## 枚举

### 什么是枚举

|  |
| --- |
| Jdk5以前，人们定义常量通常是 public static final RED…,通常会定义很多的这个，有了枚举之后，可以以另外的一种方式将这些常量组织起来。形式如:  public enum Size{  SMALL, MEDIUM, LARGE, EXTRA\_LARGE  };  参见：http://www.tuicool.com/articles/YvQZFf |

### 用法一：用作常量

|  |
| --- |
| /\*\*  \* 用法一：用作常量  \* jdk5前，定义常量都是public static final ...,有了枚举后，可以把固定的这些常量放到枚举类面  \*/  public enum Size{//枚举类默认继承自Enum,因此不可以再继承其他类,可以实现接口  //最简单枚举，枚举的内容之间用逗号隔开即可  SMALL,MEDIUM,LARGE,EXTRA\_LARGE  }  //注意，枚举是有默认无参构造函数的，而且每个枚举项都是有索引的，按照书写的顺序0,1,2…可以通过ordinal()得到其索引顺序。  //以上定义实际上会被转化为  Public class Size extends Enum{  Public static final Size SMALL;  Public static final Size MEDIUM;…….  } |

### 用法二：与switch连用

|  |
| --- |
| Jdk6以前。Switch的表达式只能是int类型、枚举，或者可以转为int的；  /\*\*  \* 用法二：与switch连用:  \* case中只能到枚举的值上，不能精确到其中的索引或者字符串  \*/  @Test  public void test\_switch(){  Size size = Size.SMALL;  switch (size){  case SMALL://这里直接写  System.out.println("small"); break;  }  Color color = Color.GREEN;  switch (color){  case RED:  System.out.println("red"); break;  }  /\*int k = 0;  // Color color = Color.GREEN;  switch (k){  case Color.RED.ordinal():  System.out.println("red"); break;  }\*/  //此枚举是Test的内部枚举  review.enum\_test.Test.People people = review.enum\_test.Test.People.MAN;  switch (people){  case MAN:  System.out.println("man"); break;  }  /\*int i = 0;  switch (i){  // case review.enum\_test.Test.People.MAN.ordinal():  // System.out.println("");break;  case Color2.GREEN.getIndex()://这样不行，必须是个常量,枚举的不算  System.out.println();break;  }\*/  }  //通过此例，我们发现case 可以是枚举里面的具体的枚举项，但不能是通过枚举项得到的其他东西，比如此枚举的序号之类的； |

用法三：枚举构造

|  |
| --- |
| public enum Color2{  RED(1,"红色"),GREEN(2,"绿色"),YELLOW(3,"黄色");//注意分号，参数顺序严格按照构造函数来  //定义成员变量  private String name;  private Integer index;  //构造方法  Color2(Integer index,String name){  this.index = index;  this.name = name;  }  //根据index获取颜色  public static String getNameByIndex(Integer index){  for(Color2 c : Color2.values()){  //System.out.print(c.getIndex()+" "+c.getName());//1 红色2 绿色3 黄色  if(c.getIndex() == index){  return c.getName();  }  }  return "";  }  //根据颜色获取index  public static Integer getIndexByName(String name){  for(Color2 c : Color2.values()){  if(c.getName().equals(name)){  return c.getIndex();  }  }  return -1;  }  //普通的get/set方法  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public Integer getIndex() {  return index;  }  public void setIndex(Integer index) {  this.index = index;  }  }  @Test  public void test\_color2(){  System.out.println(Color2.YELLOW.getIndex());//3  System.out.println(Color2.YELLOW.getName());//黄色  //根据index获取颜色  System.out.println(Color2.getNameByIndex(1));//红色  //根据颜色获取index  System.out.println(Color2.getIndexByName("绿色"));//2  } |

## 泛型

### 泛型概念的提出

|  |
| --- |
| public void test1(){  List list = new ArrayList();  list.add("1");  list.add("2");  list.add(3);//注意此处添加的是数字，而前面都是字符串  for(Object ob: list){  String e=(String)ob;//这样在遍历的时候，我们必须把每项转为我们期待的类型  //但现实是：不是所有的项都是String类型的，这可定会导致类型转换一次  //而且，在运行期间我们的数据被转为了Object类型，这也不是我们期望的  }  }  泛型的出现解决了上面的问题。 |

### 什么是泛型

|  |
| --- |
| 就是参数化类型，即将类型进行约束，只允许此类型的参数。  public void test2(){  List<String> list = new ArrayList<String>();  list.add("1");  list.add("2");  //list.add(3);//如果添加非String类型的数据时，IDE就会帮我们检查出这种错误  for(String str : list){//此时我们就可以使用具体的类型，不再使用Object，不再强转型  System.out.println(str);  }  }  我们可以看List接口：  public interface List<E> extends Collection<E> {  Iterator<E> iterator();….//迭代器返回的就是List输入的类型  } |

### 泛型的分类

|  |
| --- |
| 在具体使用时，分为  泛型类ArrayList<T>  泛型接口List<T>  泛型方法public T getData(){} |

### 泛型实例与泛型擦除

|  |
| --- |
| class Box<T>{//泛型类  private T data;//data的类型为T对应的类型  public Box(){}  public Box(T data){  this.data= data;  }  public T getData(){  return data;  }  }  @Test  public void test3(){  Box<String> stringBox = new Box<String>("box");//此构造传入的参数必须为Box的泛型String，否则报错  Box<Integer> intBox= new Box<Integer>(123);  System.out.println(stringBox.getClass());//class review.generic.GenericTest$Box  System.out.println(intBox.getClass());//class review.generic.GenericTest$Box  System.out.println(stringBox.getClass() == intBox.getClass());//true  /\*\*  \* 以上说明：虽然Box是泛型的，但其并没有生成新的类型，仍然是Box类型，Box的各个实例的字节码对象仍然是一个；  \* 原因是：泛型只存在于代码编译阶段，在编译过程中，正确检查泛型结果后，  \* 会将泛型信息擦除，即编译后的class文件时不包含任何泛型信息的  \*/  } |

### 泛型方法(复杂些)

|  |
| --- |
| 如上，是单独定义一个泛型方法的方式，方形方法不一定存在于泛型类中；  此时，<T>是必须的，它表明这是一个方形方法，后面的T是此方法的返回值类型 |

### 通配符和上界下界

|  |
| --- |
| @Test  public void test4(){  Box<Integer> intBox= new Box<Integer>(123);  Box<Number> numBox= new Box<Number>(123);  getData(numBox);  //getData(intBox);//IDE报错，说明：Box<Number> 并不是Box<Integer> 的父类，没有继承关系  getData2(intBox);//使用了类型通配符后，不再报错  getData2(numBox);  }  public void getData(Box<Number> data){//接受Number的  System.out.println(data.getData());  }  public void getData2(Box<?> data){//类型通配符，一般使用?,Box<?>在逻辑上是Box<Integer>、  // Box<Number>...等所有Box<具体类型实参>的父类  System.out.println(data.getData());  }  public void getData3(Box<? extends Number> data){//类型通配符上界，表明参数是Number的子类  // super表示下界  System.out.println(data.getData());  } |

### 最后

|  |
| --- |
| 没有泛型数组  泛型不具备继承性：Box<Number> 和Box<Integer> 没有继承关系  参考：http://www.cnblogs.com/ljxe/p/5521840.html |