**3、具体内容（理解）**

Java的发展从1995年开始经历了许多的过程，但是其中有三个最具有代表性的JDK版本：

· 1995年推出的JDK 1.0：标志着Java彻底产生了；

· 1998年推出的JDK 1.2：加入了javax.swing组件，这是主要新特征；

· 2005年推出的JDK 1.5，标记为tiger：出现了许多一直延续到今天的新特性。

在几乎所有的书上，把JDK 1.5的新特性都会结合一些系统的类库来讲解，但是本次是将所有的内容单独抽取出来进行的讲解，所以难免会出现理解上的困难。

**3.1、可变参数**

在讲解可变参数之前，首先来思考一个问题：如果说现在要实现若干个整型变量的相加操作，问，此方法该如何设计？使用数组接收，因为参数个数不确定，而按照之前的所学，只能使用数组完成。

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  System.*out*.println(*add*(**new** **int**[] { 1, 2, 3 }));  }  **public** **static** **int** add(**int**[] data) {  **int** sum = 0;  **for** (**int** x = 0; x < data.length; x++) {  sum += data[x];  }  **return** sum;  }  } |

以上的确是实现了技术要求，但是现在有一个新的问题产生了：如果按照题目的要求，应该是可以任意的传递多个数据，但是以上实际上穿的是一个数据，只不过一个数据使用数组的形式封装。那么为了更好的解决这个问题，可以使用JDK 1.5可变参数形式来进行操作。

|  |
| --- |
| public [static] [final] 返回值类型 方法名称 (**参数类型 ... 变量**) { // 虽然定义方式改变了，但本质上还是个数组  [return [返回值] ;]  } |

发现现在在进行方法参数定义的时候有了一些变化，而这个时候的参数可以说就是数组形式。

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  System.*out*.println(*add*(**new** **int**[] { 1, 2, 3 }));  System.*out*.println(*add*(1, 2, 3));  System.*out*.println(*add*());  }  **public** **static** **int** add(**int** ... data) { // 可变参数  **int** sum = 0;  **for** (**int** x = 0; x < data.length; x++) {  sum += data[x];  }  **return** sum;  }  } |

有了可变参数，在日后进行方法调用的过程之中，就可以比较直观的传递任意多个参数，但是以上的操作在开发之中不建议使用，最好别用。

**3.2、foreach输出**

首先需要解释的是：foreach并不是新的概念，最早是在.NET中提出来的，所谓的foreach可以理解为增强性的for循环，下面来回顾一下最早的for循环。

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  **int** data[] = **new** **int**[] { 1, 2, 3, 4, 5, 6 };  **int** sum = 0;  **for** (**int** x = 0; x < data.length; x++) {  sum += data[x];  }  System.*out*.println(sum);  }  } |

但是有了foreach之后，那么对于数组或者是集合的输出就有了新的支持，语法如下：

|  |
| --- |
| for (数据类型 变量 : 数组 | 集合) {  // 操作代码  } |

这表示通过自动的方式将数组之中的每一个元素赋值给变量，而后在for循环的方法体之中进行操作。

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  **int** data[] = **new** **int**[] { 1, 2, 3, 4, 5, 6 };  **int** sum = 0;  **for** (**int** x : data) { // 自动循环，将每一个元素赋给x  sum += x;  }  System.*out*.println(sum);  }  } |

对于这种for循环依然只是要求会使用，能看懂就行，不过个人建议：最好别用。

**3.3、静态导入**

如果说现在要想导入一个不同包的类的方法，那么肯定使用import完成，即：如下是之前所采用的格式。

**范例：**定义一个MyMath类

|  |
| --- |
| **package** cn.mldn.util;  **public** **class** MyMath {  **public** **static** **int** add(**int** x, **int** y) {  **return** x + y;  }  **public** **static** **int** sub(**int** x, **int** y) {  **return** x - y;  }  **public** **static** **int** mul(**int** x, **int** y) {  **return** x \* y;  }  **public** **static** **int** div(**int** x, **int** y) {  **return** x / y;  }  } |

这个类之中的方法全部都属于static型的方法，而按照之前的方式如果要使用这个类，操作形式如下。

**范例：**按照之前的方式来使用MyMath类

|  |
| --- |
| **package** cn.mldn.demo;  **import** cn.mldn.util.MyMath;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  System.*out*.println("加法操作：" + MyMath.*add*(10, 20));  System.*out*.println("减法操作：" + MyMath.*sub*(20, 10));  }  } |

但是在JDK 1.5之后，如果一个类之中的全部方法都是static型的，则可以使用如下的语法进行静态导入：

|  |
| --- |
| import static 包.类.\* ; |

表示的是将这个指定类之中的全部方法导入进来，最后就好象这些方法全部是在主类之中定义的方法一样。

|  |
| --- |
| **package** cn.mldn.demo;  **import** **static** cn.mldn.util.MyMath.\*;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  System.*out*.println("加法操作：" + *add*(10, 20));  System.*out*.println("减法操作：" + *sub*(20, 10));  }  } |

这种比较难受的语法，也只是出现在讲课之中，本人是绝对不会使用的，你们也可以忘了它，而且就算不忘也别使。

**3.4、JDK 1.5三大主要新特性 —— 泛型**

**3.4.1、泛型的引出（重点）**

下面首先通过一个简单的分析来研究一下泛型出现的主要目的，例如：现在要求定义一个表示坐标的操作类（Point），这个类可以表示三种类型的坐标：

· 整数坐标：x = 10、y = 20；

· 小数坐标：x = 10.1、y = 20.3；

· 字符串数据：x = "东经100度"、y = "北纬20度"。

类之中如果要想保存以上的数据，一定需要定义x和y两个属性，而这两个属性可以接收三种数据类型，那么只能使用Object类来定义会比较合适，这样会发生如下的几种转换关系：

· 整数：int 🡺 自动装箱为Integer 🡺 向上转型为Object；

· 小数：double 🡺 自动装箱为Double 🡺 向上转型为Object；

· 字符串：字符串 🡺 向上转型为Object。

**范例：**定义Point类，使用Object作为属性类型

|  |
| --- |
| **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** x;  }  **public** Object getY() {  **return** y;  }  } |

下面开始设置不同的数据类型，以测试程序。

**范例：**设置整型

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  // 一层设置  Point point = **new** Point() ;  point.setX(10) ;  point.setY(20) ;  // 一层取出  **int** x = (Integer) point.getX() ;  **int** y = (Integer) point.getY() ;  System.*out*.println("X的坐标是：" + x + "，Y的坐标是：" + y);  }  } |

**范例：**设置小数

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  // 一层设置  Point point = **new** Point() ;  point.setX(10.2) ;  point.setY(20.3) ;  // 一层取出  **double** x = (Double) point.getX() ;  **double** y = (Double) point.getY() ;  System.*out*.println("X的坐标是：" + x + "，Y的坐标是：" + y);  }  } |

**范例：**设置字符串

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  // 一层设置  Point point = **new** Point() ;  point.setX("东经100度") ;  point.setY("北纬20度") ;  // 一层取出  String x = (String) point.getX() ;  String y = (String) point.getY() ;  System.*out*.println("X的坐标是：" + x + "，Y的坐标是：" + y);  }  } |

看起来现在的功能都实现了，并且根据之前所学的内容，也只能做到这些了，但是本程序是否有问题？

本程序解决问题的关键就在于Object类，所有的类型都可以向Object转换，但是成是Object，败也是Object。

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  // 一层设置  Point point = **new** Point() ;  **point.setX(10) ;** // 此处设置成int型（Integer型）  point.setY("北纬20度") ;  // 一层取出  String x = (**String**) point.getX() ;  String y = (String) point.getY() ;  System.*out*.println("X的坐标是：" + x + "，Y的坐标是：" + y);  }  } |

这个时候程序并没有出现任何的语法错误，因为数字10被装箱成了Integer，可以使用Object接收，从技术上而言，本操作没有问题，但是从实际来讲，数据是有错误的，因为没有统一，所以在取得数据并且执行向下转型的过程之中就会出现如下的错误提示信息：

|  |
| --- |
| Exception in thread "main" java.lang.**ClassCastException**: java.lang.Integer cannot be cast to java.lang.String |

所以，就可以得出一个结论：以上的代码存在了安全隐患，并且这一安全隐患并没有在程序编译的过程之中检查出来，而现在就可以利用泛型来解决这种尴尬的问题？

泛型：类之中操作的属性或方法的参数的类型不在定义的时候声明，而是在使用的时候动态设置。

|  |
| --- |
| **class** Point**<T>** { // T：Type  **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** x;  }  **public** T getY() {  **return** y;  }  } |

此时就表示point类之中的x和y两个属性的类型暂时还不知道，等待程序时候的时候动态配置。

**范例：**使用泛型

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  // 一层设置  Point<String> point = **new** Point<String>() ;  point.setX("东经100度") ;  point.setY("北纬20度") ;  // 一层取出  String x = point.getX() ;  String y = point.getY() ;  System.*out*.println("X的坐标是：" + x + "，Y的坐标是：" + y);  }  } |

此时没有了向下转型的这些操作关系，那么程序就避免了安全性的问题，而且如果设置的类型不统一，在程序编译的过程之中也是可以很好的解决了，直接会报出语法错误。

而当用户在使用Point类声明对象的时候没有设置泛型，那么程序在编译的过程之中会出现警告信息，而且为了保证程序不出现错误，所有的类型都将使用Object进行处理。**使用泛型可以很好的解决数据类型的统一问题**。

但是在此处需要提醒的是，JDK 1.5和JDK 1.7在定义泛型的时候是稍微有些区别的。

**范例：**JDK 1.5的时候声明泛型对象操作

|  |
| --- |
| Point**<String>** point = **new** Point**<String>**() ; |

以上是JDK 1.5的语法，在声明对象和实例化对象的时候必须都同时设置好泛型类型。

**范例：**JDK 1.7的时候简化了

|  |
| --- |
| Point<String> point = **new** Point<>() ; |

这个时候实例化对象时的泛型类型就通过声明时的泛型类型来定义了。

**3.4.2、通配符（重点）**

泛型的出现的确是可以解决了数据类型的统一问题以及避免了向下转型操作，但同时又会带来新的问题，下面先通过一段程序来观察一下会产生什么问题？

**范例：**为了简化操作，下面定义一个简单的泛型类

|  |
| --- |
| **class** Message<**T**> {  **private** T info;  **public** **void** setInfo(T info) {  **this**.info = info;  }  **public** T getInfo() {  **return** info;  }  } |

**范例：**使用以上类对象执行引用传递

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Message<String> msg = **new** Message<String>();  msg.setInfo("Hello World .");  ***print*(msg)**; // 引用传递  }  **public** **static** **void** print(Message<String> temp) {  System.*out*.println(temp.getInfo()); // 只是输出  }  } |

但是，如果现在定义的泛型类型不是String了呢？例如：换成了int（不能写基本类型，只能是包装类）。

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Message<Integer> msg = **new** Message<Integer>();  msg.setInfo(100);  ***print*(msg);** // 无法进行引用传递  }  **public** **static** **void** print(Message<String> temp) {  System.*out*.println(temp.getInfo()); // 只是输出  }  } |

发现这个时候的print()方法无法再接收Message<Integer>对象的引用，因为这个方法只能够接收Message<String>对象的引用，那么可以将print()方法重载，换成Message<Integer>。

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Message<Integer> msg = **new** Message<Integer>();  msg.setInfo(100);  ***print*(msg)**; // 无法进行引用传递  }  **public** **static** **void** **print(Message<String> temp)** {  System.*out*.println(temp.getInfo()); // 只是输出  }  **public** **static** **void** **print(Message<Integer> temp)** {  System.*out*.println(temp.getInfo()); // 只是输出  }  } |

这个时候发现按照之前的方式根本就无法进行方法的重载，方法的重载没有说为一个类而定义的，因为方法重载的时候观察的不是泛型类型，而是类的名称，或者是说数据类型的，那么现在就可以发现，这个给出了泛型类之后，就相当于将一个类又划分成了若干个不同的小类型。

那么现在的问题：方法接收的参数问题又严重了，而且比之前使用对象多态性解决问题时出现的麻烦更大了，至少那个时候可以利用重载来接收一个类的所有子类对象，而现在呢连重载的机会都不给了。

这个时候，有人提出了，干脆在定义方法的时候就别写泛型类型了。

**范例：**定义方法时不写上泛型类型

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Message<Integer> msg = **new** Message<Integer>();  msg.setInfo(100);  *print*(msg); // 无法进行引用传递  }  **public** **static** **void** print(**Message** temp) {  System.*out*.println(temp.getInfo()); // 只是输出  }  } |

虽然现在在print()方法的参数上出现了警告，但是现在的程序可算是正常了。但是新的问题又来了，问题就在于方法操作中，没有类型限制了：

|  |
| --- |
| **public** **static** **void** print(Message temp) {  temp.setInfo(100); // 设置Integer  System.*out*.println(temp.getInfo()); // 只是输出  } |

发现此时在print()方法之中操作的时候，由于没有设置泛型类型，那么所有的类型都统一变为了Object，也就可以修改了，而通过本程序也就发现了，必须找到一种方法：此方法可以接收任意的泛型类型的设置，并且不能够修改，只能够输出。为了解决这样的问题，可以使用通配符“?”表示。

|  |
| --- |
| **public** **static** **void** print(Message<**?**> temp) {  System.*out*.println(temp.getInfo()); // 只是输出  } |

**由于“?”出现的情况较多，尤其是在学习一些类库的时候，所以对于?就记住一点，表示任意类型，如果有参数返回的时候也是这个“?”，就当成Object进行理解。**

既然现在谈到了Object，那么现在实际上又有了另外一个问题：对于所有的子类，都是Object子类，那么如果对于之前的程序使用Message<Object>能不能接收所有的泛型类型呢？

|  |
| --- |
| Message<String> msg = **new** Message<String>();  Message<Object> temp = msg ; // 无法接收 |

因为Object要比String的范围大，把我去超市买的东西理解为“**new** Message<String>”，把整个超市的商品理解为“Message<Object>”，那么如果真的可以直接转换，就变成了，我买了整个超市的商品。

而在通配符“?”上又衍生出了两个子符号：

· 设置泛型的上限：? extends 类；

|- 例如：? extends **Number**，表示只能是Number或者是Number的子类Integer等；

· 设置泛型的下限：? super 类；

|- 例如：? super String，表示只能是String或者是String的父类（Object）。

**范例：**设置泛型上限

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message<T **extends** Number> {  **private** T info;  **public** **void** setInfo(T info) {  **this**.info = info;  }  **public** T getInfo() {  **return** info;  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Message<Integer> msg = **new** Message<Integer>(); // Integer是Number的子类  msg.setInfo(100);  *print*(msg); // 引用传递  }  **public** **static** **void** print(Message<?> temp) {  System.*out*.println(temp.getInfo()); // 只是输出  }  } |

**范例：**设置泛型下限，在方法的参数上使用

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message<T> {  **private** T info;  **public** **void** setInfo(T info) {  **this**.info = info;  }  **public** T getInfo() {  **return** info;  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Message<String> msg = **new** Message<String>(); // Integer是Number的子类  msg.setInfo("Hello World .");  *print*(msg); // 引用传递  }  // 只能是String的泛型或者是Object的泛型使用  **public** **static** **void** print(Message<? **super** String> temp) {  System.*out*.println(temp.getInfo()); // 只是输出  }  } |

所有的符号实际上只是要求可以看懂，还没到用的时候。

**3.4.3、泛型接口**

在之前所有定义的泛型之中，都是在类上定义的，而对于接口也是可以进行泛型定义的，而使用泛型定义的接口可以称为泛型接口。

|  |
| --- |
| **interface** Message<T> { // 泛型接口  **public** String echo(T msg) ;  } |

而对于泛型接口的实现，在Java之中有两种方式：

**方式一：**在子类上继续定义泛型，同时此泛型在接口上继续使用

|  |
| --- |
| **package** cn.mldn.demo;  **interface** Message<T> { // 泛型接口  **public** String echo(T msg) ;  }  **class** MessageImpl<T> **implements** Message<T> {  **public** String echo(T msg) {  **return** "ECHO : " + msg;  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Message<String> msg = **new** MessageImpl<String>() ;  System.*out*.println(msg.echo("张三"));  }  } |

**方式二：**在子类上设置具体类型

|  |
| --- |
| **package** cn.mldn.demo;  **interface** Message<T> { // 泛型接口  **public** String echo(T msg) ;  }  **class** MessageImpl **implements** Message<String> {  **public** String echo(String msg) {  **return** "ECHO : " + msg;  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Message<String> msg = **new** MessageImpl() ;  System.*out*.println(msg.echo("张三"));  }  } |

以上的两种方式要求懂就可以了，日后会遇到这样的开发，而且使用起来也会很容易。

**3.4.4、泛型方法**

对于泛型除了可以定义在类上之外，也可以在方法上进行定义，而在方法上定义泛型的时候，这个方法不一定非要在泛型类中定义。

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Integer result[] = *get*(1, 2, 3);  **for** (**int** temp : result) { // 连自动拆箱都包含了  System.*out*.println(temp);  }  }  **public** **static** **<T>** T[] get(T... args) { // T的类型由方法调用的时候来决定  **return** args;  }  } |

泛型方法在以后的学习之中是一定会见到，见到的时候只要不觉得恶心就行了。

**3.5、JDK 1.5三大主要新特性 —— 枚举**

**3.5.1、枚举的基本概念**

在讲解枚举之前来回顾一个概念：多例设计模式，构造方法私有化（非public），之后在类的内部存在若干个指定的对象，通过一个方法返回指定对象。

|  |
| --- |
| **package** cn.mldn.demo;  **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 title ;  **private** Color(String title) {  **this**.title = title ;  }  **public** String toString() {  **return** **this**.title ;  }  **public** **static** Color getInstance(**int** ch) {  **switch**(ch) {  **case** 0 :  **return** *RED* ;  **case** 1 :  **return** *GREEN* ;  **case** 2 :  **return** *BLUE* ;  **default** :  **return** **null** ;  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Color c = Color.*getInstance*(0) ;  System.*out*.println(c);  }  } |

实际上在JDK 1.5之后对于多例设计模式有了一些新的改进，使用了一个新的关键字来进行定义 —— 枚举（就是一个简化了的多例设计模式），枚举使用enum关键字来进行定义。

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Color {  *RED*,*GREEN*,*BLUE* ;  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Color c = Color.*RED* ;  System.*out*.println(c);  }  } |

很明显，现在可以发现，利用枚举实现的多例设计模式会更加的简单直白一些，但是在Java之中，枚举并不是一个新的类型，严格来讲，每一个使用enum定义的类实际上都属于一个类继承了Enum父类而已，而java.lang.Enum类定义：

|  |
| --- |
| public abstract class Enum**<E extends Enum<E>>**  extends Object  implements Comparable<E>, Serializable |

而在Enum类之中定义了两个方法：

· 取得枚举的序号：public final int ordinal()；

· 取得枚举的名称：public final String name()。

**范例：**验证以上的两个方法

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Color {  *RED*,*GREEN*,*BLUE* ;  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  **for** (Color c : Color.*values*()) {  System.*out*.println(c.ordinal() + "，" + c.name());  }  }  } |

每一个枚举对象都是根据其定义的顺序进行编号的，而且需要提醒的是，在JDK 1.5之后，switch可以接收枚举类型的判断。

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Color {  *RED*,*GREEN*,*BLUE* ;  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Color c = Color.*RED* ;  **switch**(c) {  **case** *RED*:  System.*out*.println("红色");  **break** ;  **case** *GREEN* :  System.*out*.println("绿色");  **break** ;  **case** *BLUE* :  System.*out*.println("蓝色");  **break** ;  }  }  } |

**总结：**关于switch允许的操作类型

· 在JDK 1.5之前，switch只能够操作int或char型数据；

· 在JDK 1.5之后，switch可以操作enum型；

· 在JDK 1.7之后，switch可以操作String型。

**面试题：**请解释一下enum和Enum的关系？

enum是JDK 1.5之后定义的新关键字，主要用于定义枚举类型，在Java之中每一个使用enum定义的枚举类型实际上都表示一个类默认继承了Enum类而已。

**3.5.2、枚举的其他定义**

按照之前所理解，枚举就属于多例设计模式，那么既然是多例设计模式，对于类之中就肯定有多种组成，包括属性、方法、构造方法，在枚举之中也同样可以定义以上的内容，不过需要注意的是，枚举类之中定义的构造方法绝对不能是public，必须私有化。除了这些要求之外，枚举之中的每一个定义的对象，必须写在第一行。

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Color {  *RED*("红色"),*GREEN*("绿色"),*BLUE*("蓝色") ;  **private** String title ;  **private** Color(String title) {  **this**.title = title ;  }  **public** String toString() {  **return** **this**.title ;  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Color c = Color.*RED* ;  System.*out*.println(c);  }  } |

可是除了以上的定义之外，在枚举之中也可以实现接口。

**范例：**让枚举实现接口

|  |
| --- |
| **package** cn.mldn.demo;  **interface** Message{  **public** String getColor() ;  }  **enum** Color **implements** Message {  *RED*("红色"),*GREEN*("绿色"),*BLUE*("蓝色") ;  **private** String title ;  **private** Color(String title) {  **this**.title = title ;  }  **public** String toString() {  **return** **this**.title ;  }  **public** String getColor() {  **return** **this**.toString();  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Message msg = Color.*RED* ;  System.*out*.println(msg.getColor());  }  } |

而且枚举本身还有一些更为特殊的应用，例如：可以在枚举之中直接定义抽象方法，不过这个时候就要求枚举之中的每一个枚举对象分别实现这个抽象方法。

**范例：**在枚举中定义抽象方法

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Color {  *RED*("红色") {  **public** String getColor() {  **return** **this**.toString();  }  },*GREEN*("绿色"){  **public** String getColor() {  **return** **this**.toString();  }  },*BLUE*("蓝色"){  **public** String getColor() {  **return** **this**.toString();  }  } ;  **private** String title ;  **private** Color(String title) {  **this**.title = title ;  }  **public** String toString() {  **return** **this**.title ;  }  **public** **abstract** String getColor() ; // 抽象方法  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  System.*out*.println(Color.*RED*.getColor());  }  } |

那么枚举这个技术到底在开发之中有什么用呢？

**范例：**枚举的应用（好不容易才想出来的，唯一可以用的地方）

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Sex{  *MALE*("男"),*FEMALE*("女") ;  **private** String title ;  **private** Sex(String title) {  **this**.title = title ;  }  **public** String toString() {  **return** **this**.title ;  }  }  **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** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Person per = **new** Person("张三",20,Sex.*MALE*) ;  System.*out*.println(per);  }  } |

本程序不使用枚举照样可以实现，所以对于开发而言，如果已经习惯于使用枚举的可以继续使用，如果压根就没习惯用的，那么就别用了，之所以出现枚举，是因为大部分语言都有枚举，而java是在JDK 1.5之后才引入了枚举这一概念，基本上没什么用。

**3.6、JDK 1.5三的主要特性 —— Annotation**

在JDK 1.5之后，程序允许通过注解（Annotation）的方式来进行程序的定义，而在Java SE之中存在了三种Annotation：@Override、@Deprecated、@SuppressWarnings。

**1、 准确的覆写：@Override**

如果在进行方法的覆写过程之中，那么要求是：方法名称、参数的类型及个数完全相同，而在开发之中有可能会由于手误导致方法不能被正确的覆写。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message {  **public** String **tostring**() { // 原本打算覆写toString()  **return** "Hello World ." ;  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  System.*out*.println(**new** Message()); // 错误  }  } |

而这个时候没有出现任何的语法错误，因为JDK会认为tostring()是一个新的方法，所以不会出现编译的错误，可是从实际上来讲，这个方法应该是被覆写的，所以为了保证这种错误的问题在程序编译的时候可以正常解决，那么在方法覆写时可以增加上“@Override”定义。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message {  **@Override**  **public** String toString() { // 原本打算覆写toString()  **return** "Hello World ." ;  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  System.*out*.println(**new** Message()); // 错误  }  } |

**2、 声明过期操作：@Deprecated**

对于程序开发而言，往往一些使用的类要进行修改或者是维护，那么如果说现在一个类之中的某个方法，可能一开始推出的时候就存在了一些问题，在修改之后不希望人再去使用这些方法，那么肯定不能直接删除，因为如果直接删除了，那么之前的程序就会出现问题了，所以最好的做法是告诉用户：这个方法存在了问题，不建议再使用了，这个时候就使用“@Deprecated”声明。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message {  **@Deprecated**  **public** **void** ~~print~~(){}  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  **new** Message().~~print~~();  }  } |

如果日后在文档之中发现存在了“Deprecated”，就表示这个方法不建议用户继续使用了，就别用了。

**3、 压制警告信息：@SuppressWarnings**

程序在编译的时候如果存在了某些安全隐患，肯定会提示用户，所以不想让其显示的话，就增加压制警告的信息。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message<T> {  @Deprecated  **public** **void** ~~print~~(){  System.*out*.println("sss");  }  }  **public** **class** TestDemo {  **@SuppressWarnings({ "rawtypes", "unused" })**  **public** **static** **void** main(String[] args) **throws** Exception {  Message msg = **new** Message() ;  }  } |

如果按照正常的开发来讲，以上的一些Annotation意义不大，只是一种使用上的习惯而已，但是如果要想真正的去研究Annotation使用，还是需要了解一些历史的，关于软件的开发模式：

· 第一阶段：会将所有的操作都写在程序之中，例如：网络程序，连接的服务器地址，用户验证等；

· 第二阶段：程序 + 配置文件，配置文件和程序相分离，配置文件过多；

· 第三阶段：将配置文件写回到程序之中，但是和程序进行有效的分离。

**4、总结**

所有的JDK 1.5新特性要求先能看懂，能知道语法做什么，至于说什么时候用，怎么用，还不到时候呢。

而此处讲解完成之后，Java的第一大重点面向对象部分就彻底讲完了。