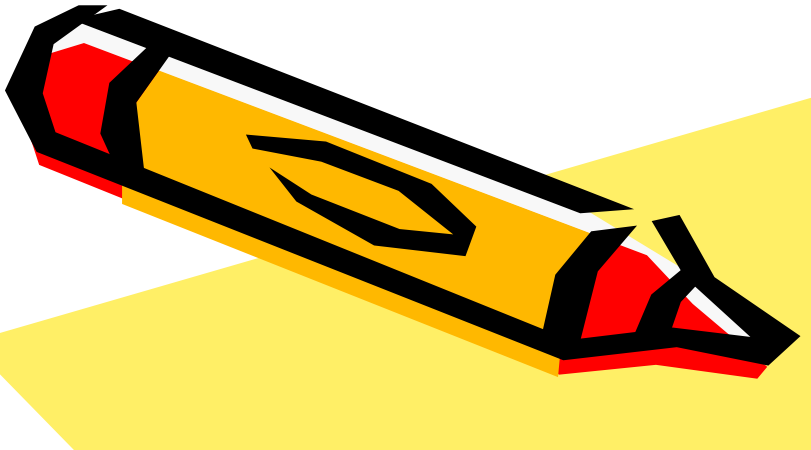
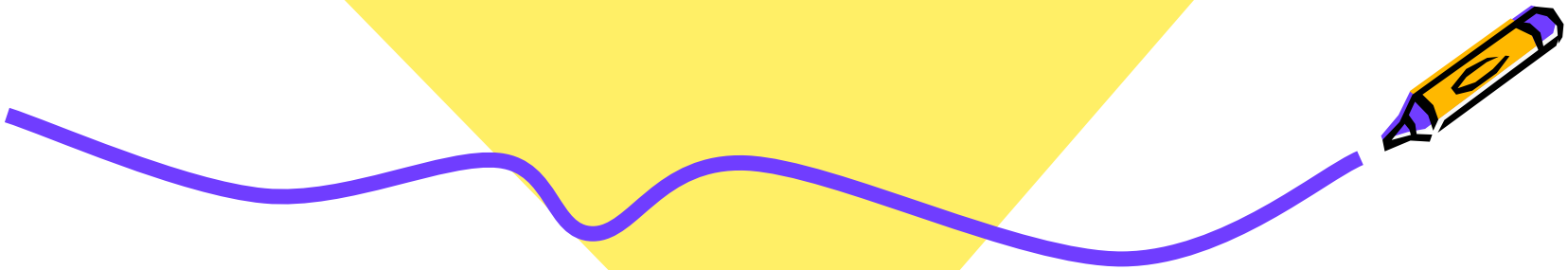


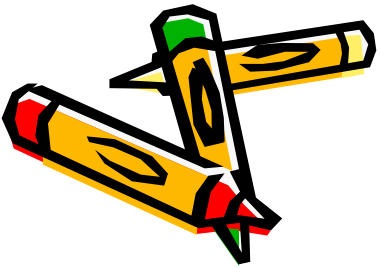
第3章 面向对象基础

(Object Oriented)



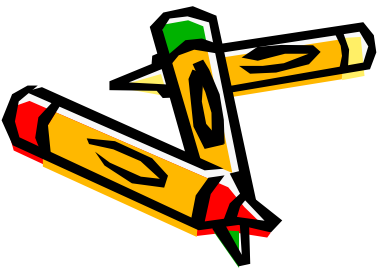
涉及到课本中的章节

- 第3章 类的封装、继承和多态



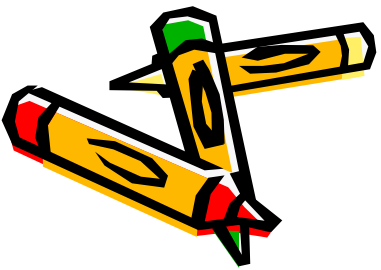
本章目标：

- 如何自己定义Java类并创建对象，使用对象
- 掌握类的三大特性
- 使用OO思想组织代码

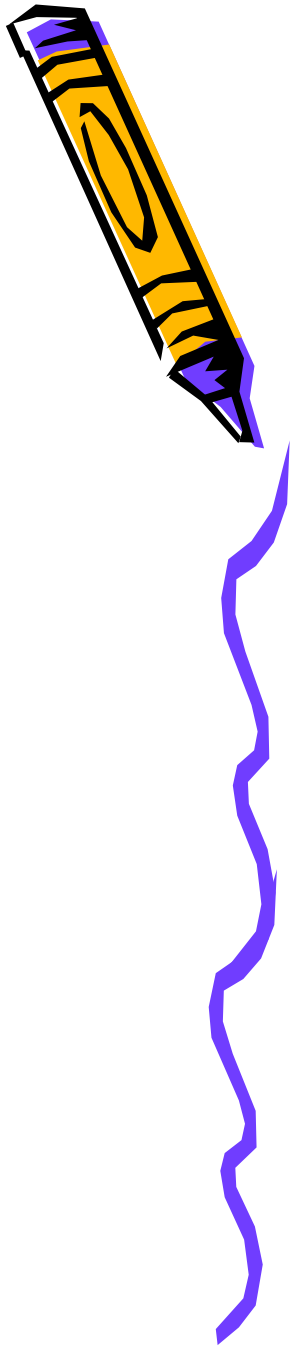


第3章： OO基础

- 3.1 类与对象
- 3.2 类的特性
- 3.3 Java中内存分配机制(补充)



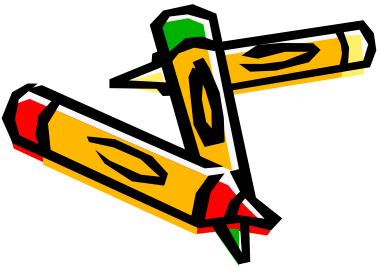
· 3.2 类的特性



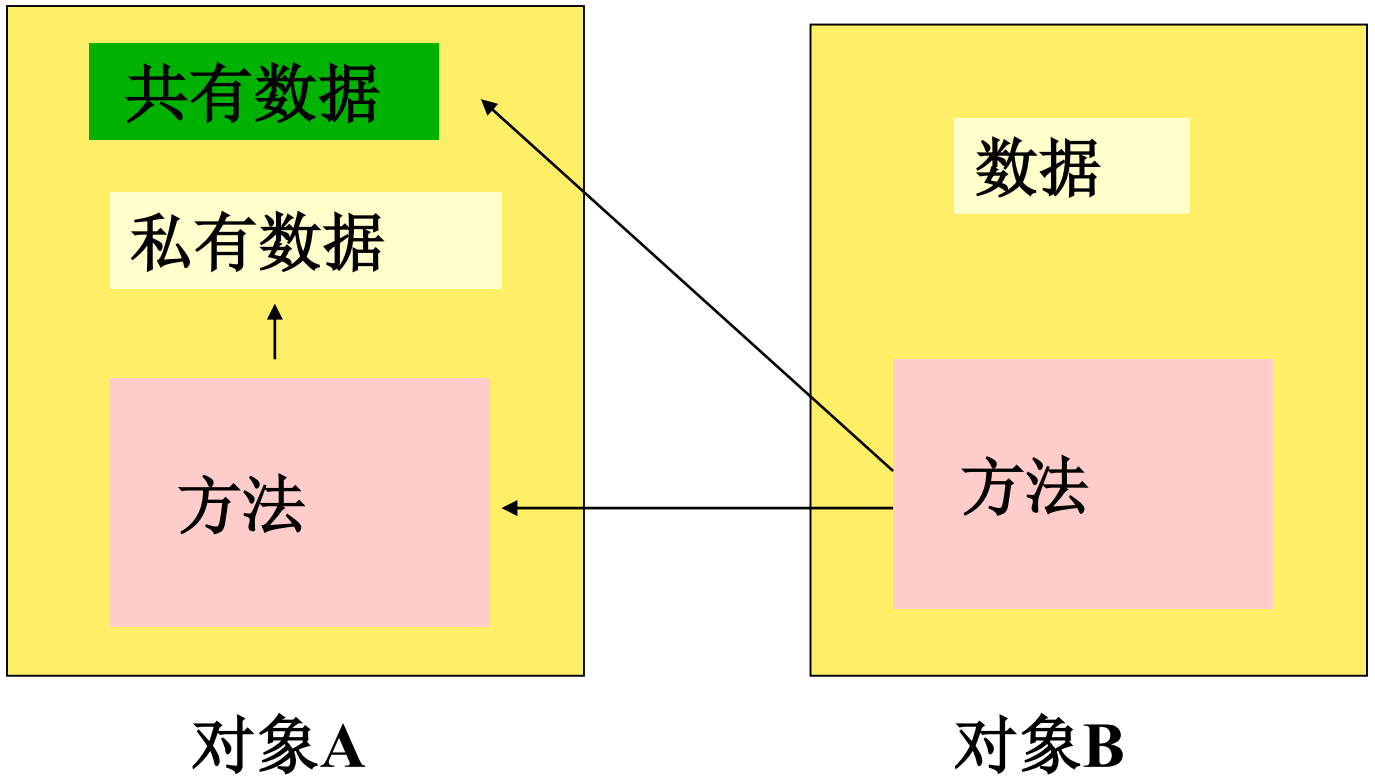
A. 封装性

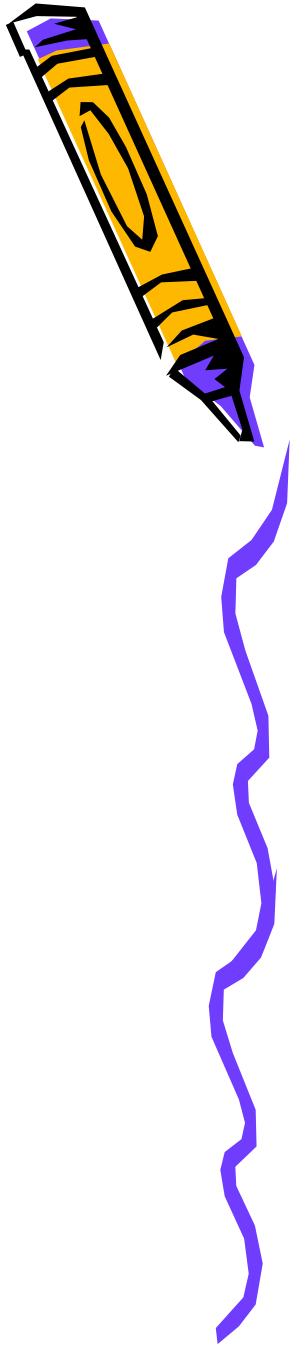
B. 继承性

C. 多态性



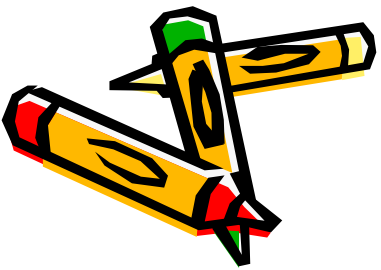
- A：封装性





1.访问控制

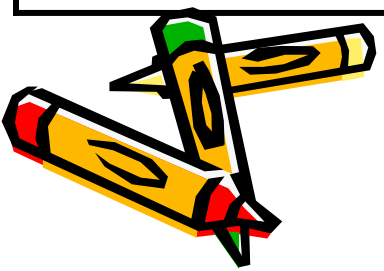
2.实例成员和类成员



- 1.访问控制：

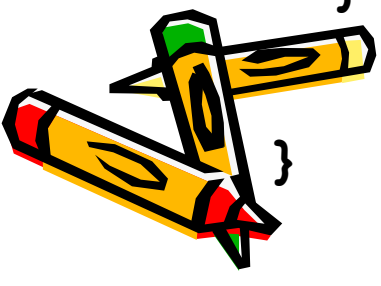
在变量和方法声明时，加权限修饰符进行访问控制。
访问控制就是当前成员在成员之外的可见性。

范围 / 修饰符	当前类	当前包	其他包的子类	其他包中的类
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

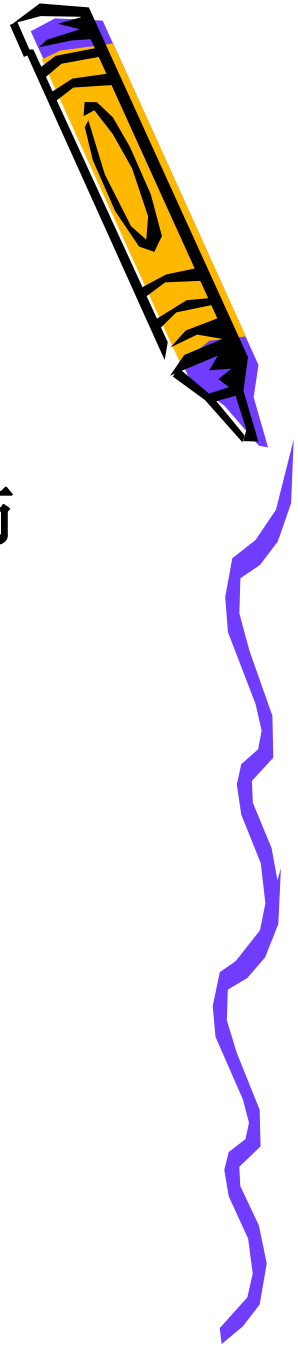


访问控制举例：

```
class Student{
    private  long  id;                  // 变量一般都声明为私有
    private  char  gender;              // 以防止其他对象任意更改
    private  int classID;

    public  long getID()                // 方法一般是对外提供服务的
          {  return id; }               // 所以声明为公共的
    public  boolean setID(long aID) {
         if(aID>=0)
               {   id = aID;
                   return true;
               }
        else {return false;}
    }
    //其他的属性的getter和setter方法.     //对外可以提供读写操作
    public String getInfo( ){
    }

}
```

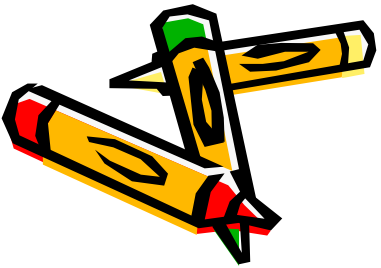


2.类成员和实例成员



- 类成员也称为静态成员，用static修饰

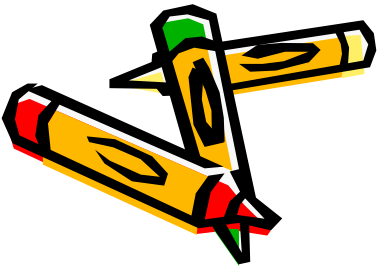
- (1) static 变量
- (2) static 方法



(1) static变量　　--类变量

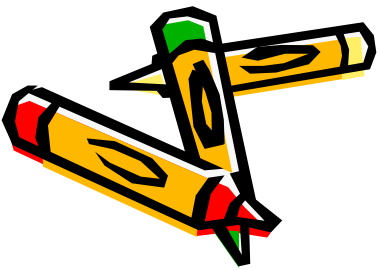
- 用static修饰的变量，在类初始化完成就开始生效。程序运行时，通过类名可直接获取，生成对象也可以获取。

- 非static修饰的成员变量叫做实例变量



引申问题？

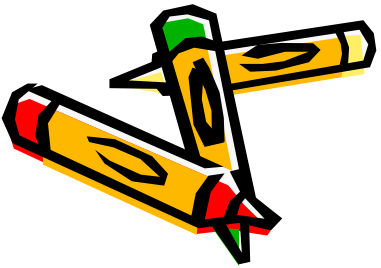
```
public class TestStatic {
    public static int myint=123;
    public void testmystatic(){
        static int myinner=125;
        System.out.println(myinner);
    }
}
```



引申问题总结：

- static关键字在修饰变量的时候只能修饰成员变量，不能修饰方法的局部变量。

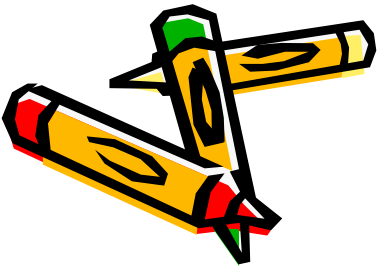
- 局部变量只在成员方法内有效，方法结束自动销毁。



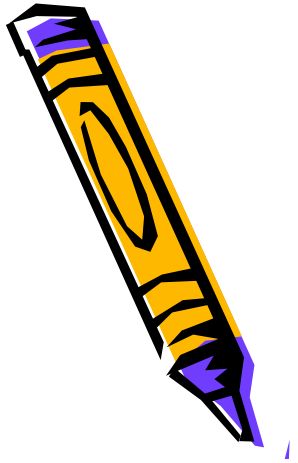
(1) static方法　　--类方法

- 只能访问类变量，
- 而且既可以通过对象来调用，也可以通过类名来调用

- 非static修饰方法叫做实例方法或者成员方法
- 实例方法可以访问成员变量或者类变量,但是只能通过对象访问实例方法



静态成员演示

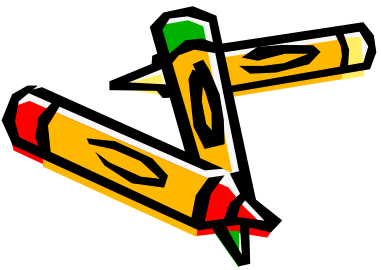


```
package cn.sdu.java.classObject;

class StaticMember{
    String name;
    int age;     //成员属性（另一种叫法（实例变量）
    static String country;     //静态成员属性（另一种叫法：类变量）
//     static String country="中国"; //【示例3】
    static void staticFunction(){
        System.out.println(country);     //静态方法里面访问静态成员属性时，不能使用this关键字，不能访问非静态成员属
    }

    public static void main(String[] args){
        StaticMember A     =     new StaticMember();
        A.country     =     "中国";       //【示例1】
//        StaticMember.country = "中国"; //【示例2】
        StaticMember B =     new StaticMember();
        System.out.println(A.country);     //对象A，定义了country成员属性的值，可以输出，这是常规的方式
        System.out.println(StaticMember.country);//类名.static成员，也可以输出

        B.staticFunction();     //调用静态方法
    }
}
```

另：static 代码块

- 语法：

```
static{
    ……//语句块
 }
```



问题：

```
public class StaticBlock {

        static {
                System.out.println("Printing in static block");
        }
        public StaticBlock(){
                System.out.println("Printing in StaticBlock construction method");
        }
        public static void main(String[] args){
                StaticBlock s1=new StaticBlock();
                StaticBlock s2=new StaticBlock();
        }
}
```

- Please give me the right output and understand the running sequence

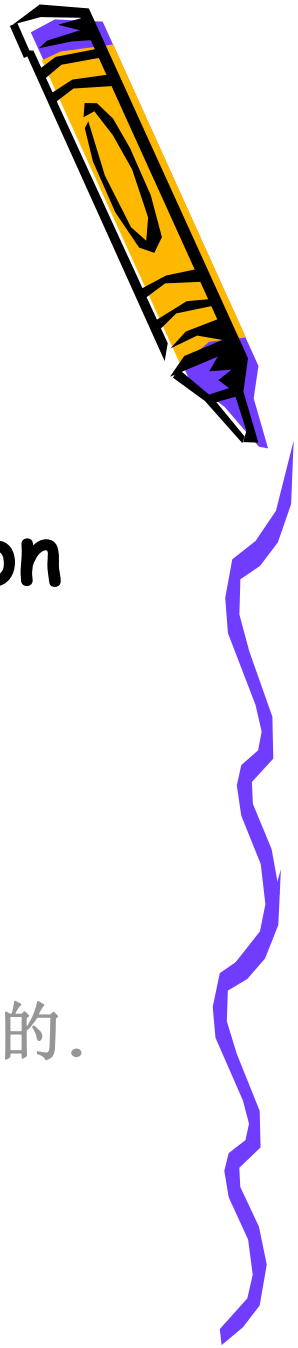
问题总结:

- static代码块只在JVM初始化类时执行一次，以后不会再执行
- 此应用一般用于一些初始化操作.

  例如数据库连接等

深入思考?

将一些初始化代码放到类的构造方法呢?

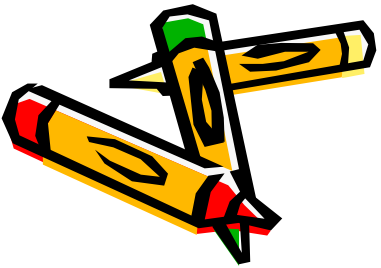


更深思考:

- static 也是实现设计模式之Singleton设计模式的一种方式

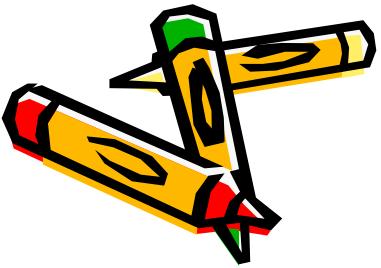
实现:

可以将所有的成员(属性和方法)都定义成static的.

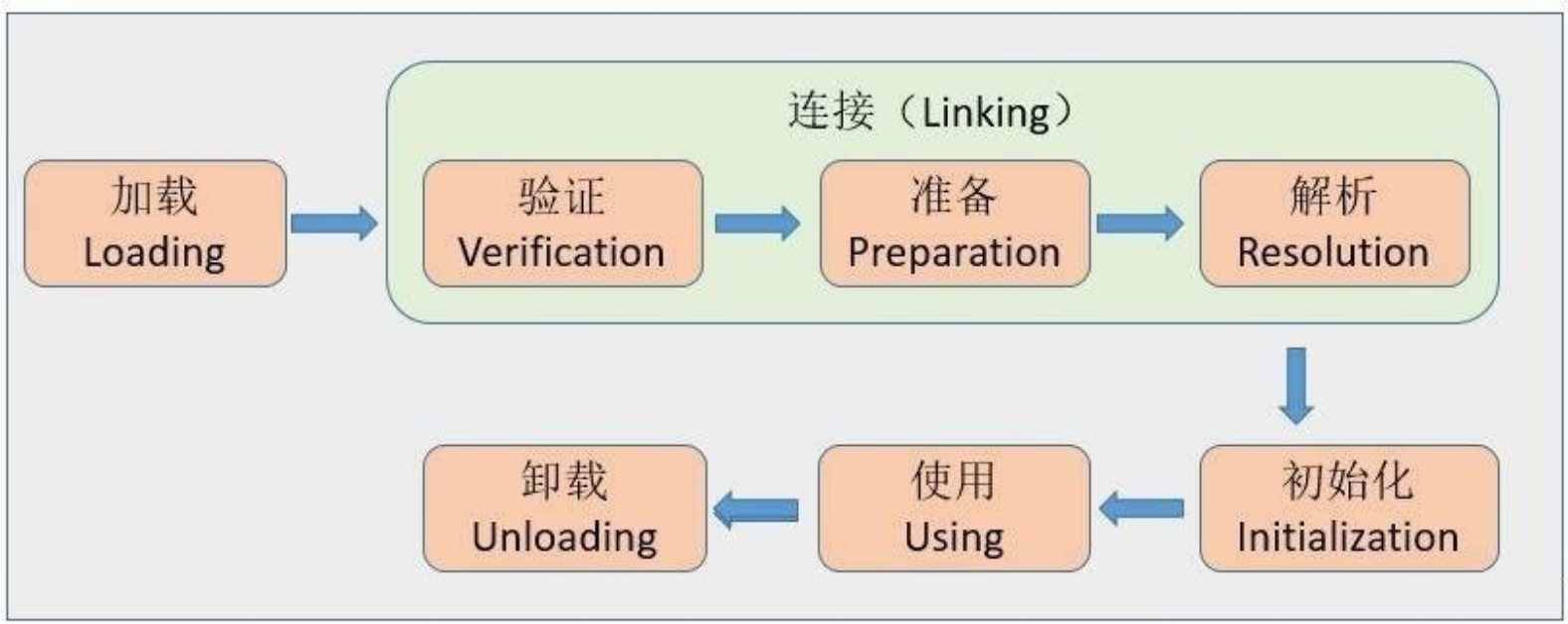


Singleton设计模式的应用：

- 见java.lang.Math类

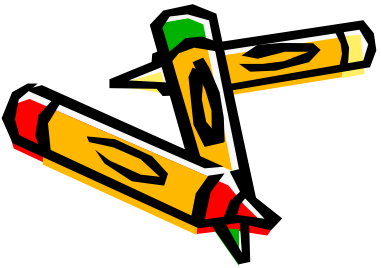


类的生命周期

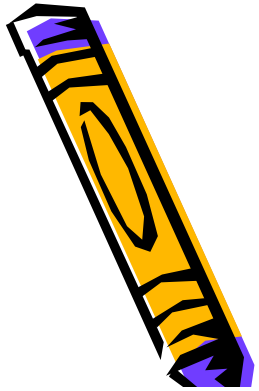

连接（Linking）
加载
Loading
验证
Verification
准备
Preparation
解析
Resolution
初始化
Initialization
使用
Using
卸载
Unloading

类初始化的情况

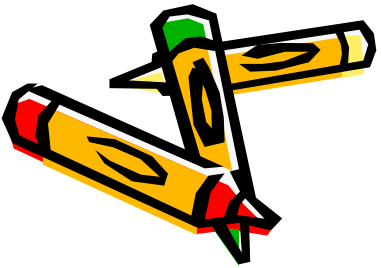
- 1. 创建类的实例
- 2. 调用类的静态方法
- 3. 读取和设置类的静态变量
- 4. 调用Java API中的某些反射方法
- 5. 某个类的子类初始化时
- 6. 含有main()方法的类启动时



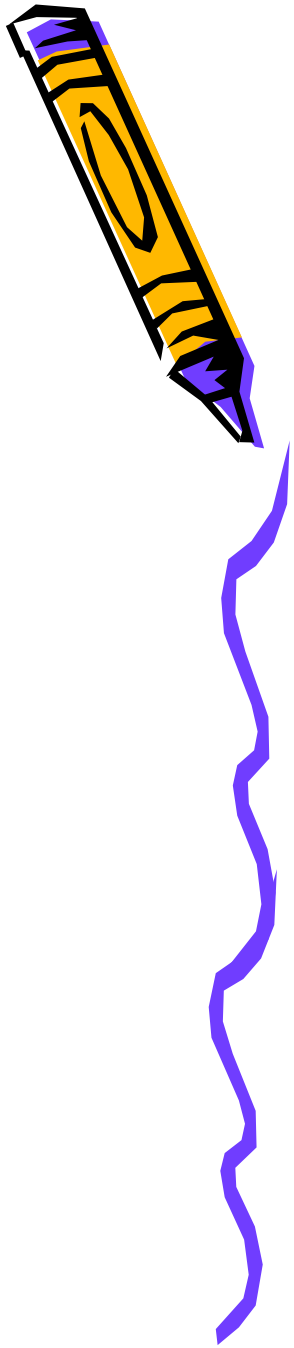
静态成员演示（初始化）



```
1  package cn.sdu.java.classObject;
2
3  class StaticMemberInitial{
4
5      static {
6          System.out.println("Static Block");
7      }
8      public static void main(String[] args){
9          System.out.println("//No Class operation");
10         StaticMemberInitial A = new StaticMemberInitial();  //此处未再执行static代码块
11         //到StaticMemberInitialTest里再试试
12     }
13
14 }
```



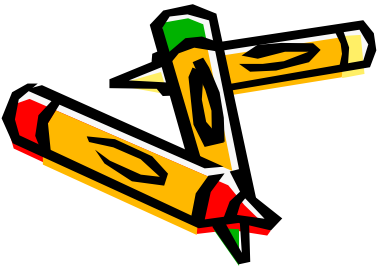
· 3.2 类的特性



A. 封装性

B. 继承性

C. 多态性



·B 类的继承

父类

自行车

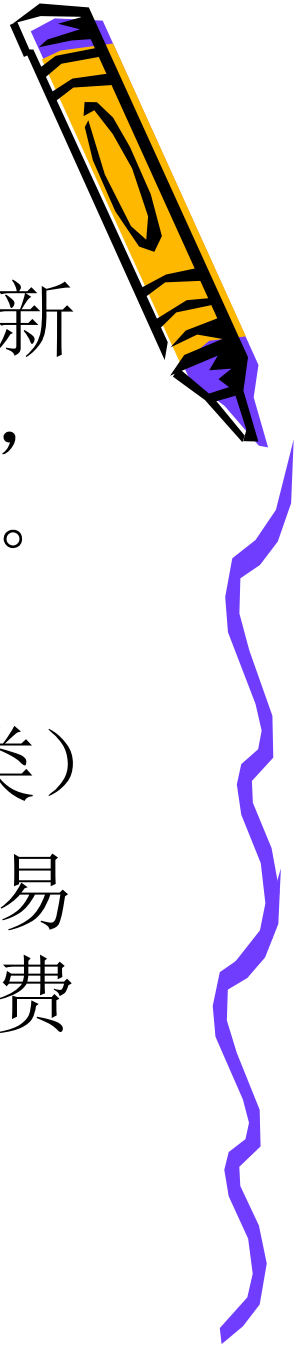
子类

山地自行车

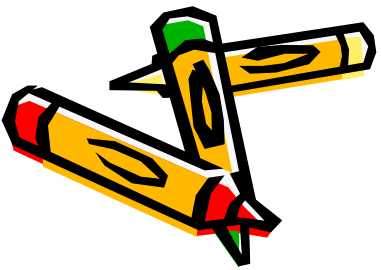
公路自行车

双座自行车

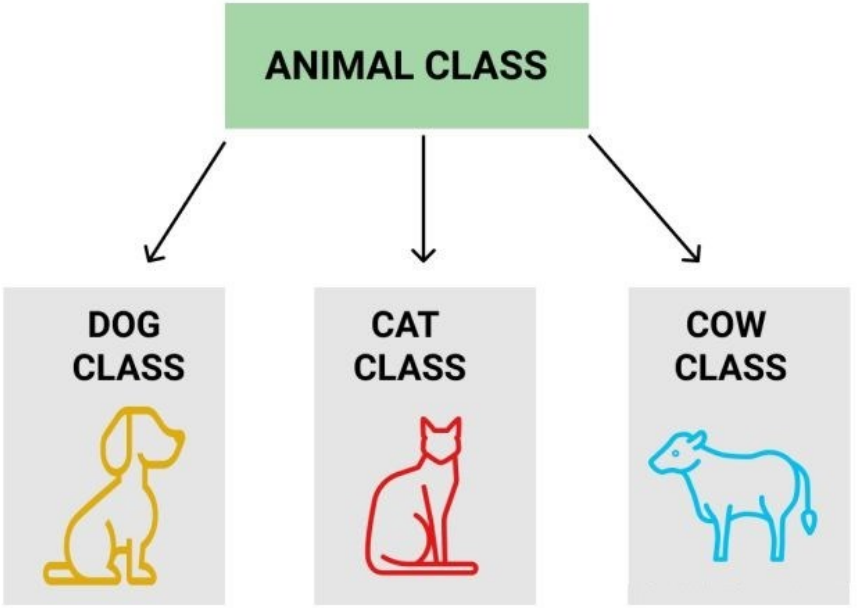
继承的概念

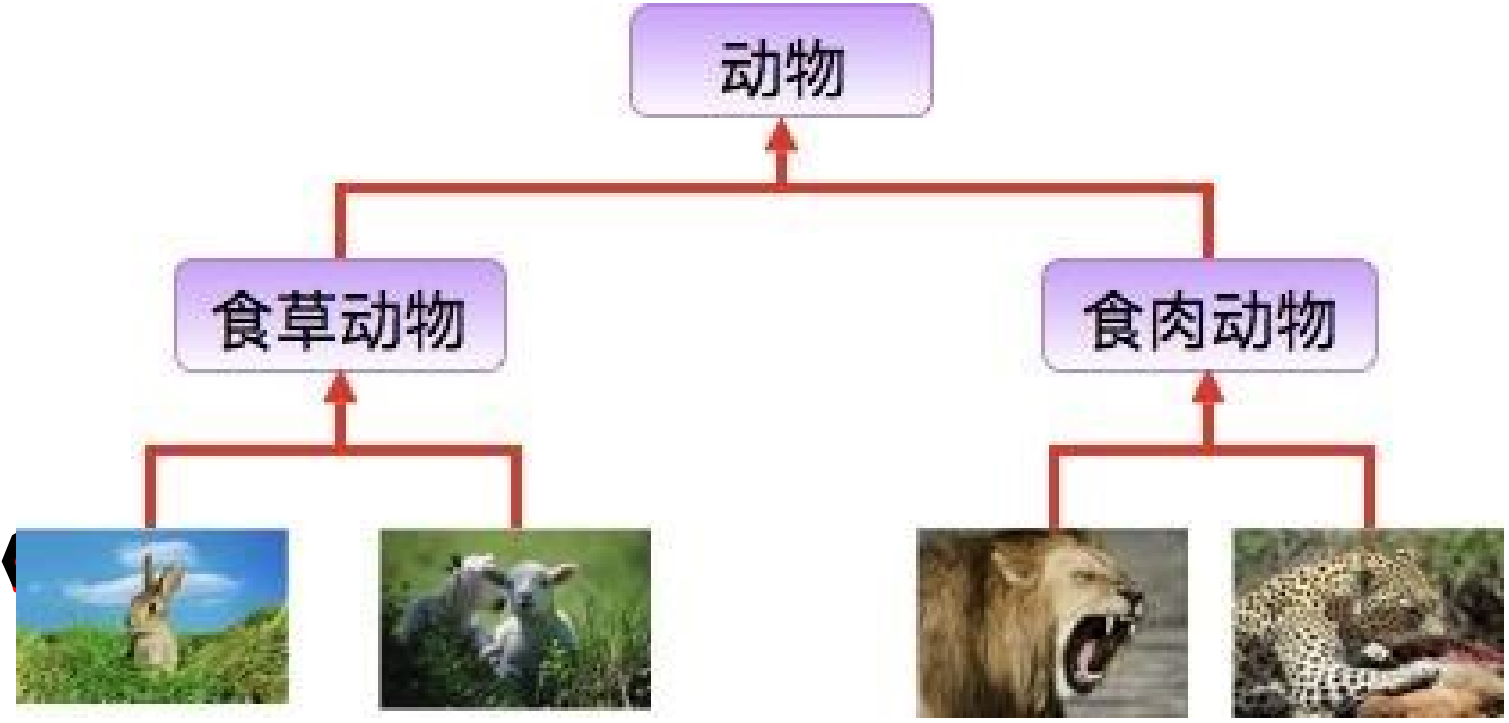


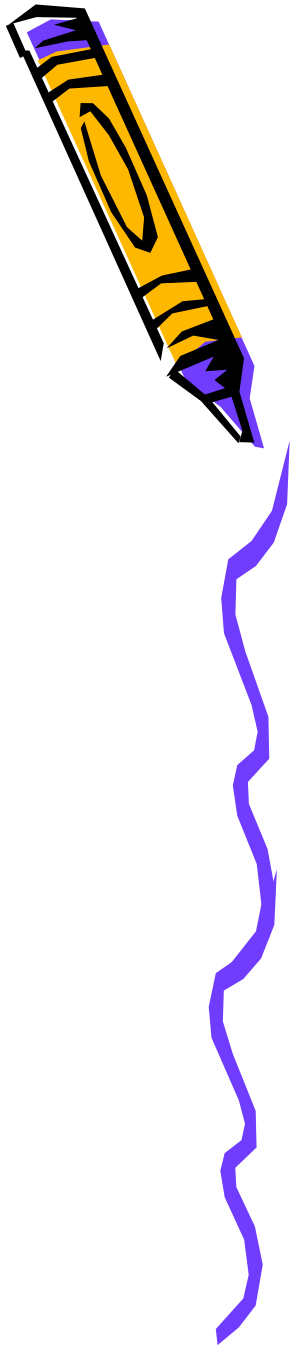
- 继承是从已有的类中派生出新的类，新的类能吸收已有类的数据属性和行为，并能扩展吸收能力或者创建新的能力。
- 被继承的类称为**父类**或超类（基类）
- 新继承的类称为**子类**（派生类/扩展类）
- 这种技术使得复用以前的代码非常容易，能够大大缩短开发周期，降低开发费用。



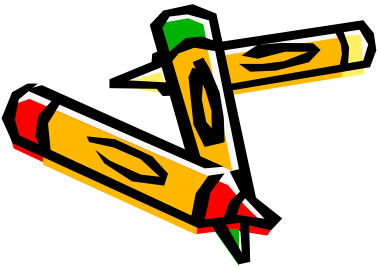
类的继承示例







1. Java子类声明
2. this,super和instanceof
3. 最终类



- 1 Java 中子类声明：

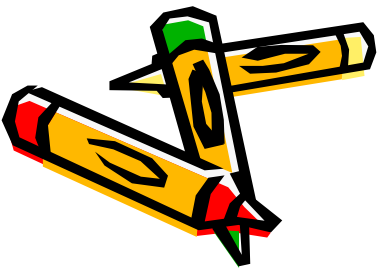
---显式声明格式为：

[<修饰符>] class <子类名> extends <超类名>

例： class Sub extends Super

class Son extends Father

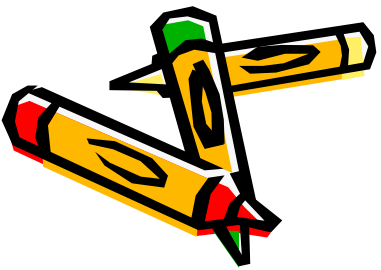
class Extend extends Base



Java 继承规范：

- **Java 只支持单继承。即extends后只有一个类**

- **Java所有类的根父类是Object类**



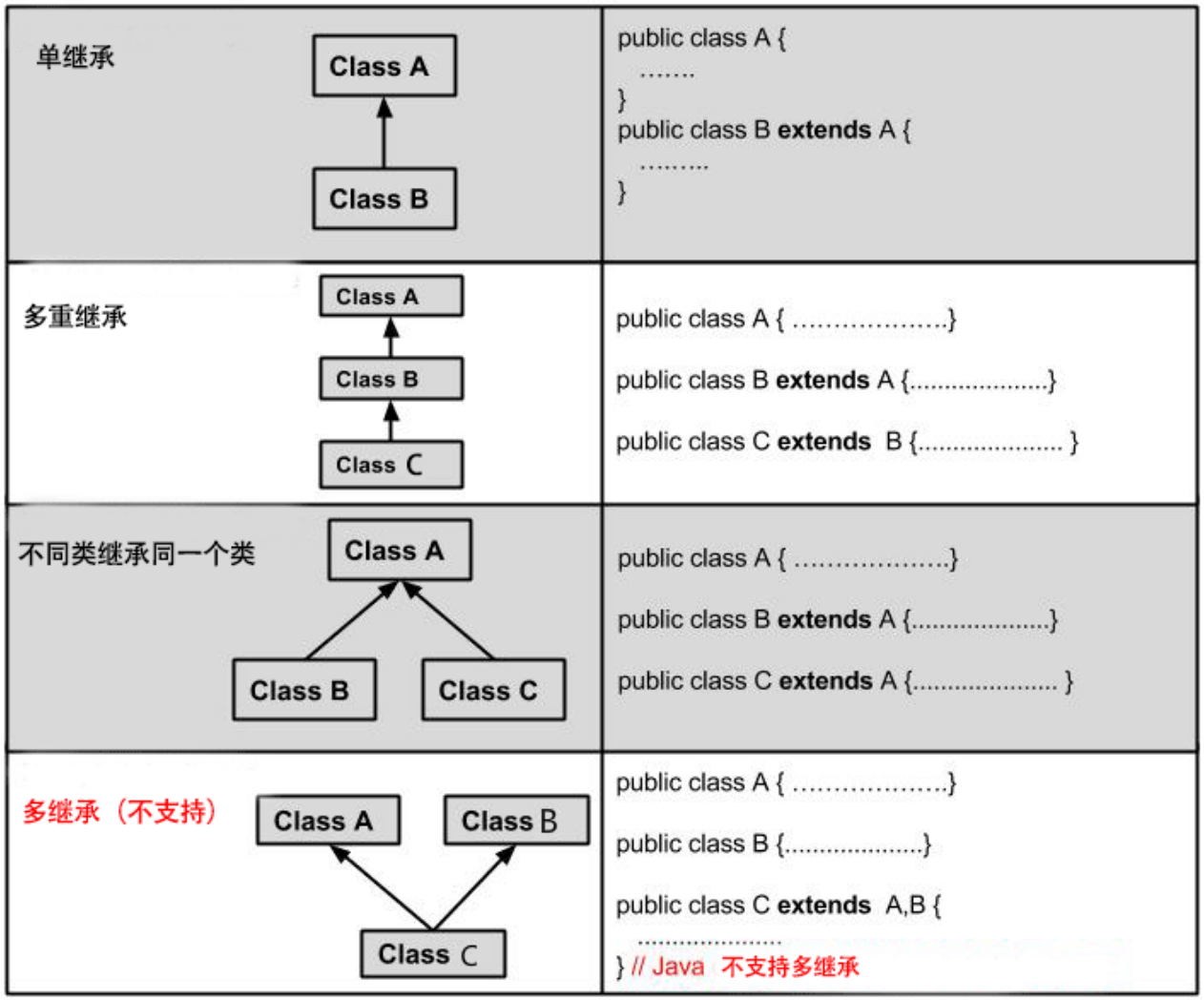
Java继承场景

智能生物
↑
人类

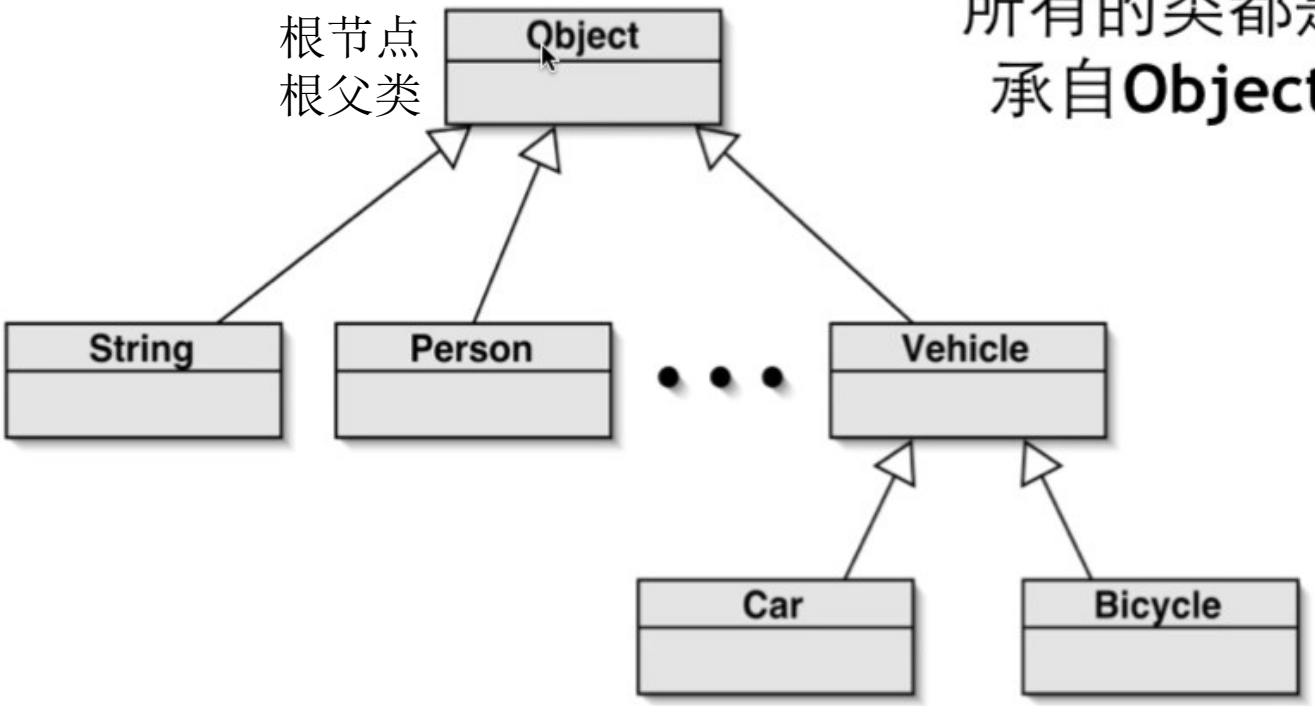
人类
↑
亚洲人
↑
中国人

人类
↑ ↑
亚洲人 非洲人

Java 类的继承中
不支持混血模式



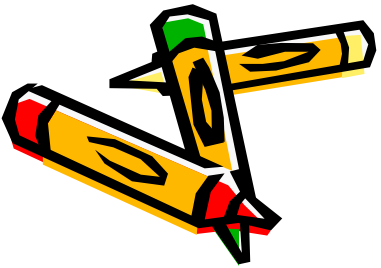
Java类树状结构
根父类Object



所有的类都是继承自Object的

Java中继承原则：

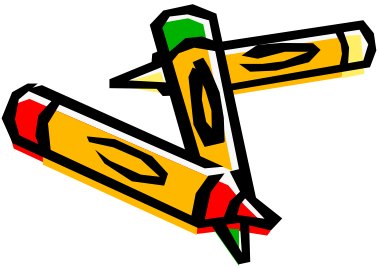
- 子类继承父类中所有的除了构造方法之外的所有的非private修饰的成员。



引申问题:

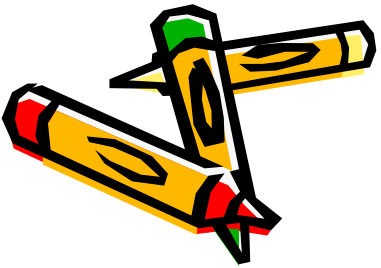
---体会生成子类对象的父子生成过程

```
public class Animal{
   static {
      System.out.println("static block in  animal");
   }
   public Animal(){
      System.out.println("create an animal object");
   }
}
```



```
public class Dog extends Animal{
    static{
        System.out.println("static block in dog");
    }
    public Dog(){
        System.out.println("create a dog object");
    }
}
```

```
public class TestStaticBlock{

    public stati void main(String[] args){
        Dog d=new Dog();
    }
}
```



继承关系父子生成过程演示

Animal.java

```java
package cn.sdu.java.classObject.extend;

public class Animal {
    static {
        System.out.println("Static block in animal");
    }
    void eat() {
        System.out.println("animal : eat");
    }
}
```

Dog.java

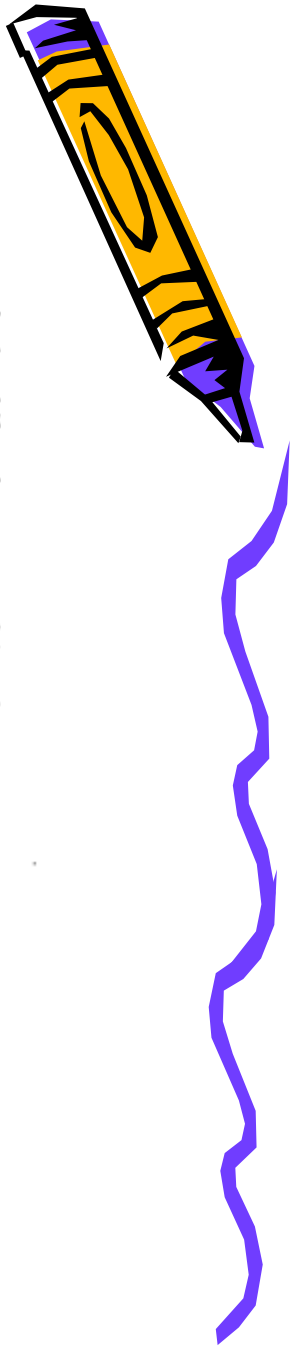
```java
package cn.sdu.java.classObject.extend;

public class Dog extends Animal {
    static {
        System.out.println("Static block in dog");
    }
    public Dog(){
        System.out.println("create a dog object");
    }
}
```

Test.java

```java
package cn.sdu.java.classObject.extend;

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

        //继承关系父子生成过程演示
        Dog d = new Dog();

    }
}
```

问题现象原理

Thinking in Java (4th)-P243



7.2.1 初始化基类

由于现在涉及基类和导出类这两个类，而不是只有一个类，所以要试着想像导出类所产生的结果对象，会有点困惑。从外部来看，它就像是一个与基类具有相同接口的新类，或许还会有一些额外的方法和域。但继承并不只是复制基类的接口。当创建了一个导出类的对象时，该对象包含了一个基类的子对象。这个子对象与你用基类直接创建的对象是一样的。二者区别在于，后者来自于外部，而基类的子对象被包装在导出类对象内部。

当然，对基类子对象的正确初始化也是至关重要的，而且也仅有一种方法来保证这一点：在构造器中调用基类构造器来执行初始化，而基类构造器具有执行基类初始化所需要的所有知识和能力。Java会自动在导出类的构造器中插入对基类构造器的调用。下例展示了上述机制在三层继承关系上是如何工作的：

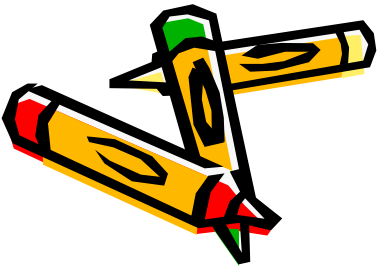
```
//: reusing/Cartoon.java
// Constructor calls during inheritance.
import static net.mindview.util.Print.*;

class Art {
  Art() { print("Art constructor"); }
}

class Drawing extends Art {
  Drawing() { print("Drawing constructor"); }
}

public class Cartoon extends Drawing {
  public Cartoon() { print("Cartoon constructor"); }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
} /* Output:
```

检验继承关系 --避免滥用继承

- 两个类如何测试是否满足继承关系?

继承关系标识子类是父类的一种特殊类型，子类对象“即是”父类对象。但反之则不然，父类对象显然不是它子类的对象。

- is-a单向测试
- Jack(Monitor) is a student
- Simba(Lion) is an animal
- John(Worker) is human



2.this ,super和instance of

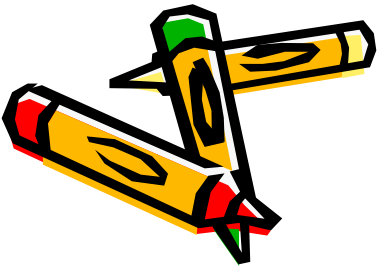
instanceof

- ----用来测试一个指定的对象是否为指定类(或其子类)的实例。若是则返回true，否则返回false

- boolean result = obj instanceof Class

①Class obj

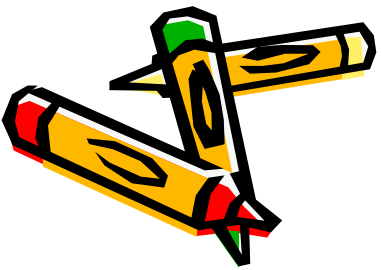
②SubClass obj



2.this ,super和instance of

- **this 引用**

Java中每个对象都具有对其自身引用的访问权



- **(1)指代对象本身：this**

```
 void equals(Object obj2)
  {Object obj1=this;        //this指调用本方法的当前对象
  }
```

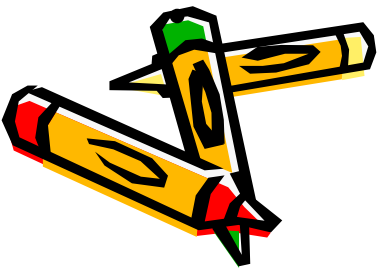
- **(2)访问本类的成员变量和方法**

  this.<变量名>
  this.<方法名>

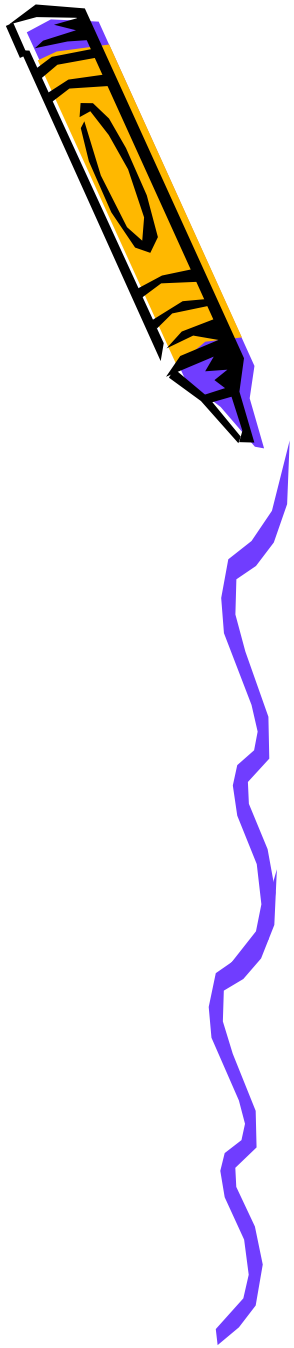
```
Class A1
{ int i=1;
  void me(int i)
  {
    this.i=2*i;      //this.i中的i指成员变量i，表达式中的i指参数i
  }
}
```

- **(3) 调用本类的构造方法**

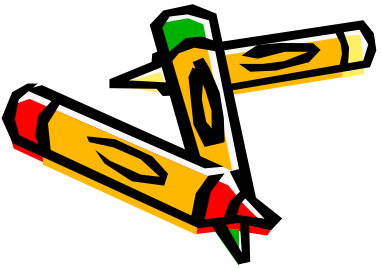
  this(<参数列表>)



this典型应用场景



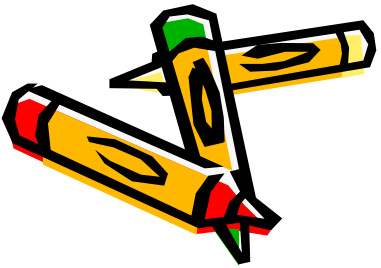
- 类构造器
- Setter方法
- 方法中调用本类的方法



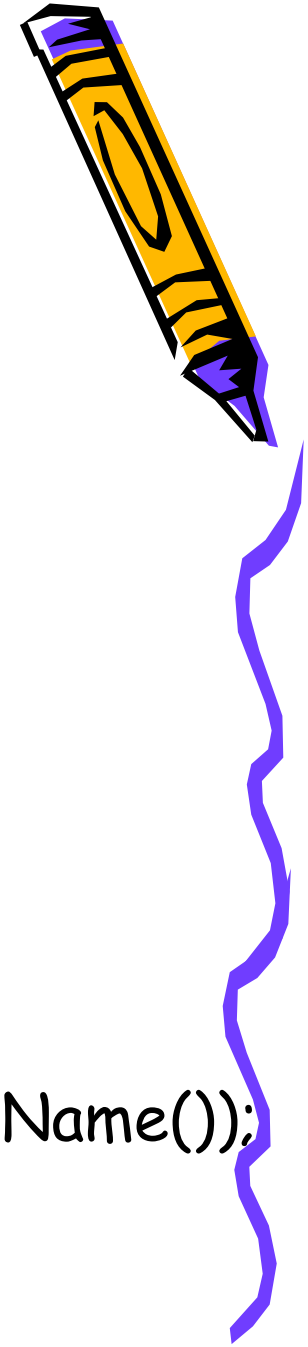
this典型应用场景

- 类构造器

```
public Human (String name, String gender, int
   height, int weight)
  {
        this.name= name;
        this.gender=gender;
        this.height= height;
        this.weight= weight;
   }
```



this典型应用场景

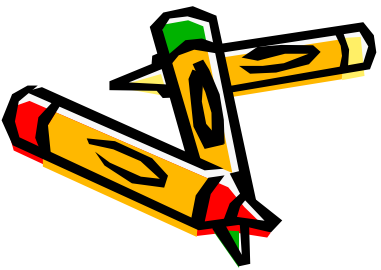


- **Setter方法**

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

- 方法中调用本类的方法

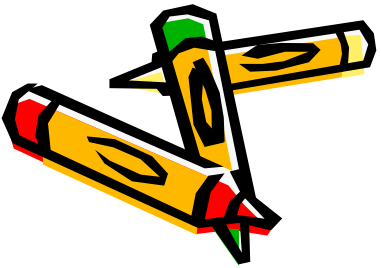
```
void work() {
    System.out.println("姓名：" + this.getName());
}
```

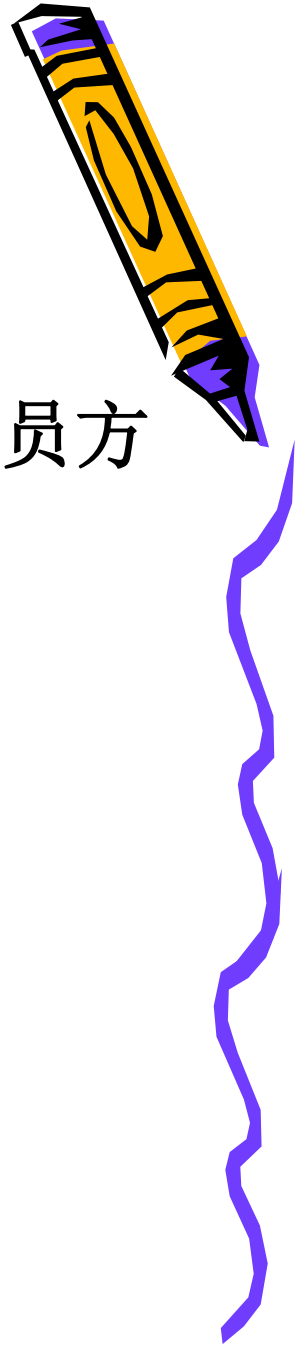


2.this ,super和instance of

super引用

- 使用关键字super,可以引用被子类隐藏的超类的成员变量和成员方法，称为super引用





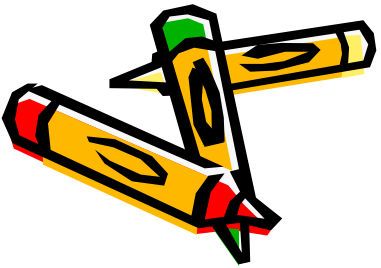
- (1)访问被子类隐藏的超类的成员变量和成员方法。格式:

  super.<变量名>

  super.<方法名>

- (2)调用超类的构造方法，格式:

  super(<参数列表>)



super、this演示

Animal.java

```
1  package cn.sdu.java.classObject.animalDemo;
2
3  public class Animal {
4      static {
5          System.out.println("Static block in ani
6      }
7      void eat() {
8          System.out.println("animal : eat");
9      }
10 }
```

Dog.java

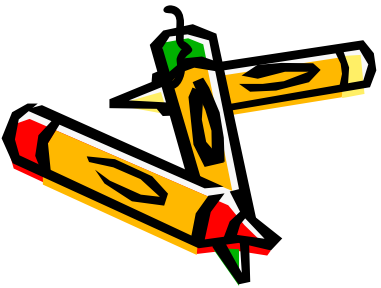
```
3  public class Dog extends Animal {
4      static {
5          System.out.println("Static block in dog
6      }
7      void eat() {
8          System.out.println("dog : eat");
9      }
10
11     void eatTest() {
12         this.eat(); // this 调用自己的方法
13         super.eat(); // super 调用父类方法
14     }
```

Test.java

```
1  package cn.sdu.java.classObject.animalDemo;
2
3  public class Test {
4      public static void main(String[] args) {
5
6          //继承关系父子生成过程演示
7  //      Dog d = new Dog();
8  ////        System.out.println("//No Class oper
9
10         //super、this演示
11         Animal a = new Animal();
12         a.eat();
13         Dog d = new Dog();
14         d.eatTest();
15     }
16 }
17
```

引申问题（方法覆盖）

```
public class Father{
   public Father(){
       System.out.println("Create a Father");
   }
   public void FatherMethod(){
   }
```

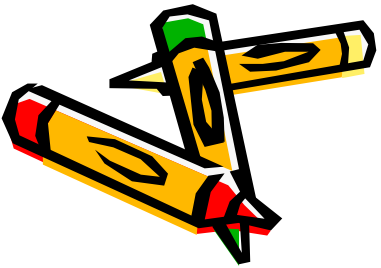


```
public class Son extends Father{
    public Son(){
       super();
       System.out.println("Create a Son");
    }
    public void FatherMethod(){
         super.FatherMethod();
         System.out.println("Son add some code");
    }
}
```

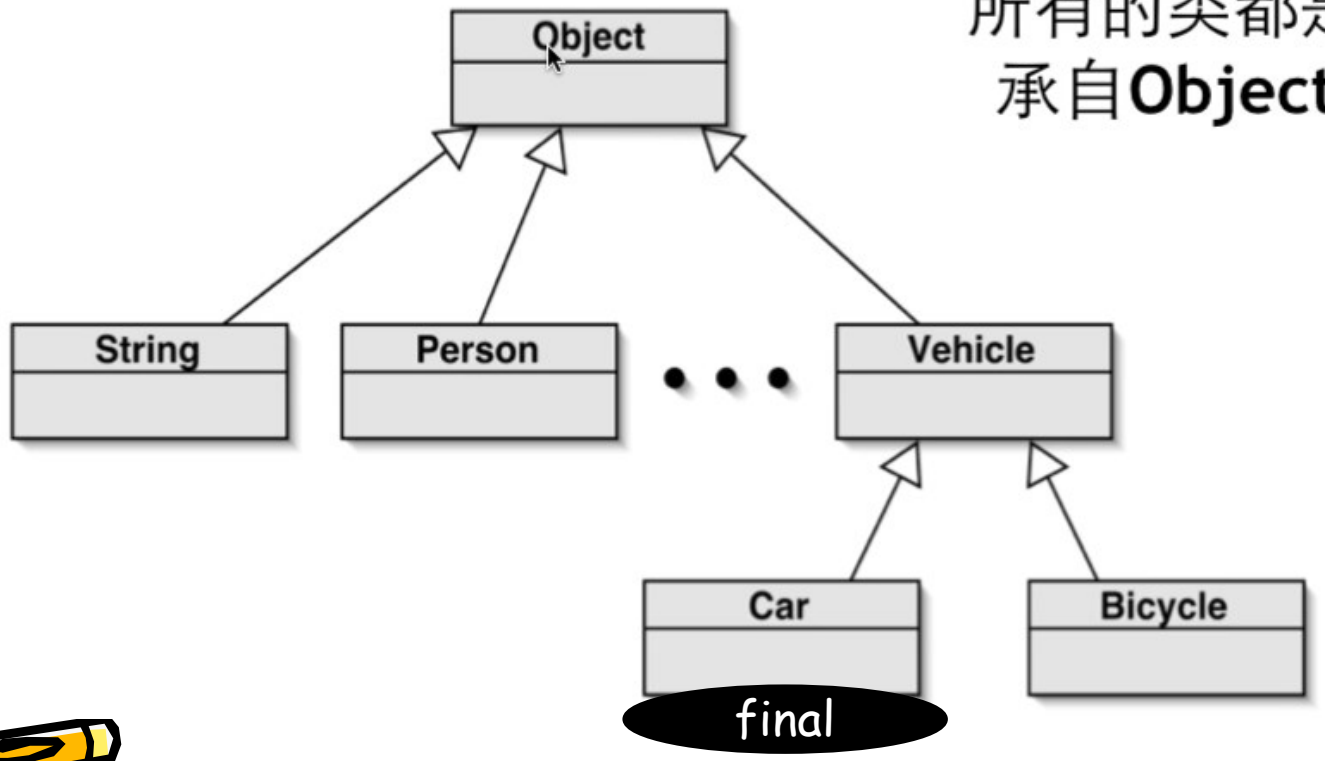
3.最终类(final)

- ---不能被继承的类，final来说明最终类
- 例： final class C1　　//合法，C1为最终类
- 　　class C2 extends C1　　//非法，C2不能为最终类的子类

- 引：最终方法---用final修饰，说明此方法不能被子类所覆盖
- final void m1( )　　//合法，m1为最终方法

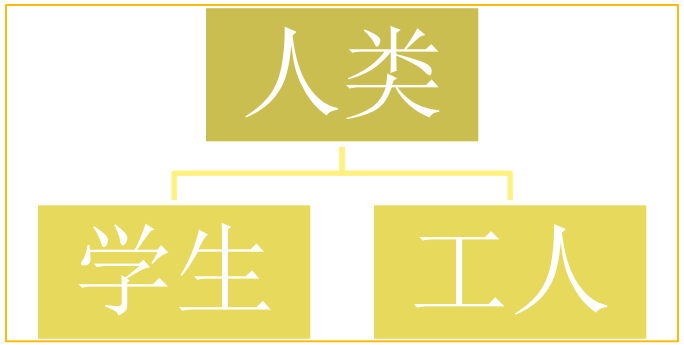


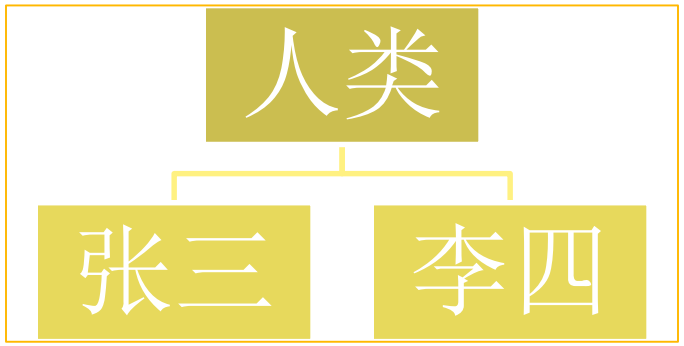
Java类树状结构的最终类

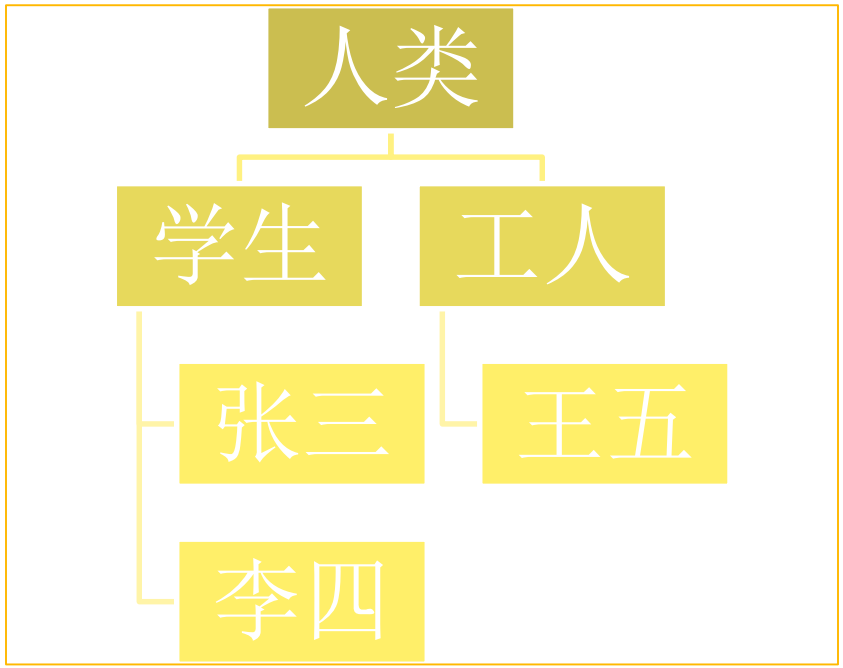


叶子节点，无法再开枝散叶

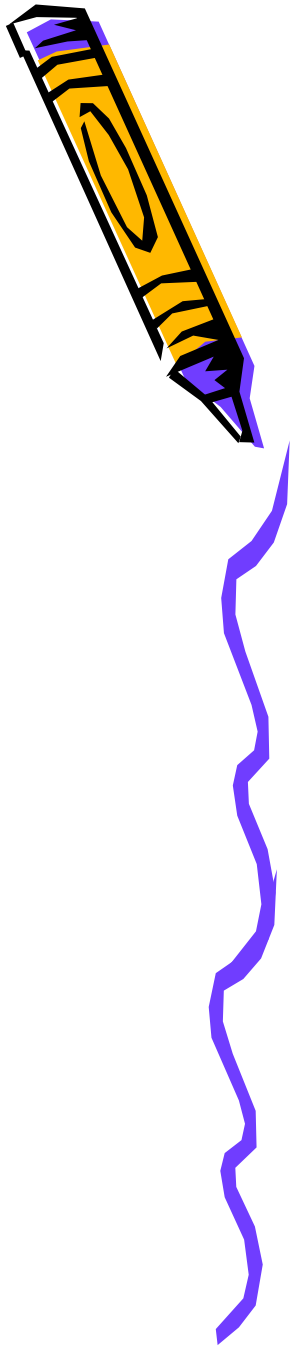
类的继承 vs 类的实例







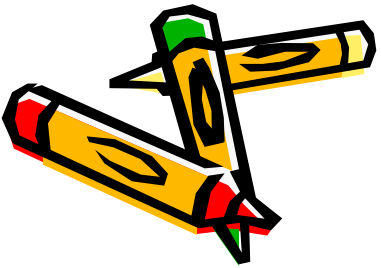
· 3.2 类的特性



A. 封装性

B. 继承性

C. 多态性



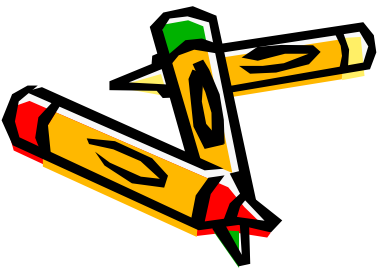
· C 类的多态——OO的核心之核心

1 方法的多态

1).方法的重载(overload)

2).方法的重写(override)—覆盖

2 类型的多态



1.方法的重载(overload)

多音字（数）

- 重载指在同一个类中至少有两个方法用同一个名字，但有不同的参数。
- 重载使得从外部来看，一个操作对于不同的对象有不同的处理方法。
- 调用时，根据参数的不同来区别调用哪个方法。
- 方法的返回类型可以各不相同，但它不足以使返回类型变成唯一的差异。重载方法的参数表必须不同。

```
class Car
{   int colorNumber;
    int doorNumber;
    int speed;

    void  pushBreak()   { speed=0; }
    void  pushBreak(int  aDeltaSpeed)   {  speed -= aDeltaSpeed; }
    void  add_oil() {  …   }
}
```

2.方法的重写(override)——覆盖

旧词新意（雷）

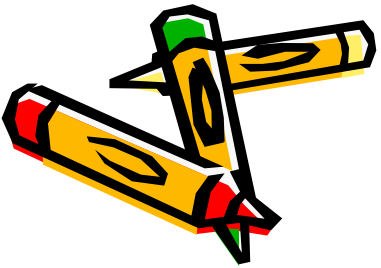
- 当用于子类的行为与父类的行为不同时，覆盖机制允许子类可以修改从父类继承来的行为。
- 覆盖就是在子类中创建一个与父类方法有不同功能的方法，但具有相同的名称、返回类型和参数表。
  - 若参数表不同，则不是覆盖，而是重载。
  - 若参数表相同，但返回值不同，则编译出错。

```
class Car
{   int colorNumber;
    int doorNumber;
    int speed;

    void  pushBreak()
    { speed=0; }
    void  addOil() {  …   }
}
```

```
class TrashCar  extends  Car
{   double amount;
    void  fillTrash() {  …  }
    void  pushBreak()
    {  speed=speed-10; }
}
```

重载与重写的区别

- 相同点：
  - 都涉及两个同名的方法。
- 不同点：
  - 类层次
    - 重载涉及的是同一个类的两个同名方法；——编译时多态
    - 重写涉及的是子类的一个方法和父类的一个方法，这两个方法同名。 ——运行时多态
  - 参数和返回值
    - 重载的两个方法具有不同的参数，可以有不同返回值类型；
    - 重写的两个方法具有相同的参数，返回值类型必需相同。



一句话总结

overload重载，方法声明像是一样入参出参不一样

override覆盖，方法声明就是一样里面功能不一样

类型多态举例：

Animal.java

```
1  package sdu.one;
2
3  public  abstract class Animal {
4
5    public Animal() {
6             System.out.println("Animal is creating
7        }
8
9     public abstract void makeSound();
10 }
11
```

Lady.java

```
3  public class Lady {
4      Animal d;
5      public Lady(Animal d) {
6          this.d=d;
7      }
8      public void showMyPetSound() {
9          d.makeSound();
10     }
11
12 }
```

Dog.java

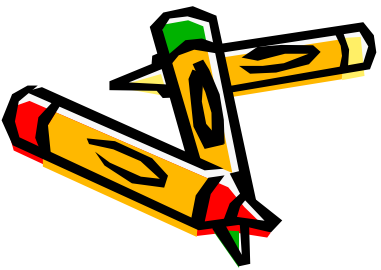
```
3  public class Dog extends Animal{
4
5        public Dog() {
6            System.out.println("Dog is creating..."
7            }
8
9      @Override
10     public void makeSound() {
11         System.out.println("Dog is making sound")
12     }
```

Test.java

```
1  package sdu.one;
2
3  public class Test {
4
5      public static void main(String[] args) {
6          // TODO Auto-generated method stub
7          Dog d=new Dog();
8          Lady l=new Lady(d);
9          l.showMyPetSound();
10     }
11
```

运行时多态的必要条件：

- ❶ 要有继承
- ❷ 要有重写（覆盖）
- ❸ 父类引用要指向子类对象



预习+思考

- 请根据多态的代码考虑一下,如果父类是抽象类或者接口,可以使用多态来增强类的扩展性和可维护性吗?