

枚举

枚举的二种实现方式

1) 自定义类实现枚举

a 构造器私有化

b 本类内部创建一组对象[四个 春夏秋冬]

c 对外暴露对象（通过为对象添加 `public final static` 修饰符）

d 可以提供 `get` 方法，但是不要提供 `set`

2) 使用 `enum` 关键字实现枚举

a. 使用关键字 `enum` 替代 `class`

b. `public static final Season SPRING = new Season("春天", "温暖")` 直接使用 `SPRING("春天", "温暖")` 解读 常量名(实参列表)

c. 如果有多个常量(对象)，使用 , 号间隔即可

d. 如果使用 `enum` 来实现枚举，要求将定义常量对象，写在前面

e. 如果我们使用的是无参构造器，创建常量对象，则可以省略 ()

`enum` 关键字注意事项：

1) 当我们使用 `enum` 关键字开发一个枚举类时，默认会继承 `Enum` 类，而且是一个 `final` 类

2) 传统的 `public static final Season2 SPRING = new Season2("春天", "温暖")`；简化成 `SPRING("春天", "温暖")`，这里必须知道，它调用的是哪个构造器。

3) 如果使用无参构造器 创建 枚举对象，则实参列表和小括号都可以省略

4) 当有多个枚举对象时，使用 , 间隔，最后有一个分号结尾

5) 枚举对象必须放在枚举类的行首。

6) 使用 `enum` 关键字后，就不能再继承其它类了，因为 `enum` 会隐式继承 `Enum`，而 `Java` 是单继承机制。

7) 枚举类和普通类一样，可以实现接口

`Enum` 常用方法：

1) `toString()`: `Enum` 类已经重写过了，返回的是当前对象名, 子类可以重写该方法，用于返回对象的属性信息

- 2) name: 返回当前对象名 (常量名), 子类中不能重写
- 3) ordinal: 返回当前对象的位置号, 默认从 0 开始
- 4) values: 返回当前枚举类中所有的常量
- 5) valueOf: 将字符串转换成枚举对象, 要求字符串必须为已有的常量名, 否则报异常!
- 6) compareTo: 比较两个枚举常量, 比较的就是编号!

注解:

- 1) 注解 (Annotation) 也被称为元数据 (Metadata), 用于修饰解释 包、类、方法、属性、构造器、局部变量等数据信息。
- 2) 和注释一样, 注解不影响程序逻辑, 但注解可以被编译或运行, 相当于嵌入在代码中的补充信息。
- 3) 在 JavaSE 中, 注解的使用目的比较简单, 例如标记过时的功能, 忽略警告等。在 JavaEE 中注解占据了更重要的角色, 例如用来配置应用程序的任何切面, 代替 java EE 旧版中所遗留的繁冗代码和 XML 配置等。

基本的 Annotation 介绍

使用 Annotation 时要在其前面增加 @ 符号, 并把该 Annotation 当成一个修饰符使用。用于修饰它支持的程序元素

三个基本的 Annotation:

- 1) @Override: 限定某个方法, 是重写父类方法, 该注解只能用于方法

@Override: 限定某个方法, 是重写父类方法, 该注解只能用于方法

```
class Father{
    public void fly(){
        System.out.println("Father fly...");
    }
}
class Son extends Father {
    @Override //说明
    public void fly() {
        System.out.println("Son fly....");
    }
}
```

➤ 补充说明: @interface 的说明
@interface 不是 interface, 是注解类 是jdk5.0之后加入的

➤ **Override 使用说明**

1. **@Override** 表示指定重写父类的方法（从编译层面验证），如果父类没有 fly 方法，则会报错
2. 如果不写 **@Override** 注解，而父类仍有 `public void fly(){}` ，仍然构成重写
3. **@Override** 只能修饰方法，不能修饰其它类，包，属性等等
4. 查看 **@Override** 注解源码为 `@Target(ElementType.METHOD)`，说明只能修饰方法
5. **@Target** 是修饰注解的注解，称为元注解，记住这个概念。

2) **@Deprecated**: 用于表示某个程序元素(类，方法等)已过时

a. **@Deprecated** 修饰某个元素，表示该元素已经过时

b. 即不在推荐使用，但是仍然可以使用

c. 可以修饰方法，类，字段，包，参数 等等

d. **@Deprecated** 可以做版本升级过渡使用

3) **@SuppressWarnings**: 抑制编译器警告

a. 当我们不希望看到这些警告的时候，可以使用 `SuppressWarnings` 注解来抑制警告信息

b. 在 `{ " " }` 中，可以写入你希望抑制(不显示)警告信息

c. 可以指定的警告类型有

`all`，抑制所有警告

`boxing`，抑制与封装/拆装作业相关的警告

`cast`，抑制与强制转型作业相关的警告

`dep-ann`，抑制与淘汰注释相关的警告

`deprecation`，抑制与淘汰的相关警告

`fallthrough`，抑制与 `switch` 陈述式中遗漏 `break` 相关的警告

`finally`，抑制与未传回 `finally` 区块相关的警告

`hiding`，抑制与隐藏变数的区域变数相关的警告

`incomplete-switch`，抑制与 `switch` 陈述式(`enum case`)中遗漏项目相关的警告

`javadoc`，抑制与 `javadoc` 相关的警告

3) 关于 `SuppressWarnings` 作用范围是和你放置的位置相关
元注解

JDK 的元 Annotation 用于修饰其他 Annotation

1) Retention //指定注解的作用范围，三种 SOURCE, CLASS, RUNTIME

说明

a. 只能用于修饰一个 Annotation 定义，用于指定该 Annotation 可以保留多长时间，@Retention 包含一个 RetentionPolicy

b. 类型的成员变量，使用 @Retention 时必须为该 value 成员变量指定值：

c. @Retention 的三种值

RetentionPolicy.SOURCE: 编译器使用后，直接丢弃这种策略的注释

RetentionPolicy.CLASS: 编译器将把注解记录在 class 文件中，当运行 Java 程序时，JVM 不会保留注解。这是默认值

RetentionPolicy.RUNTIME: 编译器将把注解记录在 class 文件中，当运行 Java 程序时，JVM 会保留注解。程序可以通过反射获取该注解

2) Target // 指定注解可以在哪些地方使用

3) Documented //指定该注解是否会在 javadoc 体现

➤ 基本说明

@Documented: 用于指定被该元 Annotation 修饰的 Annotation 类将被 javadoc 工具提取成文档，即在生成文档时，可以看到该注解。

说明: 定义为 Documented 的注解必须设置 Retention 值为 RUNTIME。

4) Inherited //子类会继承父类注解

被它修饰的 Annotation 将具有继承性.如果某个类使用了被 @Inherited 修饰的 Annotation, 则其子类将自动具有该注解