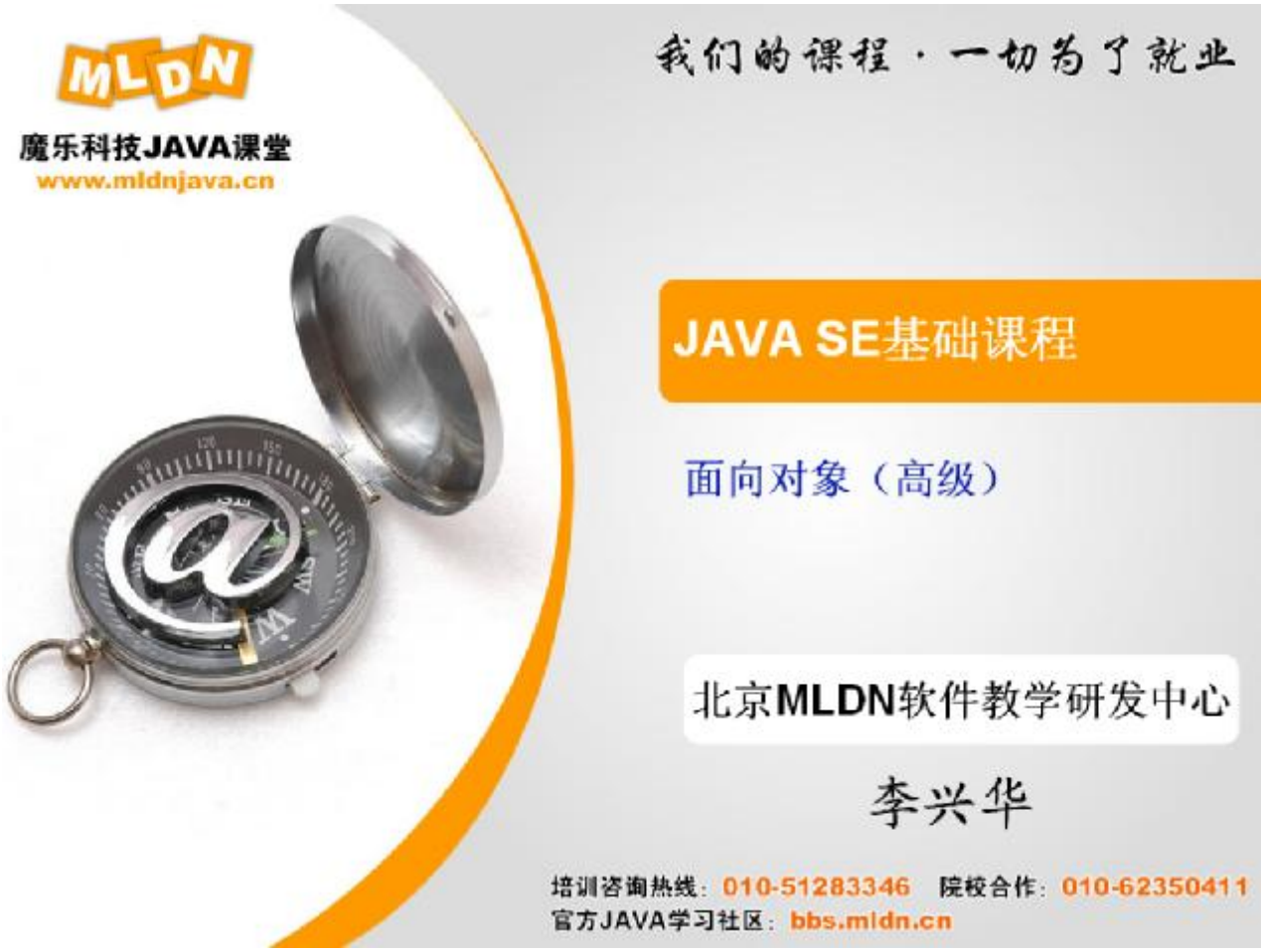
1、课程名称：面向对象（高级）



2、知识点

2.1、上次课程的主要知识点

1、 继承的实现：

- 概念：继承可以扩充已有类的功能
- 实现：class 子类 extends 父类{}，父类又称为超类，子类又称为派生类
- 限制：子类可以直接继承父类中的全部非私有操作，而只能隐式继承所有的私有操作，一个子类只能继承一个父类，但是允许多层继承。
- 子类对象的实例化过程：在进行子类对象实例化时，首先会先对父类对象进行实例化，调用父类中的构造方法，默认情况下调用的是父类中的无参构造方法，当然也可以通过 super 指定要调用的是那一个构造方法。

2、 重载与覆写的区别：

- 重载：发生在一个类之中，方法名称相同，参数的类型或个数不同；

- 覆写：发生在继承关系中，子类定义了一个与父类完全一样的方法，但是要注意方法的访问权限，即：被覆写的方法不能拥有比父类更严格的访问控制权限。

3、 this 与 super 关键字

- this 表示的是调用本类中的属性或方法，首先会从本类开始查找，如果找不到了，则再去父类中查找；而 super 表示直接调用父类中的属性或方法；
- this 与 super 调用构造方法的时候都要放在构造方法的首行，所以两者不能同时出现；
- this 可以表示当前对象，但是 super 无此概念。

4、 final 关键字：定义的类不能有子类，定义的方法不能被子类所覆写，定义的变量就称为常量，使用 public static final 声明的是全局常量。

5、 抽象类：包含一个抽象方法的类称为抽象类，抽象类必须使用 abstract 关键字声明，所有的抽象类不能直接实例化，而是需要通过子类继承，之后子类（如果不是抽象类）则要覆写全部的抽象方法。

6、 接口：抽象方法和全局常量的集合，称为接口，接口使用 interface 关键字进行声明，接口通过 implements 关键字被子类所实现，一个子类可以同时实现多个接口，接口也可以同时继承多个接口。

7、 对象多态性：

- 向上转型，子类对象变为父类实例，自动转型： 父类名称 父类对象 = 子类实例 ；
- 向下转型，将父类实例变为子类实例，强制转型： 子类名称 子类对象 = （子类名称）父类实例 ；
- 在进行向下转型之前一定要首先发生向上转型的关系
- 可以使用 instanceof 关键字判断某一个对象是否是某一个类的实例

8、 Object 类：是所有类的父类

- 所有的类默认继承自 Object 类
- 主要的方法：
  - |- toString()：对象输出时调用，用于输出对象的内容
  - |- equals()：对象比较的操作
- 使用 Object 类可以接收任意的引用数据类型的对象

9、 单例设计模式（**Singleton**）、多例设计。

2.2、本次预计讲解的知识点

1、 接口和抽象类的使用

2、 包装类的操作

3、 匿名内部类

4、 理解 JDK 1.5 之后增加的一些新特性

3、具体内容

之前的很多概念基本上已经描述了一些类的定义的结构，和一些注意特点，但是所有的概念实际上都是围绕接口进行的。

3.1、接口和抽象类（核心重点）

接口和抽象类的使用为整个 Java 面向对象的核心，包括以后讲解的 Java 应用部分，实际上都是完全围绕着接口和抽

象类的概念展开的，所以如果接口和抽象类不能理解的话，则整个的面向对象基本上就不会。

3.1.1、为抽象类实例化

抽象类中一定是要有子类的，而且子类一定要实现所有的抽象方法。

通过对象多态性的学习，可以发现，当一个父类通过子类实例化之后，调用的方法肯定是被子类所覆写过的方法。

```
abstract class Demo {
    public void fun(){
        System.out.println(this.getInfo()) ;
    }
    public abstract String getInfo() ;
};
class DemoImpl extends Demo {
    public String getInfo(){
        return "Hello World!!!" ;
    }
};
public class CaseDemo01 {
    public static void main(String args[]){
        Demo demo = new DemoImpl() ;  // 向上转型
        demo.fun() ;
    }
};
```

抽象类本身可以实例化，而且通过子类实例化之后就具备了子类的具体功能，同一种功能，会根据子类的不同而有所不同，完全由子类决定。

3.1.2、抽象类的使用 —— 模板设计（理解）

例如：现在可以将一个人的划分成学生和工人。

- 不管是工人还是学生肯定都有其共同的属性，例如：姓名、年龄。
- 但是，既然是一个类，肯定就拥有自己的信息，例如：学生有学校，工人有工作。

```
abstract class Person {
    private String name ;
    private int age ;
    public Person(String name,int age){
        this.name = name ;
        this.age = age ;
    }
    public void say(){
        System.out.println(this.getContent()) ;
    }
    public abstract String getContent() ;
    public String getName(){
```

```
        return this.name ;
    }
    public int getAge(){
        return this.age ;
    }
};
class Student extends Person {
    private String school ;
    public Student(String name,int age,String school){
        super(name,age) ;
        this.school = school ;
    }
    public String getContent(){
        return this.toString() ;
    }
    public String toString(){
        return "姓名：" + super.getName() + "，年龄：" + super.getAge() + "学校：" + this.school  ;
    }
};
class Worker extends Person {
    private String job ;
    public Worker(String name,int age,String job){
        super(name,age) ;
        this.job = job ;
    }
    public String getContent(){
        return this.toString() ;
    }
    public String toString(){
        return "姓名：" + super.getName() + "，年龄：" + super.getAge() + "工作：" + this.job  ;
    }
};
public class CaseDemo02 {
    public static void main(String args[]){
        // Person per = new Student("张三",20,"清华大学") ;
        Person per = new Worker("张三",20,"经理") ;
        per.say() ;
    }
};
```

这样的设计称为模板设计。

在实际的开发中，所有的类永远不要去继承一个已经实现好的类，而只能继承抽象类或实现接口。

3.1.3、为接口实例化

同样可以通过对象多态性完成接口对象的实例化操作。

```
interface Demo{
    public String INFO = "Hello World" ;
    public void print() ;
} ;
class DemoImpl implements Demo {
    public void print(){
        System.out.println(INFO) ;
    }
};
public class CaseDemo03 {
    public static void main(String args[]){
        Demo demo = new DemoImpl() ;
        demo.print() ;
    }
};
```

3.1.4、接口的使用 —— 制定标准（理解）

电脑上有 USB 接口，只要是 USB 设备都可以向电脑上插入并使用。

```
interface USB{   // 定义好了一个标准
    public void use() ;              // 使用
}
class Computer {
    public static void plugIn(USB usb){
        usb.use() ;
    }
};
class Flash implements USB {
    public void use(){
        System.out.println("使用 U 盘。") ;
    }
};
class Print implements USB {
    private String name ;
    public Print(String name){
        this.name = name ;
    }
    public void use(){
        System.out.println("欢迎使用" + this.name + "牌打印机！") ;
        System.out.println("开始打印！") ;
```

```
    }
};
public class CaseDemo04 {
    public static void main(String args[]){
        Computer.plugIn(new Flash()) ;
        Computer.plugIn(new Print("HP")) ;
    }
};
```

从实际的应用来看，接口主要有以下三大使用：

1、 制定标准；

2、 表示能力；

3、 将远程方法的操作视图暴露给客户端。

3.1.5、接口的使用 —— 工厂设计模式

下面先来观察以下的一段代码：

```
interface Fruit{
    public void eat() ;
}
class Apple implements Fruit {
    public void eat(){
        System.out.println("吃苹果。") ;
    }
};
class Orange implements Fruit {
    public void eat(){
        System.out.println("吃橘子。") ;
    }
};
public class CaseDemo05 {
    public static void main(String args[]){
        Fruit f = new Orange() ;
        f.eat() ;
    }
};
```

观察以上代码中存在的问题。

现在的程序中可以发现，在主方法（客户端）上，是通过关键字 new 直接为接口进行实例化，也就是说以后在使用的时候如果要不更改主方法的话，则主方法中永远只能使用一个类，这样的耦合度太深了。

回顾：Java 的可移植性原理：*.class **à** JVM **à** OS。

原本可以直接由 A **à** B 的情况，中间加入了 C：A **à** C **à** B。

```
interface Fruit{
    public void eat() ;
}
```

```
class Apple implements Fruit {
	public void eat(){
		System.out.println("吃苹果。") ;
	}
};
class Orange implements Fruit {
	public void eat(){
		System.out.println("吃橘子。") ;
	}
};
class Factory {
	public static Fruit getInstance(String className){
		Fruit f = null ;
		if("apple".equals(className)){
			f = new Apple() ;
		}
		if("orange".equals(className)){
			f = new Orange() ;
		}
		return f ;
	}
};
public class CaseDemo06 {
	public static void main(String args[]){
		Fruit f = Factory.getInstance(args[0]) ;
		f.eat() ;
	}
};
```

此时，中间加入了一个过渡端（Factory），那么都通过过渡端找到接口的实例，这样的设计称为工厂设计，以后扩充子类的时候修改工厂即可：即：某一局部的修改不影响其他环境。

3.1.6、接口的使用 —— 代理设计模式

现在，观察以下一种情况：一个代理人员可以代表一个真实的操作人员进行某些操作，但是两者的核心目的就是讨债。

```
interface Subject{
	public void give() ;
}
class RealSubject implements Subject{
	public void give(){
		System.out.println("真正的讨债者：还我的钱。") ;
	}
};
```

```
class ProxySubject implements Subject {
	private Subject sub ;  // 设置代理人
	public ProxySubject(Subject sub){
		this.sub = sub ;
	}
	public void before(){
		System.out.println("准备刀子，绳索，毒药。。。") ;
	}
	public void give(){
		this.before() ;
		this.sub.give() ;  // 真实主题
		this.after() ;
	}
	public void after(){
		System.out.println("跑路了。。。") ;
	}
};
public class CaseDemo07 {
	public static void main(String args[]){
		Subject s = new ProxySubject(new RealSubject()) ;
		s.give() ;
	}
};
```

在此设计之中可以发现，真实主题完成具体的业务操作，而代理主题将完成与真实主题有关的其他的操作。

3.1.7、接口的使用 —— 适配器设计（理解）

在正常情况下，一个接口的子类肯定是要覆写一个接口中的全部抽象方法。

那么有没有一种可能性，通过代码的变更，让一个子类可以有选择性的来覆写自己所需要的抽象方法呢？

从概念上讲这样肯定不合适，所以中间就想一想加入一个过渡端，但是这个过渡端又不能直接使用。

```
interface Fun{
	public void printA() ;
	public void printB() ;
	public void printC() ;
}
abstract class FunAdapter implements Fun {
	public void printA(){}
	public void printB(){}
	public void printC(){}
};
class Demo extends FunAdapter {
	public void printA(){
		System.out.println("Hello World!!!") ;
```

```
    }
};
```

一般在进行图形界面的开发中才会使用到适配器的设计思路。

3.1.8、抽象类和接口的区别（背下来）

接口和抽象类从使用上看非常的相似，那么下面通过以下的表格对两者进行区分：

No.	比较	抽象类	接口
1	关键字	使用 abstract class 声明	使用 interface 声明
2	定义	包含一个抽象方法的类	抽象方法和全局常量的集合
3	组成	属性、方法、构造、常量、抽象方法	全局常量、抽象方法
4	权限	抽象方法的权限可以任意	只能是 public 权限
5	使用	通过 extends 关键字继承抽象类	通过 implements 关键字实现接口
6	局限	抽象类存在单继承局限	没有此局限，一个子类可以实现多个接口
7	顺序	一个子类只能先继承抽象类再实现多个接口	
8	设计模式	模板设计	工厂设计、代理设计
		两者联合可以完成一个适配器设计	
9	实际作用	只能做一个模板使用	作为标准、表示能力
10	使用	两者没有什么本质的区别，但是从实际上来看，如果一个程序中抽象类和接口都可以使用的话，则一定要优先考虑接口，因为接口可以避免单继承所带来的局限。	
11	实例化	都是依靠对象多态性，通过子类进行对象实例化的	

观察一下以下程序的执行结果：

```
abstract class Demo {
    public Demo(){
        this.print() ;
    }
    public abstract void print() ;
};
class DemoImpl extends Demo {
    private int x = 10 ;
    public DemoImpl(int x){
        this.x = x ;
    }
    public void print(){
        System.out.println("x = " + this.x) ;
    }
};
public class CaseDemo09 {
    public static void main(String args[]){
        new DemoImpl(100) ;
    }
};
```

本程序的答案是 0，这道程序完全体现了子类对象实例化的过程。

3.2、匿名内部类（重点）

匿名内部类是在以后的框架开发中经常使用到的一种概念，但是对于一些简单的开发确实是暂时不用。

匿名内部类是在内部类及抽象类和接口的基础上发展起来的。

```
interface X{
    public void print() ;
}
class A implements X {
    public void print(){
        System.out.println("Hello World!!!") ;
    }
};
class Demo {
    public void fun(X x){
        x.print() ;
    }
    public void hello(){
        this.fun(new A()) ;
    }
};
public class NoInner {
    public static void main(String args[]){
        new Demo().hello() ;
    }
};
```

此时有一个新的问题，如果现在的 A 类只使用一次的话，那么有必要将其定义成一个具体的类吗？

此时，就可以通过匿名内部类来解决此问题。

```
interface X{
    public void print() ;
}
class Demo {
    public void fun(X x){
        x.print() ;
    }
    public void hello(final String info){
        this.fun(new X(){
            public void print(){
                System.out.println(info) ;
            }
        }) ;
    }
};
public class NoInner {
```

```
    public static void main(String args[]){
        new Demo().hello("Hello World!!!") ;
    }
};
```

3.3、包装类（重点）

在 Java 中一直提倡：一切皆对象。

但是，此时一个问题就出现了，基本数据类型不是类，所以为了可以让基本数据类型变为类的形式进行操作的话，在 Java 中引入了包装类的概念，可以通过包装类完成基本数据类型的封装。

在 Java 中一共定义了八种基本数据类型：int（Integer）、char（Character）、float（Float）、double（Double）、long（Long）、short（Short）、byte（Byte）、boolean（Boolean）

但是，需要注意的是，以上的八种包装类也分为两种类型：

- 数值型：指的是 **Number** 的子类，有：Integer、Float、Double、Long、Short、Byte
- 对象型：指的是 Object 的子类，有：Character、Boolean

在 Number 类中规定了一系列的 xxxValue()方法，可以将被包装的数字取出，变回基本数据类型。

3.3.1、装箱及拆箱

装箱：将一个基本数据类型变为包装类称为装箱操作，装箱的方法由各个子类完成

拆箱：将一个包装类变回基本数据类型，称为拆箱操作，转换的方法由 Number 类提供。

范例：以整型为例

```
public class WrapperDemo01 {
    public static void main(String args[]){
        int x1 = 10 ;       // 基本数据类型
        Integer temp = new Integer(x1) ;  // 装箱
        int x2 = temp.intValue() ;     // 拆箱
        System.out.println(x2 * x2) ;
    }
};
```

范例：以浮点型为例

```
public class WrapperDemo02 {
    public static void main(String args[]){
        float x1 = 10.3f ;
        Float temp = new Float(x1) ;
        float x2 = temp.floatValue() ;
        System.out.println(x2 * x2) ;
    }
};
```

对于以上的操作，实际上也是随着版本的不同而有所不同的，上面的代码是在 JDK 1.4 的时候使用的，所有的操作必须进行手工的装箱及拆箱操作，而且所有的包装类是不能直接进行各种计算的，但是到了 **JDK 1.5 之后，Java 向.NET 学习，增加了自动装箱和拆箱的操作**，而且所有的包装类可以自动的执行数学计算的功能。

范例：观察自动装箱和拆箱的操作

```
public class WrapperDemo03 {
    public static void main(String args[]){
        Float temp = 10.3f ;    // 自动装箱
        // Object obj = 10.3f ; // 基本 --> 包装 --> 变为 Object\
        // 先自动拆箱，之后计算再装箱
        System.out.println(temp * temp) ;
        float x = temp ;  // 自动拆箱
    }
};
```

3.3.2、数据转换

包装类的最大好处是可以将字符串中的数据变为基本数据类型，例如：将一个字符串变为 int 或者是 float 型的数据，使用的方法是各个包装类中提供的：static 数据类型 parse 数据类型() 。

```
public class WrapperDemo04 {
    public static void main(String args[]){
        String str = "133" ;
        int x = Integer.parseInt(str) ; // 将字符串边回基本数据类型
        System.out.println(x * x) ;
    }
};
```

但是需要注意的是，在使用以上方法操作的时候一定要小心，字符串中的内容必须完全由数字组成。

```
public class WrapperDemo05 {
    public static void main(String args[]){
        String str = "133" ;
        float x = Float.parseFloat(str) ;    // 将字符串边回基本数据类型
        System.out.println(x * x) ;
    }
};
```

3.3.3、题目

现在要求可以设计一个表示坐标的操作类（Point），此类中可以分别表示以下的三种坐标：

- 第一种： x=10，y=30
- 第二种： x=10.3，y=30.9
- 第三种： x="东经 110 度"，y="北纬 200 度"

问，此类该如何设计？

由于现在存在了三种数据类型的数据，所以为了保证接收，只能通过 Object 接收：

- int à Integer à Object
- float à Float à Object
- String à Object

```
class Point {
    private Object x ;
```

```
    private Object y ;
    public void setX(Object x){
        this.x = x ;
    }
    public void setY(Object y){
        this.y = y ;
    }
    public Object getX(){
        return this.x ;
    }
    public Object getY(){
        return this.y ;
    }
};
```

下面分别通过不同的数据类型对程序进行验证：

1、 使用 int 型

```
public class PointDemo {
    public static void main(String args[]){
        Point p = new Point() ;
        p.setX(10) ;        // int --> Integer --> Object
        p.setY(30) ;        // int --> Integer --> Object
        int x = (Integer) p.getX() ;    // Object --> Integer --> int
        int y = (Integer) p.getY() ;    // Object --> Integer --> int
        System.out.println("X 的坐标：" + x) ;
        System.out.println("Y 的坐标：" + y) ;
    }
};
```

2、 使用 float 型

```
public class PointDemo {
    public static void main(String args[]){
        Point p = new Point() ;
        p.setX(10.3f) ;
        p.setY(30.5f) ;
        float x = (Float) p.getX() ;
        float y = (Float) p.getY() ;
        System.out.println("X 的坐标：" + x) ;
        System.out.println("Y 的坐标：" + y) ;
    }
};
```

3、 使用 String 型

```
public class PointDemo {
    public static void main(String args[]){
        Point p = new Point() ;
        p.setX("东经 120 度") ;
```

```
        p.setY("北纬 100 度") ;
        String x = (String) p.getX() ;
        String y = (String) p.getY() ;
        System.out.println("X 的坐标：" + x) ;
        System.out.println("Y 的坐标：" + y) ;
    }
};
```

现在已经解决了本题目的要求，按照之前所学，现在肯定是最合理的解决方案，但是，本程序又存在了一个**安全隐患**，因为所有的类型都使用 Object 进行接收，那么有没有一种可能，将 X 的坐标设置成了整型，而 Y 的坐标设置成了字符串？

```
public class PointDemo {
    public static void main(String args[]){
        Point p = new Point() ;
        p.setX(100) ;
        p.setY("北纬 100 度") ;
        String x = (String) p.getX() ;
        String y = (String) p.getY() ;
        System.out.println("X 的坐标：" + x) ;
        System.out.println("Y 的坐标：" + y) ;
    }
};
```

那么，对于以上的问题，在 JDK 1.5 之后就可以解决了，因为 JDK 1.5 有一个最大的特点，就是加入了泛型的操作。

```
class Point <T> {
    private T x ;
    private T y ;
    public void setX(T x){
        this.x = x ;
    }
    public void setY(T y){
        this.y = y ;
    }
    public T getX(){
        return this.x ;
    }
    public T getY(){
        return this.y ;
    }
};
```

程序中的“<T>”就表示的是一种类型，只是这种类型现在属于未知的，在类使用的时候设置类型。

```
public class PointDemo {
    public static void main(String args[]){
        Point<String> p = new Point<String>() ;
        p.setX("东经 100 度") ;
        p.setY("北纬 100 度") ;
```

```
        String x = p.getX() ;
        String y = p.getY() ;
        System.out.println("X 的坐标：" + x) ;
        System.out.println("Y 的坐标：" + y) ;
    }
};
```

泛型中的类型可以由外部决定，但是在设置基本数据类型的时候，只能使用包装类。

```
public class PointDemo {
    public static void main(String args[]){
        Point<Integer> p = new Point<Integer>() ;
        p.setX(10) ;
        p.setY(20) ;
        int x = p.getX() ;
        int y = p.getY() ;
        System.out.println("X 的坐标：" + x) ;
        System.out.println("Y 的坐标：" + y) ;
    }
};
```

设置泛型之后，程序更加安全了，可以避免掉类转换异常的出现。

3.4、泛型（理解）

JDK 1.5 之后增加了很多的新特性，其中有三项是最重要的新特性：泛型、枚举、Annotation，但是这三项中并不能作为程序开发的重点。

泛型作为一个较大的新特性，在类库中使用较多，但是一般都是结合类集框架来看的，本次只是将一些基本的语法进行讲解，先来观察基本概念。

3.4.1、泛型的作用

泛型的主要目的是为了解决在进行类转换过程中发生的类转换异常的问题，用于处理安全隐患的。所以为了避免掉 ClassCastException 在 JDK 1.5 之后增加了泛型的操作，泛型的具体含义就是一个类中的某些属性的操作类型由使用此类的时候决定。

在设置烦型的时候是通过“<T>”的形式设置的，这里只是设置了一个标记，以后会根据设置的情况换成不同的类型，但是需要注意的是，为了保证 JDK 1.5 之前代码的使用正常，所以在泛型中也可以不设置类型，如果不设置的话，称为擦除泛型，将全部使用 Object 进行接收。

当然，也可以同时设置多个泛型类型，例如：

```
class Point <T,K,V> {}
```

3.4.2、通配符

泛型确实可以保证程序避免安全隐患问题，但是程序中一旦使用了泛型之后对于引用传递上又会存在问题。

```
class Info<T> {
```

```
    private T content ;
    public void setContent(T content){
        this.content = content ;
    }
    public T getContent(){
        return this.content ;
    }
};
public class GenDemo01 {
    public static void main(String args[]){
        Info<String> info = new Info<String>() ;
        info.setContent("Hello World") ;
        fun(info) ;
    }
    public static void fun(Info<String> temp){
        System.out.println(temp.getContent()) ;
    }
};
```

以上的程序非常的容易，但是这个时候有一个问题出现了，如果现在设置的泛型类型不是 String 呢？

由于 fun()方法上设置的 Info 只能接收 String，所以肯定无法传递，那么如果现在在 fun()方法中不写呢？

```
public class GenDemo01 {
    public static void main(String args[]){
        Info<Integer> info = new Info<Integer>() ;
        info.setContent(30) ;
        fun(info) ;
    }
    public static void fun(Info temp){
        System.out.println(temp.getContent()) ;
    }
};
```

如果这样写的话，则意味着，以后可以通过 fun()方法向 Info 的对象中设置任意样的内容。

```
    public static void fun(Info temp){
        temp.setContent("Hello") ;
        System.out.println(temp.getContent()) ;
    }
```

这种操作是存在安全隐患的。那么如果写成如下的形式？

```
    public static void fun(Info<Object> temp){
        System.out.println(temp.getContent()) ;
    }
```

这种做法肯定没意义，而且不能编译通过，那么该如何修饰呢？此时关键性的问题，是如何解决调用设置的问题，而不是取得属性输出的问题，所以说在泛型中增加了一个“?”表示的是接收所有的泛型类型，而且一旦接收之后只能取得，不能设置。

```
public class GenDemo01 {
    public static void main(String args[]){
```

```
        Info<Integer> info = new Info<Integer>() ;
        info.setContent(30) ;
        fun(info) ;
    }
    public static void fun(Info<?> temp){
        System.out.println(temp.getContent()) ;
    }
};
```

但是一个新的问题又出现了，如果现在的 Info 中的 content 属性只能是数字，不能是其他的任意一种类型呢？

所有的数字的包装类，都是 Number 的子类，所以这种条件下泛型只能设置 Number 或 Number 的子类，就可以使用如下的语法：? extends 类，表示可以设置指定类或指定类的子类。

```
class Info<T extends Number> {
    private T content ;
    public void setContent(T content){
        this.content = content ;
    }
    public T getContent(){
        return this.content ;
    }
};
public class GenDemo01 {
    public static void main(String args[]){
        Info<Number> info = new Info<Number>() ;
        info.setContent(30) ;
        fun(info) ;
    }
    public static void fun(Info<?> temp){
        System.out.println(temp.getContent()) ;
    }
};
```

现在还有一种情况，方法中接收 Info 类型的时候只能是 String 或其父类，语法：<? super 类>。

```
class Info<T> {
    private T content ;
    public void setContent(T content){
        this.content = content ;
    }
    public T getContent(){
        return this.content ;
    }
};
public class GenDemo01 {
    public static void main(String args[]){
        Info<Object> info = new Info<Object>() ;
        info.setContent("Hello World!!!") ;
```

```
        fun(info) ;
    }
    public static void fun(Info<? super String> temp){
        System.out.println(temp.getContent()) ;
    }
};
```

通配符一共有三种：

- **?：可以接收任意的泛型类型**
- **? extends 类：指定上限**
- **? super 类：指定下限**

3.4.3、在方法上使用泛型

之前的所有泛型都是在一个类上使用的，那么泛型也可以在方法中使用，当然了，泛型方法可以不用编写在泛型类之中，而可以单独存在。

```
public class GenDemo02 {
    public static void main(String args[]){
        fun("Hello","World") ;
    }
    public static <T> void fun(T t1,T t2){
        System.out.println(t1 + " -- " + t2) ;
    }
};
```

特别需要注意的是，以后在观察 DOC 文档中，泛型方法的使用几率将是非常大的。

3.4.4、在接口上使用泛型

泛型的操作不光在类上使用，也可以在接口上应用。

```
interface Info<T>{
    public void fun(T t) ;
}
```

此时，对于这种接口就有两种实现方式了。

第一种：继续指定泛型

```
interface Info<T>{
    public void fun(T t) ;
}
class InfoImpl<T> implements Info<T> {
    public void fun(T t){
        System.out.println(t) ;
    }
};
public class GenDemo03 {
    public static void main(String args[]){
```

```
        Info<String> info = new InfoImpl<String>() ;
        info.fun("Hello") ;
    }
};
```

第二种：直接设置好具体的类型

```
interface Info<T>{
    public void fun(T t) ;
}
class InfoImpl implements Info<String> {
    public void fun(String t){
        System.out.println(t) ;
    }
};
public class GenDemo03 {
    public static void main(String args[]){
        InfoImpl info = new InfoImpl() ;
        info.fun("Hello") ;
    }
};
```

以后更多的代码中是很少会用泛型进行开发的，尤其是各个通配符号更是在个人编写的代码中出现较少，但是在类库中使用的特别多。

3.5、可变参数（理解）

一般而言，一个方法定义完成之后，如果已经设置了三个参数，则在调用的时候一定要传递三个参数，但是一旦存在了可变参数之后，这个规则就打破了，可以任意的传递参数，而且所有的参数最终都是依靠数组接收。

范例：完成整型的加法操作，传递的参数可以任意多

```
public class ArgDemo {
    public static void main(String args[]){
        System.out.println(add(1)) ;
        System.out.println(add(1,2)) ;
        System.out.println(add(1,2,3)) ;
        System.out.println(add()) ;
        System.out.println(add(new int[]{1,3,4,5,67,78,8})) ;
    }
    public static int add(int...data){
        int sum = 0 ;
        for(int x=0;x<data.length;x++){
            sum += data[x] ;
        }
        return sum ;
    }
};
```

可变参数可以应用在任意的数据类型上。

```
class Person {
	private String name ;
	public Person(String name){
		this.name = name ;
	}
	public String toString(){
		return this.name ;
	}
};
public class ArgDemo {
	public static void main(String args[]){
		print(new Person("张三")) ;
		print(new Person("张三"),new Person("李四")) ;
	}
	public static void print(Person...data){
		for(int x=0;x<data.length;x++){
			System.out.println(data[x]) ;
		}
	}
};
```

可变参数也可以使用泛型。

```
public class ArgDemo {
	public static void main(String args[]){
		Integer i[] = fun(1,3,5,7,9) ;
		for(int x=0;x<i.length;x++){
			System.out.println(i[x]) ;
		}
	}
	public static <T> T[] fun(T...data){
		return data ;
	}
};
```

与泛型的概念一样，可变参数并不是开发中的绝对选择，基本上也不用，但是在类库中肯定是存在这种用法的。

3.6、foreach 输出（理解）

foreach 实际属于 for 语句的加强版，最早也是从.NET 中学来的技术，即：可以通过如下的语法完成输出：

```
for(类型 对象：数组或集合){}
```

范例：输出数组

```
public class ForDemo {
	public static void main(String args[]){
		Integer i[] = fun(1,3,5,7,9) ;
```

```
        for(Integer x:i){
            System.out.println(x) ;
        }
    }
    public static <T> T[] fun(T...data){
        return data ;
    }
};
```

与老规矩一样，正经人基本上不使用这种输出方法。

3.7、枚举（理解）

3.7.1、枚举的概念

枚举是在 JDK 1.5 之后增加的一个主要新特性，但是这个增加的特性来讲对于 90%的人员来讲都是无用的，因为只有对于那些已经习惯于使用枚举的 C 开发人员来讲，这些才算是有用的东西。

多例设计：一个类只能产生有限多个实例化对象，那么枚举的功能实际上就属于这种功能的实现，JDK 1.5 之后增加了新的关键字：enum。

范例：定义一个枚举类

```
enum Color{
    RED,GREEN,BLUE ;
}
public class EnumDemo01 {
    public static void main(String args[]){
        Color c = Color.RED ;
        System.out.println(c) ;
    }
};
```

现在只是取得了一个，也可以通过循环的方式输出全部。

```
enum Color{
    RED,GREEN,BLUE ;
}
public class EnumDemo01 {
    public static void main(String args[]){
        for(Color c:Color.values()){
            System.out.println(c) ;
        }
    }
};
```

3.7.2、enum 关键字和 Enum 类

当使用一个 enum 声明了一个枚举的时候，实际上就相当于一个类继承了 Enum 类。

```
enum Color{
	RED,GREEN,BLUE ;
}
public class EnumDemo02 {
	public static void main(String args[]){
		for(Color c:Color.values()){
			System.out.println(c.ordinal() + " --> " + c.name()) ;
		}
	}
};
```

枚举通过 enum 关键字定义，定义的枚举就相当于一个类继承了 Enum 类。

3.7.3、定义属性及方法

使用 enum 定义了一个枚举之后也可以像普通类那样定义属性和方法，包括构造方法，但是一定要注意的是，所有的构造方法一定是 private。

```
enum Color{
	RED("红色"),GREEN("绿色"),BLUE("蓝色") ;
	private Color(String title){
		this.setTitle(title) ;
	}
	private String title ;
	public void setTitle(String title){
		this.title = title ;
	}
	public String getTitle(){
		return this.title ;
	}
}
public class EnumDemo03 {
	public static void main(String args[]){
		for(Color c:Color.values()){
			System.out.println(c.ordinal() + " --> " + c.name() + " --> " + c.getTitle()) ;
		}
	}
};
```

在一般的开发中完全可以通过类的设计来达到枚举的功能。

3.7.4、实现接口定义抽象方法

枚举本身也可以实现接口，但是需要注意的是，一旦一个枚举实现了接口之后，枚举中的每个对象都必须分别的实例化这些接口中提供的抽像方法。

```
interface Info{
	public String getColorInfo() ;
}
enum Color implements Info{
	RED("红色"){
		public String getColorInfo(){
			return this.getTitle() ;
		}
	},GREEN("绿色"){
		public String getColorInfo(){
			return this.getTitle() ;
		}
	},BLUE("蓝色"){
		public String getColorInfo(){
			return this.getTitle() ;
		}
	} ;
	private Color(String title){
		this.setTitle(title) ;
	}
	private String title ;
	public void setTitle(String title){
		this.title = title ;
	}
	public String getTitle(){
		return this.title ;
	}
}
public class EnumDemo04 {
	public static void main(String args[]){
		for(Color c:Color.values()){
			System.out.println(c.ordinal() + " --> " + c.name() + " --> " + c.getColorInfo()) ;
		}
	}
};
```

在枚举中还可以定义抽象方法，但是与实现接口一样的是，每一个枚举对象都要分别实现此方法。

```
enum Color implements Info{
	RED("红色"){
		public String getColorInfo(){
```

```
            return this.getTitle() ;
        }
    },GREEN("绿色"){
        public String getColorInfo(){
            return this.getTitle() ;
        }
    },BLUE("蓝色"){
        public String getColorInfo(){
            return this.getTitle() ;
        }
    } ;
    private Color(String title){
        this.setTitle(title) ;
    }
    private String title ;
    public void setTitle(String title){
        this.title = title ;
    }
    public String getTitle(){
        return this.title ;
    }
    public abstract String getColorInfo() ;
}
public class EnumDemo05 {
    public static void main(String args[]){
        for(Color c:Color.values()){
            System.out.println(c.ordinal() + " --> " + c.name() + " --> " + c.getColorInfo()) ;
        }
    }
};
```

3.7.5、使用枚举

枚举的最大特点实际上是限制了一个类的取值范围，例如：在设置习惯年别的时候，只能是男或女，那么此时应用枚举是最合适的地方。

```
enum Sex{
    MALE("男"),FEMALE("女") ;
    private String name ;
    private Sex(String name){
        this.name = name ;
    }
    public String toString(){
        return this.name ;
```

```
	}
}
class Person {
	private String name ;
	private int age ;
	private Sex sex ;
	public Person(String name,int age,Sex sex){
		this.name = name ;
		this.age = age ;
		this.sex = sex ;
	}
	public String toString(){
		return "姓名：" + this.name + "，年龄：" + this.age + "，性别：" + this.sex ;
	}
};
public class EnumDemo06 {
	public static void main(String args[]){
		System.out.println(new Person("张三",20,Sex.MALE)) ;
	}
};
```

但是通过其他的代码也可以满足此种要求，所以枚举在开发中是否使用并不是绝对的。

3.8、链表（了解）

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 data[]) ;	// 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。

```
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 data[]) ;	// 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
}
class LinkImpl implements Link {
	class Node {
		private Object data ;
		private Node next ;
		public Node(Object data){
			this.data = data ;
		}
		public void addNode(Node newNode){
			if(this.next == null){
				this.next = newNode ;
			} else {
				this.next.addNode(newNode) ;
			}
		}
		public void deleteNode(Node previous,Object data){
			if(this.data.equals(data)){
				previous.next = this.next ;
			} else {
				if(this.next != null){
					this.next.deleteNode(this,data) ;
				}
			}
		}
		public void getAll(){
			retdata[foot++] = this.data ;  // 取出当前节点中的数据
			if(this.next != null){
				this.next.getAll() ;
			}
		}
	};
	private int foot = 0 ;
	private Node root ;	// 根节点
	private int len ;
	private Object retdata[] ; // 接收全部的返回值数据
	public void add(Object data){
		if(data != null) {
			len++ ;	// 保存个数
```

```
            Node newNode = new Node(data) ;
            if(this.root == null){
                this.root = newNode ; // 第一个节点作为根节点
            } else {
                this.root.addNode(newNode) ;
            }
        }
    }
    public void add(Object data[]){
        for(int x=0;x<data.length;x++){
            this.add(data[x]) ;              // 循环增加
        }
    }
    public void delete(Object data){
        if(this.exists(data)){    // 如果存在，则执行删除
            if(this.root.equals(data)){
                this.root = this.root.next ;    // 第二个作为根节点
            } else {
                this.root.next.deleteNode(this.root,data) ;
            }
        }
    }
    public boolean exists(Object data){
        if(data == null){
            return false ;
        }
        if(this.root == null){
            return false ;
        }
        Object d[] = this.getAll() ;   // 取得全部的数据
        boolean flag = false ;
        for(int x=0;x<d.length;x++){
            if(data.equals(d[x])){
                flag = true ;
                break ;
            }
        }
        return flag ;
    }
    public Object[] getAll(){
        this.foot = 0 ;     // 重新清空
        if(this.len != 0){
            this.retdata = new Object[this.len] ;// 根据大小开辟数组
            this.root.getAll() ;
```

```
            return this.retdata ;
        } else {
            return null ;
        }
    }
    public Object get(int index){
        Object d[] = this.getAll() ;
        if(index<d.length){
            return d[index] ;
        } else {
            return null ;
        }
    }
    public int length(){
        return this.len ;
    }
};
public class LinkDemo {
    public static void main(String args[]){
        Link link = new LinkImpl() ;
        link.add("A") ;
        link.add("B") ;
        link.add("C") ;
        link.add(new String[]{"X","Y"}) ;
        Object obj[] = link.getAll() ;
        for(Object o : obj){
            System.out.println(o) ;
        }
        System.out.println(link.exists(null)) ;
        System.out.println(link.get(10)) ;
        link.delete("X") ;
        obj = link.getAll() ;
        for(Object o : obj){
            System.out.println(o) ;
        }
    }
};
```

3.9、习题（重点）

在一个图书超市中可以存在多种图书，现在要求通过一个程序表示出此种关系，而且可以实现增加图书和关键字检索的功能，图书的信息由自己设置。

要求的是，可以清晰的表示出关系。

多种图书：图书本身是一个标准，不同的种类不管如何都要实现此标准。

超市认的是标准，而不是具体的图书。

定义出图书的标准：

```
interface Book{  // 是一个标准
    public String getTitle() ;
    public float getPrice() ;
}
```

用户关心的只是书的相关信息，具体是什么，不管。

```
class BookMarket {
    private Book books[] ; // 全部的书
    private int foot ;
    public BookMarket(int len){
        if(len > 0){
            this.books = new Book[len] ;
        } else {
            this.books = new Book[1] ;
        }
    }
    public boolean add(Book book){
        if(this.foot <   this.books.length){
            this.books[this.foot++] = book ;
            return true ;
        } else {
            return false ;
        }
    }
    public Book[] search(String keyWord){
        Book temp[] = null ;
        int count = 0     ;     // 保存记录数
        // 确认有多少个记录符合要求
        for(int x=0;x<this.books.length;x++){
            if(this.books[x].getTitle().indexOf(keyWord) != -1){
                count++ ;
            }
        }
        if(count != 0){   // 有满足的记录
            temp = new Book[count] ;   // 根据大小开辟
            count = 0 ;
            for(int x=0;x<this.books.length;x++){
                if(this.books[x].getTitle().indexOf(keyWord) != -1){
                    temp[count++] = this.books[x] ;
                }
            }
        }
```

```
            return temp ;
        }
};
```

完成书具体种类：

```
class ComputerBook implements Book {
        private String title ;
        private float price ;
        public ComputerBook(){}
        public ComputerBook(String title,float price){
            this.setTitle(title) ;
            this.setPrice(price) ;
        }
        public void setTitle(String title){
            this.title = title ;
        }
        public void setPrice(float price){
            this.price = price ;
        }
        public String getTitle(){
            return this.title ;
        }
        public float getPrice(){
            return this.price ;
        }
        public String toString(){
            return "[计算机图书] 名称：" + this.getTitle() + "，价格：" + this.getPrice() ;
        }
};
```

下面在主方法中完成功能的测试。

```
public class FinalDemo {
        public static void main(String args[]){
            BookMarket bm = new BookMarket(5) ;
            System.out.println(bm.add(new ComputerBook("JAVA",30.0f))) ;
            System.out.println(bm.add(new ComputerBook("JSP",50.0f))) ;
            System.out.println(bm.add(new ComputerBook("Oracle",60.0f))) ;
            System.out.println(bm.add(new ComputerBook("SQL Server",64.0f))) ;
            System.out.println(bm.add(new ComputerBook("C++",20.0f))) ;
            Book b[] = bm.search("J") ;
            for(int x=0;x<b.length;x++){
                System.out.println(b[x]) ;
            }
        }
};
```

4、总结

1、 抽象类和接口的使用，作为重点掌握
2、 包装类的使用
3、 匿名内部类
4、 对于 JDK 1.5 的新特性，理解概念即可

5、预习任务

异常的处理、包及访问权限、多线程（两种实现方式、同步及死锁的区别）