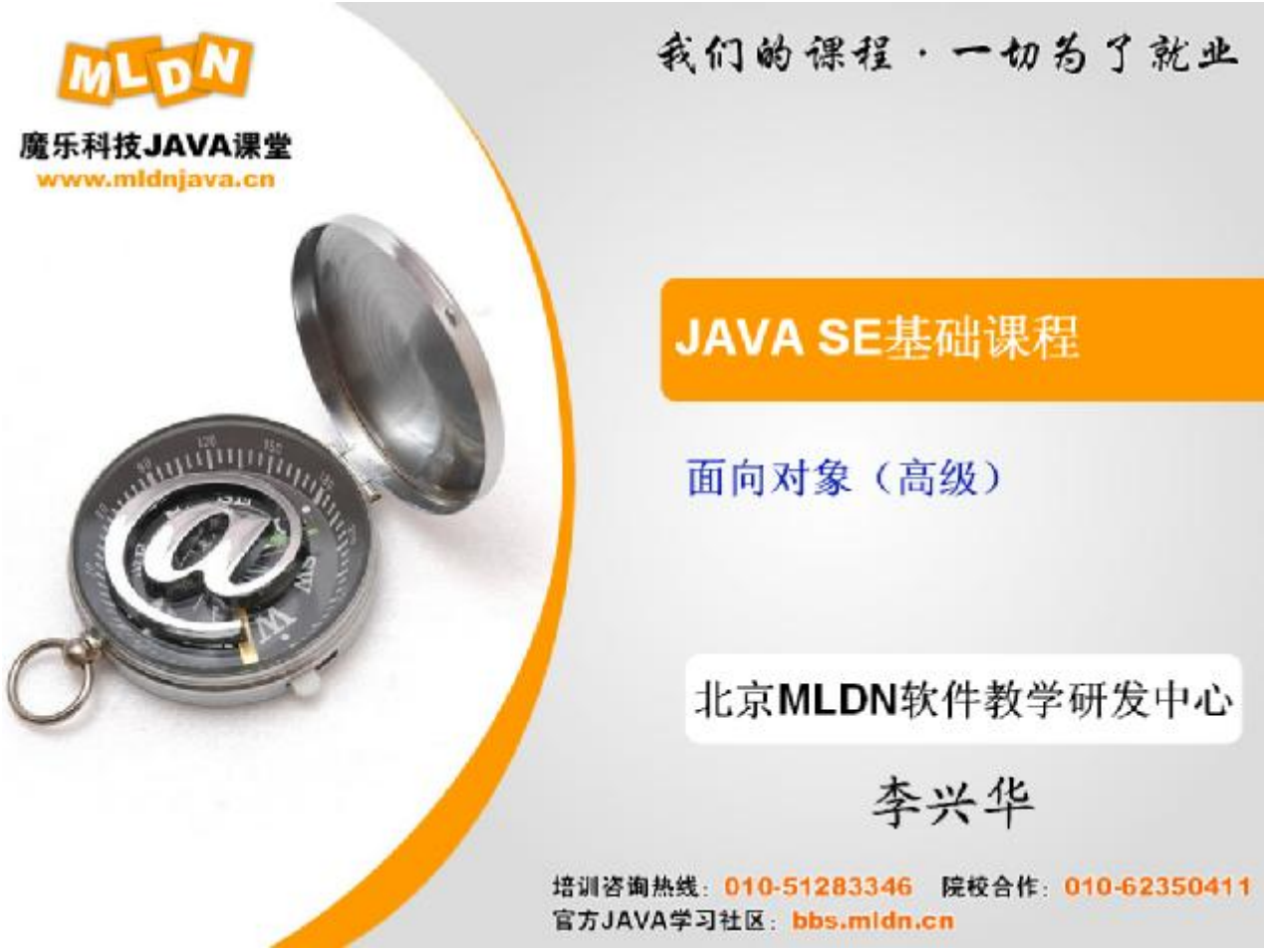
1、课程名称：面向对象（高级）



2、知识点

2.1、上次课程的主要知识点

1、 this 关键字：
- 可以访问类中的属性；
- 调用方法，如果调用本类中其他构造的时候，一定要放在首行，且至少留有一个构造方法作为出口；
- 当前对象：表示当前正在调用类中方法的对象。

2、 static 关键字：
- static 声明的属性是类变量，所有对象所共拥有，可以由类名称直接调用；
- static 声明方法，不能调用非 static 操作，可以由类名称直接调用。
- 主方法的组成：public static void main(**String args[]**)

3、 内部类：

- 定义在一个类内部的类，被包含的类称为内部类，包含的类称为外部类。
- 内部类可以直接访问外部类中的私有操作，但是会破坏原有程序的结构。
- 内部类可以被 static 修饰，修饰之后的类成为外部类
- 要是在类外部进行实例化，使用外部类.内部类
- 如果没有被修饰，需要先实例化外部类对象，再实例化内部类对象
- 可以在一个方法中定义一个内部类，这个内部类可以直接访问外部类中的私有操作，如果要访问方法中的变量，这些变量需要使用 final 来修饰。

2.2、题目讲解

链表程序：完全就是引用传递及 this 关键字的使用。

```
class Link {       // 这属于一个链表的操作类
    class Node {      // 每一个保存的数据都被封装在节点之中
        private String data ;    // 保存的数据，现在只是 String 类型
        private Node next ;            // 保存下一个节点的引用
        public Node(String data){
            this.data = data ; // 通过构造传递具体的内容
        }
        public void addNode(Node newNode){
            if(this.next == null){   // 当前节点的后面没有节点
                this.next = newNode ;
            } else {
                this.next.addNode(newNode) ;
            }
        }
        public void printNode(){
            System.out.println(this.data) ;      // 输出当前节点的内容
            if(this.next != null){   // 当前节点下还有节点
                this.next.printNode() ;
            }
        }
        public boolean existsNode(String data){ // 至少存在要查找的数据
            if(data.equals(this.data)){
                return true ;
            } else {
                if(this.next != null){
                    return this.next.existsNode(data) ;
                }
            }
            return false ;
        }
        public void deleteNode(Node previous,String data){
            if(data.equals(this.data)){    // 满足，则要删除
```

```
                    previous.next = this.next ;    // 空出当前节点
                } else {
                    if(this.next != null){
                        this.next.deleteNode(this,data) ;
                    }
                }
            }
        };
        private Node root ;                    // 设置根节点
        public void add(String data){
            Node newNode = new Node(data) ;
            if(this.root == null){    // 判断根节点是否为空
                this.root = newNode ; // 第一个节点为根节点
            } else {
                this.root.addNode(newNode) ;
            }
        }
        public boolean exists(String data){// 判断是否存在
            return this.root.existsNode(data) ;
        }
        public void delete(String data){    // 删除节点
            if(this.exists(data)){    // 如果节点存在则删除
                if(data.equals(this.root.data)){
                    this.root = this.root.next ;    // 根节点后的下一个节点
                } else {
                    this.root.next.deleteNode(this.root,data) ;
                }
            }
        }
        public void print(){    // 输出节点
            this.root.printNode() ;
        }
};
public class LinkDemo {
    public static void main(String args[]){
        Link link = new Link() ;
        link.add("A") ;
        link.add("B") ;
        link.add("C") ;
        link.add("D") ;
        link.add("E") ;
        link.delete("A") ;
        link.delete("B") ;
        link.print() ;
```

```
    }
};
```

2.3、本次预计讲解的知识点

1、 程序的开发
- 最基本的核心就是普通类，也是在以后的开发中使用最多的
- 对于 static 属性而言，当类中的某个属性需要被所有对象所共同拥有的时候才会定义。
- 如果需要为对象实例化的时候可以考虑有构造方法
- 程序需要被封装
- 而且对于传递参数的命名最好要有意义，而且最好和类中的属性一一对应，那么就需要使用 this.属性

2、 继承的基本概念及实现、继承的各个使用限制
3、 子类对象的实例化过程
4、 方法的覆写
5、 super 和 this 关键字的作用
6、 final 关键字的作用
7、 抽象类和接口的基本概念
8、 对象多态性

3、具体内容

3.1、继承（重点）

继承性是面向对象的第二大特征，主要的作用是扩充已有类的功能。

3.1.1、继承关系的引出

例如：下面要定义两个类：Person、Student，按照最早的概念，则此时的两个类定义成如下形式：

```
class Person {
    private String name ;
    private int age ;
    public void setName(String name){
        this.name = name ;
    }
    public void setAge(int age){
        this.age = age ;
    }
    public String getName(){
        return this.name ;
    }
```

```
class Student {
    private String name ;
    private int age ;
    private String school ;
    public void setName(String name){
        this.name = name ;
    }
    public void setAge(int age){
        this.age = age ;
    }
    public void setSchool(String school){
        this.school = school ;
```

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

```
    }
    public String getName(){
        return this.name ;
    }
    public int getAge(){
        return this.age ;
    }
    public String getSchool(){
        return this.school ;
    }
};
```

从以上的代码很明显的可以发现，有大量的重复代码出现，而且通过实际的问题可以发现，学生本身就是一个人。

面向对象开发的目的就是为了消除掉重复的代码，但是按照之前的编写套路很明显，已经不能满足于这种现实问题，所以下面就需要采用继承的形式，语法如下：

```
class 子类 extends 父类{}
```

但是需要说明的是，有时候父类也称为超类（super class），子类也称为派生类。

```
class Person {
    private String name ;
    private int age ;
    public void setName(String name){
        this.name = name ;
    }
    public void setAge(int age){
        this.age = age ;
    }
    public String getName(){
        return this.name ;
    }
    public int getAge(){
        return this.age ;
    }
    public String getInfo(){
        return "姓名：" + this.name + "，年龄：" + this.age ;
    }
};
class Student extends Person {      // 现在没有编写任何代码
};
public class ExtDemo02 {
    public static void main(String args[]){
        Student stu = new Student() ;
        stu.setName("张三") ;
        stu.setAge(20) ;
        System.out.println(stu.getInfo()) ;
    }
```

```
};
```

现在的 Student 类继承了 Person 类之后，发现可以将 Person 类中定义的方法任意的使用，子类也允许对父类中的定义进行扩充。

```
class Person {
    private String name ;
    private int age ;
    public void setName(String name){
        this.name = name ;
    }
    public void setAge(int age){
        this.age = age ;
    }
    public String getName(){
        return this.name ;
    }
    public int getAge(){
        return this.age ;
    }
    public String getInfo(){
        return "姓名：" + this.name + "，年龄：" + this.age ;
    }
};
class Student extends Person {      // 现在没有编写任何代码
    private String school ; // 扩充的属性
    public void setSchool(String school){
        this.school = school ;
    }
    public String getSchool(){
        return this.school ;
    }
};
public class ExtDemo03 {
    public static void main(String args[]){
        Student stu = new Student() ;
        stu.setName("张三") ;
        stu.setAge(20) ;
        stu.setSchool("清华大学") ;  // 此方法为子类扩充
        System.out.println(stu.getInfo()) ;
        System.out.println("学校：" + stu.getSchool()) ;
    }
};
```

从本程序中可以非常的清楚的发现，子类允许对父类进行扩充，所以，继承的基本作用就是扩充已有类的功能。

3.1.2、继承的限制

1、 子类可以继承父类的全部操作（属性、方法），但是对于所有的公共操作是可以直接继承的，而所有的私有操作是无法直接进程的，而是通过其他的方式间接访问。

2、 一个子类只能继承一个父类，属于单继承，而不能同时继承多个父类

```
class A {
};
class B {
};
class C extends A,B {
};
```

而以上的一个类同时继承属于多重继承，这个在 java 中是不允许出现的。

3、 在 Java 中允许多层继承。

```
class A {
};
class B extends A {
};
class C extends B {
};
```

此时的 C 类将继承 A 和 B 类中的全部操作。

3.1.3、子类对象的实例化

从一个实际的问题可以发现，在日常生活中肯定是先有父亲，再有孩子。

```
class Person {
    private String name ;
    private int age ;
    public Person(String name,int age){
    }
};
class Student extends Person {      // 现在没有编写任何代码
    private String school ; // 扩充的属性
};
public class ExtDemo06 {
    public static void main(String args[]){
        Student stu = new Student() ;
    }
};
```

此时在编译的时候出现了以下的问题：

```
ExtDemo06.java:7: cannot find symbol
symbol  : constructor Person()
location: class Person
```

```
class Student extends Person {  // 现在没有编写任何代码
^
1 error
```

此时提示的是没有找到 Person 类的无参构造方法，现在是子类出现的错误，因为一个类中一旦构造方法被调用之后，实际上就意味着此类可以使用了，对象已经产生了，就好比一个人一样已经出生了。

所以按照正常的思维来讲，肯定是先有父类产生再有子类产生，所以在子类对象实例化的时候实际上都会默认去调用父类中的无参构造方法。

```
class Person {
    private String name ;
    private int age ;
    public Person(){
        System.out.println("** 父类的无参构造！") ;
    }
    public Person(String name,int age){
    }
};
class Student extends Person {    // 现在没有编写任何代码
    private String school ; // 扩充的属性
    public Student(){
        System.out.println("** 子类的无参构造！") ;
    }
};
public class ExtDemo06 {
    public static void main(String args[]){
        Student stu = new Student() ;
    }
};
```

以上的代码的运行效果符合于现实生活中的场景，但是对于子类构造方法而言，实际上在构造方法中隐含了一条 super 语句：

```
    public Student(){
        super() ;  // 调用父类中的无参构造
        System.out.println("** 子类的无参构造！") ;
    }
```

既然在子类中可以通过 super 关键字调用父类中的无参构造，那么就一定可以通过 super 调用父类中的有参构造。

```
class Person {
    private String name ;
    private int age ;
    public Person(String name,int age){
        System.out.println("** 父类的无参构造！") ;
    }
};
class Student extends Person {    // 现在没有编写任何代码
    private String school ; // 扩充的属性
    public Student(String name,int age,String school){
```

```
            super(name,age) ;     // 调用父类中的构造
            System.out.println("** 子类的无参构造！") ;
        }
};
public class ExtDemo07 {
        public static void main(String args[]){
            Student stu = new Student("张三",30,"清华大学") ;
        }
};
```

也就是说，不管子类如何操作，肯定都会调用父类中的构造方法。

3.2、覆写（重点）

继承本身可以进行类的功能扩充，但是扩充之后也会存在问题，例如：在子类中定义了和父类完全一样的方法或者是定义了一样的属性，那么此时实际上就发生了覆写操作。

3.2.1、方法的覆写

所谓方法的覆写就是指一个子类中定义了一个与父类完全一样的方法名称，包括返回值类型、参数的类型及个数都是完全一样的，但是需要注意的是，被覆写的方法不能拥有比父类更严格的访问控制权限。

关于访问控制权限已经接触过三种：private< default（不写） < public

范例：观察方法的覆写

```
class A {
        public void print(){     // 定义方法
            System.out.println("hello") ;
        }
};
class B extends A {
        public void print(){
            System.out.println("world") ;
        }
};
public class OverrideDemo01 {
        public static void main(String args[]){
            B b = new B() ;
            b.print() ;
        }
};
```

在子类中一旦方法被覆写之后，实际上最终调用的方法就是被覆写过的方法，但是，如果此时在子类中的方法的访问权限已经降低的话，则编译的时候将出现错误：

```
OverrideDemo01.java:7: print() in B cannot override print() in A; attempting to assign weaker access privileges; was public
            void print(){
                 ^
```

一定要记住，方法名称、返回值了性、参数列表完全一样就称为覆写。

问题？如果现在父类中有一个方法是 private 声明，子类将其方法定义成了 default 访问权限，其他不变，是否叫覆写呢？

```
class A {
    public void print(){    // 定义方法
        this.getInfo() ;
    }
    private void getInfo(){
        System.out.println("A --> getInfo()") ;
    }
};
class B extends A {
    void getInfo(){
        System.out.println("B --> getInfo()") ;
    }
};
public class OverrideDemo02 {
    public static void main(String args[]){
        B b = new B() ;
        b.print() ;
    }
};
```

记住使用 private 声明的方法子类是无法覆写的，虽然语法编译上不会产生任何的问题，但是子类中被“覆写”过的方法永远无法找到，而且这种代码在实际中没有任何的意义。

如果现在子类中需要调用父类中已经被子类覆写过的方法，可以通过 super 关键字完成。

```
class A {
    public void print(){    // 定义方法
        System.out.println("hello") ;
    }
};
class B extends A {
    public void print(){
        super.print() ;    // 直接从父类中找到 print()方法
        System.out.println("world") ;
    }
};
public class OverrideDemo03 {
    public static void main(String args[]){
        B b = new B() ;
        b.print() ;
    }
};
```

super 就是直接由子类找到父类中指定的方法，而如果使用的是 this 的话，则会先从本类查找，如果查找到了就直接使用，查找不到，则再去父类中查找，super 是直接从父类中查找。

3.2.2、属性的覆盖（了解）

当子类声明了与父类完全一样的变量名称时，称为覆盖，不过这种概念基本上属于无意义的。

```
class A {
	String name = "HELLO" ;
};
class B extends A {
	int name = 30 ;
	public void print(){
		System.out.println(super.name) ;
		System.out.println(name) ;
	}
};
public class OverrideDemo04 {
	public static void main(String args[]){
		B b = new B() ;
		b.print() ;
	}
};
```

因为属性是必须封装的，所以此时的代码根本就没有任何的实际意义。

3.2.3、属性的应用

例如：之前讲解的 Person 和 Student 类。

```
class Person {
	private String name ;
	private int age ;
	public Person(){}
	public Person(String name,int age){
		this.name = name;
		this.age = age ;
	}
	public void setName(String name){
		this.name = name ;
	}
	public void setAge(int age){
		this.age = age ;
	}
	public String getName(){
		return this.name ;
	}
	public int getAge(){
		return this.age ;
```

```
    }
    public String getInfo(){
        return "姓名：" + this.name + "，年龄：" + this.age ;
    }
};
class Student extends Person {      // 现在没有编写任何代码
    private String school ; // 扩充的属性
    public Student(){}
    public Student(String name,int age,String school){
        super(name,age) ;
        this.school = school ;
    }
    public void setSchool(String school){
        this.school = school ;
    }
    public String getSchool(){
        return this.school ;
    }
    public String getInfo(){
        return super.getInfo() + "，学校：" + this.school ;
    }
};
public class OverrideDemo05 {
    public static void main(String args[]){
        Student stu = new Student("张三",20,"清华大学") ;
        System.out.println(stu.getInfo()) ;
    }
};
```

方法的覆写实际上就是根据不同的子类，同一个方法可以完成不同的功能。

3.3、两个重要概念（重点）

3.3.1、重载及覆写的区别

No.	区别点	重载	覆写
1	定义	方法名称相同，参数的类型或个数不同	方法名称、参数的类型或个数、返回值相同
2	权限	没有权限要求	被覆写的方法不能拥有比父类更严格的权限
3	范围	发生在一个类之中	发生在继承关系中
4	单词	OverLoading	Override

3.3.2、this 与 super 的区别

No.	区别点	this	super
1	使用	调用本类中的属性或方法	从子类调用父类中的属性或方法
2	构造	可以调用本类构造，且有一个构造要作为出口	从子类调用父类的构造方法，不管子类如何安排最终一定会去调用，默认调用的是父类中的无参构造方法
3	要求	调用构造的时候一定要放在构造方法首行	放在子类构造方法首行
		使用 super 和 this 调用构造方法的语句是不可能同时出现的	
4	特殊	表示当前对象	无此概念

3.4、思考题（重点）

现在要求定义一个数组类（Array），里面定义了一个整型数组，但是此整型数组属于动态分配大小，即：所有的大小由程序指定，并在此基础上实现以下的两个子类：

- 反转类：可以将数组的内容反转显示
- 排序类：可以对数组进行排序的显示

问：这样的类该如何实现？

```
class Array {
    private int num[] ;
    private int foot ; // 表示添加的下标
    public Array(int len){ // 通过构造方法传递数组的大小
        if(len > 0){
            this.num = new int[len] ;    // 根据外部指定开辟空间
        } else {
            this.num = new int[1] ;// 至少维持一个大小
        }
    }
    public boolean add(int temp){      // 向数组中添加内容
        if(this.foot < this.num.length){    // 还有地方
            this.num[this.foot++] = temp ;
            return true ;
        } else {
            return false ;
        }
    }
    public int[] getData(){ // 取得全部的数据
        return this.num ;
    }
};
class SortArray extends Array {
    public SortArray(int len){
```

```
        super(len) ;
    }
    public int[] getData(){
        java.util.Arrays.sort(super.getData()) ;
        return super.getData() ;
    }
};
class ReverseArray extends Array {
    public ReverseArray(int len){
        super(len) ;
    }
    public int[] getData(){
        int center = super.getData().length / 2 ;  // 求出中心点
        int head = 0 ;     // 开始点
        int tail = super.getData().length - 1 ;
        for(int x=0;x<center;x++){
            int temp = super.getData()[head] ;
            super.getData()[head] = super.getData()[tail] ;
            super.getData()[tail] = temp ;
            head ++ ;
            tail -- ;
        }
        return super.getData() ;
    }
};
public class ExecDemo {
    public static void main(String asgs[]){
        ReverseArray arr = new ReverseArray(4) ;
        System.out.println(arr.add(6)) ;
        System.out.println(arr.add(3)) ;
        System.out.println(arr.add(8)) ;
        System.out.println(arr.add(1)) ;
        int temp[] = arr.getData() ;
        for(int x=0;x<temp.length;x++){
            System.out.println(temp[x]) ;
        }
    }
};
```

实际代码中操作永远以父类的操作为标准。

3.5、final 关键字（重点）

在 Java 中可以使用 final 关键字定义类、方法、属性：

- 使用 final 关键字定义的类不能有子类

- 使用 final 声明的方法不能被子类所覆写
- 使用 final 声明的变量即成为常量，常量必须在声明时给出具体的内容。

范例：使用 final 声明类

```
final class A {
};
class B extends A {
};
```

范例：使用 final 声明方法

```
class A {
    public final void print(){}
};
class B extends A {
    public void print(){}
};
```

范例：使用 final 声明的变量就是常量

```
class A {
    public final String INFO = "hello" ;
    public void fun(){
        INFO = "wor" ;
    }
};
```

在声明一个常量的时候所有的单词的字母都必须采用大写的形式出现，另外需要提醒的是，如果要想声明一个全局常量的话：public static final 声明。public static final String INFO = "hello" ;

3.6、单例设计（理解）

```
class Singleton {
    private static final Singleton instance = new Singleton() ; // 在内部准备好一个对象
    public static Singleton getInstance(){    // 将 instance 传递到外部去
        return instance ;
    }
    private Singleton(){}
    public void print(){
        System.out.println("Hello World!!!") ;
    }
};
public class Test{
    public static void main(String args[]){
        Singleton s1 = Singleton.getInstance() ;
        Singleton s2 = Singleton.getInstance() ;
        Singleton s3 = Singleton.getInstance() ;
        s1.print() ;
        s2.print() ;
```

```
        s3.print() ;
    }
};
```

此时，不管外部如何变化，Singleton 类中永远只会有一个实例化对象，此种代码实现的根本原理就是在于将一个类的构造方法关闭了。

当一个类中只能产生一个实例化对象的时候，就需要将构造方法封闭，封闭之后的操作通过一个静态方法取得本类的实例化对象，这种代码的概念非常重要，而且代码的结构必须清楚。

编写 Singleton 类的话，就按照此种代码完整写出即可。

如果要想继续划分的，实际上单例设计，还分成两种类型：

- 懒汉式：当第一次使用本类的对象时，在进行对象的实例化操作。
- 饿汉式：一个单例类中不管是否使用，都始终维护一个实例化对象。

```
class Singleton {
    private static Singleton instance = null ;
    public static Singleton getInstance(){    // 将 instance 传递到外部去
        if(instance == null){
            instance = new Singleton() ;
        }
        return instance ;
    }
    private Singleton(){}
    public void print(){
        System.out.println("Hello World!!!") ;
    }
};
public class Test{
    public static void main(String args[]){
        Singleton s1 = Singleton.getInstance() ;
        Singleton s2 = Singleton.getInstance() ;
        Singleton s3 = Singleton.getInstance() ;
        s1.print() ;
        s2.print() ;
        s3.print() ;
    }
};
```

既然程序中可以存在单例设计，那么就可以存在多例设计。

```
class Color {
    private static final Color RED = new Color("红色") ;
    private static final Color GREEN = new Color("绿色") ;
    private static final Color BLUE = new Color("蓝色") ;
    private String name ;
    public static Color getInstance(int ch){  // 将 instance 传递到外部去
        if(ch==0){
            return RED ;
        } else if(ch==1){
```

```
            return GREEN ;
        } else if(ch==2){
            return BLUE ;
        } else {
            return null ;
        }
    }
    private Color(String name){
        this.name = name ;
    }
    public void print(){
        System.out.println("当前颜色：" + this.name) ;
    }
};
public class Test{
    public static void main(String args[]){
        Color s1 = Color.getInstance(0) ;
        Color s2 = Color.getInstance(1) ;
        Color s3 = Color.getInstance(2) ;
        s1.print() ;
        s2.print() ;
        s3.print() ;
    }
};
```

当一个类需要提供多个实例化对象的时候就采用如上的方式编写，不过一般这种情况使用的不多。

3.7、抽象类（重点）

抽象类和接口是整个 JAVA 面向对象的核心部分，但是要想充分理解此概念，就必须结合对象多态性，那么先来看一下基本的语法概念。

抽象类的定义比较简单，包含一个抽象方法的类就是抽象类，抽象类必须使用 abstract 关键字进行声明。

抽象方法：只声明而未定义方法体的方法称为抽象方法，抽象方法也必须使用 abstract 关键字声明。

范例：定义一个抽象类

```
abstract class Demo {  // 抽象类
    public void print(){
        System.out.println("Hello World!!!") ;
    }
    public abstract void fun() ;  // 抽象方法
};
```

从类的结构上可以清楚的发现，抽象类只是比普通类多了几个抽象方法而已，其他的定义结构都是一样的，但是一个抽象类却不能直接使用：

```
public class AbsDemo01 {
    public static void main(String args[]){
```

```
        Demo demo = null ;
        demo = new Demo() ;
        demo.print() ;
    }
};
```

此时编译的时候提示：由于 Demo 是一个抽象类，所以无法进行实例化，所以抽象类使用有以下原则：

1、 抽象类不能直接实例化。

2、 抽象类必须有子类，子类（如果不是抽象类）的话，则必须覆写抽象类中的全部抽象方法。

3、 如果一个抽象类中没有任何一个抽象方法，依然是抽象类。

范例：定义子类

```
abstract class Demo {  // 抽象类
    public void print(){
        System.out.println("Hello World!!!") ;
    }
    public abstract void fun() ;  // 抽象方法
};
class DemoImpl extends Demo {
    public void fun(){}
};
public class AbsDemo02 {
    public static void main(String args[]){
        DemoImpl di = new DemoImpl() ;
        di.print() ;
    }
};
```

在抽象类的操作中，本身依然符合于单继承的问题，即：一个子类只能继承一个抽象类。

问题：

1、 抽象类能使用 final 声明吗？

- 不能：final 声明的类不能被继承，而抽象类又必须要有子类。

2、 抽象类中能有构造方法吗？

- 可以存在，而且依然符合于子类对象的实例化过程的要求。

```
abstract class Demo {  // 抽象类
    public Demo(){
        System.out.println("抽象类中的构造方法！") ;
    }
    public void print(){
        System.out.println("Hello World!!!") ;
    }
    public abstract void fun() ;  // 抽象方法
};
class DemoImpl extends Demo {
    public DemoImpl(){
        super() ;
        System.out.println("子类中的构造方法！") ;
```

```
    }
    public void fun(){}
};
public class AbsDemo03 {
    public static void main(String args[]){
        DemoImpl di = new DemoImpl() ;
        di.print() ;
    }
};
```

抽象类和普通类相比，只是增加了抽象 abstract class 的声明，和增加了抽象方法而已。

3.8、接口（重点）

当一个类中全部是由抽象方法和全局常量组成的时候，那么就可以将这个类定义成一个接口了，接口使用 interface 关键字声明。

范例：定义接口

```
interface Demo{ // 接口
    public static final String INFO = "hello world" ;
    public abstract void print() ;
    public abstract void fun() ;
}
```

一个接口定义完成之后，实际上与抽象类的使用原则是一样的：

1、 接口必须有子类，子类（如果不是抽象类）则必须覆写接口中的全部抽象方法；
2、 接口是不能直接进行对象的实例化操作。
3、 一个子类可以同时继承（实现）多个接口，如下所示：

```
class 子类 implements 接口 A,接口 B,…{}
```

范例：实现接口

```
interface Demo{ // 接口
    public static final String INFO = "hello world" ;
    public abstract void print() ;
    public abstract void fun() ;
}
interface Flag{
    public abstract void info() ;
}
class Temp implements Demo,Flag {    // 实现两个接口
    public void print(){
        System.out.println(INFO) ;
    }
    public void fun(){}
    public void info(){}
};
public class IntDemo01 {
```

```
    public static void main(String args[]){
        Temp temp = new Temp() ;
        temp.print() ;
    }
};
```

但是，一个子类一般而言如果要实现接口又要继承抽象类的话，则必须先继承抽象类之后再实现接口。

```
interface Demo{ // 接口
    public static final String INFO = "hello world" ;
    public abstract void print() ;
    public abstract void fun() ;
}
abstract class Flag {
    public abstract void info() ;
};
class Temp extends Flag implements Demo {
    public void print(){
        System.out.println(INFO) ;
    }
    public void fun(){}
    public void info(){}
};
public class IntDemo02 {
    public static void main(String args[]){
        Temp temp = new Temp() ;
        temp.print() ;
    }
};
```

既然接口中的全部组成都是抽象方法和全局常量的话，那么以下的两种定义接口的形式是完全一样的：

```
interface Demo{ // 接口
    public static final String INFO = "hello world" ;
    public abstract void print() ;
}
```

```
interface Demo{ // 接口
    String INFO = "hello world" ;
    void print() ;
}
```

所有的修饰符在接口中是否添加本身是没有任何意义的，而且接口中的方法全部都属于公共的方法操作（public）。

对于接口而言本身还有一个很大的特点：一个接口可以同时通过 extends 关键字继承多个接口。

```
interface A{
    public void printA() ;
}
interface B{
    public void printB() ;
}
interface C extends A,B{
    public void printC() ;
}
class Demo implements C {
```

```
    public void printA(){}
    public void printB(){}
    public void printC(){}
};
```

但是在 Java 之前存在了内部类的关系，实际上接口和抽象类本身也是可以这样操作的。

即：一个接口中可以定义其他的接口、抽象类或普通类，一个抽象类也可以定义内部的接口、类或抽象类。

```
interface A{
    public void printA() ;
    static interface B{     // 外部接口
        public void printB() ;
    }
    abstract class X {
        public abstract void printX() ;
    };
}
class D1 implements A {
    public void printA(){}
    class D2 implements B{
        public void printB(){}
    };
    class D3 extends X {
        public void printX(){}
    };
};
```

一般而言，以上的代码并不常见，而且开发中也几乎是见不到的，但是对于一个内部接口使用 static 声明之后成为外部接口这个概念还是在以后会应用到的。

3.9、对象多态性（重点）

在面向对象中多态性实际上是面向对象里的一个最大的最有用的特点，对于多态性在 java 中有两种体现：

1、 方法的重载及覆写

2、 对象多态性：指的是父类对象和子类对象之间的转型操作

- 向上转型（子类 à 父类）：父类名称 父类对象 = 子类实例 ； ，自动完成
- 向下转型（父类 à 子类）：子类名称 子类对象 = （子类名称）父类实例 ； ，强制完成

但是，在讲解之前，先来观察以下的一段代码。

```
class A {
    public void print(){
        System.out.println("A --> public void print(){}") ;
    }
    public void fun(){
        this.print() ;
    }
};
```

```
class B extends A {
    public void print(){
        System.out.println("B --> public void print(){}") ;
    }
};
public class PolDemo01 {
    public static void main(String args[]){
        A a = new B() ;  // 子类对象变为父类对象
        a.fun() ;
    }
};
```

所谓的向上转型，实际上指的就是一个子类变为父类接收，但是调用的方法肯定是被子类所覆写过的操作。

```
public class PolDemo02 {
    public static void main(String args[]){
        A a = new B() ;  // 子类对象变为父类对象
        B b = (B) a ;    // 父类对象变为子类对象
        b.fun() ;
    }
};
```

再次观察以下的代码：

```
public class PolDemo03 {
    public static void main(String args[]){
        A a = new A() ;  // 父类对象实例化
        B b = (B) a ;    // 父类对象变为子类对象
        b.fun() ;
    }
};
```

以上程序执行的时候将出现以下的错误：

```
Exception in thread "main" java.lang.ClassCastException: A cannot be cast to B
```

表示的是一个类转换异常，主要的功能是由于两个类之间没有任何的关系所发生的转换。

在进行向下转型之前必须首先发生向上转型的关系，建立关系。

但是，在对象多态性中也必须注意一点，虽然可以发生向上转型的关系，但是也要考虑到一点，一旦发生了向上转型后，子类中自己的定义的操作是无法通过父类对象找到的。

```
class A {
    public void print(){
        System.out.println("A --> public void print(){}") ;
    }
    public void fun(){
        this.print() ;
    }
};
class B extends A {
    public void print(){
        System.out.println("B --> public void print(){}") ;
```

```
    }
    public void printB(){
        System.out.println("Hello B") ;
    }
};
public class PolDemo04 {
    public static void main(String args[]){
        A a = new B() ;  // 父类对象实例化
        // a.printB() ;     // 错误的
        B b = (B) a ;     // 向下转型
        b.printB() ;
    }
};
```

而且在开发中一定要明白一点，所有的操作方法一定**要以父类所规定的方法为主**，子类最好不要任意的扩充。

下面通过一段代码来研究对象多态性所带来的好处：

要求：建立一个方法，此方法可以接收 A 类的任意子类的对象

1、 第一种：使用重载完成

```
class A {
    public void print(){
        System.out.println("A --> public void print(){}") ;
    }
    public void fun(){
        this.print() ;
    }
};
class B extends A {
    public void print(){
        System.out.println("B --> public void print(){}") ;
    }
};
class C extends A {
    public void print(){
        System.out.println("C --> public void print(){}") ;
    }
};
public class PolDemo05 {
    public static void main(String args[]){
        fun(new B()) ;
        fun(new C()) ;
    }
    public static void fun(B b){
        b.fun() ;
    }
    public static void fun(C c){
```

```
            c.fun() ;
        }
};
```

这种代码虽然完成了眼前的功能，但是本身是存在问题的，因为如果现在假设 A 类有 1000 个子类呢？肯定意味着此方法要重载 1000 次。

2、 第二种：采用对象多态性完成：因为所有的子类对象都可以自动向父类对象转换

```
public class PolDemo06 {
    public static void main(String args[]){
            fun(new B()) ;
            fun(new C()) ;
    }
    public static void fun(A a){
            a.fun() ;
    }
};
```

这两种实现方式比较起来可以清楚的发现，使用第二种代码形式，即便 A 类中存在了再多的子类，则程序也根本就不需要有任何的变化。

从本程序中可以发现一点：**父类的设计是最关键的**。

3.10、instanceof 关键字（重点）

通过 instanceof 关键字可以判断某一个对象是否是某一个类的实例。

```
class A {
};
class B extends A {
};
public class InstanceDemo {
    public static void main(String args[]){
            A a = new A() ;
            System.out.println(a instanceof A) ;
            System.out.println(a instanceof B) ;
    }
};
```

所以，以后在进行向下转型操作之前，一定要先使用 instanceof 关键字进行验证，验证通过了，才可以放心安全的执行向下转型的操作。

3.11、Object（重点）

3.11.1、基本概念

Object 类是所有类的父类，如果一个类定义的时候没有明确的继承一个类的话，则默认继承 Object 类。

范例：以下的两种类的定义结果是一样的

class Person { };	class Person **extends Object** { };

所以，在整个 Java 中实际上一切都属于继承关系，在 Object 类中定义了如下的几个方法（先讲主要的）：

No.	方法名称	类型	描述
1	public String toString()	普通	对象输出时调用
2	public boolean equals(Object obj)	普通	对象比较

如果是一个设计完整的类的话，肯定是要覆写 Object 类中的三个方法：toString()、equals()、hashCode()。

3.11.2、对象信息：toString()

之前强调过，如果直接打印一个对象的话，默认情况下打印的是一个对象的地址。

```
class Person {
};
public class ObjectDemo01{
    public static void main(String args[]){
        Person per = new Person() ;
        System.out.println(per.toString()) ;
    }
};
```

发现，现在一个对象直接输出和调用 toString()方法输出的最终效果是一样的，可以得出结论：一个对象打印时，默认调用的方法就是 toString()，那么这样一来，就可以根据子类自己的需要覆写 toString()方法，以输出合适的信息。

```
class Person {
    private String name ;
    private int age ;
    public Person(String name,int age){
        this.name = name ;
        this.age = age ;
    }
    public String toString(){
        return "姓名：" + this.name + "，年龄：" + this.age ;
    }
};
public class ObjectDemo01{
    public static void main(String args[]){
        Person per = new Person("张三",20) ;
        System.out.println(per.toString()) ;
    }
};
```

3.11.3、对象比较：equals()

在使用字符串的时候曾经就强调过，如果要进行两个字符串内容比较的话，肯定使用 equals()，那么 String 是 Object 的子类，所以在 String 类中已经覆写好了 equals()方法，那么之前也强调过对象比较的操作，那么正经来说，一个对象的

比较操作应该放在 equals()方法中完成。

```
class Person {
	private String name ;
	private int age ;
	public Person(String name,int age){
		this.name = name ;
		this.age = age ;
	}
	public boolean equals(Object obj){
		if(this == obj){
			return true ;
		}
		if(!(obj instanceof Person)){
			return false ;
		}
		Person per = (Person) obj ;
		if(this.name.equals(per.name) && this.age==per.age){
			return true ;
		} else {
			return false ;
		}
	}
	public String toString(){
		return "姓名：" + this.name + "，年龄：" + this.age ;
	}
};
public class ObjectDemo02{
	public static void main(String args[]){
		Person per1 = new Person("张三",20) ;
		Person per2 = new Person("张三",20) ;
		System.out.println(per1.equals(null)) ;
	}
};
```

只要是对象比较就一定要覆写 equals()方法。

3.11.4、接收引用（重点）

对于 Java 的数据类型来讲，分为基本数据类型和引用数据类型，使用 Object 不光可以接收类的对象，只要是引用数据类型的对象都可以接收（数组、类、接口）。

范例：接收数组

```
public class ObjectDemo03{
	public static void main(String args[]){
		int data[] = {1,2,3,4,5,6,7} ;
```

```
            Object obj = data ;
            int temp[] = (int[]) obj ;
            for(int x=0;x<temp.length;x++){
                System.out.println(temp[x]) ;
            }
        }
};
```

以上的代码还可以进行替换：

int data[] = {1,2,3,4,5,6,7} ; Object obj = data ;	Object obj = new int[] {1,2,3,4,5,6,7} ;

4、总结

1、 继承可以用来扩充已有类的功能，使用 extends 关键字，在 Java 中只允许单继承而不允许多继承，而且继承时实际上是将所有的内容都继承了下来，只是有些内容是显式继承，私有的属于隐式继承。

2、 子类对象的实例化：子类对象实例化之前先去调用父类的构造方法，之后再调用子类的构造方法。

3、 this 和 super 的区别；重载及覆写的区别。

4、 覆写：子类定义了一个与父类完全一样的方法，称为覆写，覆写的时候必须注意访问控制权限。

5、 final 关键字：定义的类不能被继承、定义的方法不能被覆写、定义的变量就是常量。

6、 抽象类：包含一个抽象方法的类称为抽象类，抽象类必须有子类，子类要覆写全部的抽象方法。

7、 接口：由抽象方法和全局常量组成的特殊类，一个类可以实现多个接口，接口可以实现多继承。

8、 对象多态性及 instanceof 关键字

9、 Object 类：所有类的父类

5、预习任务

巩固所有的面向对象的概念、巩固抽象类和接口、匿名内部类、链表、异常的捕获及处理、包及访问权限

6、作业

为了进一步帮助大家理解链表，留一道链表的复杂操作题目。

现在要求定义一个链表，里面可以保存任意类型的对象，链表必须实现以下的操作接口：

```
interface Link{   // 定义了与链表操作的相关方法
    public void add(Object data) ;      // 向链表增加数据

    public void add(Object dta[]) ;     // 可以增加一组对象

    public void delete(Object data) ;   // 向链表中删除数据
```

```
    public boolean exists(Object data) ;      // 判断数据是否存在

    public Object[] getAll() ;                // 取得全部的保存对象

    public Object get(int index) ;// 根据保存的位置取出指定对象

    public int length() ;                     // 求出链表的长度
}
```

实现以上接口完成链表的操作。

此时肯定是按照对象数组操作，所以实际上完成了一个动态的对象数组。