

# 面向对象三大特性

Java Platform Standard Edition

Java教学部

# 课程目标

## CONTENTS

ITEMS **1** 封装

ITEMS **2** 访问修饰符

ITEMS **3** 继承

ITEMS **4** 方法重写

ITEMS **5** 多态

ITEMS **6** 装箱、拆箱

# 封装

Java Platform Standard Edition

# 封装的必要性

```
public class TestEncapsulation {  
    public static void main(String[] args) {  
        Student s1 = new Student();  
        s1.name = "tom";  
        s1.age = 20000;  
        s1.sex = "male";  
        s1.score = 100D;  
    }  
}  
  
class Student{  
    String name;  
    int age;  
    String sex;  
    double score;  
}
```

在对象的外部，为对象的属性赋值，可能存在非法数据的录入。

就目前的技术而言，并没有办法对属性的赋值加以控制。

# 什么是封装

- 概念：尽可能隐藏对象的内部实现细节，控制对象的修改及访问的权限。
- 访问修饰符：`private`（可将属性修饰为私有，仅本类可见）

```
public class TestEncapsulation {  
    public static void main(String[] args) {  
        Student s1 = new Student();  
        s1.age = 20000;  
    }  
}  
  
class Student {  
    String name;  
    private int age;  
    String sex;  
    double score;  
}
```

The field Student.age is not visible

编译错误：私有属性在类的外部不可访问

如何在提供正常的对外访问渠道的同时，又能控制录入的数据为有效？

# 公共访问方法

```
public class TestEncapsulation {  
    public static void main(String[] args) {  
        Student s1 = new Student();  
        s1.setAge(20000);  
        System.out.println(s1.getAge());  
    }  
}  
  
class Student{  
    String name;  
    private int age;  
    String sex;  
    double score;  
  
    public void setAge(int age){  
        this.age = age;  
    }  
  
    public int getAge(){  
        return this.age;  
    }  
}
```

以访问方法的形式，进而完成赋值与取值操作。  
问题：依旧没有解决到非法数据录入！

提供公共访问方法，以保证数据的正常录入。

命名规范：

赋值：setXXX() //使用方法参数实现赋值

取值：getXXX() //使用方法返回值实现取值

# 过滤有效数据

```
class Student{
    String name;
    private int age;
    String sex;
    double score;

    public void setAge(int age){
        if(age > 0 && age <=160){//指定有效范围
            this.age = age;
        }else{
            this.age = 18;//录入非法数据时的默认值
        }
    }

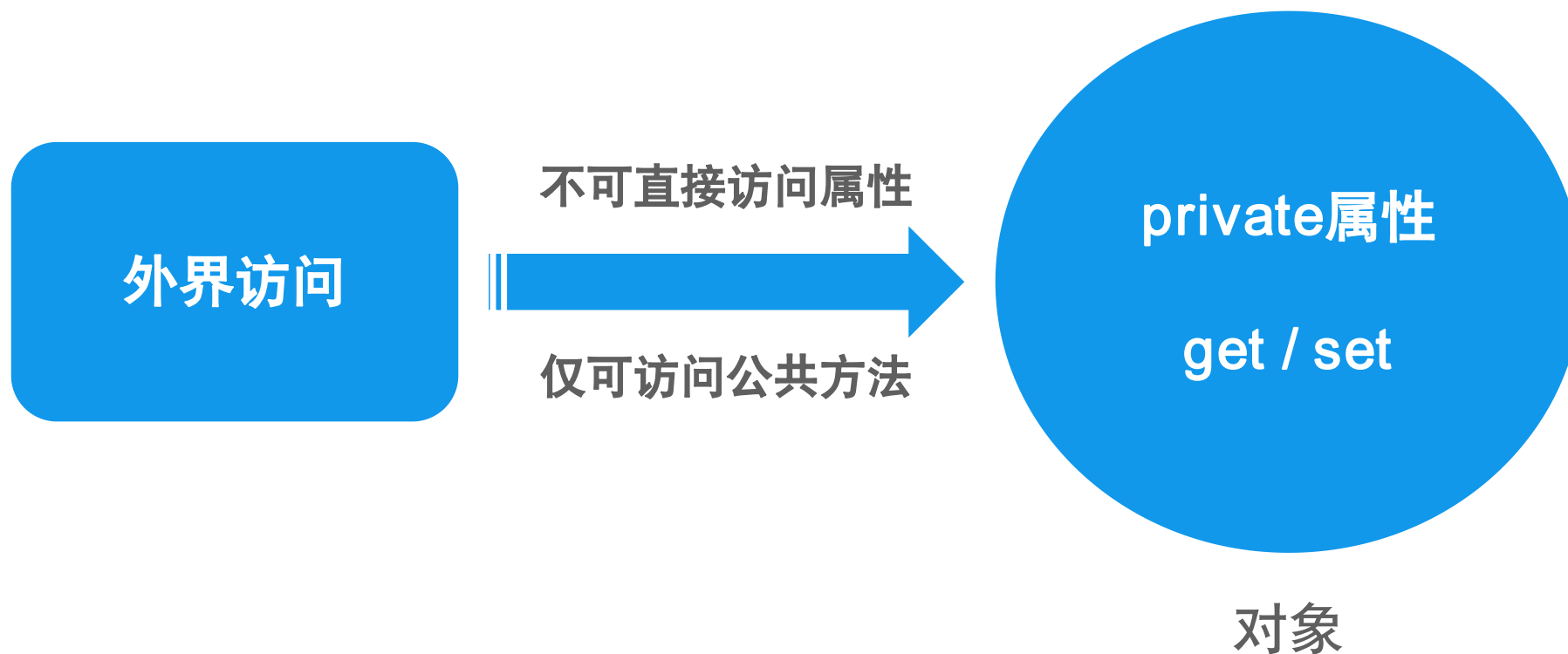
    public int getAge(){
        return this.age;
    }
}
```

```
public class TestEncapsulation {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setAge(20000);
        System.out.println(s1.getAge());
    }
}
```

在公共的访问方法内部，添加逻辑判断，进而过滤掉非法数据，以保证数据安全。

运行结果： 18





get/set方法是外界访问对象私有属性的唯一通道，方法内部可对数据进行检测和过滤。



# 继 承

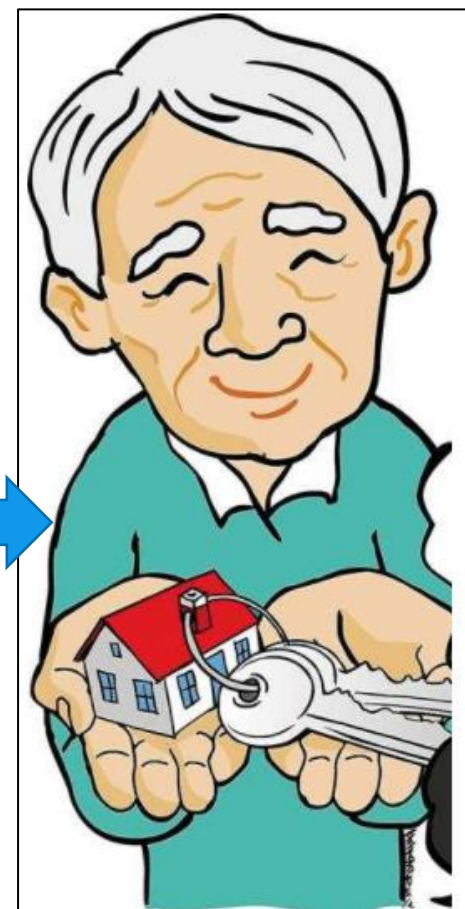
Java Platform Standard Edition

# 生活中的继承

- 生活中的“继承”是施方的一种赠与，受方的一种获得。
- 将一方所拥有的东西给予另一方。



往往二者之间  
具有继承关系  
(直系、亲属)



# 程序中的继承

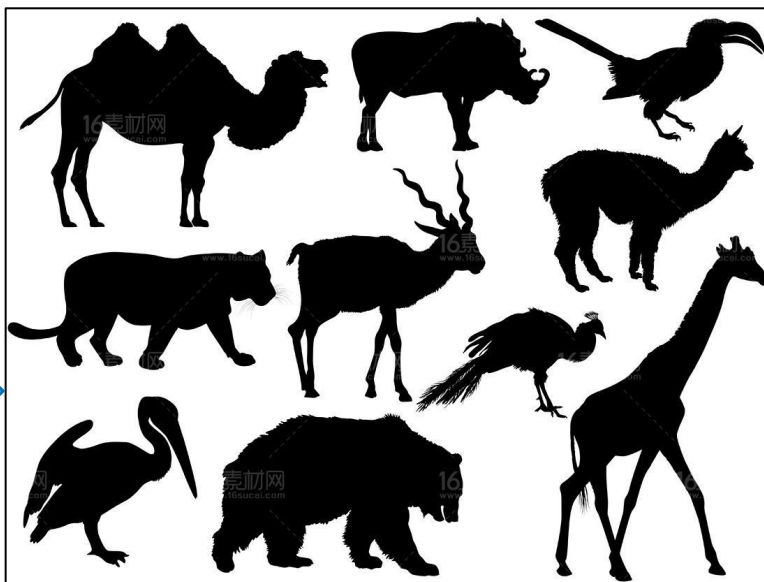
- 程序中的继承，是类与类之间特征和行为的一种赠与或获得。
- 两个类之间的继承关系，必须满足“is a”的关系。

Dog is an Animal 成立



狗 (Dog)  
子类

继承



动物 (Animal)  
父类

Cat is an Animal 成立



猫 (Cat)  
子类

继承

- 现实生活中，很多类别之间都存在着继承关系，都满足“is a”的关系。
- 狗是一种动物、狗是一种生物、狗是一种物质。
- 多个类别都可作为“狗”的父类，需要从中选择出最适合的父类。

## 动物

属性：  
品种、年龄、性别  
方法：  
呼吸、吃、睡

## 生物

属性：  
品种、年龄、性别  
方法：  
呼吸

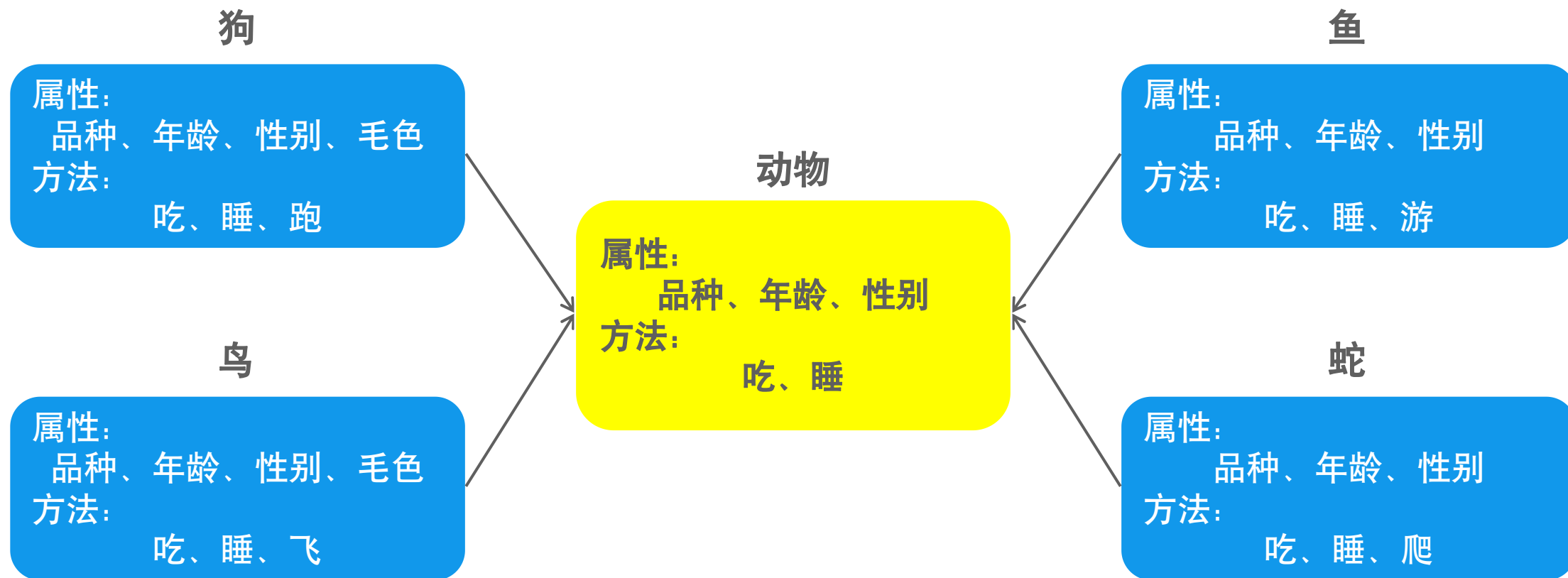
## 物质

属性：  
...  
方法：  
...

- 功能越精细，重合点越多，越接近直接父类。
- 功能越粗略，重合点越少，越接近Object类。（万物皆对象的概念）

# 父类的抽象

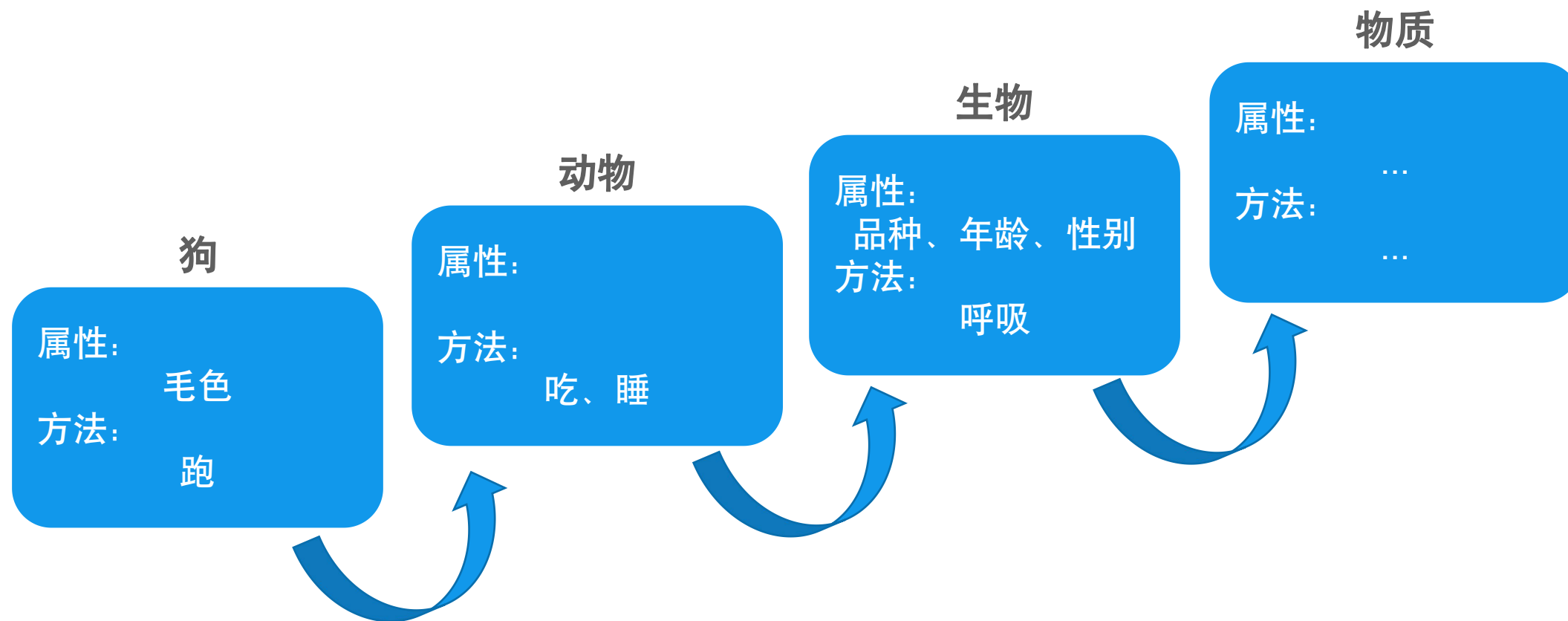
- 实战：可根据程序需要使用到的多个具体类，进行共性抽取，进而定义父类。



在一组相同或类似的类中，抽取出共性的特征和行为，定义在父类中，实现重用。

- 语法：class 子类 extends 父类{ } //定义子类时，显示继承父类
- 应用：产生继承关系之后，子类可以使用父类中的属性和方法，也可定义子类独有的属性和方法。
- 好处：既提高代码的复用性，又提高代码的可扩展性。

# 继承的特点



Java为单继承，一个类只能有一个直接父类，但可以多级继承，属性和方法逐级叠加。



- 构造方法：
  - 类中的构造方法，只负责创建本类对象，不可继承。
- `private`修饰的属性和方法：
  - 访问修饰符的一种，仅本类可见。（详见下图）
- 父子类不在同一个package中时，`default`修饰的属性和方法：
  - 访问修饰符的一种，仅同包可见。（详见下图）

# 访问修饰符

	本类	同包	非同包子类	其他
private	✓	×	×	×
default	✓	✓	×	×
protected	✓	✓	✓	×
public	✓	✓	✓	✓

严

宽

- 思考：子类中是否可以定义和父类相同的方法？
- 思考：为什么需要在子类中定义和父类相同的方法？
- 分析：当父类提供的方法无法满足子类需求时，可在子类中定义和父类相同的方法进行覆盖（Override）。

- 方法覆盖原则：
  - 方法名称、参数列表、返回值类型必须与父类相同。
  - 访问修饰符可与父类相同或是比父类更宽泛。
- 方法覆盖的执行：
  - 子类覆盖父类方法后，调用时优先执行子类覆盖后的方法。

# super关键字

- 在子类中，可直接访问从父类继承到的属性和方法，但如果父子类的属性或方法存在重名（属性遮蔽、方法覆盖）时，需要加以区分，才可专项访问。

```
public class TestSuperKeyword {  
    public static void main(String[] args) {  
        B b = new B();  
        b.upload();  
    }  
}  
  
class A{  
    public void upload(){  
        //上传文件的100行逻辑代码  
    }  
}  
  
class B extends A{  
    public void upload(){  
        //上传文件的100行逻辑代码  
        //修改文件名称的1行代码  
    }  
}
```

父类具有上传文件的功能

子类既要上传文件，又要更改文件名称，  
进而覆盖了父类的方法。  
但上传文件的逻辑代码相同，如何复用？

# super访问方法

```
public class TestSuperKeyword {  
    public static void main(String[] args) {  
        B b = new B();  
        b.upload();  
    }  
}  
  
class A{  
    public void upload(){  
        //上传文件的100行逻辑代码  
    }  
}  
  
class B extends A{  
    public void upload(){  
        super.upload(); //上传文件的100行逻辑代码  
        //修改文件名称的1行代码  
    }  
}
```

super关键字可在**子类**中访问父类的方法。

使用“super.”的形式访问父类的方法，  
进而完成在子类中的复用；  
再叠加额外的功能代码，组成新的功能。

# super访问属性

```
public class TestSuperKeyword {  
    public static void main(String[] args) {  
        B b = new B();  
        b.print();  
    }  
}  
  
class A{  
    int value = 10;  
}  
  
class B extends A{  
    int value = 20;  
  
    public void print(){  
        int value = 30;  
        System.out.println(value);  
        System.out.println(this.value);  
        System.out.println(super.value);  
    }  
}
```

父子类的同名属性不存在覆盖关系，  
两块空间同时存在（子类遮蔽父类属性），  
需使用不同前缀进行访问。

运行结果：

30  
20  
10



# 继承中的对象创建

- 在具有继承关系的对象创建中，构建子类对象会先构建父类对象。
- 由父类的共性内容，叠加子类的独有内容，组合成完整的子类对象。

```
class Father{  
    int a;  
    int b;  
    public void m1(){  
    }  
  
class Son extends Father{  
    int c;  
    public void m2(){  
    }  
}
```

子类Son所持有的属性和方法:

```
int a  
int b  
int c  
m1()  
m2()
```

# 继承后的对象构建过程

```
public class TestSuperKeyword {  
    public static void main(String[] args) {  
        new C();  
    }  
}  
  
class A{}  
class B extends A{}  
class C extends B{}
```

构建子类对象时，先构建父类对象。

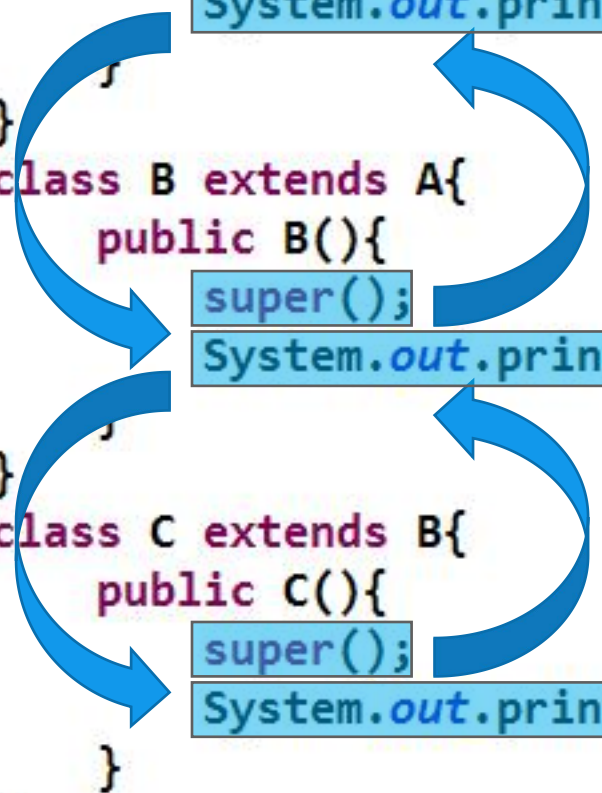
1.分配空间 (A、B、C)  
2.构建父类对象 (B)  
3.初始化属性 (C)  
4.执行构造方法代码 (C)

1.构建父类对象 (A)  
2.初始化属性 (B)  
3.执行构造方法代码 (B)

1.构建父类对象 (Object)  
2.初始化属性 (A)  
3.执行构造方法代码 (A)

# super调用父类无参构造

```
class A{
    public A(){
        System.out.println("A()");
    }
}
class B extends A{
    public B(){
        super();
        System.out.println("B()");
    }
}
class C extends B{
    public C(){
        super();
        System.out.println("C()");
    }
}
```



```
public class TestSuperKeyword {
    public static void main(String[] args) {
        new C();
    }
}
```

super(): 表示调用父类无参构造方法。  
如果没有显示书写,  
隐式存在于子类构造方法的首行。

运行结果:

A()  
B()  
C()

# super调用父类有参构造

```
class A{
    public A(){
        System.out.println("A()");
    }

    public A(int value){
        System.out.println("A(int value)");
    }
}
class B extends A{
    public B(){
        super();
        System.out.println("B()");
    }

    public B(int value){
        super(value);
        System.out.println("B(int value)");
    }
}
```

```
public class TestSuperKeyword {
    public static void main(String[] args) {
        new B();
        new B(10);
    }
}
```

super(): 表示调用父类无参构造方法。

super(实参): 表示调用父类有参构造方法。

# this与super

```
class A{
    public A(){
        System.out.println("A-无参构造");
    }

    public A(int value){
        System.out.println("A-有参构造");
    }
}
class B extends A{
    public B(){
        super();
        System.out.println("B-无参构造");
    }

    public B(int value){
        this(); //super();
        System.out.println("B-有参构造");
    }
}
```

```
public class TestSuperKeyword {
    public static void main(String[] args) {
        new B(10);
    }
}
```

运行结果：

A-无参构造  
B-无参构造  
B-有参构造

this或super使用在构造方法中时，都要求在首行。  
当子类构造中使用了this()或this(实参)，  
即不可再同时书写super()或super(实参)，  
会由this()指向的构造方法完成super()的调用。

- **super关键字的第一种用法：**
  - 在子类的方法中使用“super.”的形式访问父类的属性和方法。
  - 例如：super.父类属性、super.父类方法();
- **super关键字的第二种用法：**
  - 在子类的构造方法的首行，使用“super()”或“super(实参)”，调用父类构造方法。
- **注意：**
  - 如果子类构造方法中，没有显示定义super()或super(实参)，则默认提供super()。
  - 同一个子类构造方法中，super()、this()不可同时存在。



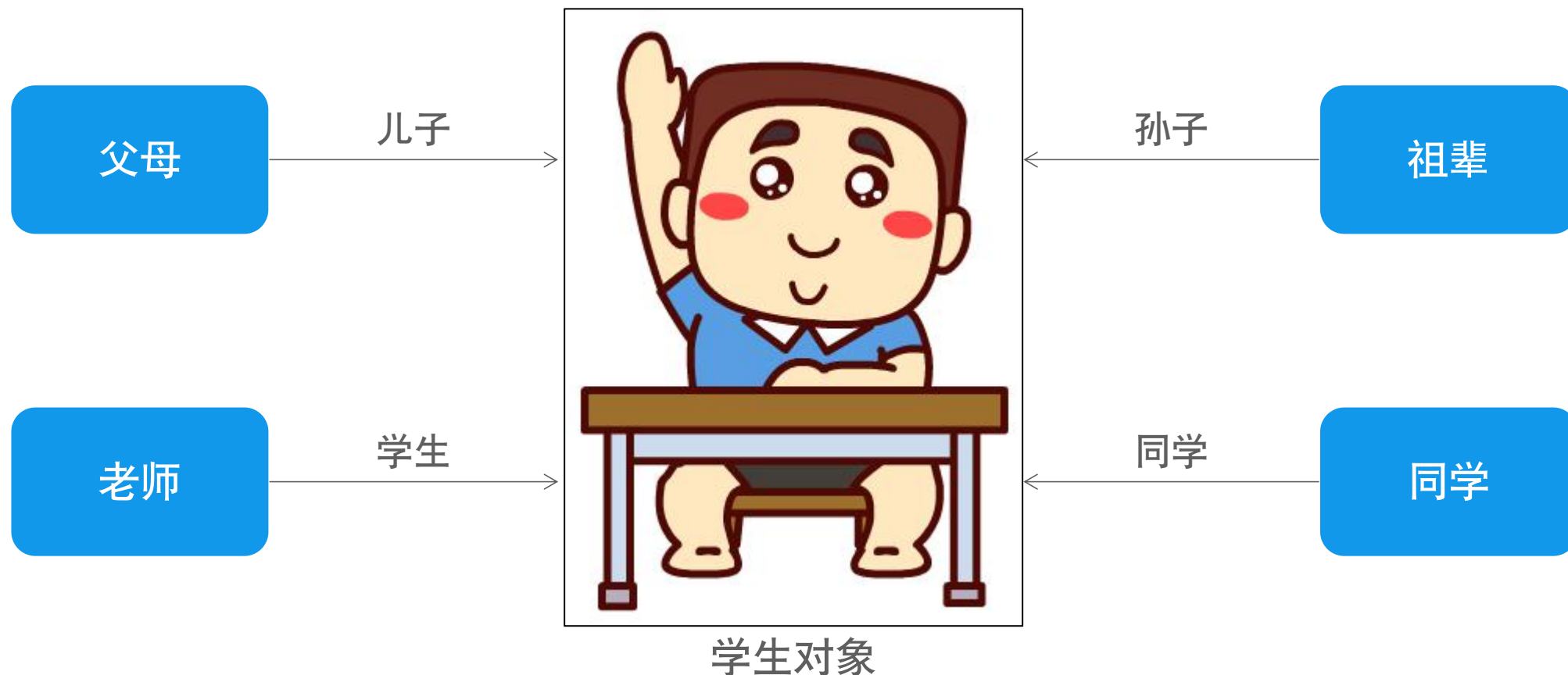
# 多 态

Java Platform Standard Edition



# 生活中的人物视角

- 生活中，不同人物角色看待同一个对象的视角不同，关注点也不相同。



主观反应



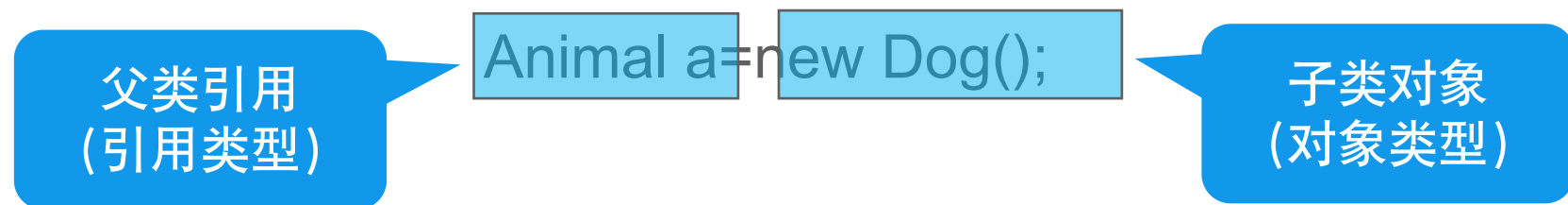
将子类对象当成父类类别看待

客观事物



- 生活中的多态是指“客观事物在人脑中的主观反应”。
- 主观意识上的类别与客观存在的对象具有“is a”关系时，即形成多态。

- 概念：父类引用指向子类对象，从而产生多种形态。



- 二者具有直接或间接的继承关系时，父类引用可指向子类对象，即形成多态。
- 父类引用仅可调用父类所声明的属性和方法，不可调用子类独有的属性和方法。

- 思考：如果子类中覆盖了父类中的方法，以父类型引用调用此方法时，优先执行父类还是子类中的方法？
- 实际运行过程中，依旧遵循覆盖原则，如果子类覆盖了父类中的方法，则执行子类中覆盖后的方法，否则执行父类中的方法。

# 多态的应用

```
class Master{  
    public void feed(Dog dog){  
        dog.eat();  
    }  
  
    public void feed(Cat cat){  
        cat.eat();  
    }  
  
    public void feed(Fish fish){  
        fish.eat();  
    }  
  
    public void feed(Bird bird){  
        bird.eat();  
    }  
  
    //.....  
}
```

方法重载可以解决接收不同对象参数的问题，但其缺点也比较明显。

首先，随着子类的增加，Master类需要继续提供大量的方法重载，多次修改并重新编译源文件。其次，每一个feed方法与具体某一种类型形成了密不可分的关系，耦合太高。

- 场景一：
  - 使用父类作为方法形参实现多态，使方法参数的类型更为宽泛。
- 场景二：
  - 使用父类作为方法返回值实现多态，使方法可以返回不同子类对象。

# 向上转型 ( 装箱 )

```
public class TestConvert {  
    public static void main(String[] args) {  
        Animal a = new Dog();  
    }  
}  
  
class Animal{  
    public void eat(){  
        System.out.println("动物在吃...");  
    }  
}  
  
class Dog extends Animal{  
    public void eat(){  
        System.out.println("狗在吃骨头...");  
    }  
}
```

父类引用中保存真实子类对象，  
称为向上转型（即多态核心概念）。

注意：  
仅可调用Animal中所声明的属性和方法。



# 向下转型 ( 拆箱 )

```
public class TestConvert {  
    public static void main(String[] args) {  
        Animal a = new Dog();  
        Dog dog = (Dog)a;  
    }  
}  
  
class Animal{  
    public void eat(){  
        System.out.println("动物在吃...");  
    }  
}  
  
class Dog extends Animal{  
    public void eat(){  
        System.out.println("狗在吃骨头...");  
    }  
}
```

将父类引用中的真实子类对象，  
强转回子类本身类型，称为向下转型。

注意：  
只有转换回子类真实类型，才可调用子类  
独有的属性和方法。

# 类型转换异常

```
class Animal{
    public void eat(){
        System.out.println("动物在吃...");
    }
}

class Dog extends Animal{
    public void eat(){
        System.out.println("狗在吃骨头...");
    }
}

class Cat extends Animal{
    public void eat(){
        System.out.println("猫在吃鱼...");
    }
}
```

```
public class TestConvert {
    public static void main(String[] args) {
        Animal a = new Dog();
        Cat cat = (Cat)a;
    }
}
```

Exception in thread "main" [java.lang.ClassCastException](#)

向下转型时，如果父类引用中的子类对象类型和目标类型不匹配，则会发生类型转换异常。

# instanceof关键字

- 向下转型前，应判断引用中的对象真实类型，保证类型转换的正确性。
- 语法：引用 instanceof 类型 //返回boolean类型结果

```
public class TestConvert {  
    public static void main(String[] args) {  
        Animal a = new Dog();  
  
        if(a instanceof Dog){  
            Dog dog = (Dog)a;  
            dog.eat();  
        }else if(a instanceof Cat){  
            Cat cat = (Cat)a;  
            cat.eat();  
        }  
    }  
}
```

当“a”引用中存储的对象类型确实为Dog时，再进行类型转换，进而调用Dog中的独有方法。

运行结果：  
狗在吃骨头...

- 多态的两种应用场景：
  - 使用父类作为方法形参，实现多态。
    - 调用方法时，可传递的实参类型包括：本类型对象+其所有的子类对象。
  - 使用父类作为方法返回值，实现多态。
    - 调用方法后，可得到的结果类型包括：本类型对象+其所有的子类对象。
- 多态的作用：
  - 屏蔽子类间的差异。
  - 灵活、耦合度低。