自J2SE5.0开始新增了枚举类型（Enumerated Types），你可以使用这个功能取代之前J2SE1.4或之前版本定义常数的方式。

枚举类型本质上还是以类的方式存在。

# 常数设置

有时会需要定义一些常数供程序使用，可以使用接口或类来定义，例如可以使用接口来定义实现时所需的共享常数。

**public** **interface** ActionConstants {

**public** **static** **final** **int** *TURN\_LEFT* = 1;

**public** **static** **final** **int** *TURN\_RIGHT* = 2;

**public** **static** **final** **int** *SHOT* = 3;

}

共享的常数通常是可以直接取用并且不可被修改的，所以在声明时加上static和final，例如可以在程序中直接使用像ActionConstants.TURN\_LEFT这样的名称来取用常数值。例如：

**public** **void** someMethod() {

doAction(ActionConstants.*TURN\_RIGHT*);

}

**public** **void** doAction(**int** action) {

**switch** (action) {

**case** ActionConstants.*TURN\_LEFT*:

System.*out*.println("向左转");

**break**;

**case** ActionConstants.*TURN\_RIGHT*:

System.*out*.println("向右转");

**break**;

**case** ActionConstants.*SHOT*:

System.*out*.println("射击");

**break**;

}

}

如果使用类来声明，方法也相似。例如：

**public** **class** CommandTool {

**public** **static** **final** String *ADMIN* = "onlyfun.caterpillar.admin";

**public** **static** **final** String *DEVELOPER* = "onlyfun.caterpillar.developer";

**public** **void** someMethod() {

// ...

}

}

如果常数只是在类内部使用，就声明其为private或是protected就可以了；声明为类外可取用的常数，通常是与类功能相联系的常数。例如使用CommandTool时若会使用到与CommandTool功能相联系的常数，将这些常数直接声明在CommandTool类上取用时就很方便。而使用接口所声明的常数，则通常是整个程序或某个模块中都会共享到的常数。

# 枚举类型入门

**public** **enum** Action {

*TURN\_LEFT*,*TURN\_RIGHT*,*SHOOT*;

}

**public** **void** someMethod() {

doAction(Action.*TURN\_RIGHT*);

}

**public** **void** doAction(Action action) {

**switch** (action) {

**case** *TURN\_LEFT*:

System.*out*.println("向左转");

**break**;

**case** *TURN\_RIGHT*:

System.*out*.println("向右转");

**break**;

**case** *SHOOT*:

System.*out*.println("射击");

**break**;

}

}

注意到doAction()参数行的类型是Action.如果doAction()方法输入其他类型的变量，编译器会报告错误，因为doAction()所接受的变量必须是Action枚举类型。

使用枚举类型还可以作更进一步的检验，如果在switch中加入了不属于Action中枚举的值，编译器也会报告错误。例如：

**public** **void** doAction(Action action) {

**switch** (action) {

**case** *TURN\_LEFT*:

System.*out*.println("向左转");

**break**;

**case** *STOP*: //Action中没有枚举这个值

System.*out*.println("停止");

**break**;

}}

可以在一个独立的文件中声明枚举值，或是在某个类中声明枚举成员。

# 深入枚举类型

当使用enum定义枚举类型时，实际上所定义出来的类型是继承自java.lang.Enum类 。 而每个被枚举的成员其实就是定义的枚举类型的一个实例，它们都被默认为final.无法改变常数名称所设置的值，它们也是 public 且static的成员。可以通过类名称直接使用它们。上面定义了Action枚举类型，其中定义的TURN\_LEFT , TURN\_RIGHT,SHOOT都是Action的一个对象实例。因为是对象。所以对象上自然有一些方法可以调用。例如从Object继承下来的toString()方法被重新定义了，可以让你直接取得枚举值得字符串描述；values()方法可以让你取得所有的枚举成员实例，并且以数组方式返回。

**public** **class** ShowEnum {

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

**for**(Action action : Action.*values*()){

System.*out*.println(action.toString());

}

}

}

基本上println()会自动调用对象的 toString(),所以不写toString()其实也是可以得，执行结果如下：

TURN\_LEFT

TURN\_RIGHT

SHOOT

由于每一个枚举的成员都是一个对象实例，所以可以使用==或是equals()方法来比较枚举对象。==会比较您提供的枚举对象是不是同一个对象，而equals()则是比较两个枚举对象的内容是否相同，使用equals()时默认会根据枚举对象的字符串值来比较。

静态valueOf()方法可以让你将指定的字符串尝试转换为枚举实例。可以使用compareTo()方法来比较两个枚举对象在枚举时的顺序。

**public** **class** EnumCompareTo {

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

*compareToAction*(Action.*valueOf*(args[0]));

}

**public** **static** **void** compareToAction(Action inputAction) {

System.*out*.println("输入："+inputAction);

**for**(Action action : Action.*values*()){

System.*out*.println(action.compareTo(inputAction));

}

}

}

compareTo()如果返回正值，表示设置为变量的枚举对象(inputAction)的顺序在比较的枚举对象(action)之前；负值表示在之后；而0则表示两个互比枚举值得位置是相同的。

执行结果如下：

java EnumCompareTo SHOOT

输入：SHOOT

-2

-1

0

对于每一个枚举成员，可以使用 ordinal()方法，依枚举顺序得到位置索引，默认以0开始。

**public** **class** EnumIndex {

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

**for**(Action action: Action.*values*()){

System.*out*.printf("%d %s%n",action.ordinal(),action);

}

}

}

执行结果：

0 TURN\_LEFT

1 TURN\_RIGHT

2 SHOOT

定义枚举类型基本上就像是在定义类，定义枚举类型时也可以定义方法。例如，也许你想要为枚举值加上一些描述，而不是使用默认的toString()返回值来描述枚举值。

**public** **enum** DetailAction {

*TURN\_LEFT*, *TURN\_RIGHT*, *SHOOT*;

**public** String getDescription() {

**switch** (**this**.ordinal()) {

**case** 0:

**return** "向左转";

**case** 1:

**return** "向右转";

**case** 2:

**return** "射击";

**default**:

**return** **null**;

}

}

}

**public** **class** DetailActionDemo {

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

**for**(DetailAction action: DetailAction.*values*()){

System.*out*.printf("%s %s%n",action,action.getDescription());

}

}

}

执行结果：

TURN\_LEFT 向左转

TURN\_RIGHT 向右转

SHOOT 射击

枚举类型既然是类，那么可以为它加上构造函数吗？答案是可以得，但是不得为公开的(Public)构造函数,这是为了避免粗心的程序设计人员直接对枚举类型实例化。一个不公开的构造函数可以做什么？看下面范例：

**public** **enum** DetailAction2 {

*TURN\_LEFT*("向左转"), *TURN\_RIGHT*("向右转"), *SHOOT*("射击");

**private** String description;

**private** DetailAction2(String description){

**this**.description = description;

}

**public** String getDescription(){

**return** description;

}

}

上面的范例可以依照之前的DetailActionDemo类(将DetailAction改为DetailAction2)进行测试，会得到相同的显示结果.

在定义枚举时也可以一并实现接口。例如：

**public** **interface** IDescription {

**public** String getDescription();

}

可以使用这个接口规定每个实现该接口的枚举，都必须返回一个描述枚举值的字符串，如下：

**public** **enum** DetailAction3 **implements** IDescription {

*TURN\_LEFT*("向左转"), *TURN\_RIGHT*("向右转"), *SHOOT*("射击");

//不公开的构造函数

**private** DetailAction3(String description){

**this**.description = description;

}

**private** String description;

@Override

**public** String getDescription() {

**return** description;

}

}

非公开的构造函数最常见的例子就是Singleton模式的应用，当某个类只能有一个实例时，可由类维护唯一的实例。

因值而异的类实现(Value-Specific Class Bodies)

这个功能简单地说，实现时像是在使用匿名内嵌类来实现Command模式，它可以为每个枚举值定义各自的类本体与方法(Method)实现。

**public** **enum** MoreAction **implements** IDescription {

*TURN\_LEFT*{

//实现接口上的方法

**public** String getDescription() {

**return** "向左转";

}

}, //记得这里的枚举值分割使用,

*TURN\_RIGHT*{

**public** String getDescription() {

**return** "向右转";

}

},

*SHOOT*{

**public** String getDescription() {

**return** "射击";

}

}; //记得这里的枚举值结束使用

}

**public** **class** MoreActionDemo {

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

**for**(MoreAction action : MoreAction.*values*()){

System.*out*.printf("%s: %s%n",action,action.getDescription());

}

}

}

这个例子是将因值而异的类实现用在返回枚举值描述上，可以按照相同的方法，为每个枚举值加上一些各自的方法实现，而调用的接口是统一的，执行结果会显示各自的枚举描述：

TURN\_LEFT: 向左转

TURN\_RIGHT: 向右转

SHOOT: 射击

也可以运用抽象方法来改写：

**public** **enum** MoreAction2 {

*TURN\_LEFT*{

//实现抽象方法

**public** String getDescription() {

**return** "向左转";

}

}, //记得这里的枚举值分割使用,

*TURN\_RIGHT*{

//实现抽象方法

**public** String getDescription() {

**return** "向右转";

}

},

*SHOOT*{

//实现抽象方法

**public** String getDescription() {

**return** "射击";

}

}; //记得这里的枚举值结束使用;

//声明抽象方法

**public** **abstract** String getDescription();

}

MoreAction2与MoreAction不同的地方在于，MoreAction2是实现抽象方法。基本上定义接口方法或抽象方法，是为了知道对象的实现接口，这样才能去实现这个对象。

枚举只有一个元素时,就可以作为一种单例的实现方式;

代码示例：

**public** **class** EnumTest {

/\*\*

\* 枚举

\*/

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

// WeekDay1 引用的对象 只能 是 WeekDay1 中定义的

WeekDay1 weekDay = WeekDay1.*MON*;

System.*out*.println(weekDay.nextDay());

WeekDay weekDay2 = WeekDay.*FRI*;

System.*out*.println(weekDay2);

System.*out*.println(weekDay2.name());

System.*out*.println(weekDay2.ordinal());

System.*out*.println(WeekDay.*valueOf*("SUN").toString());

System.*out*.println(WeekDay.*values*().length);

TrafficLamp lamp = TrafficLamp.*RED*;

System.*out*.println(lamp);

System.*out*.println(lamp.nextLamp());

System.*out*.println(lamp.ordinal());

System.*out*.println(lamp.*values*().length);

System.*out*.println(lamp.time);

Date date = EnumTest.*datetime*;

System.*out*.println(date.~~getYear~~());

}

//内部类,子类的构造方法调用父类有参的构造方法

**public** **static** Date *datetime* = **new** Date(100){

**public** **int** getYear(){

**return** 2000;

}

};

/\*\*

\* 枚举就相当于一个类,枚举只有一个元素时,就可以作为一种单例的实现方式

\* \*/

**public** **enum** WeekDay{

//枚举中的元素表示该枚举类的一个实例对象

*SUN*(1),*MON*(),*TUE*,*WED*,*THI*,*FRI*,*SAT* ;

// 若后面有其它需要定义的内容，则要在元素列表后加上“;”

//枚举构造方法必须是私有的 ,必须定义在元素列表之后

**private** WeekDay(){

System.*out*.println("first");

};

**private** WeekDay(**int** day){

System.*out*.println("second");

};

}

**public** **enum** TrafficLamp{

*RED*(30){

**public** TrafficLamp nextLamp(){

**return** *GREEN*;

}

},

*GREEN*(45){

**public** TrafficLamp nextLamp(){

**return** *YELLOW*;

}

},

// 调用下面带参数的构造方法

*YELLOW*(5){

**public** TrafficLamp nextLamp(){

**return** *RED*;

}

};

**public** **abstract** TrafficLamp nextLamp();

**private** **int** time;

**private** TrafficLamp(**int** time){

**this**.time = time;

}

}

}

**public** **abstract** **class** WeekDay1 {

**private** WeekDay1(){

}

**public** **final** **static** WeekDay1 *SUN* = **new** WeekDay1(){

// 内部类，定义WeekDay1的子类

**public** WeekDay1 nextDay(){

**return** *MON*;

}

};

**public** **final** **static** WeekDay1 *MON* = **new** WeekDay1(){

**public** WeekDay1 nextDay(){

**return** *SUN*;

}

};

/\*public WeekDay1 nextDay(){

if(this == SUN) return MON;

else return SUN;

}\*/

/\*\*

\* 采用抽象方法定义nextDay就将大量的if.else语句转移成了一个个独立的类

\* \*/

**public** **abstract** WeekDay1 nextDay();

**public** String toString(){

**return** **this**==*SUN*?"SUN":"MON";

}

}