的引用类型变量可以指向其子类的对象
 * 2.一个父类的引用不可以访问其子类对象新增加的成员
 */
Animal animal = new Cat( name: "cat", eyeColor: "blue");
System.out.println(animal.name);
System.out.println(animal.eyeColor);

```

```

public class test01 {
    public static void main(String[] args) {
        /**
         * 1.一个父类的引用类型变量可以指向其子类的对象
         * 2.一个父类的引用不可以访问其子类对象新增加的成员
         */

        Animal animal = new Cat("cat","blue");
        System.out.println(animal.name);
        /**
         * 试图访问子类对象新增加的属性，导致错误
         */
        // System.out.println(animal.eyeColor);
    }
}

```

输出:
cat

- 可以使用引用变量instanceof类名来判断该引用类型所指向的对象是否属于该类或该类的子类

格式:

```
a instanceof A
```

功能: 用来判断 **a指向的类型** 是否为A或A的子类, 并非a本身的类型

```
public class test01 {
    public static void main(String[] args) {

        /**
         * 可以使用 引用变量 + instanceof + 类名 来判断该引用变量所指向的对象是否属于该类或该类的子类
         * 最重要的是该引用变量所指向的对象类型, 并非引用变量自身的类型。
         */

        Animal animal = new Animal("name");
        System.out.println(animal instanceof Animal);
        System.out.println(animal instanceof Cat);
        System.out.println(animal instanceof Dog);

        Dog dog = new Dog("dog","pink");
        System.out.println(dog instanceof Animal);
        System.out.println(dog instanceof Dog);

        Animal a1 = new Cat("cat","blue");
        System.out.println(a1 instanceof Animal);
        System.out.println(a1 instanceof Cat);

    }
}
```

结果:

```
true
false
false
true
true
true
true
```

- 子类的对象可以当做基类的对象来使用, 这称为向上转型, 反之称为向下转型
 - 向上转型: 把子类对象直接赋值给父类引用, **不需要强制类型转换**
 - 向下转型: 把指向子类对象的父类引用赋值给子类对象, **需要强制类型转换**

```
public class test01 {
    public static void main(String[] args) {
```

```

/**
 * 把子类对象直接赋值给父类引用叫做向上转型，向上转型不用强制转换
 * 把指向子类对象的父类引用赋值给子类引用叫做向下转型，要强制转换
 */

// 向上转型
Animal animal = new Dog("dog","black");
// 向下转型
Dog dog = (Dog)animal;
}
}

```

- 安全的向下转型：引用类型变量当前指向的内存空间隐含目标引用类型变量的内存空间，反之则会发送异常。

```

Animal animal = new Animal( name: "animal");
/**
 * Cat cat1 = (Cat) animal;
 * 注意该代码：
 * 在运行时会抛出 java.lang.ClassCastException
 * 原因：
 * 编译器在编译时，只会检查类型之间是否存在继承关系，有则通过；无则报错。
 * 由于Cat 和 Animal 具有继承关系，所以在编译的时候并不会报错，
 * 但是在运行时，编译器会检查它的真实类型，
 * 在此处，animal的真实类型为Animal，与Cat类型不一致，所以运行时报错
 *
 * 总结：父类强制转化为子类时，只有当引用类型的真正身份为子类时才会强制转换成功
 */
Cat cat1 = (Cat) animal;
System.out.println(cat1.name);

Dog dog = new Dog( name: "dog", furColor: "pink");
// 向上转型
animal = dog;
// 向下转型
Dog dog1 = (Dog)animal;
System.out.println(dog1.furColor);

```

编译正常，运行报错

正常编译、运行

注意animal指向的对象

```

public class test01 {
    public static void main(String[] args) {
        Animal animal = new Animal("animal");
        /**
         * Cat cat1 = (Cat) animal;
         * 注意该代码：
         * 在运行时会抛出 java.lang.ClassCastException
         * 原因：
         * 编译器在编译时，只会检查类型之间是否存在继承关系，有则通过；无则报错。
         * 由于Cat 和 Animal 具有继承关系，所以在编译的时候并不会报错，
         * 但是在运行时，编译器会检查它的真实类型，
         * 在此处，animal的真实类型为Animal，与Cat类型不一致，所以运行时报错
         *
         * 总结：父类强制转化为子类时，只有当引用类型的真正身份为子类时才会强制转换成功
         */
        Cat cat1 = (Cat) animal;
        System.out.println(cat1.name);
    }
}

```

```
        Dog dog = new Dog("dog","pink");
//    向上转型
    animal = dog;
    Dog dog1 = (Dog)animal;
    System.out.println(dog1.furColor);
    dog1.swim();

    System.out.println(animal.name);
//    System.out.println(animal.furColor);

//    ((Dog) animal).swim();

    System.out.println(animal instanceof Animal);
    System.out.println(animal instanceof Dog);

//    向下转型

    }
}
```