Xiaomage\_biji

十六进制：0 1 2 3 4 5 6 7 8 9 A B C D E F

10 11 12 13 14 15

**007 原码 补码和反码**

原码 补码（所有运算都是补码运算） 反码

正数的原码、反码、补码都相同

负数的反码是在保持符号位为1的前提下，将各位取反

负数的补码则是在负数的反码的基础上在末位加1

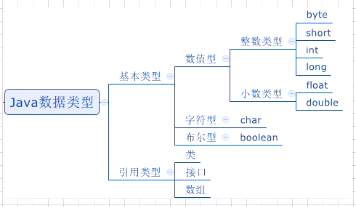


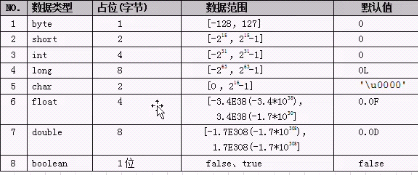
**配置环境变量：JAVA\_HOME**

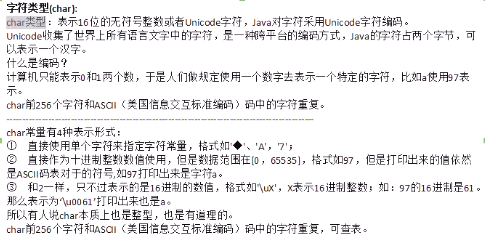
**在path中增加：%JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin**

**配置环境变量CLASS\_PATH:%JAVA\_HOME%\lib\...**

Java语言是编译性和解释性语言:源代码需要编译为字节码文件，而字节码文件有需要在jvm中解释执行。



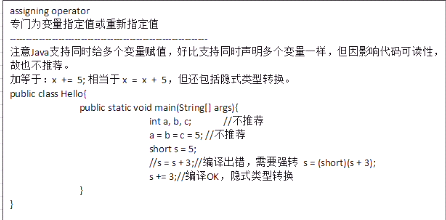


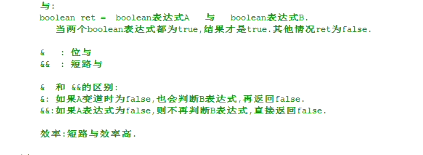


Char是无符号的。可以赋值为整数，范围为0~65535（0~2^16-1）

从某方面来讲，char就是整数类型，0~2^16-1，最高位不是符号位

+= -= \*= /= 包含隐式类型转换



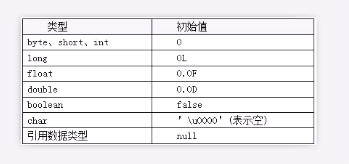


一个短路一个不短路



For循环性能更高

成员变量初始值



Foreach是for的语法糖，底层仍然是由for来循环的，只是又重新将数组元素赋值给了一个新的变量。

正因为如此，不能在foreach循环中改变数组的值

底层实现仍然是数组类型，如果定义两个同名的方法，一个参数为数组，一个为…，编译的时候会报错，定义了两个相同的方法。

这是编译器级别的新特性；只是编译器在接受成参数时将所有的可变参数组合为一个数组；因此可变参数必须作为方法参数的最后一个。

**软件的生命周期：**

**需求-分析-设计-实现-测试-部署-维护**

匿名对象new的是接口或者抽象类

变量在内存中停留的时间越短则性能越高。因此访问局部变量比访问成员变量更快

不要随意使用static修饰，非常消耗内存，static存在于整个程序运行的声明周期

也不要使用成员变量，因为存在线程的不安全问题（单例模式下）、

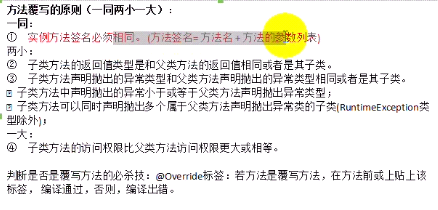
能使用局部变量就使用局部变量，因为局部变量是使用时单独开辟空间，用完即销毁。

This可以用于构造器重载时的互相调用。而且只能出现在另一个构造器代码块的第一句话，调用父类的构造器语句也只能出现在子类构造器的第一句话

多参数构造器调用少参数的构造器

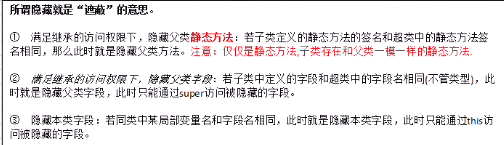
子类继承父类之后可以拥有父类的某些状态和行为，子类到底继承了父类的哪些成员根据访问控制修饰符来判断。

* Public修饰的可以被子类继承
* 缺省表示的字段和方法可以被同一个包中的子类继承
* Protected修饰的字段和方法可以被子类继承
* Private修饰的字段和方法不能被子类继承
* 父类的构造器不能被子类继承，构造器只能和当前的类名相同，但是可以通过super来引用父类的构造器

两大一小的原则也符合了多态，便于多态的使用，符合里氏替换原则。

因此在子类构造器中需要调用父类的构造器，如果没显示的写出调用父类的某个构造器则默认调用父类无参数构造器；如果父类不存在无参数构造器则需要在子类构造器中显示的调用有参数的构造器，否则子类对象的创建失败

字段不存在覆盖，叫做隐藏

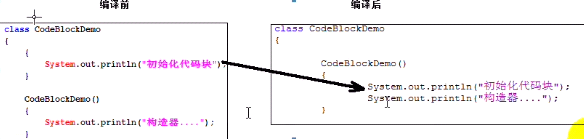


多态里氏替换



判断类型，使用反射字节码类型：

//判断准确的类型，使用字节码  
System.out.println(dog2.getClass()==Dog.class); //true  
System.out.println(dog2.getClass()==Animal.class); //error，编译不通过



通过反编译工具，初始化代码块在编译后也是作为构造器的最初语句，所以一般很少这样写，而是直接将初始化代码块放在构造器中。

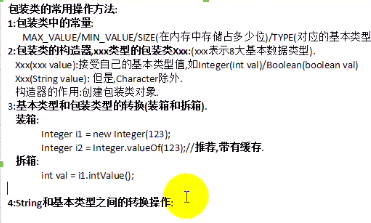
A类依赖B类，则B类先编译

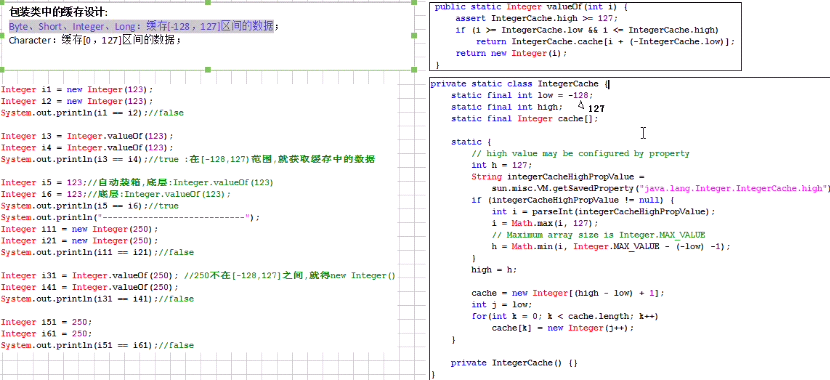


Final修饰的基本类型不能被再赋值；final修饰的引用类型变量的引用地址不能变，而不是引用地址的数据不能变。

Final 修饰类变量时不需要在定义时初始化但需要在构造器中为初始化的常量赋值，而final修饰的局部变量需要在定义时初始化。

包装类使用时，如果数值在指定的范围内会使用缓存



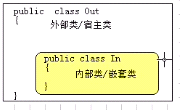


开发中建议使用包装类型，基本数据类型存储在栈中方法的栈帧中，包装类存储在堆中。

* 接口中没有构造器，接口不能创建对象
* 接口中定义的成员变量，实质是全局静态常量，默认使用public static final修饰
* 接口中定义的方法都是公共的抽象方法，默认使用public abstract修饰
* 接口中定义的内部类都是public static修饰的



Jar—类是java的压缩文件（因此可以使用wrar打开），存储的是java的字节码文件

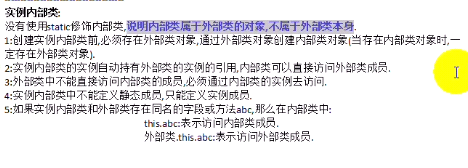




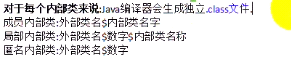
内部类可以看做是内部类的成员，可以使用通用的访问控制符



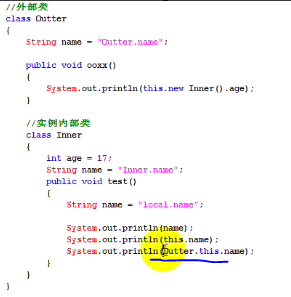
实例内部类：



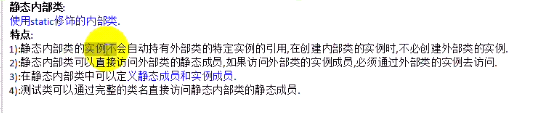
外部类.this.abc表示访问外部类成员。



**内部类访问外部类成员使用outter.this.字段或方法名**



静态内部类：

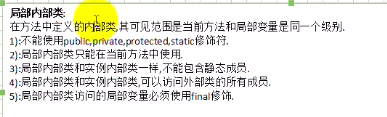


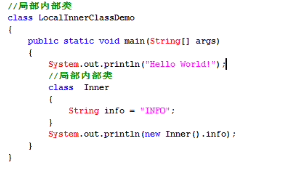
有静态内部类的时候可以没有外部类对象

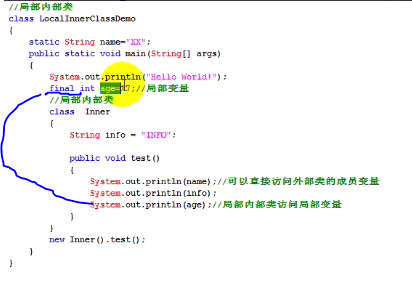


**10局部内部类**

可以把局部内部类当做局部变量(**所以修饰符可以是final但是不能是访问控制符，因为局部变量不能被访问控制符修饰**)



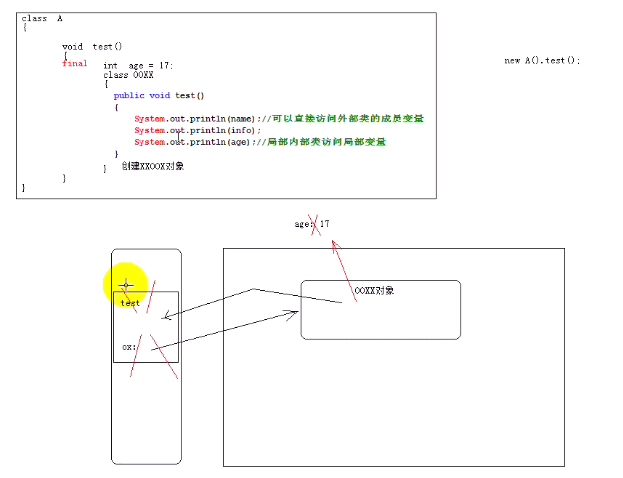






原因：如果当前方法不是main方法，那么当前方法调用完毕之后，当前方法的栈帧被销毁，方法内的局部变量的空间全部被销毁的时候，还存在堆内存，依然持有对局部的引用，但是方法被销毁的时候局部变量已经被销毁了。、

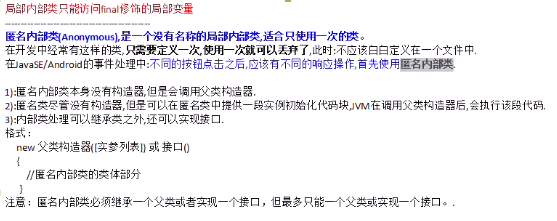
此时出现：在堆内存中，一个对象引用一个不存在的数据，为了避免该问题我们使用final修饰局部变量，从而编程常量永驻内存空间，即使方法销毁后也可以被持有。



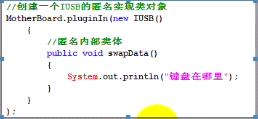
Age作为常量时就会被放在常量池里面，所以当方法调用完毕栈帧被销毁后仍然能够存在，而此时未被垃圾回收器回收的对象也可以引用此常量。

局部内部类几乎不会使用。

**11匿名内部类分析**



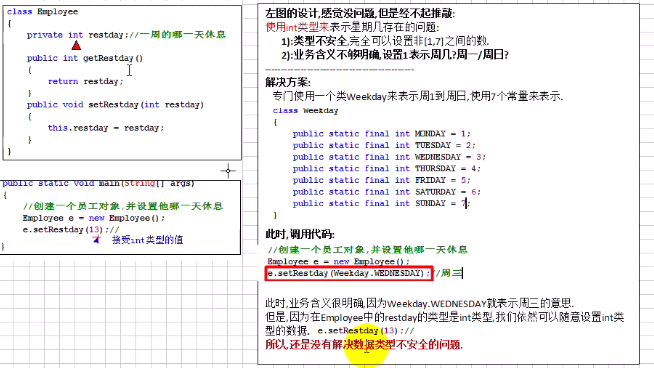
**匿名内部类**连类名都没有因此也就**没有构造器**。



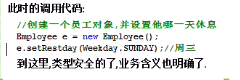
虽然是new接口，这只是语法格式，表名该匿名类实现了此接口。

匿名内部类访问局部变量也要加final修饰符，因为**匿名内部类是特殊的局部内部类**。

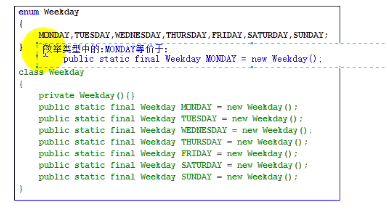
12引入枚举类型和枚举的模拟



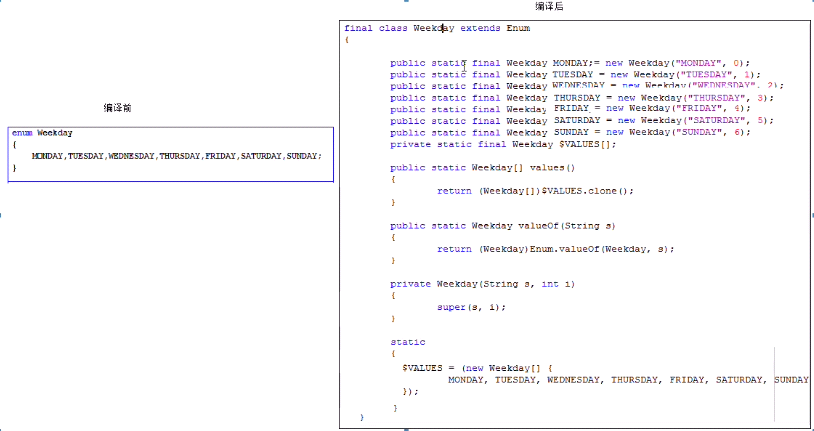


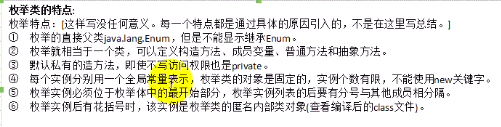


13枚举的定义和特点以及底层分析



反编译即可

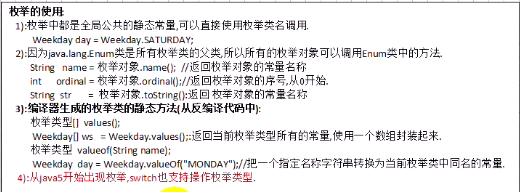


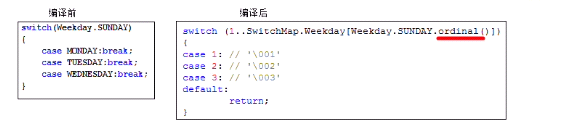


底层实现是

枚举底层没有无参数构造器

14枚举的操作细节





枚举类默认的是final的，因此不能被继承

完

使用魔法数值，代之以有名字的Static final或者enum值

静态：存放在jvm的常量方法区中，存在于jvm运行的整个生命周期；如果一个类要被声明为static的，只有一种情况，就是静态内部类，外部类不能使用static修饰会出现编译错误。

静态内部类builder模式即建造者模式，使用this链式，内部类和外部类保存了相同的成员变量。然后外部类的构造方法将内部类的实例作为参数来构造外部类的实例。

publicclass Outer {

private String name;

privateint age;

publicstaticclass Builder {

private String name;

privateint age;

publicBuilder(int age) {

this.age = age;

}

public Builder withName(String name) {

this.name = name;

return this;

}

public Builder withAge(int age) {

this.age = age;

return this;

}

public Outer build() {

return new Outer(this);

}

}

privateOuter(Builder b) {

this.age = b.age;

this.name = b.name;

}

}

**Volatile修饰符 强制从主存读和写数据**

安装jdk三步：

1. jdk下载安装
2. jdk环境变量配置:
3. JAVA\_HOME：jdk的安装路径/your/~/path/java-1.0.1
4. PATH:jdk的命令执行路径/your/~/path/java-1.0.1/bin
5. CLASSPATH:jdk类库文件的路径/your/~/path/java-1.0.1/lib
6. 安装配置验证:java –v javac -v

**注：位移运算**

左移运算符：<<

丢弃最高位，0补最低位

如果运算数小于int则先转换为int后运算

左移的位数是对数据类型的位数取模（例如：对int—4字节32位—左移32为相当于左移了32%32=0位）

在数字没有溢出的情况下左移一位相当于乘以2

右移运算符：>>

丢弃最低位，高位符号位扩展（保留符号位sign extension），即以符号位来填充

如果运算数小于int则先转换为int后运算

右移的位数是对数据类型的位数取模（例如：对int 右移32为相当于右移了32%32=0位）

在数字没有溢出的情况下，右移一位相当于除以2

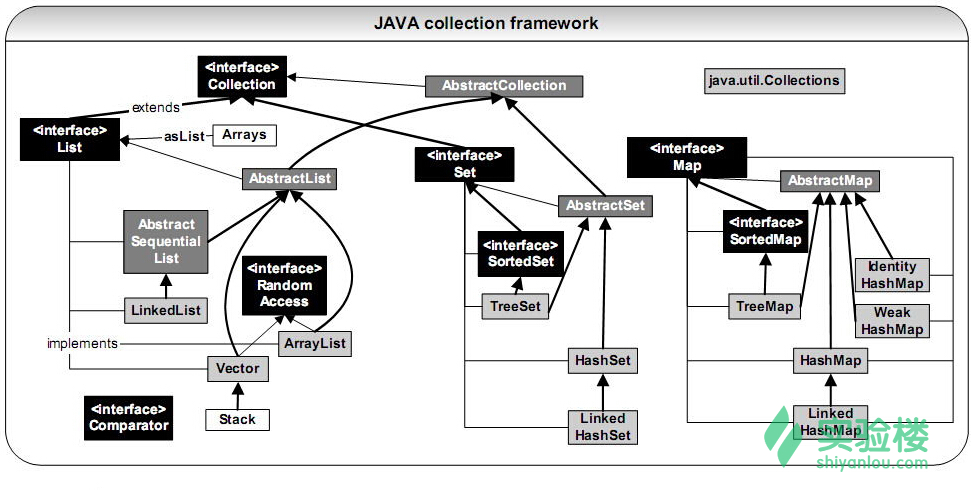
Java中的数字都是有符号的

一种引用类型转换是向上类型转换（隐式/自动类型转换），是小类型到大类型的转换（无风险）。另一种是向下类型转换（强制类型转换），是大类型到小类型（有风险）。

这种向上转换是不存在风险的，计算机可以自动帮我们完成，所以向上类型转换又叫做自动类型转换或隐式类型转换。而如果我们想将壶里的水倒进杯子里，就可能会产生溢出，这是有风险的。计算机是不会自动完成这样的操作。但如果程序猿想要完成这样的操作，就只能强制转换。所以向下类型转换又叫做强制类型转换

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

Java集合框架



Java 自带了各种 Map 类，这些 Map 类可归为三种类型：

* 通用 Map（HashMap、Hashtable、Properties、LinkedHashMap、IdentityHashMap等）
* 专用 Map（java.util.jar.Attributes、javax.print.attribute.standard.PrinterStateReasons等）
* 一个用于帮助实现您自己的 Map 类的抽象类

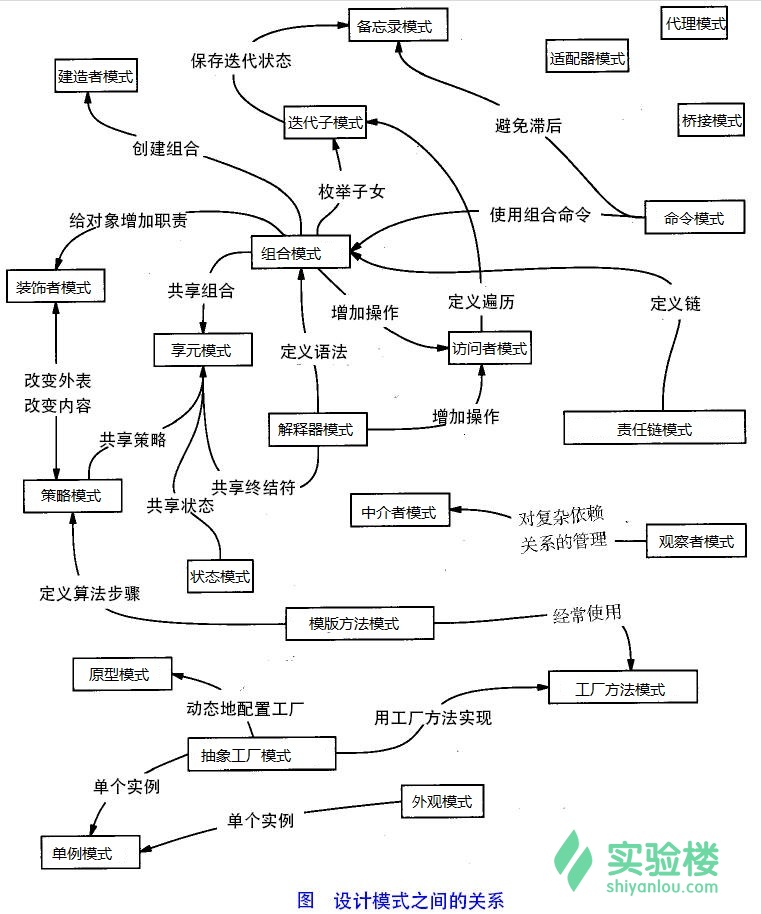
异常通常有四类：

* Error：系统内部错误，这类错误由系统进行处理，程序本身无需捕获处理
* Exception：可以处理的异常
* RuntimeException：可以捕获，也可以不捕获的异常
* 继承Exception的其他类：必须捕获，通常在API文档中会说明这些方法抛出哪些异常

try语句块是必不可少的，catch和finally语句块可以根据情况选择其一或者全选（catch和finally语句块至少有一个）

j2se核心开发实战\_shiyanlou，未完待续，实验楼java 进阶

Java进阶之设计模式



Atomicinteger理解

在使用**Integer**的时候，必须加上**synchronized**保证不会出现并发线程同时访问的情况，而在**AtomicInteger**中却不用加上**synchronized**，在这里**AtomicInteger**是提供原子操作的

AtomicInteger可以在并发情况下达到原子化更新，避免使用synchronized，而且性能非常高

深入解析单例模式：

静态内部类实现单例模式：

1package singleton;

2

3publicclass GracefulSingleton {

4private GracefulSingleton(){

5 System.out.println("创建GracefulSingleton实例一次！");

6 }

7  
     //类级的内部类，也就是静态的成员式内部类，该内部类的实例与外部类的实例没有绑定关系，而且只有被调用到才会装载，从而实现了延迟加载

8private static class SingletonHoder{  
　　　　　　　//静态初始化器，由JVM来保证线程安全

9private static GracefulSingleton instance = new GracefulSingleton();

10 }

11

12public static GracefulSingleton getInstance(){

13return SingletonHoder.instance;

14 }

15 }

Web应用是否能够实现单例，要看web容器是如何实现的：

Java+servlet和以此技术为基础的框架是可以实现的；（这也间接说明了static类型的变量在）

Php+ngin则不能，每次都是新的线程。

small\_model\_study

java\_enum

枚举也可以实现接口，一般很少这样做。

public interface Food {  
enum Coffee implements Food{  
 *BLACK\_COFFEE*,*DECAF\_COFFEE*,*LATTE*,*CAPPUCCINO*}  
  
enum Dessert implements Food{  
 *FRUIT*,*CAKE*,*GELATO*  
}

}

枚举类似全局静态常量，所以可以在外部访问，即使枚举是内部类。