

纲要

面向过程与面向对象的区别

面向对象的三大特性

类与对象的概念

类的定义

对象的创建和使用

Java 中的封装特性

构造函数

对象内存分析及引用传递

this 关键字

static 关键字

单例模式初步

类的继承

方法的覆盖

super 关键字

final 关键字

抽象类

接口

多态

抽象类与接口的区别

Object 类

package 和 import

访问权限控制

内容

1.1、面向过程与面向对象的区别（盖饭、蛋炒饭）

为什么会出现面向对象分析方法？

因为现实世界太复杂多变，面向过程的分析方法无法满足

面向过程？

采用面向过程必须了解整个过程，每个步骤都有因果关系，每个因果关系都构成了一个步骤，多个步骤就构成了一个系统，因为存在因果关系每个步骤很难分离，非常紧密，当任何一步骤出现问题，将会影响到所有的系统。如：采用面向过程生产电脑，那么他不会分 CPU、主板和硬盘，它会按照电脑的工作流程一次成型。

面向对象？

面向对象对会将现实世界分割成不同的单元（对象），实现各个对象，如果完成某个功能，

只需要将各个对象协作起来就可以。

1.2、面向对象的三大特性

- 封装
- 继承
- 多态

1.3、类与对象的概念

类是对具有共性事物的抽象描述，是在概念上的一个定义，那么如何发现类呢？

通常根据名词(概念)来发现类，如在成绩管理系统中：学生、班级、课程、成绩

学生—张三

班级—622

课程—J2SE

成绩—张三成绩

以上“张三”、“622”、“J2SE”和“张三的成绩”他们是具体存在的，称为对象，也叫**实例**，**也就是说一个类的具体化（实例化），就是对象或实例**

为什么面向对象成为主流技术，主要就是因为**更符合人的思维模式**，更容易的分析现实世界，所以在程序设计中采用了面向对象的技术，从软件的开发的生命周期来看，基于面向对象可以分为三个阶段：

- **OOA**（面向对象的分析）
- **OOD**（面向对象的设计）
- **OOP**（面向对象的编程）-----Java 就是一个纯面向对象的语言

我们再进一步的展开，首先看看学生：

学生：学号、姓名、性别、地址，班级

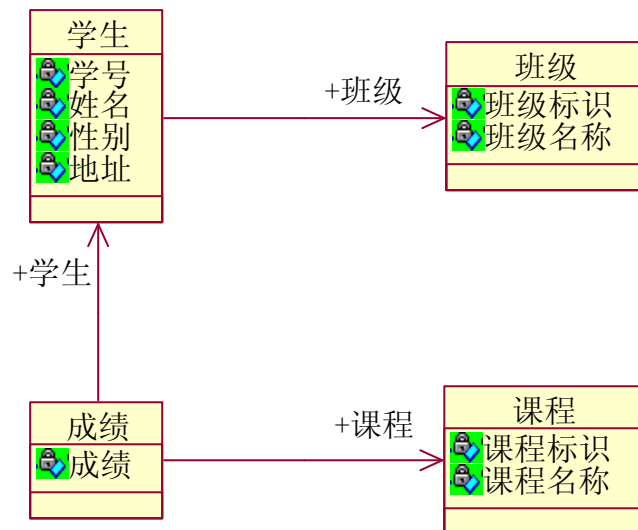
班级：班级代码、班级名称

课程：课程代码、课程名称

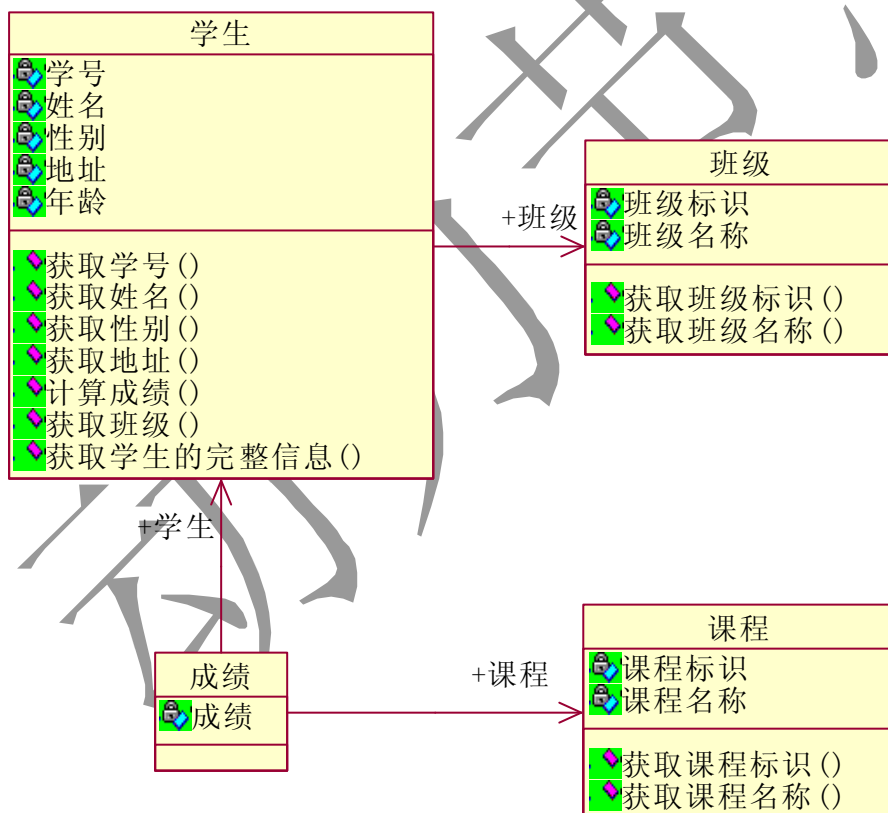
成绩：学生、课程、成绩

大家看到以上我们分析出来的都是类的属性

接下来采用简易的图形来描述一下，来描述我们的概念（来源成绩管理系统的概念，来源于领域的概念，这个领域就是成绩系统管理领域）



以上描述的是类的属性，也就是状态信息，接下来，再做进一步的细化



通过以上分析，大家应该了解：

类=属性+方法

属性来源于类的状态，而方法来源于动作

以上模型完全可以使用面向对象的语言，如 Java 来实现

1.4、类的定义

在 Java 中如何定义类？

具体格式：

```
类的修饰符 class 类名 extends 父对象名称 implements 接口名称 {  
    类体：属性和方法组成  
}
```

【示例代码】

```
public class Student {  
  
    //学号  
    int id;  
  
    //姓名  
    String name;  
  
    //性别  
    boolean sex;  
  
    //地址  
    String address;  
  
    //年龄  
    int age;  
  
}
```

以上属性称为成员变量，局部变量是在方法中定义的变量，方法的参数，方法的返回值，局部变量使用前必须初始化，而成员变量会默认初始化，初始化的值名为该类型的默认值

1.5、对象的创建和使用

必须使用 new 创建出来，才能用。

【示例代码】

```
public class OOTest01 {  
  
    public static void main(String[] args) {  
        //创建一个对象  
        Student zhangsan = new Student();  
        System.out.println("id=" + zhangsan.id);  
        System.out.println("name=" + zhangsan.name);  
        System.out.println("sex=" + zhangsan.sex);  
        System.out.println("address=" + zhangsan.address);  
        System.out.println("age=" + zhangsan.age);  
    }  
}
```

```

    }
}

class Student {

    //学号
    int id;

    //姓名
    String name;

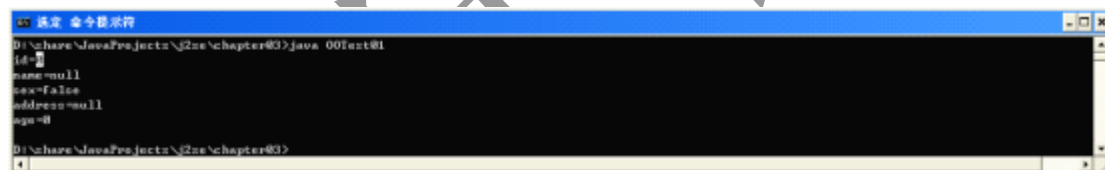
    //性别
    boolean sex;

    //地址
    String address;

    //年龄
    int age;

}

```



```

D:\share\JavaProjects\J2se\chapter03>java OOTest01
id=0
name=null
sex=false
address=null
age=0
D:\share\JavaProjects\J2se\chapter03>

```

具体默认值如下:

类型	默认值
byte	0
short	0
int	0
long	0L
char	'\u0000'
float	0.0f
double	0.0d
boolean	false
引用类型	null

对成员变量进行赋值

```

public class OOTest02 {

    public static void main(String[] args) {
        //创建一个对象
        Student zhangsan = new Student();
    }
}

```

```
        zhangsan.id = 1001;
        zhangsan.name = "张三";
        zhangsan.sex = true;
        zhangsan.address = "北京";
        zhangsan.age = 20;

        System.out.println("id=" + zhangsan.id);
        System.out.println("name=" + zhangsan.name);
        System.out.println("sex=" + zhangsan.sex);
        System.out.println("address=" + zhangsan.address);
        System.out.println("age=" + zhangsan.age);
    }
}

class Student {

    //学号
    int id;

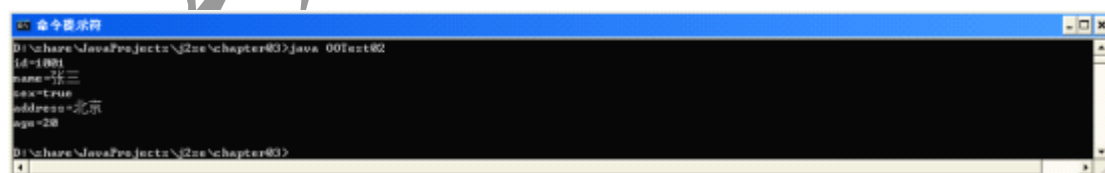
    //姓名
    String name;

    //性别
    boolean sex;

    //地址
    String address;

    //年龄
    int age;

}
```



```
命令提示符
D:\share\JavaProjects\J2se\chapter03>java 00Test02
id=1001
name=张三
sex=true
address=北京
age=20
D:\share\JavaProjects\J2se\chapter03>
```

一个类可以创建 N 个对象，成员变量只属于当前的对象（只属于对象，不属于类），只有通过对象才可以访问**成员变量**，通过类不能直接访问成员变量

以上程序存在缺点，年龄可以赋值为负数，怎么控制不能赋值为负数？

1.6、面向对象的封装性

控制对年龄的修改，年龄只能为大于等于 0 并且小于等于 120 的值

```
public class OOTest03 {

    public static void main(String[] args) {
        //创建一个对象
        Student zhangsan = new Student();
        /*
        zhangsan.id = 1001;
        zhangsan.name = "张三";
        zhangsan.sex = true;
        zhangsan.address = "北京";
        zhangsan.age = 20;
        */
        zhangsan.setId(1001);
        zhangsan.setName("张三");
        zhangsan.setSex(true);
        zhangsan.setAddress("北京");
        zhangsan.setAge(-20);

        System.out.println("id=" + zhangsan.id);
        System.out.println("name=" + zhangsan.name);
        System.out.println("sex=" + zhangsan.sex);
        System.out.println("address=" + zhangsan.address);
        System.out.println("age=" + zhangsan.age);
    }
}

class Student {

    //学号
    int id;

    //姓名
    String name;

    //性别
    boolean sex;

    //地址
    String address;
```

```
//年龄
int age;

//设置学号
public void setId(int studentId) {
    id = studentId;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}

public boolean getSex() {
    return sex;
}

public void setAddress(String studentAddress) {
    address = studentAddress;
}

public String getAddress() {
    return address;
}

public void setAge(int studentAge) {
    if (studentAge >=0 && studentAge <=120) {
        age = studentAge;
    }
}
```



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

从上面的示例,采用方法可以控制赋值的过程,加入了对年龄的检查,避免了直接操纵 student 属性,这就是封装,封装其实就是**封装属性**,让外界知道这个类的状态越少越好。以上仍然不完善,采用属性仍然可以赋值,如果屏蔽掉属性的赋值,只采用方法赋值,如下:

```
public class OOTest04 {  
  
    public static void main(String[] args) {  
        //创建一个对象  
        Student zhangsan = new Student();  
        zhangsan.id = 1001;  
        zhangsan.name = "张三";  
        zhangsan.sex = true;  
        zhangsan.address = "北京";  
        zhangsan.age = 20;  
  
        /*  
        zhangsan.setId(1001);  
        zhangsan.setName("张三");  
        zhangsan.setSex(true);  
        zhangsan.setAddress("北京");  
        zhangsan.setAge(20);  
        */  
  
        /*  
        System.out.println("id=" + zhangsan.id);  
        System.out.println("name=" + zhangsan.name);  
        System.out.println("sex=" + zhangsan.sex);  
        System.out.println("address=" + zhangsan.address);  
        System.out.println("age=" + zhangsan.age);  
        */  
  
        System.out.println("id=" + zhangsan.getId());  
        System.out.println("name=" + zhangsan.getName());  
        System.out.println("sex=" + zhangsan.getSex());  
        System.out.println("address=" + zhangsan.getAddress());  
        System.out.println("age=" + zhangsan.getAge());  
    }  
}  
  
class Student {
```

```
//学号
private int id;

//姓名
private String name;

//性别
private boolean sex;

//地址
private String address;

//年龄
private int age;

//设置学号
public void setId(int studentId) {
    id = studentId;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String studentName){
    name = studentName;
}

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}

public boolean getSex() {
    return sex;
}

public void setAddress(String studentAddress) {
    address = studentAddress;
}
```

```
}

public String getAddress() {
    return address;
}

public void setAge(int studentAge) {
    if (studentAge >=0 && studentAge <=120) {
        age = studentAge;
    }
}

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

以上采用 `private` 来声明成员变量，那么此时的成员变量只属于 `Student`，外界无法访问，这样就封装了我们的属性，那么属性只能通过方法访问，通过方法我们就可以控制对内部状态的读取权利。

封装属性，暴露方法(行为)

1.7、构造函数（构造方法，构造器，Constructor）

构造方法主要用来创建类的实例化对象，可以完成创建实例化对象的初始化工作，声明格式：构造方法修饰词列表 **类名**（方法参数列表）

构造方法修饰词列表： `public`、`protected`、`private`

类的构造方法和普通方法一样可以进行重载

构造方法具有的特点：

- 构造方法名称必须与类名一致
- 构造方法不具有任何返回值类型，即没有返回值，关键字 `void` 也不能加入，加入后就不是构造方法了，就成了普通的方法了
- 任何类都有构造方法，如果没有显示的定义，则系统会为该类定义一个默认的构造器，这个构造器不含任何参数，如果显示了定义了构造器，系统就不会创建默认的不含参数的构造器了。

【代码示例】，默认构造方法（也就是无参构造方法）

```
public class ConstructorTest01 {

    public static void main(String[] args) {
        //创建一个对象
        Student zhangsan = new Student();
        zhangsan.setId(1001);
        zhangsan.setName("张三");
    }
}
```

```
        zhangsan.setSex(true);
        zhangsan.setAddress("北京");
        zhangsan.setAge(20);

        System.out.println("id=" + zhangsan.getId());
        System.out.println("name=" + zhangsan.getName());
        System.out.println("sex=" + zhangsan.getSex());
        System.out.println("address=" + zhangsan.getAddress());
        System.out.println("age=" + zhangsan.getAge());
    }
}
```

```
class Student {
```

```
    //学号
```

```
    private int id;
```

```
    //姓名
```

```
    private String name;
```

```
    //性别
```

```
    private boolean sex;
```

```
    //地址
```

```
    private String address;
```

```
    //年龄
```

```
    private int age;
```

```
    //默认构造方法
```

```
    public Student() {
```

```
        //在创建对象的时候会执行该构造方法
```

```
        //在创建对象的时候，如果需要做些事情，可以放在构造方法中
```

```
        System.out.println("-----Student-----");
```

```
    }
```

```
    //设置学号
```

```
    public void setId(int studentId) {
```

```
        id = studentId;
```

```
    }
```

```
    //读取学号
```

```
    public int getId() {
```

```
        return id;
    }

    public void setName(String studentName) {
        name = studentName;
    }

    public String getName() {
        return name;
    }

    public void setSex(boolean studentSex) {
        sex = studentSex;
    }

    public boolean getSex() {
        return sex;
    }

    public void setAddress(String studentAddress) {
        address = studentAddress;
    }

    public String getAddress() {
        return address;
    }

    public void setAge(int studentAge) {
        if (studentAge >= 0 && studentAge <= 120) {
            age = studentAge;
        }
    }

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

【代码示例】，带参数的构造方法

```
public class ConstructorTest02 {

    public static void main(String[] args) {

        //调用带参数的构造方法对成员变量进行赋值
        Student zhangsan = new Student(1001, "张三", true, "北京", 20);
    }
}
```

```
        System.out.println("id=" + zhangsan.getId());
        System.out.println("name=" + zhangsan.getName());
        System.out.println("sex=" + zhangsan.getSex());
        System.out.println("address=" + zhangsan.getAddress());
        System.out.println("age=" + zhangsan.getAge());
    }
}

class Student {

    //学号
    private int id;

    //姓名
    private String name;

    //性别
    private boolean sex;

    //地址
    private String address;

    //年龄
    private int age;

    public Student(int studentId, String studentName, boolean studentSex, String studentAddress,
int studentAge) {
        id = studentId;
        name = studentName;
        sex = studentSex;
        address = studentAddress;
        age = studentAge;
    }

    //设置学号
    public void setId(int studentId) {
        id = studentId;
    }

    //读取学号
    public int getId() {
        return id;
    }
}
```

```
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}

public boolean getSex() {
    return sex;
}

public void setAddress(String studentAddress) {
    address = studentAddress;
}

public String getAddress() {
    return address;
}

public void setAge(int studentAge) {
    if (studentAge >=0 && studentAge <=120) {
        age = studentAge;
    }
}

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

【代码示例】，进一步理解默认构造方法

```
public class ConstructorTest03 {

    public static void main(String[] args) {

        //调用带参数的构造方法对成员变量进行赋值
        //Student zhangsan = new Student(1001,"张三", true,"北京", 20);
    }
}
```

```
Student zhangsan = new Student();
zhangsan.setId(1001);
zhangsan.setName("张三");
zhangsan.setSex(true);
zhangsan.setAddress("北京");
zhangsan.setAge(20);

System.out.println("id=" + zhangsan.getId());
System.out.println("name=" + zhangsan.getName());
System.out.println("sex=" + zhangsan.getSex());
System.out.println("address=" + zhangsan.getAddress());
System.out.println("age=" + zhangsan.getAge());
}
}

class Student {

    //学号
    private int id;

    //姓名
    private String name;

    //性别
    private boolean sex;

    //地址
    private String address;

    //年龄
    private int age;

    public Student(int studentId, String studentName, boolean studentSex, String studentAddress,
int studentAge) {
        id = studentId;
        name = studentName;
        sex = studentSex;
        address = studentAddress;
        age = studentAge;
    }

    //设置学号
    public void setId(int studentId) {
```



```
        id = studentId;
    }

    //读取学号
    public int getId() {
        return id;
    }

    public void setName(String studentName) {
        name = studentName;
    }

    public String getName() {
        return name;
    }

    public void setSex(boolean studentSex) {
        sex = studentSex;
    }

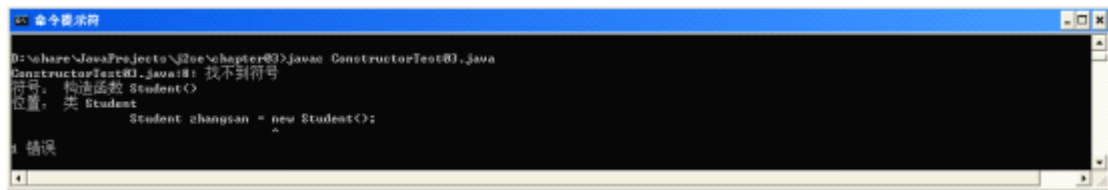
    public boolean getSex() {
        return sex;
    }

    public void setAddress(String studentAddress) {
        address = studentAddress;
    }

    public String getAddress() {
        return address;
    }

    public void setAge(int studentAge) {
        if (studentAge >=0 && studentAge <=120) {
            age = studentAge;
        }
    }

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



出现错误，主要原因是默认构造函数不能找到，手动建立的构造方法，将会把默认的构造方法覆盖掉，所以在这种情况下必须显示的建立默认构造方法

```
public class ConstructorTest04 {

    public static void main(String[] args) {

        //调用带参数的构造方法对成员变量进行赋值
        //Student zhangsan = new Student(1001,"张三", true, "北京", 20);

        Student zhangsan = new Student();
        zhangsan.setId(1001);
        zhangsan.setName("张三");
        zhangsan.setSex(true);
        zhangsan.setAddress("北京");
        zhangsan.setAge(20);

        System.out.println("id=" + zhangsan.getId());
        System.out.println("name=" + zhangsan.getName());
        System.out.println("sex=" + zhangsan.getSex());
        System.out.println("address=" + zhangsan.getAddress());
        System.out.println("age=" + zhangsan.getAge());
    }
}

class Student {

    //学号
    private int id;

    //姓名
    private String name;

    //性别
    private boolean sex;

    //地址
    private String address;
```

```
//年龄
private int age;

//手动提供默认的构造函数
public Student() {}

public Student(int studentId, String studentName, boolean studentSex, String studentAddress,
int studentAge) {
    id = studentId;
    name = studentName;
    sex = studentSex;
    address = studentAddress;
    age = studentAge;
}

//设置学号
public void setId(int studentId) {
    id = studentId;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}

public boolean getSex() {
    return sex;
}

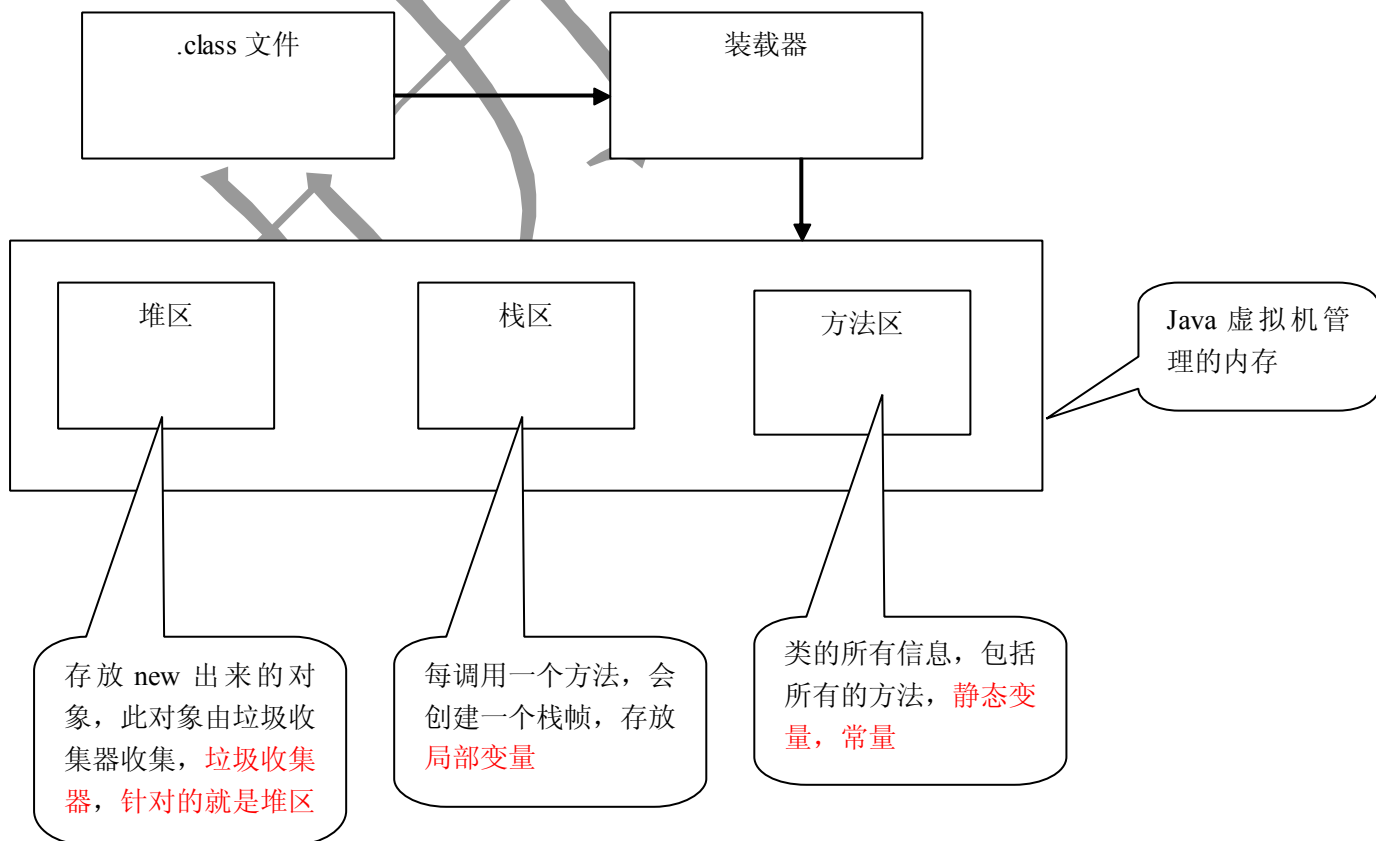
public void setAddress(String studentAddress) {
    address = studentAddress;
}
```

```
public String getAddress() {  
    return address;  
}  
  
public void setAge(int studentAge) {  
    if (studentAge >=0 && studentAge <=120) {  
        age = studentAge;  
    }  
}  
  
public int getAge() {  
    return age;  
}  
}
```

以上示例执行正确，因为加入了默认的构造方法，同时也演示了构造方法的重载，构造方法重载原则和普通方法是一样的。

1.8、对象和引用

1.8.1、Java 内存的主要划分

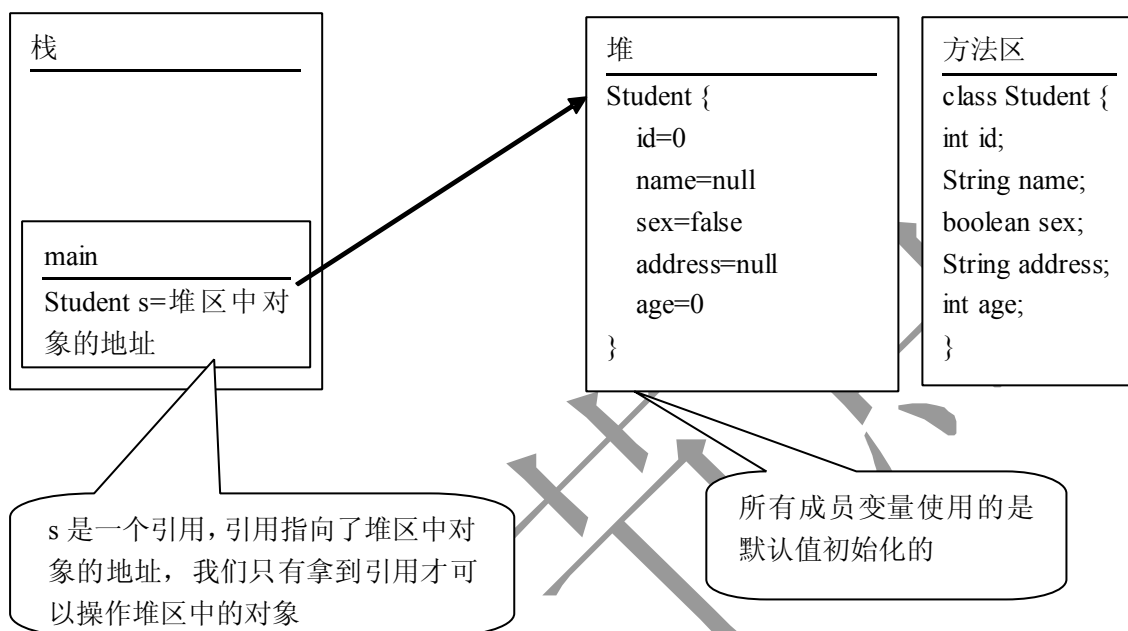


1.8.2、OOTest02.java 内存的表示

- 第一步，执行 main 方法，将 main 方法压入栈，然后 new Student 对象

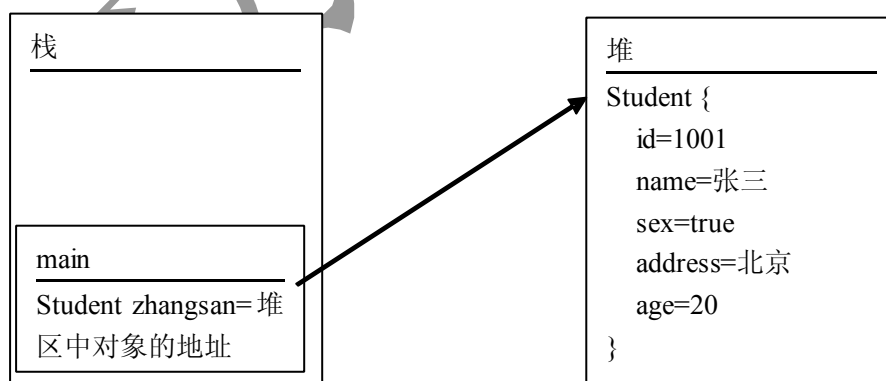
```
//创建一个对象
```

```
Student zhangsan = new Student();
```



- 第二步，对 student 赋值

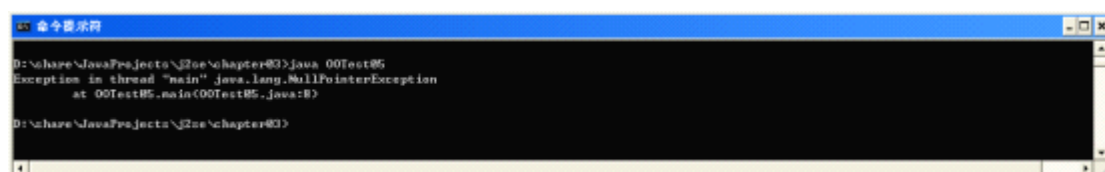
```
zhangsan.id = 1001;  
zhangsan.name = "张三";  
zhangsan.sex = true;  
zhangsan.address = "北京";  
zhangsan.age = 20;
```



1.8.3、当不使用 new 关键字时，出现的问题

```
public class OOTest05 {  
  
    public static void main(String[] args) {  
        //创建一个对象  
        //Student zhangsan = new Student();  
  
        Student zhangsan = null;  
        zhangsan.id = 1001;  
        zhangsan.name = "张三";  
        zhangsan.sex = true;  
        zhangsan.address = "北京";  
        zhangsan.age = 20;  
  
        System.out.println("id=" + zhangsan.id);  
        System.out.println("name=" + zhangsan.name);  
        System.out.println("sex=" + zhangsan.sex);  
        System.out.println("address=" + zhangsan.address);  
        System.out.println("age=" + zhangsan.age);  
    }  
}
```

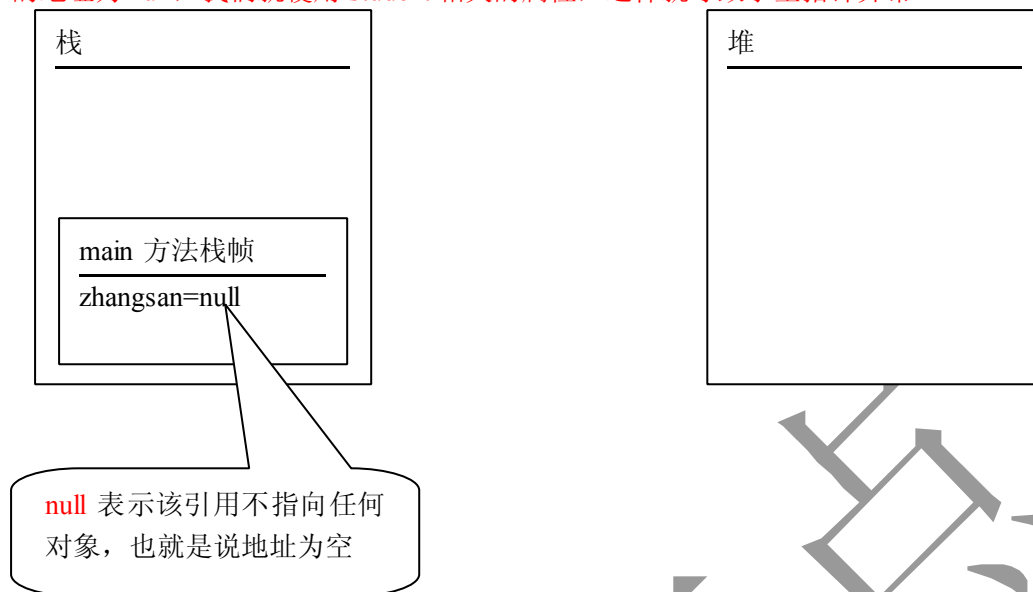
```
class Student {  
  
    //学号  
    int id;  
  
    //姓名  
    String name;  
  
    //性别  
    boolean sex;  
  
    //地址  
    String address;  
  
    //年龄  
    int age;  
}
```



The screenshot shows a Java IDE console window with the following text:

```
D:\share\JavaProjects\J2ee\chapter03>java OOTest05  
Exception in thread "main" java.lang.NullPointerException  
    at OOTest05.main(OOTest05.java:8)  
D:\share\JavaProjects\J2ee\chapter03>
```

抛出了空指针异常，为什么会抛出此一样，因为 zhangsan 没有指向任何对象，所以 zhangsan 的地址为 null，我们就使用 student 相关的属性，这样就导致了空指针异常



1.8.4、参数传递

- 值传递

```
public class OOTest06 {
```

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

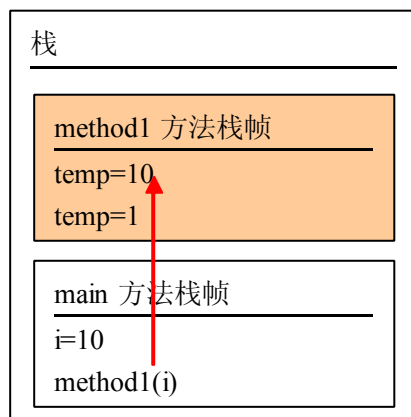
```
        int i=10;
        method1(i);
        System.out.println(i);
    }
```

```
    public static void method1(int temp) {
```

```
        temp = 1;
```

```
    }
```

```
}
```



以上代码执行过程，首先在栈中创建 main 方法栈帧，初始化 i 的值为 10，将此栈帧压入栈，接着调用 method1（）方法，通常是在栈区创建新的栈帧，并赋值 temp 为 1，压入栈中，**两栈帧都是独立的，他们之间的值是不能互访的**。然后 method1（）方法栈帧弹出，接下来执行输出 i 值的语句，因为传递的是值，所以不会改变 i 的值，**输出结果 i 的值仍然为 10**，最后将 main 方法栈帧弹出，main 方法执行完毕。

结论：只要是基本类型，传递过去的都是值

● 引用传递（传址）

```
public class OOTest07 {

    public static void main(String[] args) {
        //创建一个对象
        Student student = new Student();
        student.id = 1001;
        student.name = "张三";
        student.sex = true;
        student.address = "北京";
        student.age = 20;

        method1(student);

        System.out.println("id=" + student.id);
        System.out.println("name=" + student.name);
        System.out.println("sex=" + student.sex);
        System.out.println("address=" + student.address);
        System.out.println("age=" + student.age);
    }

    public static void method1(Student temp) {
        temp.name="李四";
    }
}

class Student {

    //学号
    int id;

    //姓名
    String name;

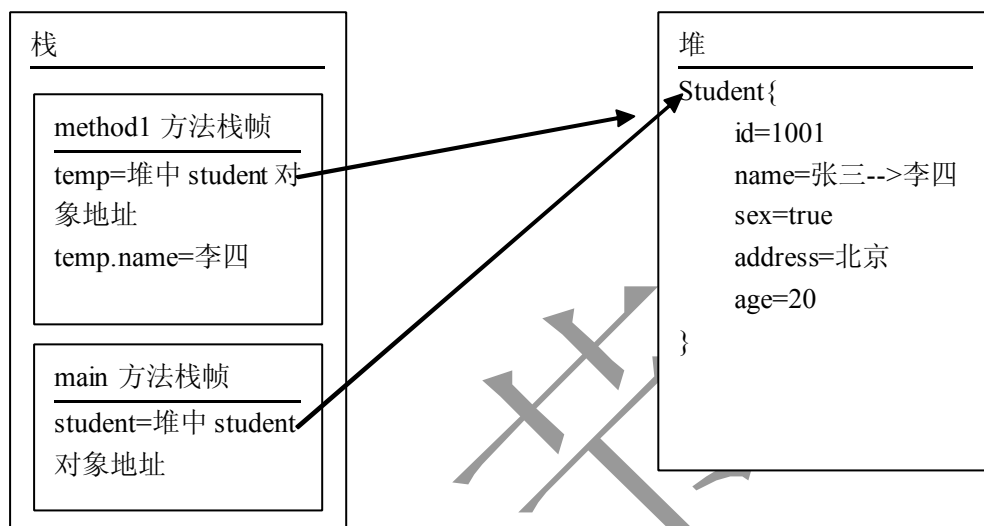
    //性别
    boolean sex;
```



```
//地址
String address;

//年龄
int age;

}
```



以上执行流程为，

- 1、执行 main 方法时，会在栈中创建 main 方法的栈帧，将局部变量 student 引用保存到栈中
- 2、将 Student 对象放到堆区中，并赋值
- 3、调用 method1 方法，创建 method1 方法栈帧
- 4、将局部变量 student 引用（也就是 student 对象的地址），传递到 method1 方法中，并赋值给 temp
- 5、此时 temp 和 student 引用指向的都是一个对象
- 6、当 method1 方法中将 name 属性改为“李四”时，改的是堆中的属性
- 7、所以会影响输出结果，最后的输出结果为：李四
- 8、main 方法执行结束，此时的 student 对象还在堆中，我们无法使用了，那么他就变成了“垃圾对象”，java 的垃圾收集器会在某个时刻，将其清除

以上就是址传递，也就是引用的传递

结论：除了基本类型外都是值传递

1.9、this 关键字

this 关键字指的是当前调用的对象，如果有 100 个对象，将有 100 个 this 对象指向各个对象
this 关键字可以使用在：

- 当局部变量和成员变量重名的时候可以使用 this 指定调用成员变量
- 通过 this 调用另一个构造方法

需要注意：**this** 只能用在构造函数和成员方法内部，还可以应用在成员变量的声明上，**static** 标识的方法里是不能使用 **this** 的，关于 **static** 以后再讲

1.9.1、当局部变量和成员变量重名的时候可以使用 **this** 指定调用成员变量

```
public class OOTest08 {  
  
    public static void main(String[] args) {  
  
        Student zhangsan = new Student();  
        zhangsan.setId(1001);  
        zhangsan.setName("张三");  
        zhangsan.setSex(true);  
        zhangsan.setAddress("北京");  
        zhangsan.setAge(20);  
  
        System.out.println("id=" + zhangsan.getId());  
        System.out.println("name=" + zhangsan.getName());  
        System.out.println("sex=" + zhangsan.getSex());  
        System.out.println("address=" + zhangsan.getAddress());  
        System.out.println("age=" + zhangsan.getAge());  
    }  
}  
  
class Student {  
    //学号  
    private int id;  
  
    //姓名  
    private String name;  
  
    //性别  
    private boolean sex;  
  
    //地址  
    private String address;  
  
    //年龄  
    private int age;
```

```
//设置学号
public void setId(int id) {
    id = id;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String name) {
    name = name;
}

public String getName() {
    return name;
}

public void setSex(boolean sex) {
    sex = sex;
}

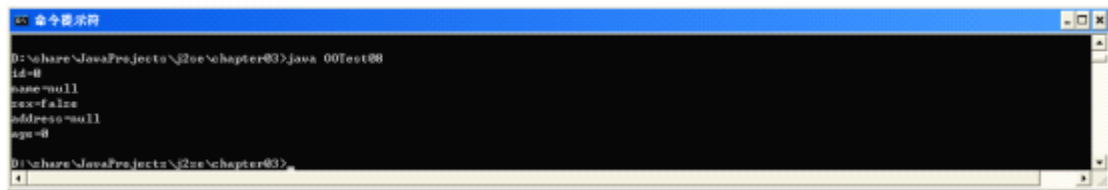
public boolean getSex() {
    return sex;
}

public void setAddress(String address) {
    address = address;
}

public String getAddress() {
    return address;
}

public void setAge(int age) {
    age = age;
}

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



```
D:\chare\JavaProjects\2ee\chapter03>java 00Test08
id=0
name=null
sex=false
address=null
age=0
D:\chare\JavaProjects\2ee\chapter03>
```

输出错误，输出的都是成员变量的默认值，赋值没有起作用，这是为什么？

如：setId(int id) 方法中，我们的赋值语句为 id=id,这样写 Java 无法知道是向成员变量 id 赋值，它遵循“**谁近谁优先**”，当然局部变量优先了，所以会忽略成员变量

【代码示例】，修正以上代码

```
public class OOTest09 {

    public static void main(String[] args) {

        Student zhangsan = new Student();
        zhangsan.setId(1001);
        zhangsan.setName("张三");
        zhangsan.setSex(true);
        zhangsan.setAddress("北京");
        zhangsan.setAge(20);

        System.out.println("id=" + zhangsan.getId());
        System.out.println("name=" + zhangsan.getName());
        System.out.println("sex=" + zhangsan.getSex());
        System.out.println("address=" + zhangsan.getAddress());
        System.out.println("age=" + zhangsan.getAge());
    }
}

class Student {

    //学号
    private int id;

    //姓名
    private String name;

    //性别
    private boolean sex;

    //地址
    private String address;

    //年龄
```

```
private int age;

//设置学号
public void setId(int id) {
    this.id = id;
}

//读取学号
public int getId() {
    return id;
}

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

public String getName() {
    return name;
}

public void setSex(boolean sex) {
    this.sex = sex;
}

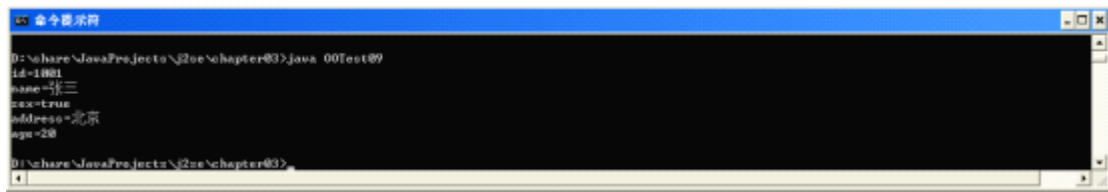
public boolean getSex() {
    return sex;
}

public void setAddress(String address) {
    this.address = address;
}

public String getAddress() {
    return address;
}

public void setAge(int age) {
    this.age = age;
}

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



通过 `this` 可以取得当前调用对象，采用 `this` 明确指定了成员变量的名称，所以就不会赋值到局部变量，了解狭义和广义上的 `javabean` 的概念

1.9.2、通过 `this` 调用另一个构造方法

- 通过 `new` 调用当前对象的构造函数（这样是不对的）

```
public class OOTest10 {  
  
    public static void main(String[] args) {  
  
        //Student zhangsan = new Student(1001, "张三", true, "北京", 20);  
  
        //Student zhangsan = new Student(1001, "张三");  
        Student zhangsan = new Student(1002, "李四");  
  
        System.out.println("id=" + zhangsan.getId());  
        System.out.println("name=" + zhangsan.getName());  
        System.out.println("sex=" + zhangsan.getSex());  
        System.out.println("address=" + zhangsan.getAddress());  
        System.out.println("age=" + zhangsan.getAge());  
    }  
}  
  
class Student {  
  
    //学号  
    private int id;  
  
    //姓名  
    private String name;  
  
    //性别  
    private boolean sex;  
  
    //地址  
    private String address;  
  
    //年龄
```

```
private int age;

public Student(int id, String name) {
    /*
    this.id = id;
    this.name = name;
    this.sex=true;
    this.address="北京";
    this.age=20;
    */
    //调用构造函数,部分值采用默认值
    new Student(id, name, true, "北京", 20);
}

public Student(int id, String name, boolean sex, String address, int age) {
    this.id = id;
    this.name = name;
    this.sex = sex;
    this.address = address;
    this.age = age;
}

//设置学号
public void setId(int studentId) {
    id = studentId;
}

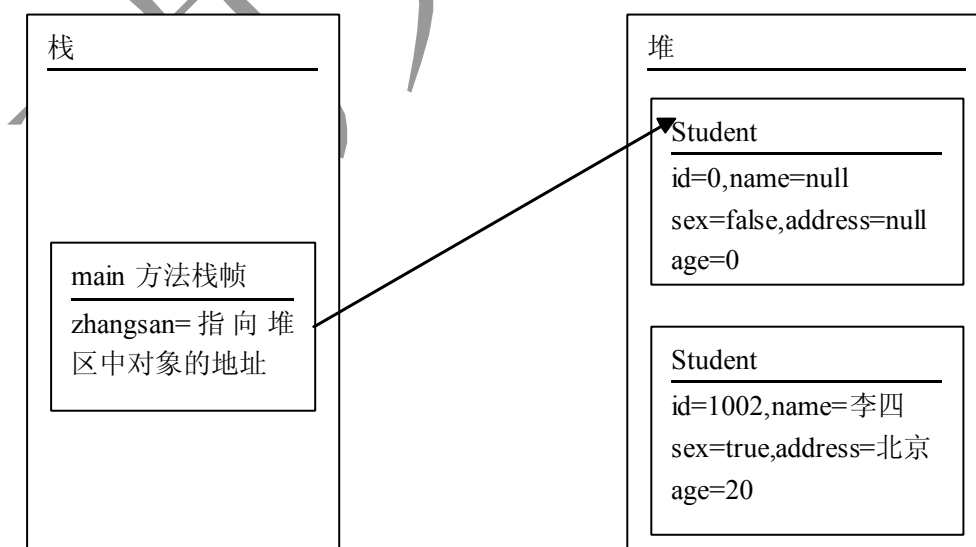
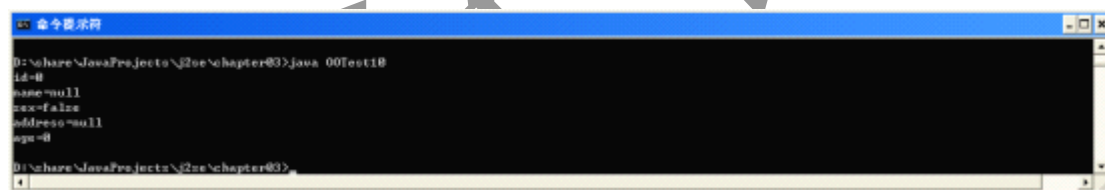
//读取学号
public int getId() {
    return id;
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}
```

```
public boolean getSex() {  
    return sex;  
}  
  
public void setAddress(String studentAddress) {  
    address = studentAddress;  
}  
  
public String getAddress() {  
    return address;  
}  
  
public void setAge(int studentAge) {  
    if (studentAge >=0 && studentAge <=120) {  
        age = studentAge;  
    }  
}  
  
public int getAge() {  
    return age;  
}  
}
```



- 通过 this 调用当前对象的构造函数

```
public class OOTest11 {
```



```
public static void main(String[] args) {  
  
    Student zhangsan = new Student(1002, "李四");  
  
    System.out.println("id=" + zhangsan.getId());  
    System.out.println("name=" + zhangsan.getName());  
    System.out.println("sex=" + zhangsan.getSex());  
    System.out.println("address=" + zhangsan.getAddress());  
    System.out.println("age=" + zhangsan.getAge());  
  
}  
}
```

```
class Student {
```

```
    //学号
```

```
    private int id;
```

```
    //姓名
```

```
    private String name;
```

```
    //性别
```

```
    private boolean sex;
```

```
    //地址
```

```
    private String address;
```

```
    //年龄
```

```
    private int age;
```

```
    public Student(int id, String name) {
```

```
        /*
```

```
        this.id = id;
```

```
        this.name = name;
```

```
        this.sex=true;
```

```
        this.address="北京";
```

```
        this.age=20;
```

```
        */
```

```
        //调用构造函数,部分值采用默认值
```

```
        //new Student(id, name, true, "北京", 20);
```

```
        //采用 this 调用当前对象对象的构造函数，不能采用 new,以上调用时错误的  
        this(id, name, true, "北京", 20);
```

```
}

public Student(int id, String name, boolean sex, String address, int age) {
    this.id = id;
    this.name = name;
    this.sex = sex;
    this.address = address;
    this.age = age;
}

//设置学号
public void setId(int studentId) {
    id = studentId;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
    return name;
}

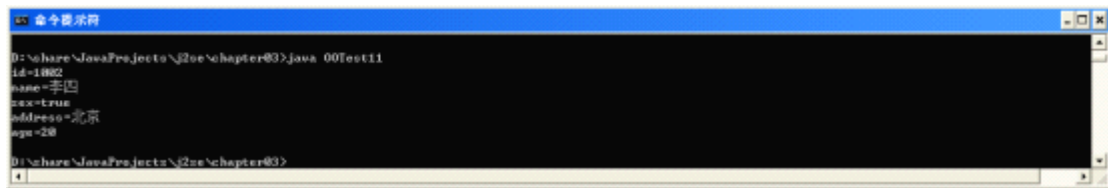
public void setSex(boolean studentSex) {
    sex = studentSex;
}

public boolean getSex() {
    return sex;
}

public void setAddress(String studentAddress) {
    address = studentAddress;
}

public String getAddress() {
    return address;
}
```

```
public void setAge(int studentAge) {  
    if (studentAge >=0 && studentAge <=120) {  
        age = studentAge;  
    }  
}  
  
public int getAge() {  
    return age;  
}  
}
```



```
D:\share\JavaProjects\200\chapter03>java 00Test11  
id=1001  
name=张三  
sex=true  
address=北京  
age=20  
D:\share\JavaProjects\200\chapter03>
```

以上输出正确

1.10、 static 关键字

static 修饰符可以修饰：变量、方法和代码块

- 用 static 修饰的变量和方法，可以采用类名直接访问
- 用 static 声明的代码块为静态代码块，JVM 第一次使用类的时候，会执行静态代码块中的内容

1.10.1、采用静态变量实现累加器

【代码示例】

```
public class StaticTest01 {  
  
    public static void main(String[] args) {  
        Student student1 = new Student(1001, "张三", true, "北京", 20);  
        Student student2 = new Student(1002, "李四", true, "上海", 30);  
  
        System.out.println(student1.getCount());  
        System.out.println(student2.getCount());  
    }  
}  
  
class Student {  
  
    //学号  
    private int id;
```

```
//姓名
private String name;

//性别
private boolean sex;

//地址
private String address;

//年龄
private int age;

//计数器，计算 student 的创建个数
private int count;

public Student(int id, String name, boolean sex, String address, int age) {
    count++;
    this.id = id;
    this.name = name;
    this.sex = sex;
    this.address = address;
    this.age = age;
}

//返回计数器
public int getCount() {
    return count;
}

//设置学号
public void setId(int studentId) {
    id = studentId;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
```

```
        return name;
    }

    public void setSex(boolean studentSex) {
        sex = studentSex;
    }

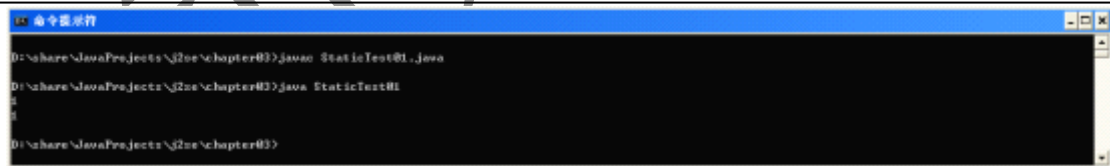
    public boolean getSex() {
        return sex;
    }

    public void setAddress(String studentAddress) {
        address = studentAddress;
    }

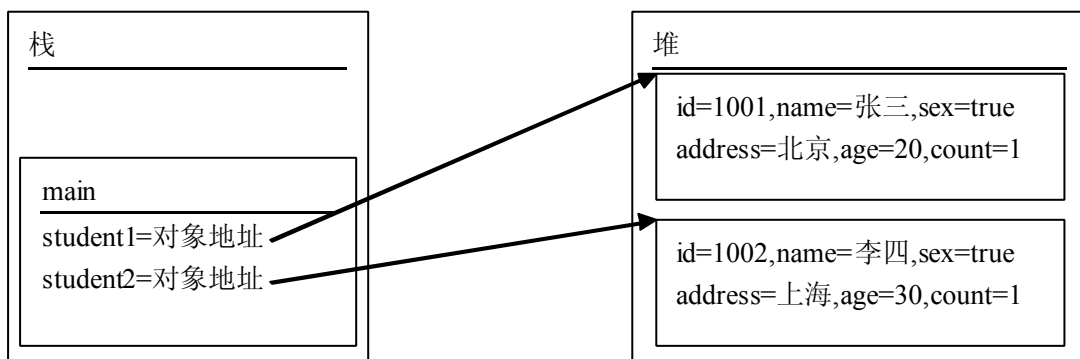
    public String getAddress() {
        return address;
    }

    public void setAge(int studentAge) {
        if (studentAge >=0 && studentAge <=120) {
            age = studentAge;
        }
    }

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



以上输出不正确，因为采用的是成员变量作为累加器，成员变量只属于某一个对象，如果多个对象那么成员变量会有多个拷贝，对象中的成员变量都是私有的，**某一个对象中的成员变量的改变不会影响到另一个对象中的成员变量的改变**



【代码示例】

```
public class StaticTest02 {

    public static void main(String[] args) {
        Student student1 = new Student(1001, "张三", true, "北京", 20);
        Student student2 = new Student(1002, "李四", true, "上海", 30);

        System.out.println(student1.getCount());
        System.out.println(student2.getCount());
    }
}

class Student {

    //学号
    private int id;

    //姓名
    private String name;

    //性别
    private boolean sex;

    //地址
    private String address;

    //年龄
    private int age;

    //计数器，计算 student 的创建个数
    private static int count;

    public Student(int id, String name, boolean sex, String address, int age) {
        count++;
        this.id = id;
        this.name = name;
        this.sex = sex;
        this.address = address;
        this.age = age;
    }

    //返回计数器
    public int getCount() {
        return count;
    }
}
```

```
}

//设置学号
public void setId(int studentId) {
    id = studentId;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}

public boolean getSex() {
    return sex;
}

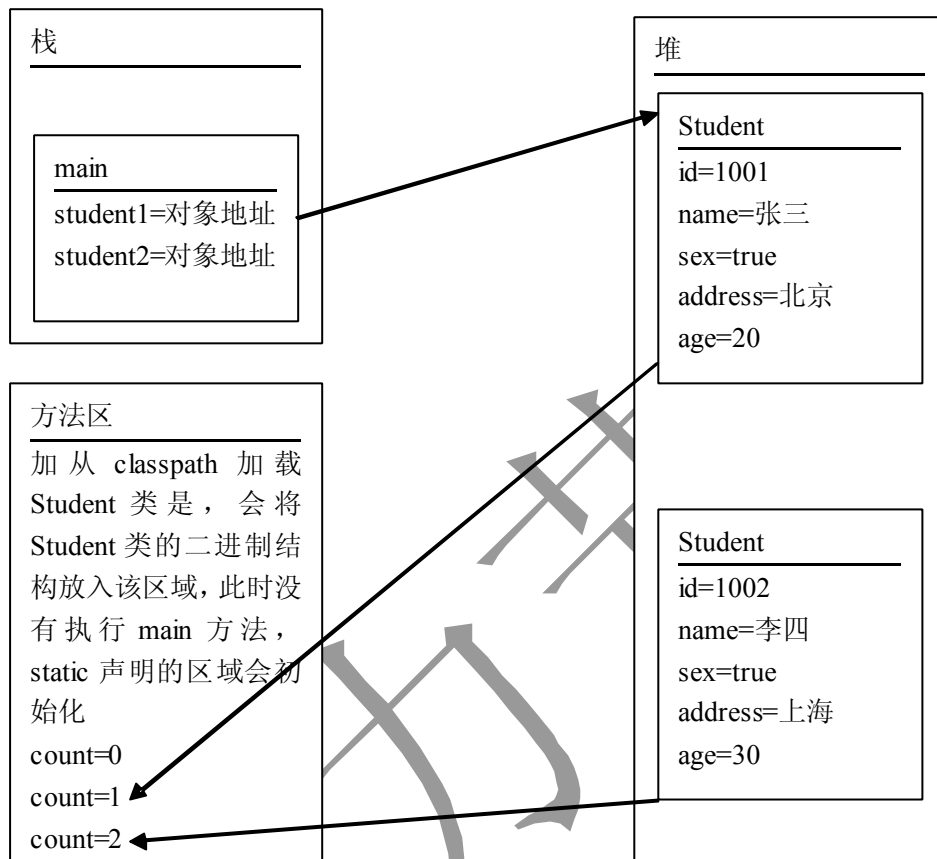
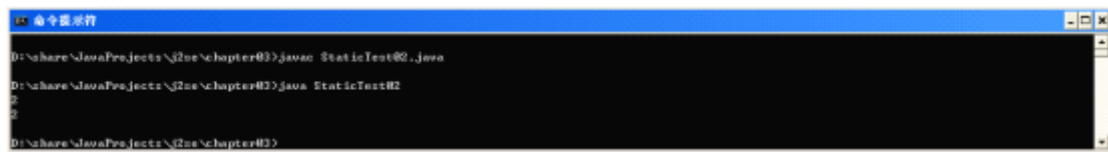
public void setAddress(String studentAddress) {
    address = studentAddress;
}

public String getAddress() {
    return address;
}

public void setAge(int studentAge) {
    if (studentAge >=0 && studentAge <=120) {
        age = studentAge;
    }
}

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

```
}  
}
```



static 声明的变量，所有通过该类 new 出的对象，都可以共享，通过该对象都可以直接访问 static 声明的变量，也可以采用类直接访问，所以我们也称其为类变量

【代码示例】，采用类直接访问 count

```
public class StaticTest03 {  
  
    public static void main(String[] args) {  
        Student student1 = new Student(1001, "张三", true, "北京", 20);  
        Student student2 = new Student(1002, "李四", true, "上海", 30);  
  
        //System.out.println(student1.getCount());  
        //System.out.println(student2.getCount());  
        System.out.println(Student.count);  
    }  
}
```



```
class Student {  
  
    //学号  
    private int id;  
  
    //姓名  
    private String name;  
  
    //性别  
    private boolean sex;  
  
    //地址  
    private String address;  
  
    //年龄  
    private int age;  
  
    //计数器，计算 student 的创建个数  
    static int count;  
  
    public Student(int id, String name, boolean sex, String address, int age) {  
        count++;  
        this.id = id;  
        this.name = name;  
        this.sex = sex;  
        this.address = address;  
        this.age = age;  
    }  
  
    //返回计数器  
    public int getCount() {  
        return count;  
    }  
  
    //设置学号  
    public void setId(int studentId) {  
        id = studentId;  
    }  
  
    //读取学号  
    public int getId() {  
        return id;  
    }  
}
```

```
public void setName(String studentName) {  
    name = studentName;  
}  
  
public String getName() {  
    return name;  
}  
  
public void setSex(boolean studentSex) {  
    sex = studentSex;  
}  
  
public boolean getSex() {  
    return sex;  
}  
  
public void setAddress(String studentAddress) {  
    address = studentAddress;  
}  
  
public String getAddress() {  
    return address;  
}  
  
public void setAge(int studentAge) {  
    if (studentAge >=0 && studentAge <=120) {  
        age = studentAge;  
    }  
}  
  
public int getAge() {  
    return age;  
}  
}
```

通过以上分析，**static** 声明的变量会放到方法区中，**static** 声明的变量只初始化一次，加在类的时候初始化，如果多个静态变量，会按照静态变量在类中的顺序进行初始化。

1.10.2、静态方法中访问实例变量、实例方法或 **this** 关键字

静态方法中是不能直接调用实例变量、实例方法和使用 **this** 的，也就是说和实例相关的他都不能直接调用

【示例代码】，在静态区域中调用实例方法

```
public class StaticTest04 {
```

```
public static void main(String[] args) {  
  
    //在静态区域中不能直接调用成员方法  
    //因为静态方法的执行不需要 new 对象  
    //而成员方法必须 new 对象后才可以执行  
    //所以执行静态方法的时候，对象还没有创建  
    //所以无法执行成员方法  
    method1();  
}  
  
public void method1() {  
    System.out.println("method1");  
}  
}
```

【代码示例】，改善以上程序，在静态区域中成功调用实例方法

```
public class StaticTest05 {  
  
    public static void main(String[] args) {  
        StaticTest05 staticTest05 = new StaticTest05();  
        //在静态区域中如果访问成员方法必须具有  
        //该成员方法对应的对象引用  
        staticTest05.method1();  
    }  
  
    public void method1() {  
        System.out.println("method1");  
    }  
}
```

【代码示例】，改善以上程序，在静态区域中可以直接调用静态方法

```
public class StaticTest06 {  
  
    public static void main(String[] args) {  
        //可以直接调用  
        //method1();  
        //可以采用类名+方法调用  
        StaticTest06.method1();  
    }  
  
    public static void method1() {  
        System.out.println("method1");  
    }  
}
```

【代码示例】，在静态区域中调用成员（实例）变量

```
public class StaticTest07 {  
  
    //private int age = 100;  
  
    private static int age = 100;  
  
    public static void main(String[] args) {  
  
        //在静态区域中无法直接访问成员变量  
        //System.out.println(age);  
  
        //可以采用对象的引用调用  
        //StaticTest07 staticTest07 = new StaticTest07();  
        //System.out.println(staticTest07.age);  
  
        //直接取得静态变量的值  
        //System.out.println(age);  
  
        //可以采用类名+属性名 访问  
        System.out.println(StaticTest07.age);  
    }  
}
```

【代码示例】，在静态区域中使用 this

```
public class StaticTest08 {  
  
    private int age = 100;  
  
    public static void main(String[] args) {  
  
        //因为执行 main 方法的时候，还没有该对象  
        //所以 this 也就不存在,无法使用  
        System.out.println(this.age);  
    }  
}
```

1.10.3、静态方法的初始化顺序

上面已经说到，静态变量，在类加载时就会初始化，也就是将类的字节码读取到方法区时就会初始化

【代码示例】

```
public class StaticTest09 {
```

```
private static int age = 100;

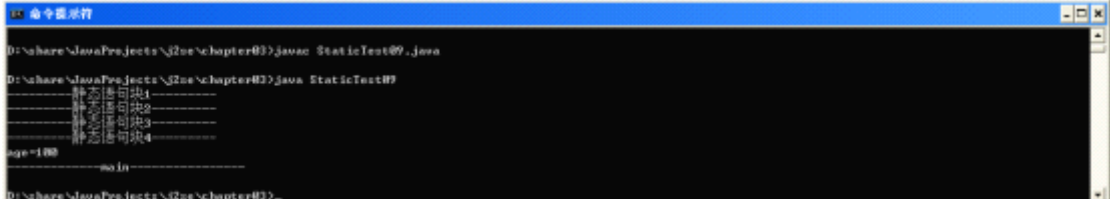
static {
    System.out.println("-----静态语句块 1-----");
}

static {
    System.out.println("-----静态语句块 2-----");
}

static {
    System.out.println("-----静态语句块 3-----");
}

static {
    System.out.println("-----静态语句块 4-----");
    System.out.println("age=" + age);
}

public static void main(String[] args) {
    System.out.println("-----main-----");
}
}
```



Static 声明的变量或 static 语句块在类加载时就会初始化，而且只初始化一次

1.10.4、解释 main 方法

解释 main 方法：

- public：表示全局所有，其实就是封装性
- static：静态的，也就是说它描述的方法只能通过类调用
- main：系统规定的
- String[] args 参数类型也是系统规定的

以上为什么是静态方法，应该很显然，这样 java 虚拟机调用起来会更方便，直接拿到类就可以调用 main 方法了，而静态的东西都在方法区中，那么你在静态方法里调用成员属性，他在方法区中无法找到，就算到堆区中查找，可能此成员属性的对象也没有创建，所以静态方法中是不能直接访问成员属性和成员方法的。

1.11、 单例模式初步

什么是设计模式：设计模式是**可以重复利用的解决方案**

设计模式的提出是在 1995 年，是由 4 人为作者提出的，称为 **GoF**,也就是“四人组”

设计模式从结构上分为三类：

- 创建型
- 结构性
- 行为型

其中最简单的设计模式就是单例了，单例这种模式，尽量少用，也有将其称为“反模式”，关于缺点以后做项目时再说

单例模式有什么好处

我们知道对象实例创建完成后，会放到堆中，如果堆中的实例过多，将会存在特别多的垃圾，这样会导致一些问题，如内存溢出等，使用单例模式后，只会创建一个实例，显著减少对象实例的个数，同时也会提高性能，因为不会频繁地创建对象，这只是他的一个好处，其他方面项目中再说。

单例模式的三要素：

- 在类体中需要具有静态的私有的本类型的变量
- 构造方法必须是私有的
- 提供一个公共的静态的入口点方法

```
public class SingletonPatternTest01 {  
  
    public static void main(String[] args) {  
        //Singleton s1 = new Singleton();  
        //Singleton s2 = new Singleton();  
        //.....  
        Singleton.getInstance().test();  
    }  
}  
  
class Singleton {  
  
    //静态私有的本类的成员变量  
    private static Singleton instance = new Singleton();  
  
    //私有的构造方法  
    private Singleton() {  
  
    }  
  
    //提供一个公共的静态的入口点方法
```

```
public static Singleton getInstance() {  
    return instance;  
}  
  
public void test() {  
    System.out.println("-----test-----");  
}  
}
```

1.12、 类的继承

面向对象的三大特性：

- 封装（封装细节）
- 继承
- 多态

- 继承是面向对象的重要概念，软件中的继承和现实中的继承概念是一样的
- 继承是实现软件可重用性的重要手段，如：A 继承 B，A 就拥有了 B 的所有特性，如现实世界中的儿子继承父亲的财产，儿子不用努力就有了财产，这就是重用性
- java 中只支持类的单继承，也就是说 A 只能继承 B，A 不能同时继承 C
- java 中的继承使用 extends 关键字，语法格式：
[修饰符] class 子类 extends 父类 {

}

【代码示例】

```
public class ExtendsTest01 {  
  
    public static void main(String[] args) {  
  
        //如果参数比较多尽量使用 setter 方法，不要使用构造函数  
        //因为参数比较多，构造函数表达的不是很明确  
        Student student = new Student();  
        student.setId(1001);  
        student.setName("张三");  
        student.setSex(true);  
        student.setAddress("北京");  
        student.setAge(20);  
        student.setClassesId(10);  
  
        System.out.println("id=" + student.getId());  
        System.out.println("name=" + student.getName());  
        System.out.println("sex=" + student.getSex());  
        System.out.println("address=" + student.getAddress());  
    }  
}
```

```
System.out.println("age=" + student.getAge());
System.out.println("classid=" + student.getClassesId());

System.out.println("");
System.out.println("");
Employee emp = new Employee();
emp.setId(1002);
emp.setName("李四");
emp.setSex(true);
emp.setAddress("上海");
emp.setAge(30);
emp.setWorkYear(10);

System.out.println("id=" + emp.getId());
System.out.println("name=" + emp.getName());
System.out.println("sex=" + emp.getSex());
System.out.println("address=" + emp.getAddress());
System.out.println("age=" + emp.getAge());
System.out.println("workYear=" + emp.getWorkYear());
}
}
```

```
class Student {

    //学号
    private int id;

    //姓名
    private String name;

    //性别
    private boolean sex;

    //地址
    private String address;

    //年龄
    private int age;

    //班级编号
    private int classesId;

    /*
```



```
public Student(int id, String name, boolean sex, String address, int age) {
    this.id = id;
    this.name = name;
    this.sex = sex;
    this.address = address;
    this.age = age;
}
*/

//设置学号
public void setId(int studentId) {
    id = studentId;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}

public boolean getSex() {
    return sex;
}

public void setAddress(String studentAddress) {
    address = studentAddress;
}

public String getAddress() {
    return address;
}

public void setAge(int studentAge) {
```

```
        if (studentAge >=0 && studentAge <=120) {  
            age = studentAge;  
        }  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setClassesId(int classesId) {  
        this.classesId = classesId;  
    }  
  
    public int getClassesId() {  
        return classesId;  
    }  
}  
  
class Employee {
```

//员工编号

private int id;

//姓名

private String name;

//性别

private boolean sex;

//地址

private String address;

//年龄

private int age;

//工作年限

private int workYear;

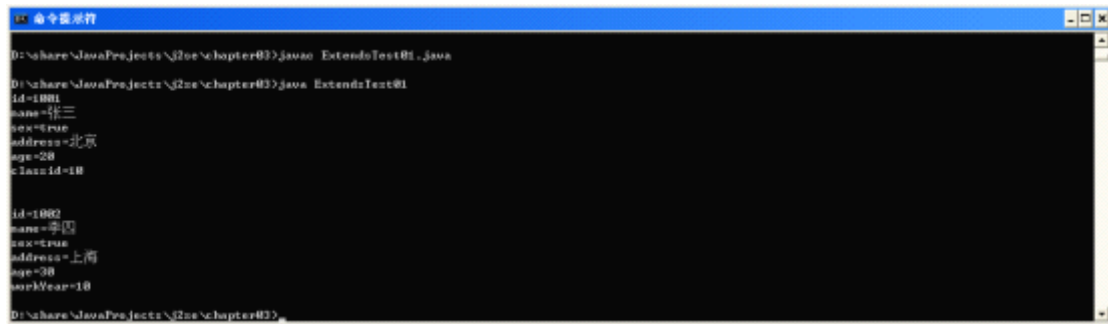
//设置学号

```
public void setId(int studentId) {  
    id = studentId;  
}
```

//读取学号

```
public int getId() {  
    return id;  
}  
  
public void setName(String studentName) {  
    name = studentName;  
}  
  
public String getName() {  
    return name;  
}  
  
public void setSex(boolean studentSex) {  
    sex = studentSex;  
}  
  
public boolean getSex() {  
    return sex;  
}  
  
public void setAddress(String studentAddress) {  
    address = studentAddress;  
}  
  
public String getAddress() {  
    return address;  
}  
  
public void setAge(int studentAge) {  
    if (studentAge >=0 && studentAge <=120) {  
        age = studentAge;  
    }  
}  
  
public int getAge() {  
    return age;  
}  
  
public void setWorkYear(int workYear) {  
    this.workYear = workYear;  
}  
  
public int getWorkYear() {  
    return workYear;  
}
```

```
}  
}
```



```
D:\share\javaProjects\J2se\chapter03>java ExtendsTest01.java  
D:\share\javaProjects\J2se\chapter03>java ExtendsTest01  
id=1001  
name=张三  
sex=true  
address=北京  
age=20  
class id=10  
  
id=1002  
name=李四  
sex=true  
address=上海  
age=30  
workYear=10  
D:\share\javaProjects\J2se\chapter03>
```

以上输出是完全正确的，但程序上存在一些冗余，因为 Student 和 Employee 都重复出现了 id,name,address, sex,age 属性，而 Student 和 Employee 其实都是 Person，所以应该具有 person 的相关属性，所以完全可以抽取出一个 Person 类出来，让 Student 和 Employee 继承它，这样我们的冗余代码就会减少

【代码示例】，提取 Person 类，让 Student 和 Employee 继承 Person

```
public class ExtendsTest02 {  
  
    public static void main(String[] args) {  
  
        Student student = new Student();  
        student.setId(1001);  
        student.setName("张三");  
        student.setSex(true);  
        student.setAddress("北京");  
        student.setAge(20);  
        student.setClassesId(10);  
  
        System.out.println("id=" + student.getId());  
        System.out.println("name=" + student.getName());  
        System.out.println("sex=" + student.getSex());  
        System.out.println("address=" + student.getAddress());  
        System.out.println("age=" + student.getAge());  
        System.out.println("classid=" + student.getClassesId());  
  
        System.out.println("");  
        System.out.println("");  
        Employee emp = new Employee();  
        emp.setId(1002);  
        emp.setName("李四");  
        emp.setSex(true);  
        emp.setAddress("上海");  
        emp.setAge(30);  
        emp.setWorkYear(10);  
    }  
}
```

```
        System.out.println("id=" + emp.getId());
        System.out.println("name=" + emp.getName());
        System.out.println("sex=" + emp.getSex());
        System.out.println("address=" + emp.getAddress());
        System.out.println("age=" + emp.getAge());
        System.out.println("workYear=" + emp.getWorkYear());
    }
}
```

```
class Person {
```

```
    //姓名
```

```
    private String name;
```

```
    //性别
```

```
    private boolean sex;
```

```
    //地址
```

```
    private String address;
```

```
    //年龄
```

```
    private int age;
```

```
    //设置学号
```

```
    public void setId(int studentId) {
```

```
        id = studentId;
```

```
    }
```

```
    //读取学号
```

```
    public int getId() {
```

```
        return id;
```

```
    }
```

```
    public void setName(String studentName) {
```

```
        name = studentName;
```

```
    }
```

```
    public String getName() {
```

```
        return name;
```

```
    }
```

```
    public void setSex(boolean studentSex) {
```

```
        sex = studentSex;
    }

    public boolean getSex() {
        return sex;
    }

    public void setAddress(String studentAddress) {
        address = studentAddress;
    }

    public String getAddress() {
        return address;
    }

    public void setAge(int studentAge) {
        if (studentAge >=0 && studentAge <=120) {
            age = studentAge;
        }
    }

    public int getAge() {
        return age;
    }
}

class Student extends Person {
    //学号
    private int sid;
    //班级编号
    private int classesId;

    public void setClassesId(int classesId) {
        this.classesId = classesId;
    }

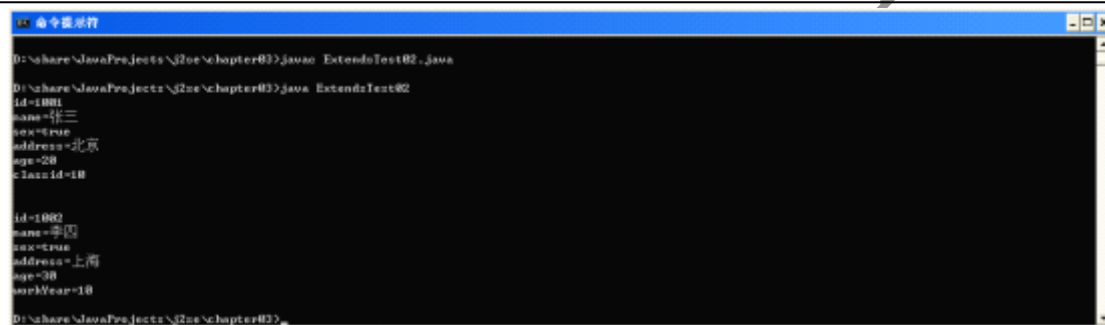
    public int getClassesId() {
        return classesId;
    }
}

class Employee extends Person {
    //编号
    private int eno;
```

```
//工作年限
private int workYear;

public void setWorkYear(int workYear) {
    this.workYear = workYear;
}

public int getWorkYear() {
    return workYear;
}
}
```



```
D:\share\JavaProjects\j2se\chapter03>java ExtendTest02.java
Id=10001
name=张三
sex=男性
address=北京
age=28
classId=100

Id=10002
name=李四
sex=女性
address=上海
age=38
workYear=18
D:\share\JavaProjects\j2se\chapter03>
```

运行结果和前面的一样，先从代码数量上来看，比以前少多了，没有重复了，软件设计有一个原则“**重复的代码最好不要出现两次或多次**”，如果出现两次以上相同的代码，那么就不好维护了，如果要改的话，那么多处都要改，给维护带来不便，而从我们的这个示例，大家看到采用继承可以达到**重用性**，把父类中的所有属性都继承下来了。

1.13、 方法的覆盖(Override)

首先看一下方法重载(Overload)，回顾方法重载的条件：

- **方法名称相同**
- 方法参数类型、个数、顺序至少有一个不同
- **方法的返回类型可以不同，因为方法重载和返回类型没有任何关系**
- 方法的修饰符可以不同，因为方法重载和修饰符没有任何关系
- 方法重载只出现在**同一个类中**

方法的覆盖(Override)的条件：

- **必须要有继承关系**
- 覆盖只能出现在子类中，如果没有继承关系，不存在覆盖，只存在重载
- 在子类中被覆盖的方法，必须和父类中的方法**完全一样**，也就是方法名，**返回类型**、参数列表，完全一样
- 子类方法的访问权限不能小于父类方法的访问权限
- **子类方法不能抛出比父类方法更多的异常，但可以抛出父类方法异常的子异常**
- **父类的静态方法不能被子类覆盖**
- 父类的私有方法不能覆盖
- **覆盖是针对成员方法，而非属性**

为什么需要覆盖？

就是要改变父类的行为。

1.13.1、对成员方法覆盖

【代码示例】，继承父类方法，不覆盖

```
public class OverrideTest01 {  
  
    public static void main(String[] args) {  
  
        Student student = new Student();  
        student.setId(1001);  
        student.setName("张三");  
        student.setSex(true);  
        student.setAddress("北京");  
        student.setAge(20);  
        student.setClassesId(10);  
  
        student.printInfo();  
  
        System.out.println("");  
        Employee emp = new Employee();  
        emp.setId(1002);  
        emp.setName("李四");  
        emp.setSex(true);  
        emp.setAddress("上海");  
        emp.setAge(30);  
        emp.setWorkYear(10);  
  
        emp.printInfo();  
    }  
}  
  
class Person {  
    //学号  
    private int id;  
  
    //姓名  
    private String name;  
  
    //性别  
    private boolean sex;  
  
    //地址  
    private String address;
```



```
//年龄
private int age;

public void printInfo() {
    System.out.println("id=" + id + ", name=" + name + ",sex=" + sex + ", address=" +
address + ", age=" + age);
}

//设置学号
public void setId(int studentId) {
    id = studentId;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}

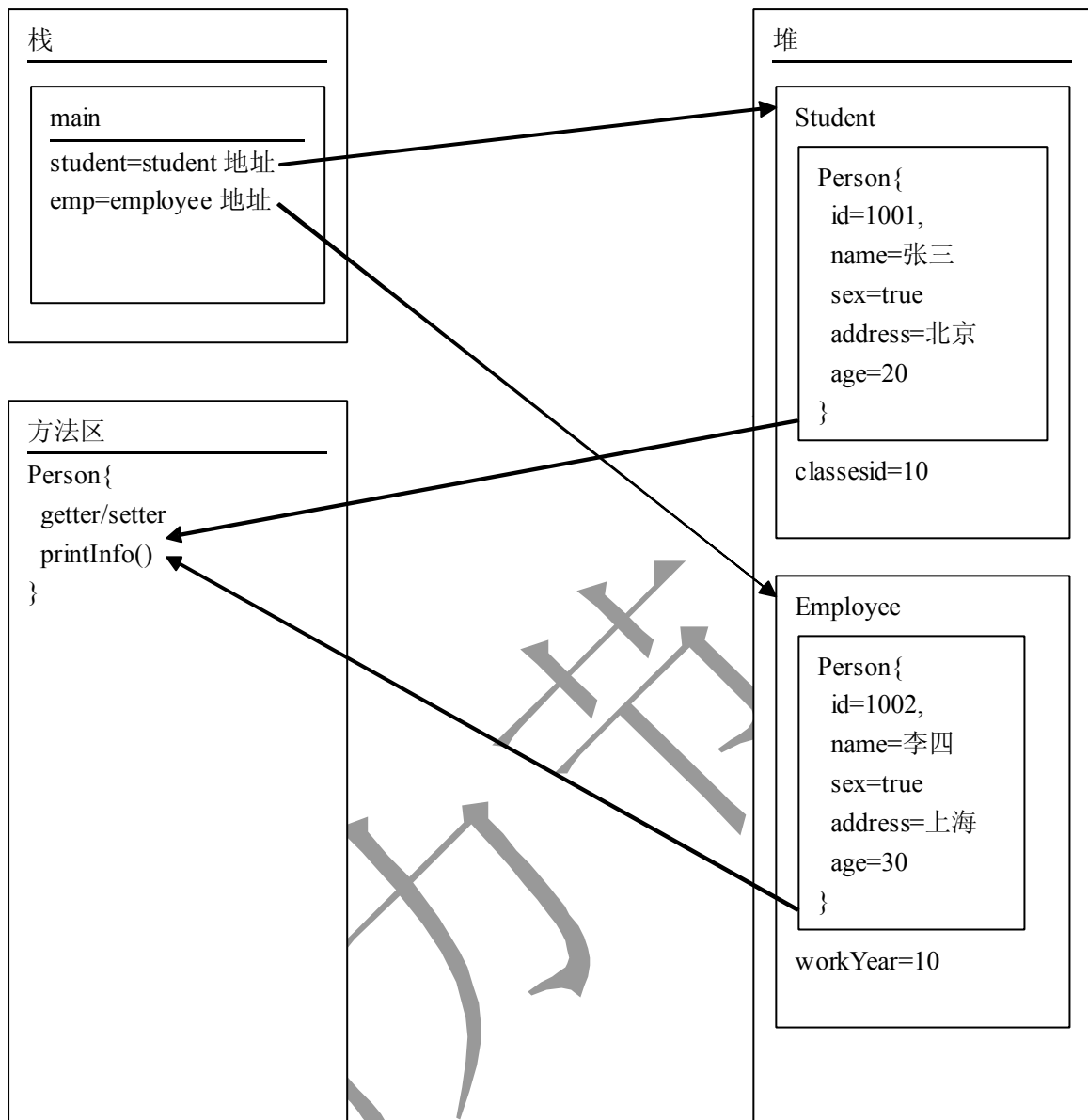
public boolean getSex() {
    return sex;
}

public void setAddress(String studentAddress) {
    address = studentAddress;
}

public String getAddress() {
    return address;
}

public void setAge(int studentAge) {
```

```
        if (studentAge >=0 && studentAge <=120) {  
            age = studentAge;  
        }  
    }  
  
    public int getAge() {  
        return age;  
    }  
}  
  
class Student extends Person {  
  
    //班级编号  
    private int classesId;  
  
    public void setClassesId(int classesId) {  
        this.classesId = classesId;  
    }  
  
    public int getClassesId() {  
        return classesId;  
    }  
}  
  
class Employee extends Person {  
  
    //工作年限  
    private int workYear;  
  
    public void setWorkYear(int workYear) {  
        this.workYear = workYear;  
    }  
  
    public int getWorkYear() {  
        return workYear;  
    }  
}
```



【代码示例】，继承父类方法，覆盖父类中的方法（改变父类的行为）

```
public class OverrideTest02 {  
  
    public static void main(String[] args) {  
  
        Student student = new Student();  
        student.setId(1001);  
        student.setName("张三");  
        student.setSex(true);  
        student.setAddress("北京");  
        student.setAge(20);  
        student.setClassesId(10);  
  
        student.printInfo();  
    }  
}
```

```
        System.out.println("");
        Employee emp = new Employee();
        emp.setId(1002);
        emp.setName("李四");
        emp.setSex(true);
        emp.setAddress("上海");
        emp.setAge(30);
        emp.setWorkYear(10);

        emp.printInfo();
    }
}

class Person {
    //学号
    private int id;

    //姓名
    private String name;

    //性别
    private boolean sex;

    //地址
    private String address;

    //年龄
    private int age;

    public void printInfo() {
        System.out.println("id=" + id + ", name=" + name + ",sex=" + sex + ", address=" +
address + ", age=" + age);
    }

    //设置学号
    public void setId(int studentId) {
        id = studentId;
    }

    //读取学号
    public int getId() {
        return id;
    }
}
```

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

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}

public boolean getSex() {
    return sex;
}

public void setAddress(String studentAddress) {
    address = studentAddress;
}

public String getAddress() {
    return address;
}

public void setAge(int studentAge) {
    if (studentAge >=0 && studentAge <=120) {
        age = studentAge;
    }
}

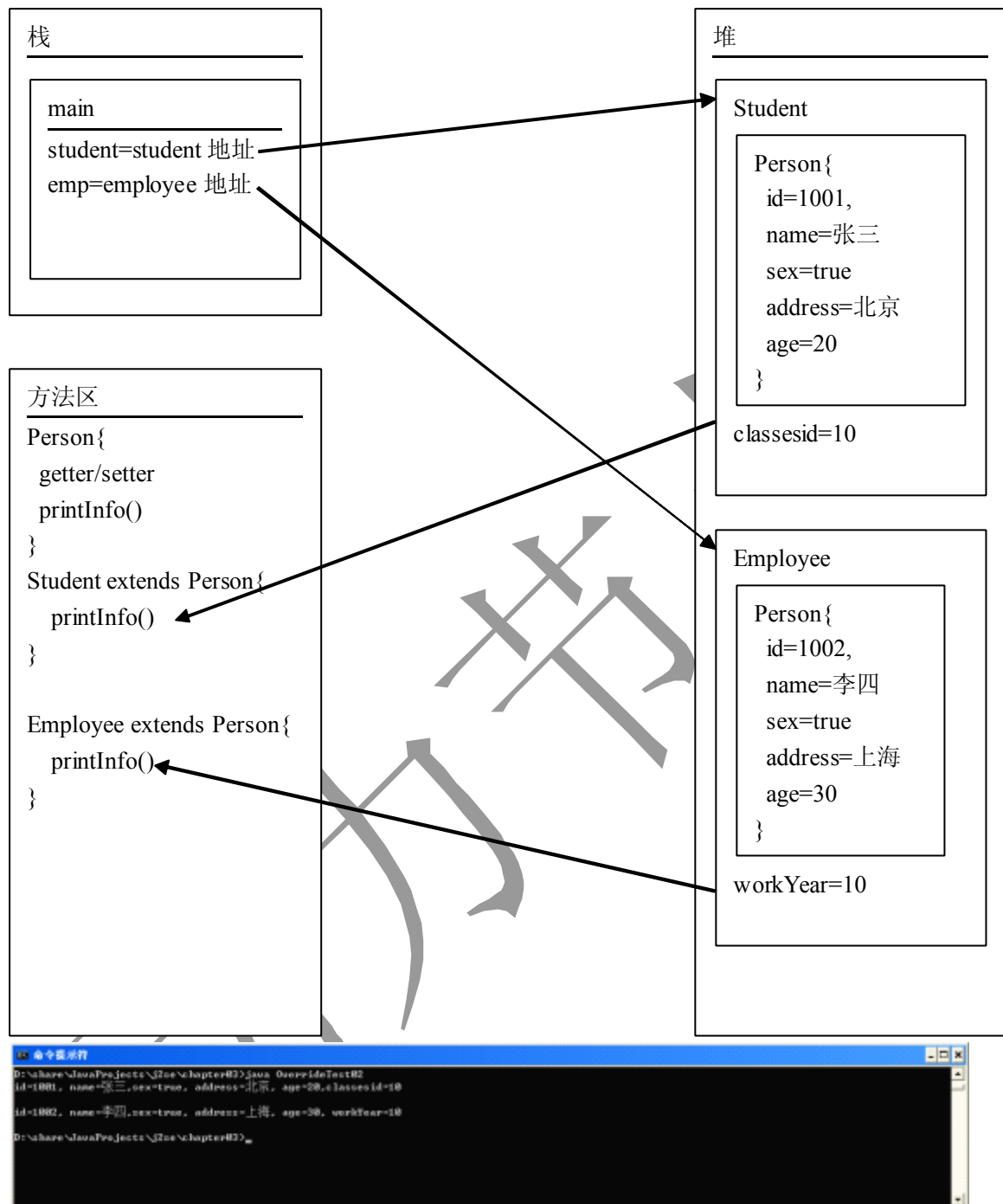
public int getAge() {
    return age;
}
}

class Student extends Person {

    //班级编号
    private int classesId;

    public void setClassesId(int classesId) {
        this.classesId = classesId;
    }
}
```

```
public int getClassesId() {  
    return classesId;  
}  
  
public void printInfo() {  
    System.out.println("id=" + getId() + ", name=" + getName() + ",sex=" + getSex() + ",  
address=" + getAddress() + ", age=" + getAge() + ",classesid=" + classesId);  
}  
}  
  
class Employee extends Person {  
  
    //工作年限  
    private int workYear;  
  
    public void setWorkYear(int workYear) {  
        this.workYear = workYear;  
    }  
  
    public int getWorkYear() {  
        return workYear;  
    }  
  
    public void printInfo() {  
        System.out.println("id=" + getId() + ", name=" + getName() + ",sex=" + getSex() + ",  
address=" + getAddress() + ", age=" + getAge() + ", workYear=" + workYear);  
    }  
}
```



以上子类对父类的方法进行了覆盖，改变了父类的行为，当我们 new 子类的时候，它不会再调用父类的方法了，而直接调用子类的方法，所以我们就完成了对父类行为的扩展。

【代码示例】，改善以上示例

```
public class OverrideTest03 {  
  
    public static void main(String[] args) {  
  
        //Student student = new Student();  
  
    }  
}
```

```
//此种写法是错误的，Person 不是 Student
//Student person_student = new Person();

//此种写法是正确的，Student 是 Person
Person person_student = new Student();

person_student.setId(1001);
person_student.setName("张三");
person_student.setSex(true);
person_student.setAddress("北京");
person_student.setAge(20);

//编译出错，因为 Person 看不到子类 Student 的属性
//person_student.setClassesId(10);

person_student.printInfo();

System.out.println("");
//Employee person_emp = new Employee();
Person person_emp = new Employee();

person_emp.setId(1002);
person_emp.setName("李四");
person_emp.setSex(true);
person_emp.setAddress("上海");
person_emp.setAge(30);
//编译出错，因为 Person 看不到子类 Employee 的属性
//person_emp.setWorkYear(10);

person_emp.printInfo();
}
}

class Person {
    //学号
    private int id;

    //姓名
    private String name;

    //性别
    private boolean sex;

    //地址
```



```
private String address;

//年龄
private int age;

public void printInfo() {
    System.out.println("id=" + id + ", name=" + name + ",sex=" + sex + ", address=" +
address + ", age=" + age);
}

//设置学号
public void setId(int studentId) {
    id = studentId;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}

public boolean getSex() {
    return sex;
}

public void setAddress(String studentAddress) {
    address = studentAddress;
}

public String getAddress() {
    return address;
}
```

```
public void setAge(int studentAge) {
    if (studentAge >=0 && studentAge <=120) {
        age = studentAge;
    }
}

public int getAge() {
    return age;
}
}

class Student extends Person {

    //班级编号
    private int classesId;

    public void setClassesId(int classesId) {
        this.classesId = classesId;
    }

    public int getClassesId() {
        return classesId;
    }

    public void printInfo() {
        System.out.println("id=" + getId() + ", name=" + getName() + ",sex=" + getSex() + ",
address=" + getAddress() + ", age=" + getAge() + ",classesid=" + classesId);
    }
}

class Employee extends Person {

    //工作年限
    private int workYear;

    public void setWorkYear(int workYear) {
        this.workYear = workYear;
    }

    public int getWorkYear() {
        return workYear;
    }
}
```

```
public void printInfo() {  
    System.out.println("id=" + getId() + ", name=" + getName() + ",sex=" + getSex() + ",  
address=" + getAddress() + ", age=" + getAge() + ", workYear=" + workYear);  
}  
}
```

【代码示例】，改善以上示例

```
public class OverrideTest04 {  
  
    public static void main(String[] args) {  
        Person person_student = new Student();  
  
        person_student.setId(1001);  
        person_student.setName("张三");  
        person_student.setSex(true);  
        person_student.setAddress("北京");  
        person_student.setAge(20);  
  
        print(person_student);  
  
        Person person_emp = new Employee();  
  
        person_emp.setId(1002);  
        person_emp.setName("李四");  
        person_emp.setSex(true);  
        person_emp.setAddress("上海");  
        person_emp.setAge(30);  
  
        print(person_emp);  
  
        Person person = new Person();  
        person.setId(1003);  
        person.setName("王五");  
        person.setSex(true);  
  
        print(person);  
    }  
  
    private static void print(Person person) {  
        person.printInfo();  
    }  
}
```

```
class Person {  
    //学号  
    private int id;  
  
    //姓名  
    private String name;  
  
    //性别  
    private boolean sex;  
  
    //地址  
    private String address;  
  
    //年龄  
    private int age;  
  
    public void printInfo() {  
        System.out.println("id=" + id + ", name=" + name + ",sex=" + sex + ", address=" +  
address + ", age=" + age);  
    }  
  
    //设置学号  
    public void setId(int studentId) {  
        id = studentId;  
    }  
  
    //读取学号  
    public int getId() {  
        return id;  
    }  
  
    public void setName(String studentName) {  
        name = studentName;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setSex(boolean studentSex) {  
        sex = studentSex;  
    }  
}
```

```
public boolean getSex() {
    return sex;
}

public void setAddress(String studentAddress) {
    address = studentAddress;
}

public String getAddress() {
    return address;
}

public void setAge(int studentAge) {
    if (studentAge >=0 && studentAge <=120) {
        age = studentAge;
    }
}

public int getAge() {
    return age;
}
}

class Student extends Person {

    //班级编号
    private int classesId;

    public void setClassesId(int classesId) {
        this.classesId = classesId;
    }

    public int getClassesId() {
        return classesId;
    }

    public void printInfo() {
        System.out.println("id=" + getId() + ", name=" + getName() + ",sex=" + getSex() + ",
address=" + getAddress() + ", age=" + getAge() + ",classesid=" + classesId);
    }
}

class Employee extends Person {
```

```
//工作年限
private int workYear;

public void setWorkYear(int workYear) {
    this.workYear = workYear;
}

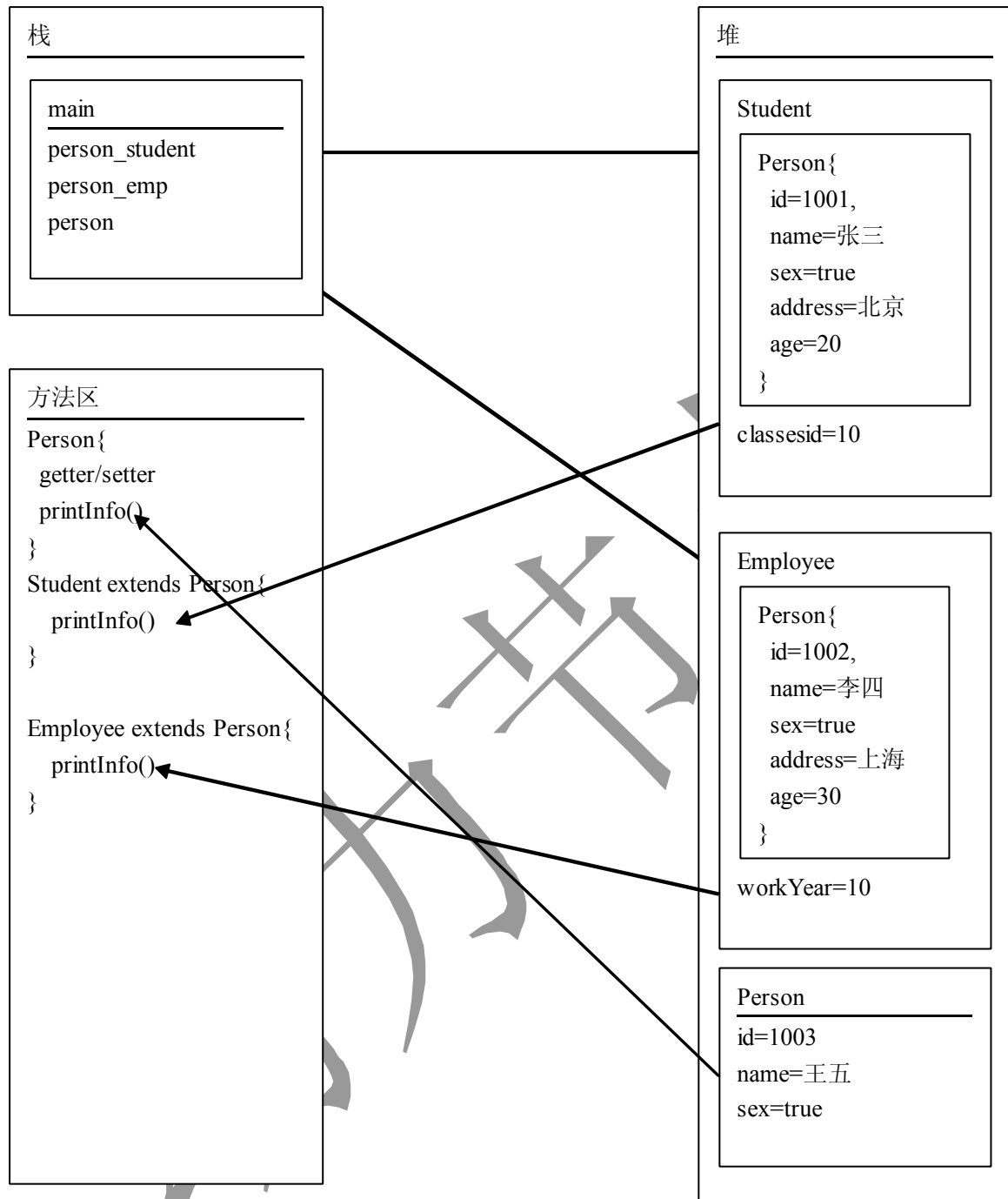
public int getWorkYear() {
    return workYear;
}

public void printInfo() {
    System.out.println("id=" + getId() + ", name=" + getName() + ",sex=" + getSex() + ",
address=" + getAddress() + ", age=" + getAge() + ", workYear=" + workYear);
}
}
```

多态其实就是多种状态，overload（重载）是多态的一种，属于编译期绑定，也就是静态绑定（前期绑定），override 是运行期间绑定（后期绑定）。

多态存在的条件：

- 有继承
- 有覆盖
- 父类指向子类的引用



1.13.2、对静态方法覆盖

【代码示例】

```
public class OverrideTest05 {

    public static void main(String[] args) {
        Person person_student = new Student();

        person_student.setId(1001);
    }
}
```

```
        person_student.setName("张三");
        person_student.setSex(true);
        person_student.setAddress("北京");
        person_student.setAge(20);

        print(person_student);

        Person person_emp = new Employee();

        person_emp.setId(1002);
        person_emp.setName("李四");
        person_emp.setSex(true);
        person_emp.setAddress("上海");
        person_emp.setAge(30);

        print(person_emp);
    }

    private static void print(Person person) {
        person.printInfo();
    }
}

class Person {
    //学号
    private int id;

    //姓名
    private String name;

    //性别
    private boolean sex;

    //地址
    private String address;

    //年龄
    private int age;

    public static void printInfo() {
        System.out.println("-----Person-----");
    }
}
```



```
}

//设置学号
public void setId(int studentId) {
    id = studentId;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}

public boolean getSex() {
    return sex;
}

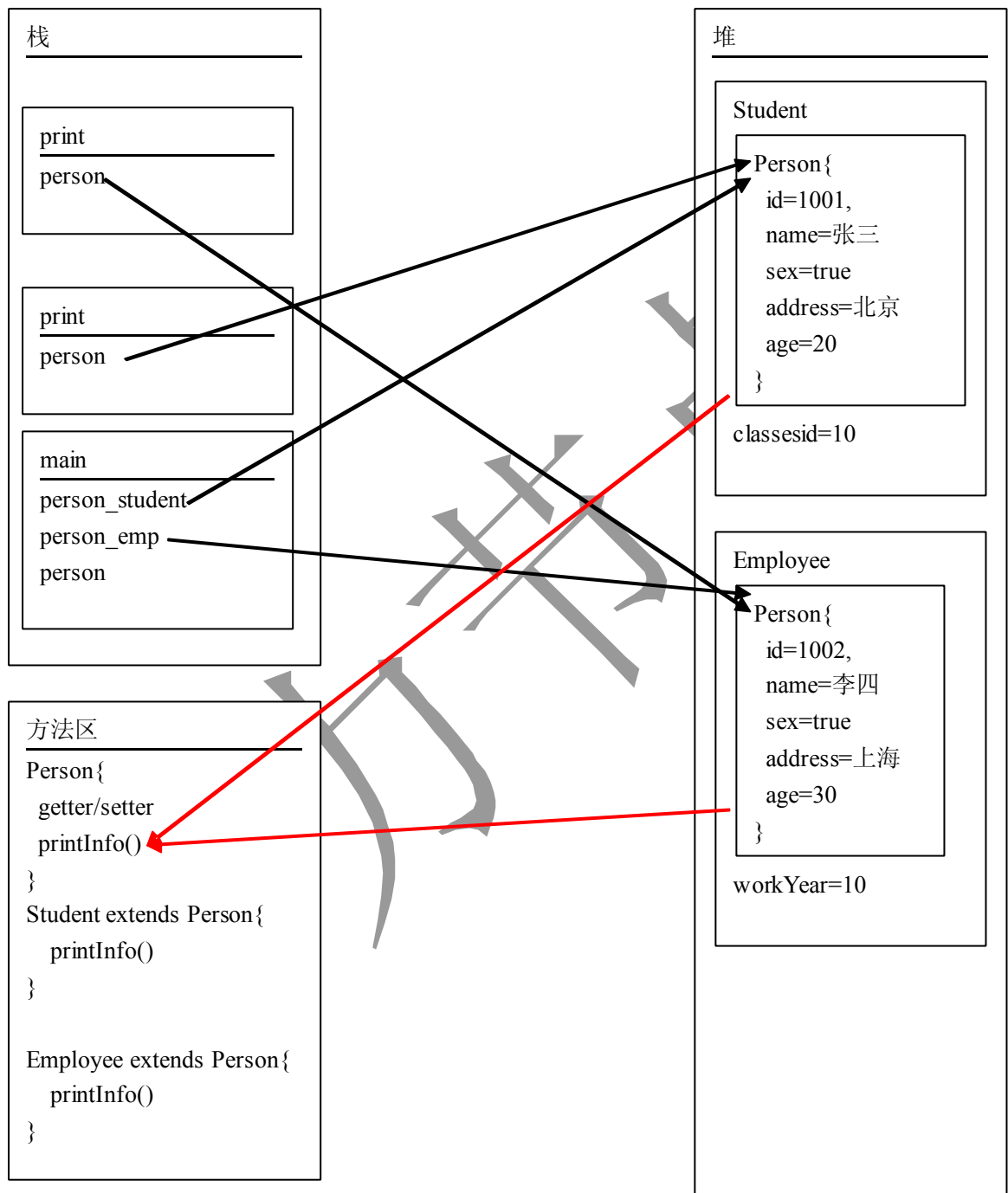
public void setAddress(String studentAddress) {
    address = studentAddress;
}

public String getAddress() {
    return address;
}

public void setAge(int studentAge) {
    if (studentAge >=0 && studentAge <=120) {
        age = studentAge;
    }
}

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

```
}  
}  
  
class Student extends Person {  
  
    //班级编号  
    private int classesId;  
  
    public void setClassesId(int classesId) {  
        this.classesId = classesId;  
    }  
  
    public int getClassesId() {  
        return classesId;  
    }  
  
    public static void printInfo() {  
        System.out.println("-----Student-----");  
    }  
}  
  
class Employee extends Person {  
  
    //工作年限  
    private int workYear;  
  
    public void setWorkYear(int workYear) {  
        this.workYear = workYear;  
    }  
  
    public int getWorkYear() {  
        return workYear;  
    }  
  
    public static void printInfo() {  
        System.out.println("-----Employee-----");  
    }  
}
```



```
命令提示符
D:\share\JavaProjects\j2se\chapter03>java OverrideTest05
-----
Person
Person
D:\share\JavaProjects\j2se\chapter03>
```

从上面的输出可以看到，静态方法不存在多态的概念，**多态和成员方法相关**，静态方法只属于某一个类，**声明的是那个一个类就调用的是哪一个类的静态方法和子类没有关系**，不像覆盖了成员方法，**new 谁调谁**，为什么覆盖成员方法可以构成多态（多种状态），主要是运行

期的动态绑定（**动态绑定**构成多态），**静态绑定**的含义是在编译成 class 文件（字节码）的时候已经确定该调用哪个方法。

1.14、 super 关键字

super 关键字的作用：

- 调用父类的构造方法
- 调用父类的成员方法

需要注意：super 只能应用在成员方法和构造方法中，不能应用在静态方法中（和 this 是一样的），如果在构造方法中使用必须放在第一行

为什么会有 super 关键字？

- 因为子类必须要调用父类的构造方法，先把父类构造完成，因为子类依赖于父类，没有父，也就没有子
- 有时需要在子类中显示的调用父类的成员方法

那么我们以前为什么没有看到 super，而且我们也有继承，如：Student 继承了 Person？

- 因为子类中我们没有显示的调用构造方法，那么他会默认调用父类的无参构造方法，此种情况下如果父类中没有无参构造方法，那么编译时将会失败

注意构造方法不存在覆盖的概念，构造方法可以重载

1.14.1、调用默认构造方法

```
public class SuperTest01 {  
  
    public static void main(String[] args) {  
        Person student = new Student();  
    }  
}  
  
class Person {  
    //学号  
    private int id;  
  
    //姓名  
    private String name;  
  
    //性别  
    private boolean sex;  
  
    //地址  
    private String address;  
  
    //年龄  
    private int age;
```

```
public Person() {  
    //System.out.println(this);  
    System.out.println("-----Person-----");  
}  
  
//设置学号  
public void setId(int studentId) {  
    id = studentId;  
}  
  
//读取学号  
public int getId() {  
    return id;  
}  
  
public void setName(String studentName) {  
    name = studentName;  
}  
  
public String getName() {  
    return name;  
}  
  
public void setSex(boolean studentSex) {  
    sex = studentSex;  
}  
  
public boolean getSex() {  
    return sex;  
}  
  
public void setAddress(String studentAddress) {  
    address = studentAddress;  
}  
  
public String getAddress() {  
    return address;  
}  
  
public void setAge(int studentAge) {  
    if (studentAge >=0 && studentAge <=120) {  
        age = studentAge;  
    }  
}
```

```
}

public int getAge() {
    return age;
}
}

class Student extends Person {

    //班级编号
    private int classesId;

    public Student() {
        //System.out.println(this);
        //显示调用父类的构造方法
        //调用父类的无参的构造函数
        //super();

        System.out.println("-----Student-----");
        //编译错误，必须将 super 放到构造函数的第一语句
        //必须先创建父类，才能创建子类
        //super();
    }

    public void setClassesId(int classesId) {
        this.classesId = classesId;
    }

    public int getClassesId() {
        return classesId;
    }
}
```

关于构造方法的执行顺序，先执行父类的再执行子类的，必须完成父类的所有初始化，才能创建子类

1.14.2、调用带参数的构造方法

```
public class SuperTest02 {

    public static void main(String[] args) {

        /*
        Person person_student = new Student();
        */
    }
}
```

```
person_student.setId(1001);
person_student.setName("张三");
person_student.setSex(true);
person_student.setAddress("北京");
person_student.setAge(20);
*/

//编译出错，因为 Person 看不到子类 Student 的属性
//person_student.setClassesId(10);

/*
person_student.printInfo();

System.out.println("");
Person person_emp = new Employee();

person_emp.setId(1002);
person_emp.setName("李四");
person_emp.setSex(true);
person_emp.setAddress("上海");
person_emp.setAge(30);
*/

//编译出错，因为 Person 看不到子类 Employee 的属性
//person_emp.setWorkYear(10);

//person_emp.printInfo();
Person person_student = new Student(1001, "张三", true, "北京", 20, 10);
person_student.printInfo();
}
}

class Person {
    //学号
    private int id;

    //姓名
    private String name;

    //性别
    private boolean sex;
```

```
//地址
private String address;

//年龄
private int age;

public Person() {
    System.out.println("-----Person-----");
}

public Person(int id, String name, boolean sex, String address, int age) {
    this.id = id;
    this.name = name;
    this.sex = sex;
    this.address = address;
    this.age = age;
}

public void printInfo() {
    System.out.println("id=" + id + ", name=" + name + ",sex=" + sex + ", address=" +
address + ", age=" + age);
}

//设置学号
public void setId(int studentId) {
    id = studentId;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}
```



```
public boolean getSex() {  
    return sex;  
}  
  
public void setAddress(String studentAddress) {  
    address = studentAddress;  
}  
  
public String getAddress() {  
    return address;  
}  
  
public void setAge(int studentAge) {  
    if (studentAge >=0 && studentAge <=120) {  
        age = studentAge;  
    }  
}  
  
public int getAge() {  
    return age;  
}  
}  
  
class Student extends Person {  
  
    //班级编号  
    private int classesId;  
  
    public Student(int id, String name, boolean sex, String address, int age, int classesId) {  
        /*  
        this.id = id;  
        this.name = name;  
        this.sex = sex;  
        this.address = address;  
        this.age = age;  
        this.classesId = id;  
        */  
  
        /*  
        setId(id);  
        setName(name);  
        setSex(sex);  
        setAddress(address);  
    }  
}
```

```
        setAge(age);
        this.classesId = classesId;
        */
        //手动调用调用带参数的构造函数
        super(id, name, sex, address, age);
        this.classesId = classesId;

    }

    public void setClassesId(int classesId) {
        this.classesId = classesId;
    }

    public int getClassesId() {
        return classesId;
    }

    public void printInfo() {
        System.out.println("id=" + getId() + ", name=" + getName() + ",sex=" + getSex() + ",
address=" + getAddress() + ", age=" + getAge() + ",classesid=" + classesId);
    }
}

class Employee extends Person {

    //工作年限
    private int workYear;

    public void setWorkYear(int workYear) {
        this.workYear = workYear;
    }

    public int getWorkYear() {
        return workYear;
    }

    public void printInfo() {
        System.out.println("id=" + getId() + ", name=" + getName() + ",sex=" + getSex() + ",
address=" + getAddress() + ", age=" + getAge() + ", workYear=" + workYear);
    }
}
```

1.14.3、采用 **super** 调用父类的方法

```
public class SuperTest03 {

    public static void main(String[] args) {
        Person person_student = new Student();

        person_student.setId(1001);
        person_student.setName("张三");
        person_student.setSex(true);
        person_student.setAddress("北京");
        person_student.setAge(20);

        print(person_student);

        Person person_emp = new Employee();

        person_emp.setId(1002);
        person_emp.setName("李四");
        person_emp.setSex(true);
        person_emp.setAddress("上海");
        person_emp.setAge(30);

        print(person_emp);
    }

    private static void print(Person person) {
        person.printInfo();
    }
}

class Person {
    //学号
    private int id;

    //姓名
    private String name;

    //性别
    private boolean sex;
```

```
//地址
private String address;

//年龄
private int age;

public void printInfo() {
    System.out.println("id=" + id + ", name=" + name + ",sex=" + sex + ", address=" +
address + ", age=" + age);
}

//设置学号
public void setId(int studentId) {
    id = studentId;
}

//读取学号
public int getId() {
    return id;
}

public void setName(String studentName) {
    name = studentName;
}

public String getName() {
    return name;
}

public void setSex(boolean studentSex) {
    sex = studentSex;
}

public boolean getSex() {
    return sex;
}

public void setAddress(String studentAddress) {
    address = studentAddress;
}

public String getAddress() {
    return address;
}
```

```
public void setAge(int studentAge) {
    if (studentAge >=0 && studentAge <=120) {
        age = studentAge;
    }
}

public int getAge() {
    return age;
}
}

class Student extends Person {

    //班级编号
    private int classesId;

    public void setClassesId(int classesId) {
        this.classesId = classesId;
    }

    public int getClassesId() {
        return classesId;
    }

    public void printInfo() {
        //System.out.println("id=" + getId() + ", name=" + getName() + ",sex=" + getSex() + ",
        address=" + getAddress() + ", age=" + getAge() + ",classesid=" + classesId);
        //采用 super 调用父类的方法
        super.printInfo();
        System.out.println("classesId=" + classesId);
    }
}

class Employee extends Person {

    //工作年限
    private int workYear;

    public void setWorkYear(int workYear) {
        this.workYear = workYear;
    }
}
```

```
public int getWorkYear() {
    return workYear;
}

public void printInfo() {
    //System.out.println("id=" + getId() + ", name=" + getName() + ",sex=" + getSex() + ",
address=" + getAddress() + ", age=" + getAge() + ", workYear=" + workYear);
    System.out.println("workYear=" + workYear);
    //采用 super 调用父类的方法
    super.printInfo();
}
}
```

1.15、 final 关键字

final 表示不可改变的含义

- 采用 final 修饰的类不能被继承
- 采用 final 修饰的方法不能被覆盖
- 采用 final 修饰的变量不能被修改
- final 修饰的变量必须显示初始化
- 如果修饰的引用，那么这个引用只能指向一个对象，也就是说这个引用不能再次赋值，但被指向的对象是可以修改的
- 构造方法不能被 final 修饰
- 会影响 JAVA 类的初始化:final 定义的静态常量调用时不会执行 java 的类初始化方法，也就是说不会执行 static 代码块等相关语句，这是由 java 虚拟机规定的。我们不需要了解的很深，有个概念就可以了。

1.15.1、采用 final 修饰的类不能被继承

```
public class FinalTest01 {

    public static void main(String[] args) {

    }

}

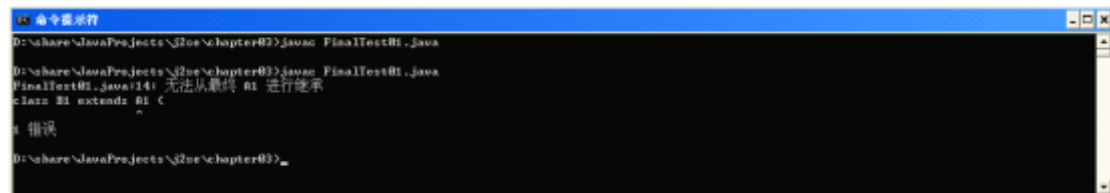
final class A1 {
    public void test1() {

    }

}
```

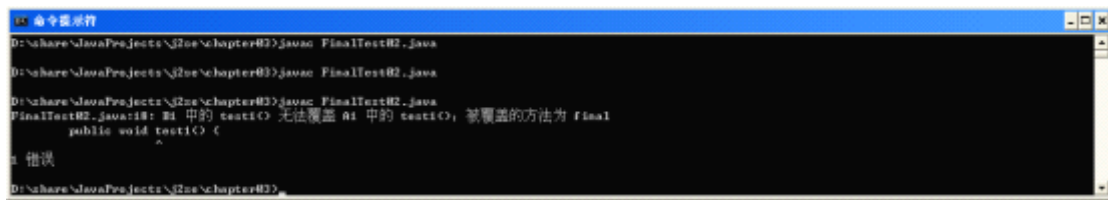
//不能继承 A1，因为 A1 采用 final 修饰了

```
class B1 extends A1 {  
  
    public void test2() {  
  
    }  
  
}
```



1.15.2、采用 final 修饰的方法不能被覆盖

```
public class FinalTest02 {  
  
    public static void main(String[] args) {  
  
    }  
  
}  
  
class A1 {  
  
    public final void test1() {  
  
    }  
  
}  
  
class B1 extends A1 {  
  
    //覆盖父类的方法，改变其行为  
    //因为父类的方法是 final 修饰的，所以不能覆盖  
    public void test1() {  
  
    }  
  
    public void test2() {  
  
    }  
  
}
```



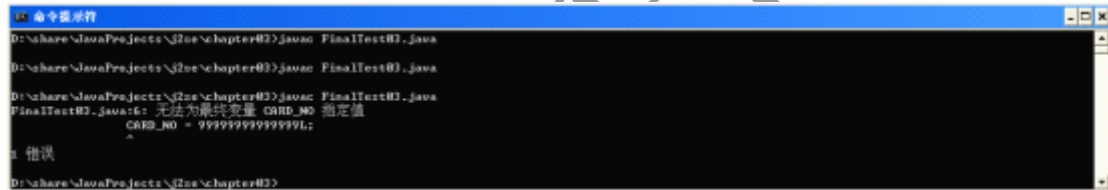
```
D:\share\javaProjects\202\chapter02>javac FinalTest02.java
D:\share\javaProjects\202\chapter02>javac FinalTest02.java
D:\share\javaProjects\202\chapter02>javac FinalTest02.java
FinalTest02.java:18: B1 中的 test() 无法覆盖 B1 中的 test(); 被覆盖的方法为 final
    public void test() {
            ^
1 错误
D:\share\javaProjects\202\chapter02>
```

1.15.3、采用 **final** 修饰的变量(基本类型)不能被修改

```
public class FinalTest03 {

    private static final long CARD_NO = 878778878787878L;

    public static void main(String[] args) {
        //不能进行修改，因为 CARD_NO 采用 final 修改了
        CARD_NO = 999999999999999L;
    }
}
```



```
D:\share\javaProjects\202\chapter02>javac FinalTest03.java
D:\share\javaProjects\202\chapter02>javac FinalTest03.java
D:\share\javaProjects\202\chapter02>javac FinalTest03.java
FinalTest03.java:6: 无法为最终变量 CARD_NO 指定值
    CARD_NO = 999999999999999L;
            ^
1 错误
D:\share\javaProjects\202\chapter02>
```

1.15.4、**final** 修饰的变量必须显示初始化

```
public class FinalTest04 {

    //如果是 final 修饰的变量必须初始化
    private static final long CARD_NO = 0L;

    public static void main(String[] args) {
        int i;
        //局部变量必须初始化
        //如果不使用可以不初始化
        System.out.println(i);
    }
}
```


1.15.5、如果修饰的引用，那么这个引用只能指向一个对象，也就是说这个引用不能再次赋值，但被指向的对象是可以修改的

```
public class FinalTest05 {
```

```
    public static void main(String[] args) {  
        Person p1 = new Person();
```

```
        //可以赋值  
        p1.name = "张三";  
        System.out.println(p1.name);
```

```
        final Person p2 = new Person();  
        p2.name = "李四";  
        System.out.println(p2.name);
```

```
        //不能编译通过
```

//p2 采用 final 修饰，主要限制了 p2 指向堆区中的地址不能修改(也就是 p2 只能指向一个对象)

//p2 指向的对象的属性是可以修改的

```
        p2 = new Person();
```

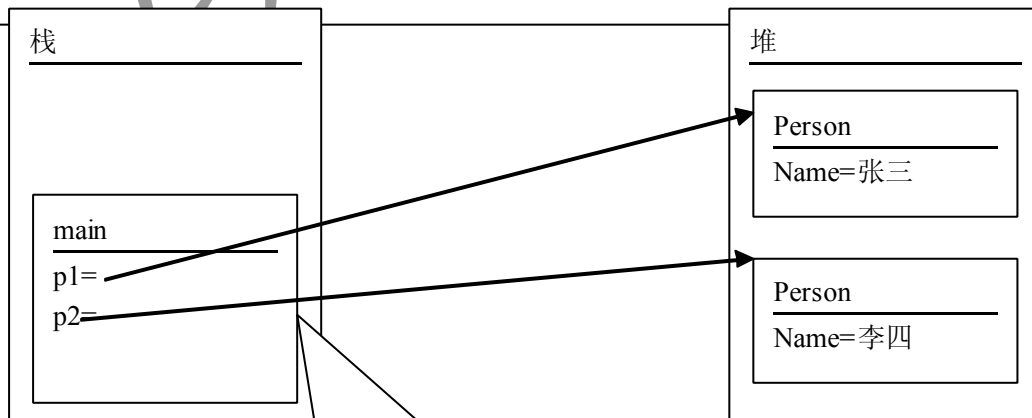
```
    }
```

```
}
```

```
class Person {
```

```
    String name;
```

```
}
```



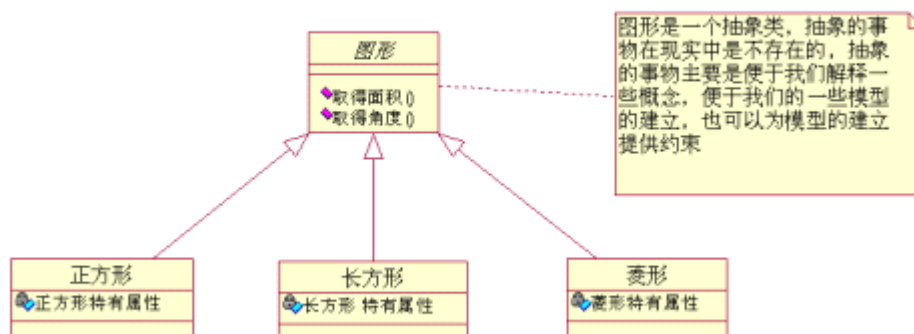
p2 采用 final 声明，p2 的地址是不能改变的,但他指向的对象是可以改变的,final 修饰引用类型限制的是地址不能修改，而不是引用指向的对象

1.16、 抽象类

看我们以前示例中的 Person、Student 和 Employee，从我们使用的角度来看主要对 Student 和 Employee 进行实例化，Person 中主要包含了一些公共的属性和方法，而 Person 我们通常不会实例化，所以我们可以把它定义成抽象的：

- 在 java 中采用 **abstract** 关键字定义的类就是抽象类，采用 **abstract** 关键字定义的方法就是抽象方法
- 抽象的方法只需在抽象类中，提供声明，不需要实现
- 如果一个类中含有抽象方法，那么这个类必须定义成抽象类
- 如果这个类是抽象的，那么这个类被子类继承，抽象方法必须被重写。如果在子类中不复写该抽象方法，那么必须将此类再次声明为抽象类
- 抽象的类是不能实例化的，就像现实世界中人其实是抽象的，张三、李四才是具体的
- 抽象类不能被 **final** 修饰
- 抽象方法不能被 **final** 修饰，因为抽象方法就是被子类实现的

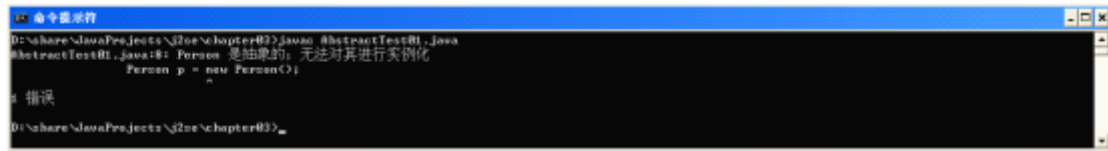
抽象类中可以包含方法实现，可以将一些公共的代码放到抽象类中，另外在抽象类中可以定义一些抽象的方法，这样就会存在一个约束，而子类必须实现我们定义的方法，如：teacher 必须实现 printInfo 方法，Student 也必须实现 printInfo 方法，方法名称不能修改，必须为 printInfo，这样就能实现多态的机制，有了多态的机制，我们在运行期就可以动态的调用子类的方法。所以在运行期可以灵活的互换实现。



1.16.1、采用 **abstract** 声明抽象类

```
public class AbstractTest01 {  
  
    public static void main(String[] args) {  
  
        //不能实例化抽象类  
        //抽象类是不存在，抽象类必须有子类继承  
        Person p = new Person();  
  
        //以下使用是正确的，因为我们 new 的是具体类  
        Person p1 = new Employee();  
        p1.setName("张三");  
        System.out.println(p1.getName());  
  
    }  
}  
  
//采用 abstract 定义抽象类  
//在抽象类中可以定义一些子类公共的方法或属性  
//这样子类就可以直接继承下来使用了，而不需要每个  
//子类重复定义  
abstract class Person {  
  
    private String name;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    //此方法各个子类都可以使用  
    public void commonMethod1() {  
        System.out.println("-----commonMethod1-----");  
    }  
}  
  
class Employee extends Person {
```

```
}  
  
class Student extends Person {  
  
}
```



1.16.2、抽象的方法只需在抽象类中，提供声明，不需要实现，起到了一个强制的约束作用，要求子类必须实现

```
public class AbstractTest02 {  
  
    public static void main(String[] args) {  
        //Person p = new Employee();  
        //Person p = new Student();  
        //Person p = new Person();  
        p.setName("张三");  
        p.printInfo();  
    }  
}  
  
abstract class Person {  
  
    private String name;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    //此方法各个子类都可以使用  
    public void commonMethod1() {  
        System.out.println("-----commonMethod1-----");  
    }  
}
```

```
//public void printInfo() {  
//    System.out.println("-----Person.printInfo()-----");  
//}  
  
//采用 abstract 定义抽象方法  
//如果有一个方法为抽象的，那么此类必须为抽象的  
//如果一个类是抽象的，并不要求具有抽象的方法  
public abstract void printInfo();  
}  
  
class Employee extends Person {  
  
    //必须实现抽象的方法  
    public void printInfo() {  
        System.out.println("Employee.printInfo()");  
    }  
}  
  
class Student extends Person {  
  
    //必须实现抽象的方法  
    public void printInfo() {  
        System.out.println("Student.printInfo()");  
    }  
}
```

1.16.3、如果这个类是抽象的，那么这个类被子类继承，抽象方法必须被覆盖。如果在子类中不覆盖该抽象方法，那么必须将此方法再次声明为抽象方法

```
public class AbstractTest03 {  
  
    public static void main(String[] args) {  
        //此时不能再 new Employee 了  
        Person p = new Employee();  
    }  
}  
  
abstract class Person {
```

```
private String name;

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

public String getName() {
    return name;
}

//此方法各个子类都可以使用
public void commonMethod1() {
    System.out.println("-----commonMethod1-----");
}

//采用 abstract 定义抽象方法
public abstract void printInfo();
}

abstract class Employee extends Person {

    //再次声明该方法为抽象的
    public abstract void printInfo();
}

class Student extends Person {

    //实现抽象的方法
    public void printInfo() {
        System.out.println("Student.printInfo()");
    }
}
```

1.16.4、抽象类不能被 **final** 修饰

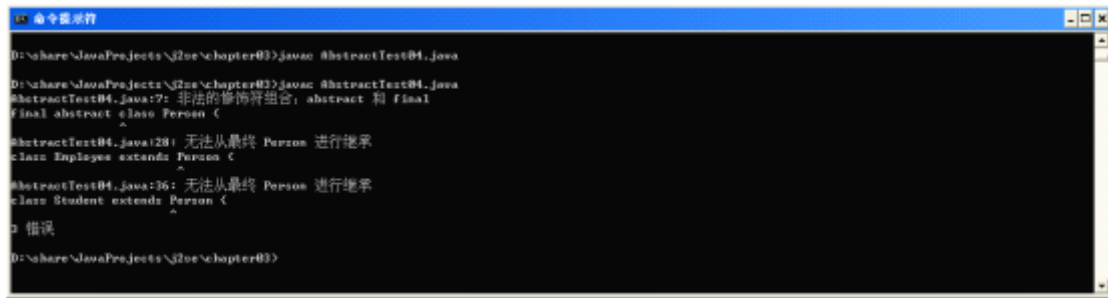
```
public class AbstractTest04 {

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

//不能采用 **final** 修改抽象类

//两个关键字是矛盾的

```
final abstract class Person {  
  
    private String name;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    //此方法各个子类都可以使用  
    public void commonMethod1() {  
        System.out.println("-----commonMethod1-----");  
    }  
  
    //采用 abstract 定义抽象方法  
    public abstract void printInfo();  
}  
  
class Employee extends Person {  
  
    //实现抽象的方法  
    public void printInfo() {  
        System.out.println("Student.printInfo()");  
    }  
}  
  
class Student extends Person {  
  
    //实现抽象的方法  
    public void printInfo() {  
        System.out.println("Student.printInfo()");  
    }  
}
```



```
D:\share\JavaProjects\j2se\chapter03>javac AbstractTest04.java
D:\share\JavaProjects\j2se\chapter03>javac AbstractTest04.java
AbstractTest04.java:7: 非法的修饰符组合: abstract 和 final
final abstract class Person {
    ^
AbstractTest04.java:28: 无法从最终 Person 进行继承
class Employee extends Person {
    ^
AbstractTest04.java:36: 无法从最终 Person 进行继承
class Student extends Person {
    ^
D 错误
D:\share\JavaProjects\j2se\chapter03>
```

1.16.5、抽象方法不能被 **final** 修饰

```
public class AbstractTest05 {

    public static void main(String[] args) {
    }
}

abstract class Person {

    private String name;

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

    public String getName() {
        return name;
    }

    //此方法各个子类都可以使用
    public void commonMethod1() {
        System.out.println("-----commonMethod1-----");
    }

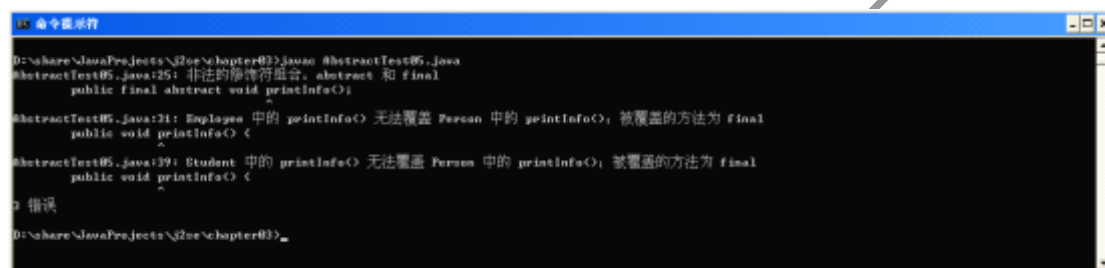
    //不能采用 final 修饰抽象的方法
    //这两个关键字存在矛盾
    public final abstract void printInfo();
}

class Employee extends Person {

    //实现抽象的方法
    public void printInfo() {
        System.out.println("Student.printInfo()");
    }
}
```



```
}  
}  
  
class Student extends Person {  
  
    //实现抽象的方法  
    public void printInfo() {  
        System.out.println("Student.printInfo()");  
    }  
}
```



```
D:\share\JavaProjects\32ee\chapter03\java0 AbstractTest05.java  
AbstractTest05.java:25: 非法的修饰符组合, abstract 和 final  
    public final abstract void printInfo();  
                           ^  
AbstractTest05.java:31: Employee 中的 printInfo() 无法覆盖 Person 中的 printInfo(); 被覆盖的方法为 final  
    public void printInfo() {  
                           ^  
AbstractTest05.java:39: Student 中的 printInfo() 无法覆盖 Person 中的 printInfo(); 被覆盖的方法为 final  
    public void printInfo() {  
                           ^  
编译错误  
D:\share\JavaProjects\32ee\chapter03>_
```

1.16.6、抽象类中可以没有抽象方法（参见 1.16.1）

1.17、 接口(行为)

接口我们可以~~看作~~是抽象类的一种特殊情况，在接口中只能定义抽象的方法和常量

- 1) 在 java 中接口采用 interface 声明
- 2) 接口中的方法默认都是 **public abstract** 的，不能更改
- 3) 接口中的变量默认都是 **public static final** 类型的，不能更改，所以必须显示的初始化
- 4) 接口不能被实例化，接口中没有构造函数的概念
- 5) 接口之间可以继承，但接口之间不能实现
- 6) 接口中的方法只能通过类来实现，通过 implements 关键字
- 7) 如果一个类实现了接口，那么接口中所有的方法必须实现
- 8) 一类可以实现多个接口

1.17.1、接口中的方法默认都是 **public abstract** 的，不能更改

```
public class InterfaceTest01 {
```

```
public static void main(String[] args) {  
    }  
}  
  
//采用 interface 定义接口  
//定义功能，没有实现  
//实现委托给类实现  
interface StudentManager {  
  
    //正确，默认 public abstract 等同 public abstract void addStudent(int id, String name);  
    public void addStudent(int id, String name);  
  
    //正确  
    //public abstract void addStudent(int id, String name);  
  
    //正确，可以加入 public 修饰符，此种写法较多  
    public void delStudent(int id);  
  
    //正确，可以加入 abstract，这种写法比较少  
    public abstract void modifyStudent(int id, String name);  
  
    //编译错误，因为接口就是让其他人实现  
    //采用 private 就和接口原本的定义产生矛盾了  
    private String findStudentById(int id);  
  
}
```

1.17.2、接口中的变量是 **public static final** 类型的，不能更改，所以必须显示的初始化

```
public class InterfaceTest02 {  
  
    public static void main(String[] args) {  
  
        //不能修改，因为 final 的  
        //StudentManager.YES = "abc";  
  
        System.out.println(StudentManager.YES);  
    }  
}
```

```
interface StudentManager {  
  
    //正确，默认加入 public static final  
    String YES = "yes";  
  
    //正确，开发中一般就按照下面的方式进行声明  
    public static final String NO = "no";  
  
    //错误，必须赋值，因为是 final 的  
    //int ON;  
  
    //错误，不能采用 private 声明  
    private static final int OFF = -1;  
}
```

1.17.3、接口不能被实例化，接口中没有构造函数的概念

```
public class InterfaceTest03 {  
  
    public static void main(String[] args) {  
  
        //接口是抽象类的一种特例，只能定义方法和变量，没有实现  
        //所以不能实例化  
        StudentManager studentManager = new StudentManager();  
    }  
}  
  
interface StudentManager {  
  
    public void addStudent(int id, String name);  
}
```

1.17.4、接口之间可以继承，但接口之间不能实现

```
public class InterfaceTest04 {  
  
    public static void main(String[] args) {  
    }  
}  
  
interface inter1 {  
    public void method1();  
}
```

```
public void method2();
}

interface inter2 {
    public void method3();
}

//接口可以继承
interface inter3 extends inter1 {

    public void method4();
}

//接口不能实现接口
//接口只能被类实现
interface inter4 implements inter2 {
    public void method3();
}
```

1.17.5、如果一个类实现了接口，那么接口中所有的方法必须实现

```
public class InterfaceTest05 {

    public static void main(String[] args) {
        //Iter1Impl 实现了 Inter1 接口
        //所以它是 Inter1 类型的产品
        //所以可以赋值
        Inter1 iter1 = new Iter1Impl();
        iter1.method1();

        //Iter1Impl123 实现了 Inter1 接口
        //所以它是 Inter1 类型的产品
        //所以可以赋值
        iter1 = new Iter1Impl123();
        iter1.method1();

        //可以直接采用 Iter1Impl 来声明类型
        //这种方式存在问题
        //不利于互换，因为面向具体编程了
        Iter1Impl iter1Impl = new Iter1Impl();
        iter1Impl.method1();
    }
}
```

```
//不能直接赋值给 iter1Impl  
//因为 Iter1Impl123 不是 Iter1Impl 类型  
//iter1Impl = new Iter1Impl123();  
//iter1Impl.method1();  
  
}  
}
```

//接口中的方法必须全部实现

```
class Iter1Impl implements Inter1 {  
  
    public void method1() {  
        System.out.println("method1");  
    }  
  
    public void method2() {  
        System.out.println("method2");  
    }  
  
    public void method3() {  
        System.out.println("method3");  
    }  
}  
  
class Iter1Impl123 implements Inter1 {  
  
    public void method1() {  
        System.out.println("method1_123");  
    }  
  
    public void method2() {  
        System.out.println("method2_123");  
    }  
  
    public void method3() {  
        System.out.println("method3_123");  
    }  
}  
  
abstract class Iter1Impl456 implements Inter1 {  
  
    public void method1() {  
        System.out.println("method1_123");  
    }  
}
```

```
}

public void method2() {
    System.out.println("method2_123");
}

//再次声明成抽象方法
public abstract void method3();
}

//定义接口
interface Inter1 {

    public void method1();

    public void method2();

    public void method3();
}
```

1.17.6、一类可以实现多个接口

```
public class InterfaceTest06 {

    public static void main(String[] args) {

        //可以采用 Inter1 定义
        Inter1 inter1 = new InterImpl();
        inter1.method1();

        //可以采用 Inter1 定义
        Inter2 inter2 = new InterImpl();
        inter2.method2();

        //可以采用 Inter1 定义
        Inter3 inter3 = new InterImpl();
        inter3.method3();
    }
}
```

//实现多个接口，采用逗号隔开
//这样这个类就拥有了多种类型
//等同于现实中的多继承
//所以采用 java 中的接口可以实现多继承
//把接口粒度划分细了，主要使功能定义的含义更明确
//可以采用一个大的接口定义所有功能，替代多个小的接口，
//但这样定义功能不明确，粒度太粗了

```
class InterImpl implements Inter1, Inter2, Inter3 {
```

```
    public void method1() {  
        System.out.println("----method1-----");  
    }
```

```
    public void method2() {  
        System.out.println("----method2-----");  
    }
```

```
    public void method3() {  
        System.out.println("----method3-----");  
    }
```

```
}
```

```
interface Inter1 {
```

```
    public void method1();  
}
```

```
interface Inter2 {
```

```
    public void method2();  
}
```

```
interface Inter3 {
```

```
    public void method3();  
}
```

```
/*
```

```
interface Inter {
```

```
    public void method1();
```

```
    public void method2();
```

```
public void method3();  
}  
*/
```

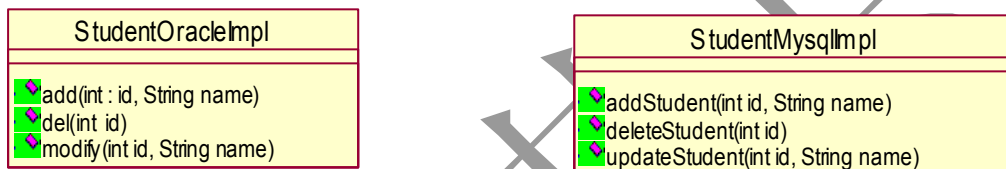
1.18、 接口的进一步应用

在 java 中接口其实描述了类需要做的事情，类要遵循接口的定义来做事，使用接口到底有什么本质的好处？可以归纳为两点：

- 采用接口明确的声明了它能提供的服务
- 解决了 Java 单继承的问题
- 实现了可插拔性（重要）

示例：完成学生信息的增删改操作，系统要求适用于多个数据库，如：适用于 Oracle 和 MySQL；

- 第一种方案，不使用接口，每个数据库实现一个类：



//Oracle 的实现

```
public class StudentOracleImpl {  
  
    public void add(int id, String name) {  
        System.out.println("StudentOracleImpl.add()");  
    }  
  
    public void del(int id) {  
        System.out.println("StudentOracleImpl.del()");  
    }  
  
    public void modify(int id, String name) {  
        System.out.println("StudentOracleImpl.modify()");  
    }  
  
}
```

需求发生了变化了，客户需要将数据移植 Mysql 上

////Mysql 的实现

```
public class StudentMysqlImpl {  
  
    public void addStudent(int id, String name) {  
        System.out.println("StudentMysqlImpl.addStudent()");  
    }  
  
    public void deleteStudent(int id) {
```



```

        System.out.println("StudentMysqlImpl.deleteStudent()");
    }

    public void ud pateStudent(int id, String name) {
        System.out.println("StudentMysqlImpl.ud pateStudent()");
    }
}

```

调用以上两个类完成向 Oracle 数据库和 Mysql 数据存储数据

```

public class StudentManager {

    public static void main(String[] args) {
        //对 Oracle 数据库的支持
        /*
        StudentOracleImpl studentOracleImpl = new StudentOracleImpl();
        studentOracleImpl.add(1, "张三");
        studentOracleImpl.del(1);
        studentOracleImpl.modify(1, "张三");
        */

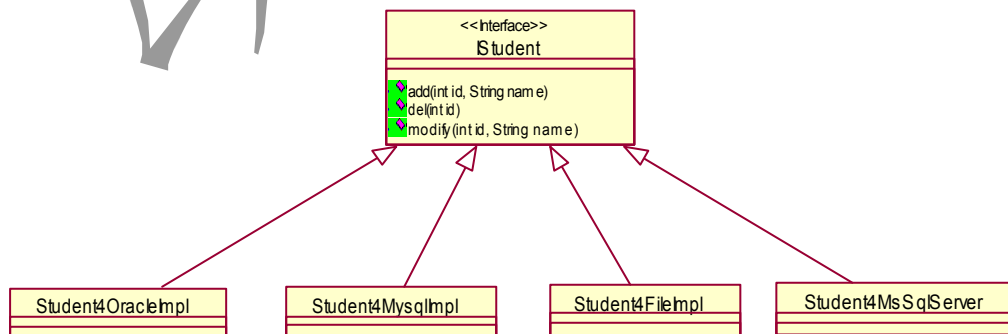
        //需要支持 Mysql 数据库
        StudentMysqlImpl studentMysqlImpl = new StudentMysqlImpl();
        studentMysqlImpl.addStudent(1, "张三");
        studentMysqlImpl.deleteStudent(1);
        studentMysqlImpl.ud pateStudent(1, "张三");
    }
}

```

以上代码不能灵活的适应需求，当需求发生改变需要改动的代码量太大，这样可能会导致代码的冗余，另外可能会导致项目的失败，为什么会导致这个问题，在开发中没有考虑到程序的扩展性，就是一味的实现，这样做程序是不行的，所以大的项目比较追求程序扩展性，有了扩展性才可以更好的适应需求。

- 第二种方案，使用接口

UML，统一建模语言



```

public class Student4OracleImpl implements IStudent {

    public void add(int id, String name) {

```

```
        System.out.println("Student4OracleImpl.add()");
    }

    public void del(int id) {
        System.out.println("Student4OracleImpl.del()");
    }

    public void modify(int id, String name) {
        System.out.println("Student4OracleImpl.modify()");
    }
}
```

```
public class Student4MysqlImpl implements IStudent {

    public void add(int id, String name) {
        System.out.println("Student4MysqlImpl.add()");
    }

    public void del(int id) {
        System.out.println("Student4MysqlImpl.del()");
    }

    public void modify(int id, String name) {
        System.out.println("Student4MysqlImpl.modify()");
    }
}
```

```
public class StudentService {

    public static void main(String[] args) {
        /*
        IStudent istudent = new Student4OracleImpl();
        IStudent istudent = new Student4MysqlImpl();
        istudent.add(1, "张三");
        istudent.del(1);
        istudent.modify(1, "张三");
        */
        //IStudent istudent = new Student4OracleImpl();
        //IStudent istudent = new Student4MysqlImpl();
        //doCrud(istudent);
        //doCrud(new Student4OracleImpl());
        //doCrud(new Student4MysqlImpl());

        //doCrud(new Student4OracleImpl());
        doCrud(new Student4MysqlImpl());
    }
}
```

```
}

//此种写法没有依赖具体的实现
//而只依赖的抽象，就像你的手机电池一样：你的手机只依赖电池（电池是一个抽象的事物），
//而不依赖某个厂家的电池(某个厂家的电池就是具体的事物了)
//因为你依赖了抽象的事物，每个抽象的事物都有不同的实现
//这样你就可以利用多态的机制完成动态绑定，进行互换，是程序具有较高的灵活
//我们尽量遵循面向接口（抽象）编程,而不要面向实现编程
public static void doCrud(IStudent istudent) {
    istudent.add(1, "张三");
    istudent.del(1);
    istudent.modify(1, "张三");
}

//以下写法不具有扩展性
//因为它依赖了具体的实现
//建议不要采用此种方法，此种方案是面向实现编程，就依赖于具体的东西了
/*
public static void doCrud(Student4OracleImpl istudent) {
    istudent.add(1, "张三");
    istudent.del(1);
    istudent.modify(1, "张三");
}
*/
}
```

1.19、多态

多态其实就是多种状态的含义，如我们方法重载，相同的方法名称可以完成不同的功能，这就是多态的一种表现，此时成为静态多态。另外就是我们将学生保存到数据的示例，当我们调用 IStudent 接口中的方法，那么 java 会自动调用实现类的方法，如果是 Oracle 实现就调用 Oracle 的方法，如果是 MySQL 实现就调用 MySQL 中的方法，这是在运行期决定的。多态的条件是：有继承或实现，有方法覆盖或实现，父类对象(接口)指向子类对象

1.20、接口和抽象类的区别？

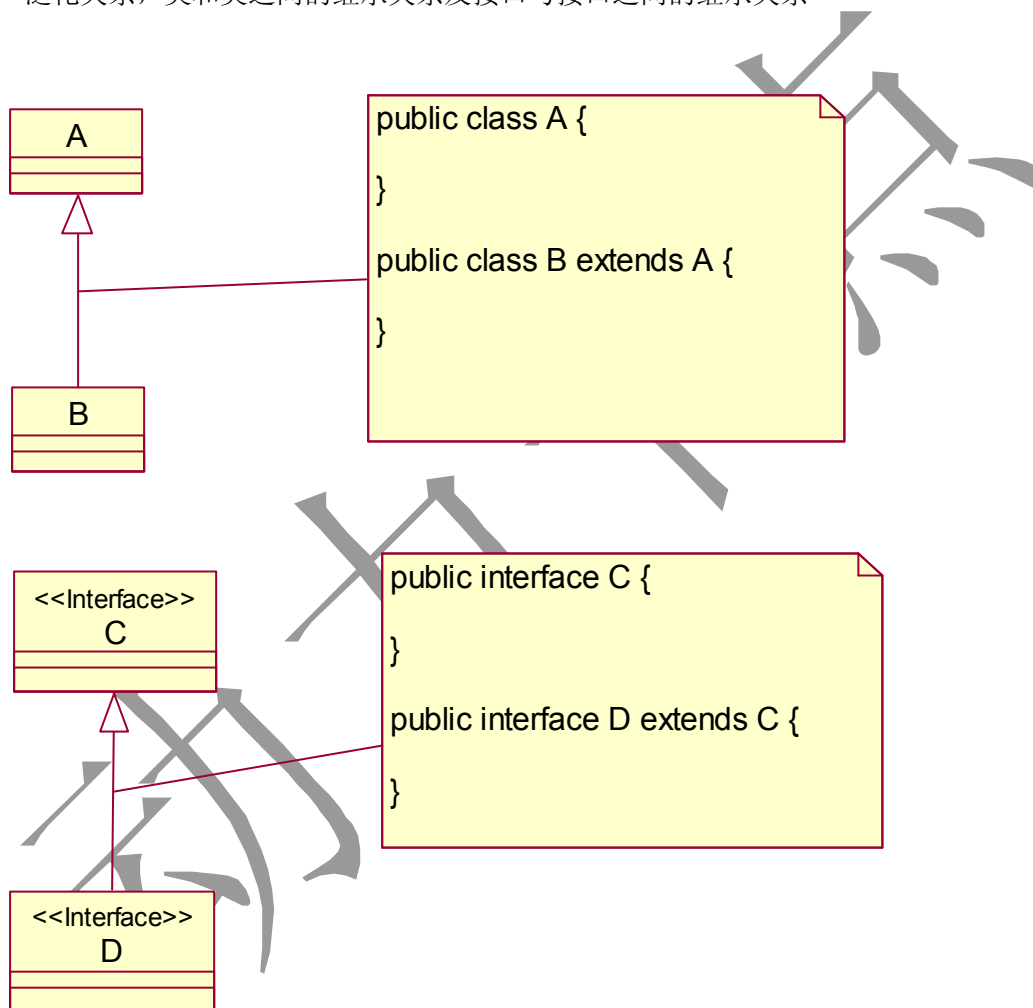
- a) 接口描述了方法的特征，不给出实现，一方面解决 java 的单继承问题，实现了强

大的可插拔性

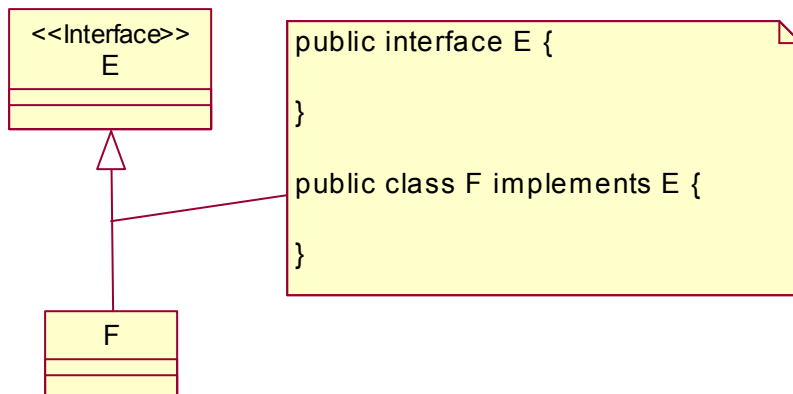
- b) 抽象类提供了部分实现，抽象类是不能实例化的，抽象类的存在主要是可以把公共的代码移植到抽象类中
- c) 面向接口编程，而不要面向具体编程（面向抽象编程，而不要面向具体编程）
- d) 优先选择接口（因为继承抽象类后，此类将无法再继承，所以会丧失此类的灵活性）

1.21、类之间的关系

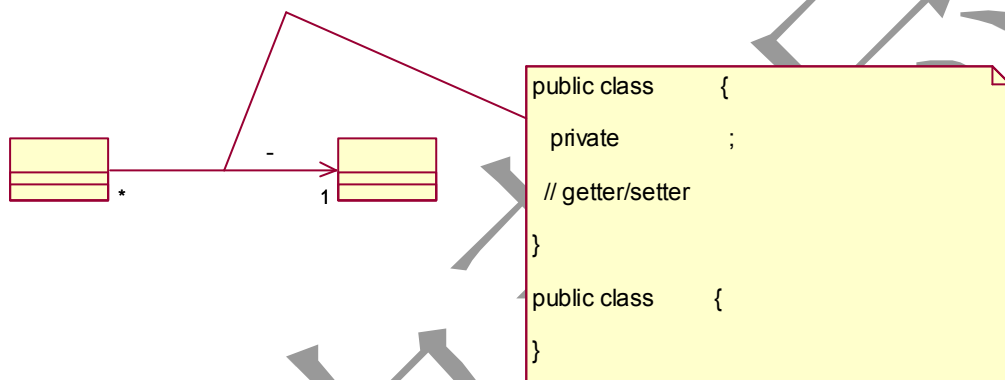
1. 泛化关系，类和类之间的继承关系及接口与接口之间的继承关系



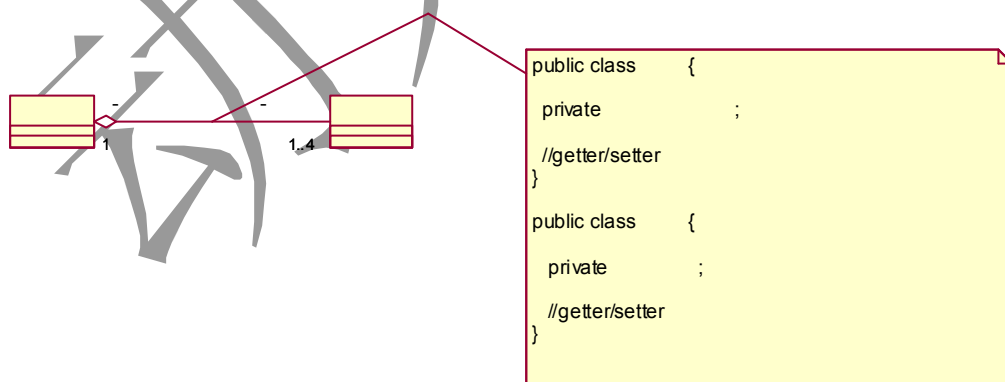
2. 实现关系，类对接口的实现



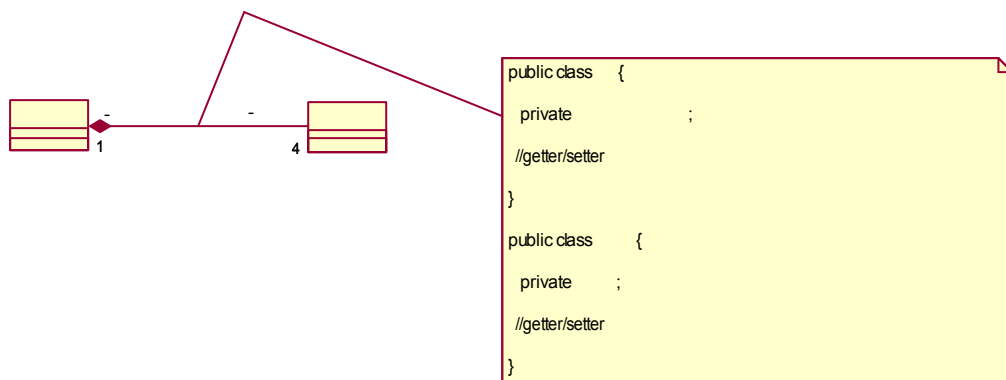
3. 关联关系，类与类之间的连接，一个类可以知道另一个类的属性和方法，在 java 语言中使用成员变量体现



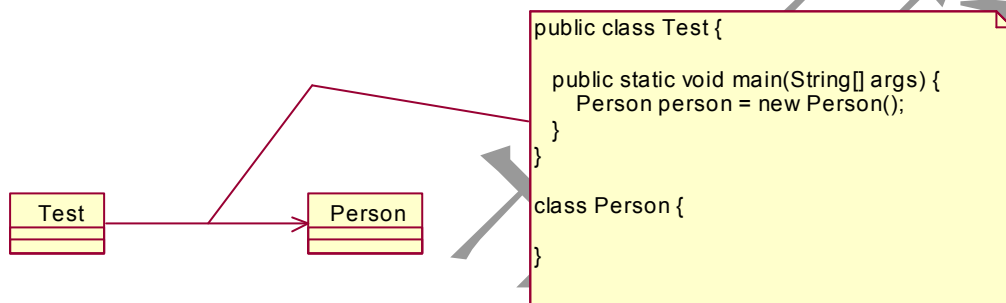
4. 聚合关系，是关联关系的一种，是较强的关联关系，是整体和部分的关系，如：汽车和轮胎，它与关联关系不同，关联关系的类处在同一个层次上，而聚合关系的类处在不平等的层次上，一个代表整体，一个代表部分，在 java 语言中使用实例变量体现



5. 合成关系，是关系的一种，比聚合关系强的关联关系，如：人和四肢，整体对象决定部分对象的生命周期，部分对象每一时刻只与一个对象发生合成关系，在 java 语言中使用实例变量体现



6. 依赖关系，依赖关系是比关联关系弱的关系，在 java 语言中体现为返回值，参数，局部变量和静态方法调用



1.22、is-a、is-like-a、has-a

● Is-a

```

public class Animal{
    public void method1();
}

public class Dog extends Animal { //Dog is a Animal
}
  
```

● is-like-a

```

public interface I {
    public void method1();
}

public class A implements I { //A is like a I;
    public void method1() {
        //实现
    }
}
  
```

● has-a

```
public class A {    //A has a B;
    private B b;

}

public class B {

}
```

1.23、Object 类

- a) Object 类是所有 Java 类的根基类
- b) 如果在类的声明中未使用 `extends` 关键字指明其基类，则默认基类为 Object 类如：

```
public class User {
    .....
}
相当于
public class User extends Object {
    .....
}
```

1.23.1、toString()

返回该对象的字符串表示。通常 `toString` 方法会返回一个“以文本方式表示”此对象的字符串，Object 类的 `toString` 方法返回一个字符串，该字符串由类名加标记 `@` 和此对象哈希码的无符号十六进制表示组成，Object 类 `toString` 源代码如下：

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

在进行 String 与其它类型数据的连接操作时，如：

`System.out.println(student);`，它自动调用该对象的 `toString()` 方法

【代码示例】

```
public class ToStringTest01 {

    public static void main(String[] args) {
        int i = 100;
        System.out.println(100);

        Person person = new Person();
        person.id = 200;
        person.name = "张三";
    }
}
```

```
//会输出 Person@757aef
//因为它调用了 Object 中的 toString 方法
//输出的格式不友好，无法看懂
System.out.println(person);

    }
}

//class Person extends Object { //和以下写法等同
class Person{

    int id;

    String name;

}
```

【代码示例】，覆盖 Person 中的 toString 方法

```
public class ToStringTest02 {

    public static void main(String[] args) {
        Person person = new Person();
        person.id = 200;
        person.name = "张三";

        //System.out.println(person.toString());

        //输出结果为：{id=200, name=张三}
        //因为 println 方法没有带 Person 参数的
        //而 Person 是 Object，所以他会调用 println(Object x)方法
        //这样就是产生 object 对其子类 Person 的指向，而在 Person 中
        //覆盖了父类 Object 的 toString 方法，所以运行时会动态绑定
        //Person 中的 toString 方法，所以将会按照我们的需求进行输出
        System.out.println(person);

    }
}

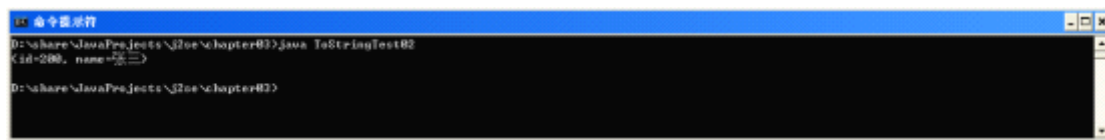
//class Person extends Object { //和以下写法等同
class Person{

    int id;
```



```
String name;

public String toString() {
    return "{id=" + id + ", name=" + name + "}";
}
}
```



1.23.2、finalize

垃圾回收器（Garbage Collection），也叫 **GC**，垃圾回收器主要有以下特点：

- 当对象不再被程序使用时，垃圾回收器将会将其回收
- 垃圾回收是在后台运行的，我们无法命令垃圾回收器**马上**回收资源，但是我们可以告诉他，尽快回收资源（**System.gc** 和 **Runtime.getRuntime().gc()**）
- 垃圾回收器在回收某个对象的时候，首先会调用该对象的 **finalize** 方法
- **GC 主要针对堆内存**
- **单例模式的缺点**

当垃圾收集器将要收集某个垃圾对象时将会调用 **finalize**，建议不要使用此方法，因为此方法的运行时间不确定，如果执行此方法出现错误，程序不会报告，仍然继续运行

```
public class FinalizeTest01 {

    public static void main(String[] args) {
        Person person = new Person();
        person.id = 1000;
        person.name = "张三";

        //将 person 设置为 null 表示，person 不再执行堆中的对象
        //那么此时堆中的对象就是垃圾对象
        //垃圾收集（GC）就会收集此对象
        //GC 不会马上收集，收集时间不确定
        //但是我们可以告诉 GC，马上来收集垃圾，但也不确定，会马上来
        //也许不会来
        person = null;

        //通知垃圾收集器，来收集垃圾
        System.gc();
        /*
        try {
```

```
        Thread.sleep(5000);
    } catch (Exception e) {
    }
    */
}

}

class Person {

    int id;

    String name;

    //此方法垃圾收集器会调用
    public void finalize() throws Throwable {
        System.out.println("Person.finalize()");
    }
}
```

注意以下写法

```
public class FinalizeTest02 {

    public static void main(String[] args) {
        method1();
    }

    private static void method1() {
        Person person = new Person();
        person.id = 1000;
        person.name = "张三";

        //这种写法没有多大的意义，
        //执行完成方法，所有的局部变量的生命周期全部结束
        //所以堆区中的对象就变成垃圾了（因为没有引用指向对象了）
        //person = null;
    }
}

class Person {

    int id;

    String name;
```

```
public void finalize() throws Throwable {  
    System.out.println("Person.finalize()");  
}  
}
```

2.23.3、==与 equals 方法

■ 等号 “==”

等号可以比较基本类型和引用类型，等号比较的是值，特别是比较引用类型，比较的是引用的内存地址

```
public class EqualsTest01 {  
  
    public static void main(String[] args) {  
        int a = 100;  
        int b = 100;  
  
        //可以成功比较  
        //采用等号比较基本它比较的就是具体的值  
        System.out.println((a == b)?"a==b":"a!=b");  
  
        Person p1 = new Person();  
        p1.id = 1001;  
        p1.name = "张三";  
  
        Person p2 = new Person();  
        p2.id = 1001;  
        p2.name="张三";  
  
        //输出为 p1!=p2  
        //采用等号比较引用类型比较的是引用类型的地址（地址也是值）  
        //这个是不符合我们的比较需求的  
        //我们比较的应该是对象的具体属性，如：id 相等，或 id 和 name 相等  
        System.out.println((p1 == p2)?"p1==p2":"p1!=p2");  
  
        Person p3 = p1;  
  
        //输出为 p1==p3  
        //因为 p1 和 p3 指向的是一个对象，所以地址一样  
        //所以采用等号比较引用类型比较的是地址  
        System.out.println((p1 == p3)?"p1==p3":"p1!=p3");  
  
        String s1 = "abc";  
        String s2 = "abc";  
    }  
}
```

```
//输出 s1==s2
System.out.println((s1==s2)? "s1==s2": "s1!=s2");

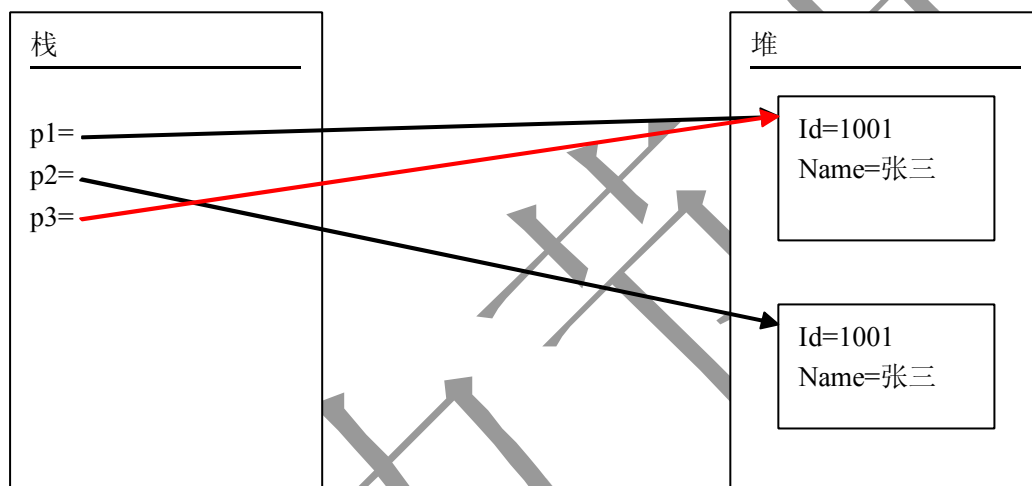
}
}

class Person{

    int id;

    String name;

}
```



■ 采用 equals 比较两个对象是否相等

```
public class EqualsTest02 {

    public static void main(String[] args) {
        String s1 = "abc";
        String s2 = "abc";
        //输出 s1==s2
        System.out.println((s1==s2)? "s1==s2": "s1!=s2");

        String s3 = new String("abc");
        String s4 = new String("abc");
        System.out.println((s3==s4)? "s3==s4": "s3!=s4");

        //输出 s3 等于 s4，所以确定 string 的 equals 比较的是具体的内容
        System.out.println(s3.equals(s4)? "s3 等于 s4": "s3 不等于 s4");

        Person p1 = new Person();
        p1.id = 1001;
    }
}
```

```
p1.name = "张三";

Person p2 = new Person();
p2.id = 1001;
p2.name = "张三";

//输出: p1 不等于 p2
//因为它默认调用的是 Object 的 equals 方法
//而 Object 的 equals 方法默认比较的就是地址, Object 的 equals 方法代码如下:
//    public boolean equals(Object obj) {
//    return (this == obj);
//    }
//如果不准备调用父类的 equals 方法, 那么必须覆盖父类的 equals 方法行为
System.out.println(p1.equals(p2)? "p1 等于 p2": "p1 不等于 p2");
}
}

class Person{

    int id;

    String name;

}
```

在进一步完善

```
public class EqualsTest03 {

    public static void main(String[] args) {

        Person p1 = new Person();
        p1.id = 1001;
        p1.name = "张三";

        Person p2 = new Person();
        p2.id = 1001;
        p2.name = "张三";

        System.out.println(p1.equals(p2)? "p1 等于 p2": "p1 不等于 p2");

    }

}

class Person{

    int id;

    String name;

}
```

```
//覆盖父类的方法
//加入我们自己的比较规则
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    //确定比较类型为 person
    //同一类型，才具有可比性
    if (obj instanceof Person) {
        //强制转换，必须实现知道该类型是什么
        Person p = (Person)obj;
        //如果 id 相等就认为相等
        if (this.id == p.id) {
            return true;
        }
    }
    return false;
}
```

以上输出完全正确，因为执行了我们自定义的 equals 方法，按照我们的规则进行比较的，注意 instanceof 的使用，注意强制转换的概念。将父类转换成子类叫做“**向下转型**（造型）”，向下造型是不安全的。“**向上转型**（造型）”是安全，子类转换成父类，如：将 Student 转成 Person,如 Dog 转成动物

1.24、包和 import

1.24.1、包

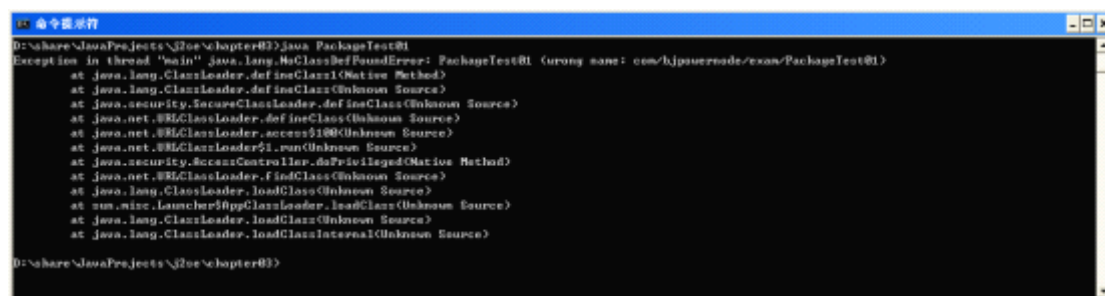
包其实就是目录，特别是项目比较大，java 文件特别多的情况下，我们应该分目录管理,在 java 中称为分包管理，包名称**通常采用小写**

```
/*
    1、包最好采用小写字母
    2、包的命名应该有规则，不能重复，一般采用公司网站逆序，
    如：com.bjpowernode.项目名称.模块名称
    com.bjpowernode.exam
*/

//package 必须放到 所有语句的第一行，注释除外

package com.bjpowernode.exam;
```

```
public class PackageTest01 {  
  
    public static void main(String[] args) {  
        System.out.println("Hello Package!!!");  
    }  
}
```



```
D:\share\javaProjects\202\chapter03>java PackageTest01  
Exception in thread "main" java.lang.NoClassDefFoundError: PackageTest01 (wrong name: com/bjpowernode/exam/PackageTest01)  
    at java.lang.ClassLoader.defineClass(Native Method)  
    at java.lang.ClassLoader.defineClass(Unknown Source)  
    at java.security.SecureClassLoader.defineClass(Unknown Source)  
    at java.net.URLClassLoader.defineClass(Unknown Source)  
    at java.net.URLClassLoader.access$100(Unknown Source)  
    at java.net.URLClassLoader$1.run(Unknown Source)  
    at java.security.AccessController.doPrivileged(Native Method)  
    at java.net.URLClassLoader.findClass(Unknown Source)  
    at java.lang.ClassLoader.loadClass(Unknown Source)  
    at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)  
    at java.lang.ClassLoader.loadClass(Unknown Source)  
    at java.lang.ClassLoader.loadClassInternal(Unknown Source)  
D:\share\javaProjects\202\chapter03>
```

运行出现类不能找到错误，提示给的很明显，如 `com.bjpowernode.exam.PackageTest01` 类不能找到，因为我们加入了包，所以我们的 `class` 文件必须放到和包一样的目录里才可以，这就是采用包来管理类，也就是采用目录来管理类，建立目录 `com/bjpowernode/exam`，采用 `java PackageTest01` 执行，同样出现上面的错误，如果采用了包在执行该类时必须加入完整的包名称，正确的执行方式为 `java com.bjpowernode.exam.PackageTest01`



```
D:\share\javaProjects\202\chapter03>java com.bjpowernode.exam.PackageTest01  
Hello Package!!!  
D:\share\javaProjects\202\chapter03>
```

正确执行通过。

另外还有一点需要注意：必须在最外层包采用 `java` 来执行，也就是 `classpath` 必须设置在 `chapter03` 目录上，以 `chapter03` 目录为起点开始找我们的 `class` 文件

1.24. 2、import

如何使用包下的 `class` 文件

```
package com.bjpowernode.exam;  
  
//采用 import 引入需要使用的类  
//import com.bjpowernode.exam.model.User;  
  
//import com.bjpowernode.exam.model.Student;  
//import com.bjpowernode.exam.model.Employee;  
  
//可以采用 * 通配符引入包下的所有类  
//此种方式不明确，但简单  
import com.bjpowernode.exam.model.*;  
  
//package 必须放到 所有语句的第一行，注释除外
```

```
//package com.bjpowernode.exam;

public class PackageTest02 {

    public static void main(String[] args) {
        User user = new User();
        user.setUserId(10000);
        user.setUserName("张三");

        System.out.println("user.id=" + user.getUserId());
        System.out.println("user.name=" + user.getUserName());
    }
}
```

如果都在同一个包下就不需要 import 引入了，以上的示例都没有包，可以理解为都在同一个包下，在实际开发过程中不应该这样做，必须建立包

1.24.3、JDK 常用开发包

- java.lang，此包 Java 语言标准包，使用此包中的内容无需 import 引入
- java.sql，提供了 JDBC 接口类
- java.util，提供了常用工具类
- java.io，提供了各种输入输出流

1.25、访问控制权限

java 访问级别修饰符主要包括：private protected 和 public，可以限定其他类对该类、属性和方法的使用权限，

修饰符	类的内部	同一个包里	子类	任何地方
private	Y	N	N	N
缺省	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

注意以上对类的修饰只有：public 和 default，内部类除外

1.24.1、private

【示例代码】

```
public class PrivateTest01 {

    public static void main(String[] args) {
        A a = new A();
        //不能访问，private 声明的变量或方法，只能在同一个类中使用
    }
}
```



```
        System.out.println(a.id);
    }
}

class A {

    private int id;

}
```

1.24.2、protected

【代码实例】，在同一个包下，建立类 ProtectedTest01、A，并建立 B 继承 A

```
public class ProtectedTest01 {

    public static void main(String[] args) {

        A a = new A();
        a.method1();
        A b = new B();
        b.method1();

        B b1 = new B();
        b1.method3();

    }

}

class A {

    //采用 protected 声明的变量或方法只有子类或同一个包下的类可以调用
    protected int id = 100;

    public void method1() {

        System.out.println(id);

    }

    protected void method2() {

        System.out.println("A.method2()");

    }

}

class B extends A {

    public void method1() {
```

```
//可以正确调用
//因为和 A 在同一个包下
System.out.println(id);
}

public void method3() {
    //可以正确调用
    method2();
}
}
```

【代码示例】，在 test1 下建立类 C1，在 test2 下建立 ProtectedTest02 和 C2 类

```
package test2;

import test1.C1;

public class ProtectedTest02 {

    public static void main(String[] args) {
        new C2().method2();
    }
}

class C2 extends C1 {

    public void method2() {
        //可以调用，主要原因 C2 是 C1 的子类
        //所以可以访问 protected 声明的变量
        System.out.println(id);
        method1();
    }
}

class C3 {

    public void method3() {
        //不能在其他类中访问 protected 声明的方法或变量
        //new C1().method1();
    }
}
```

1.24.3、default

如果 class 不采用 public 修饰，那么此时的 class，只能被该包下的类访问，其他包下无法访问

```
import test3.D;

public class DefaultTest01 {

    public static void main(String[] args) {
        D d = new D();
        d.method1();

        //不能访问，只有在一个类中或在同一个包下可以访问
        //在子类中也不能访问
        d.method2();
    }
}
```

1.26、内部类

在一个类的内部定义的类，称为内部类

内部类主要分类：

- 实例内部类
- 局部内部类
- 静态内部类

1.26.1、实例内部类

- 创建实例内部类，外部类的实例必须已经创建
- 实例内部类会持有外部类的引用
- 实例内部不能定义 `static` 成员，只能定义实例成员

```
public class InnerClassTest01 {

    private int a;

    private int b;

    InnerClassTest01(int a, int b) {
        this.a = a;
        this.b = b;
    }

    //内部类可以使用 private 和 protected 修饰
    private class Inner1 {
```

```
int i1 = 0;
int i2 = 1;

int i3 = a;
int i4 = b;

//实例内部类不能采用 static 声明
//static int i5 = 20;
}

public static void main(String[] args) {
    InnerClassTest01.Inner1 inner1 = new InnerClassTest01(100, 200).new Inner1();
    System.out.println(inner1.i1);
    System.out.println(inner1.i2);
    System.out.println(inner1.i3);
    System.out.println(inner1.i4);
}
}
```

1.23.2、静态内部类

- 静态内部类不会持有外部的类的引用，创建时不用创建外部类
- 静态内部类可以访问外部的静态变量，如果访问外部类的成员变量必须通过外部类的实例访问

【示例代码】

```
public class InnerClassTest02 {

    static int a = 200;

    int b = 300;

    static class Inner2 {
        //在静态内部类中可以定义实例变量
        int i1 = 10;
        int i2 = 20;

        //可以定义静态变量
        static int i3 = 100;

        //可以直接使用外部类的静态变量
        static int i4 = a;
    }
}
```

```
//不能直接引用外部类的实例变量
//int i5 = b;

//采用外部类的引用可以取得成员变量的值
int i5 = new InnerClassTest02().b;

}

public static void main(String[] args) {
    InnerClassTest02.Inner2 inner = new InnerClassTest02.Inner2();
    System.out.println(inner.i1);
}
}
```

1.26.3、局部内部类

局部内部类是在方法中定义的，它只能在当前方法中使用。和局部变量的作用一样
局部内部类和实例内部类一致，不能包含静态成员

```
public class InnerClassTest03 {

    private int a = 100;

    //局部变量，在内部类中使用必须采用 final 修饰
    public void method1(final int temp) {
        class Inner3 {
            int i1 = 10;

            //可以访问外部类的成员变量
            int i2 = a;

            int i3 = temp;
        }

        //使用内部类
        Inner3 inner3 = new Inner3();

        System.out.println(inner3.i1);
        System.out.println(inner3.i3);
    }

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

```
        InnerClassTest03 innerClassTest03 = new InnerClassTest03();
        innerClassTest03.method1(300);
    }
}
```

1.26.4、匿名内部类

是一种特殊的内部类，该类没有名字

- 没有使用匿名类

```
public class InnerClassTest04 {

    public static void main(String[] args) {
        MyInterface myInterface = new MyInterfaceImpl();
        myInterface.add();
    }
}

interface MyInterface {

    public void add();
}

class MyInterfaceImpl implements MyInterface {

    public void add() {
        System.out.println("-----add-----");
    }
}
```

- 使用匿名类

```
public class InnerClassTest05 {

    public static void main(String[] args) {
        /*
        MyInterface myInterface = new MyInterface() {
            public void add() {
                System.out.println("-----add-----");
            }
        };
        myInterface.add();
        */
    }
}
```

```
/*
MyInterface myInterface = new MyInterfaceImpl();
InnerClassTest05 innerClassTest05 = new InnerClassTest05();
innerClassTest05.method1(myInterface);
*/

InnerClassTest05 innerClassTest05 = new InnerClassTest05();
innerClassTest05.method1(new MyInterface() {
    public void add() {
        System.out.println("-----add-----");
    }
});

}

private void method1(MyInterface myInterface) {
    myInterface.add();
}
}

interface MyInterface {

    public void add();
}

/*
class MyInterfaceImpl implements MyInterface {

    public void add() {
        System.out.println("-----add-----");
    }
}
*/
```