概念

enum的全稱為 enumeration， 是 JDK 1.5 中引入的新特性。

在Java中，被 enum 關鍵字修飾的型別就是列舉型別。形式如下：

```
enum Color { RED, GREEN, BLUE }
```

如果列舉不新增任何方法，列舉值預設為從0開始的有序數值。以 Color 列舉型別舉例，它的列舉常量依次為RED：0，GREEN：1，BLUE：2

列舉的好處：可以將常量組織起來，統一進行管理。

列舉的典型應用場景：錯誤碼、狀態機等。

列舉型別的本質

儘管enum 看起來像是一種新的資料型別，事實上，enum是一種受限制的類，並且具有自己的方法。

建立enum時，編譯器會為你生成一個相關的類，這個類繼承自 java.lang.Enum。

java.lang.Enum類宣告

```
public abstract class Enum<E extends Enum<E>>
implements Comparable<E>, Serializable { ... }
```

列舉的方法

在enum中，提供了一些基本方法：

values()：返回enum例項的陣列，而且該陣列中的元素嚴格保持在enum中宣告時的順序。

name()：返回例項名。

ordinal()：返回例項宣告時的次序，從0開始。

getDeclaringClass()：返回例項所屬的enum型別。

equals()：判斷是否為同一個物件。

可以使用 == 來比較enum例項。

此外，java.lang.Enum實現了Comparable和 Serializable 介面，所以也提供 compareTo() 方法。

例：展示enum的基本方法

```
public class EnumMethodDemo {
enum Color {RED, GREEN, BLUE;}
enum Size {BIG, MIDDLE, SMALL;}
public static void main(String args[]) {
System.out.println("=========== Print all Color ===========");
for (Color c : Color.values()) {
System.out.println(c   " ordinal: "   c.ordinal());
}
System.out.println("=========== Print all Size ===========");
for (Size s : Size.values()) {
System.out.println(s   " ordinal: "   s.ordinal());
}
Color green = Color.GREEN;
System.out.println("green name(): "   green.name());
System.out.println("green getDeclaringClass(): "   green.getDecla
System.out.println("green hashCode(): "   green.hashCode());
System.out.println("green compareTo Color.GREEN: "   green.compar
System.out.println("green equals Color.GREEN: "   green.equals(Co
System.out.println("green equals Size.MIDDLE: "   green.equals(Si
System.out.println("green equals 1: "   green.equals(1));
System.out.format("green == Color.BLUE: %b\n", green == Color.BLU
}
}
```

輸出

```
=========== Print all Color ===========
RED ordinal: 0
GREEN ordinal: 1
BLUE ordinal: 2
=========== Print all Size ===========
BIG ordinal: 0
MIDDLE ordinal: 1
SMALL ordinal: 2
green name(): GREEN
green getDeclaringClass(): class org.zp.javase.enumeration.EnumDe
```

```
green hashCode(): 460141958
green compareTo Color.GREEN: 0
green equals Color.GREEN: true
green equals Size.MIDDLE: false
green equals 1: false
green == Color.BLUE: false
```

列舉的特性

列舉的特性，歸結起來就是一句話：

除了不能繼承，基本上可以將enum 看做一個常規的類。

但是這句話需要拆分去理解，讓我們細細道來。

列舉可以新增方法

在概念章節提到了，列舉值預設為從0開始的有序數值 。那麼問題來了：如何為列舉顯示的賦值。

Java 不允許使用 = 為列舉常量賦值

如果你接觸過C/C ，你肯定會很自然的想到賦值符號 = 。在C/C 語言中的enum，可以用賦值符號=顯示的為列舉常量賦值；但是 ，很遺憾，Java 語法中卻不允許使用賦值符號 = 為列舉常量賦值。

例：C/C 語言中的列舉宣告

```
typedef enum{
ONE = 1,
TWO,
THREE = 3,
TEN = 10
} Number;
```

enum 可以新增普通方法、靜態方法、抽象方法、構造方法

Java雖然不能直接為例項賦值，但是它有更優秀的解決方案：為 enum 新增方法來間接實現顯示賦值。

建立 enum 時，可以為其新增多種方法，甚至可以為其新增構造方法。

注意一個細節：如果要為enum定義方法，那麼必須在enum的最後一個例項尾部新增一個分號。此外，在enum中，必須先定義例項，不能將欄位或方法定義在例項前面。否則，編譯器會報錯。

例：全面展示如何在列舉中定義普通方法、靜態方法、抽象方法、構造方法

```
public enum ErrorCode {
OK(0) {
public String getDescription() {
return "成功";
}
},
ERROR_A(100) {
public String getDescription() {
return "錯誤A";
}
},
ERROR_B(200) {
public String getDescription() {
return "錯誤B";
}
};
private int code;
// 構造方法：enum的構造方法只能被宣告為private許可權或不宣告許可權
private ErrorCode(int number) { // 構造方法
this.code = number;
}
public int getCode() { // 普通方法
return code;
} // 普通方法
public abstract String getDescription(); // 抽象方法
public static void main(String args[]) { // 靜態方法
for (ErrorCode s : ErrorCode.values()) {
System.out.println("code: "   s.getCode()   ", description: "   s
}
}
}
```

注：上面的例子並不可取，僅僅是為了展示列舉支援定義各種方法。下面是一個簡化的例子

例：一個錯誤碼列舉型別的定義

本例和上例的執行結果完全相同。

```
public enum ErrorCodeEn {
OK(0, "成功"),
ERROR_A(100, "錯誤A"),
ERROR_B(200, "錯誤B");
ErrorCodeEn(int number, String description) {
this.code = number;
this.description = description;
}
private int code;
private String description;
public int getCode() {
return code;
}
public String getDescription() {
return description;
}
public static void main(String args[]) { // 靜態方法
for (ErrorCodeEn s : ErrorCodeEn.values()) {
System.out.println("code: "   s.getCode()   ", description: "   s
}
}
}
```

列舉可以實現介面

enum 可以像一般類一樣實現介面。

同樣是實現上一節中的錯誤碼列舉類，通過實現介面，可以約束它的方法。

```
public interface INumberEnum {
int getCode();
String getDescription();
}
public enum ErrorCodeEn2 implements INumberEnum {
OK(0, "成功"),
ERROR_A(100, "錯誤A"),
ERROR_B(200, "錯誤B");
ErrorCodeEn2(int number, String description) {
this.code = number;
this.description = description;
}
```

```
private int code;
private String description;
@Override
public int getCode() {
return code;
}
@Override
public String getDescription() {
return description;
}
}
```

列舉不可以繼承

enum 不可以繼承另外一個類，當然，也不能繼承另一個 enum 。

因為 enum 實際上都繼承自 java.lang.Enum 類，而 Java 不支援多重繼承，所以enum不能再繼承其他類，當然也不能繼承另一個enum。

列舉的應用場景

組織常量

在JDK1.5 之前，在Java中定義常量都是public static final TYPE a; 這樣的形式。有了列舉，你可以將有關聯關係的常量組織起來，使程式碼更加易讀、安全，並且還可以使用列舉提供的方法。

列舉宣告的格式

注：如果列舉中沒有定義方法，也可以在最後一個例項後面加逗號、分號或什麼都不加。

下面三種宣告方式是等價的：

```
enum Color { RED, GREEN, BLUE }
enum Color { RED, GREEN, BLUE, }
enum Color { RED, GREEN, BLUE; }
```

switch 狀態機

我們經常使用switch語句來寫狀態機。JDK7以後，switch已經支援 int、char、String、enum 型別的引數。這幾種型別的引數比較起來，使用列舉的switch程式碼更具有可讀性。

```
enum Signal {RED, YELLOW, GREEN}
public static String getTrafficInstruct(Signal signal) {
String instruct = "訊號燈故障";
switch (signal) {
case RED:
instruct = "紅燈停";
break;
case YELLOW:
instruct = "黃燈請注意";
break;
case GREEN:
instruct = "綠燈行";
break;
default:
break;
}
return instruct;
}
```

組織列舉

可以將型別相近的列舉通過介面或類組織起來。

但是一般用介面方式進行組織。

原因是：Java介面在編譯時會自動為enum型別加上public static修飾符；Java類在編譯時會自動為 enum 型別加上static修飾符。看出差異了嗎？沒錯，就是說，在類中組織 enum，如果你不給它修飾為 public，那麼只能在本包中進行訪問。

例：在介面中組織 enum

```
public interface Plant {
enum Vegetable implements INumberEnum {
POTATO(0, "土豆"),
TOMATO(0, "西紅柿");
Vegetable(int number, String description) {
this.code = number;
this.description = description;
}
private int code;
private String description;
```

```
@Override
public int getCode() {
return 0;
}
@Override
public String getDescription() {
return null;
}
}
enum Fruit implements INumberEnum {
APPLE(0, "蘋果"),
ORANGE(0, "桔子"),
BANANA(0, "香蕉");
Fruit(int number, String description) {
this.code = number;
this.description = description;
}
private int code;
private String description;
@Override
public int getCode() {
return 0;
}
@Override
public String getDescription() {
return null;
}
}
}
```

例：在類中組織 enum

本例和上例效果相同。

```
public class Plant2 {
public enum Vegetable implements INumberEnum {...} // 省略程式碼
public enum Fruit implements INumberEnum {...} // 省略程式碼
}
```

策略列舉

EffectiveJava中展示了一種策略列舉。這種列舉通過列舉巢狀列舉的方式，將列舉常量分類處理。

這種做法雖然沒有switch語句簡潔，但是更加安全、靈活。

例：EffectvieJava中的策略列舉範例

```
enum PayrollDay {
MONDAY(PayType.WEEKDAY), TUESDAY(PayType.WEEKDAY), WEDNESDAY(
PayType.WEEKDAY), THURSDAY(PayType.WEEKDAY), FRIDAY(PayType.WEEKD
PayType.WEEKEND), SUNDAY(PayType.WEEKEND);
private final PayType payType;
PayrollDay(PayType payType) {
this.payType = payType;
}
double pay(double hoursWorked, double payRate) {
return payType.pay(hoursWorked, payRate);
}
// 策略列舉
private enum PayType {
WEEKDAY {
double overtimePay(double hours, double payRate) {
return hours <= HOURS_PER_SHIFT ? 0 : (hours - HOURS_PER_SHIFT)
* payRate / 2;
}
},
WEEKEND {
double overtimePay(double hours, double payRate) {
return hours * payRate / 2;
}
};
private static final int HOURS_PER_SHIFT = 8;
abstract double overtimePay(double hrs, double payRate);
double pay(double hoursWorked, double payRate) {
double basePay = hoursWorked * payRate;
return basePay   overtimePay(hoursWorked, payRate);
}
}
}
```

測試

```
System.out.println("時薪100的人在週五工作8小時的收入："    PayrollDay
System.out.println("時薪100的人在週六工作8小時的收入："    PayrollDay
```

EnumSet和EnumMap

Java中提供了兩個方便操作enum的工具類——EnumSet和EnumMap。

`EnumSet` 是列舉型別的高效能Set實現。它要求放入它的列舉常量必須屬於同一列舉型別。

`EnumMap` 是專門為列舉型別量身定做的Map實現。雖然使用其它的Map實現（如HashMap）也能完成列舉型別例項到值得對映，但是使用EnumMap會更加高效：它只能接收同一列舉型別的例項作為鍵值，並且由於列舉型別例項的數量相對固定並且有限，所以EnumMap使用陣列來存放與列舉型別對應的值。這使得EnumMap的效率非常高。

```
// EnumSet的使用
System.out.println("EnumSet展示");
EnumSet<ErrorCodeEn> errSet = EnumSet.allOf(ErrorCodeEn.class);
for (ErrorCodeEn e : errSet) {
System.out.println(e.name()   " : "   e.ordinal());
}
// EnumMap的使用
System.out.println("EnumMap展示");
EnumMap<StateMachine.Signal, String> errMap = new EnumMap(StateMa
errMap.put(StateMachine.Signal.RED, "紅燈");
errMap.put(StateMachine.Signal.YELLOW, "黃燈");
errMap.put(StateMachine.Signal.GREEN, "綠燈");
for (Iterator<Map.Entry<StateMachine.Signal, String>> iter = errM
Map.Entry<StateMachine.Signal, String> entry = iter.next();
System.out.println(entry.getKey().name()   " : "   entry.getValue
}
```