## 1 考虑用静态工厂方法代替构造器

优势：

1. 有名称，用方法名标识获取对象的名称
2. 不必每次调用都创建新的对象
3. 可以返回原返回类型的任何子类型的对象
4. 在创建参数化类型实力的时候，它们使代码变得更加简洁

缺点：

1. 类如果不含有公有的或者受保护的构造器，就不能被子类化
2. 静态工厂方法与其它静态方法没有任何区别

常用名称：

valueOf

of

getInstance

newInstance

getType

newType

## 2 遇到多个构造器参数时考虑使用构建器

当类的属性较多时，用一般的构造方法，重载情况多。用对象的构建器来创建对象。

如：

Component c=new Component.Builder(a,b).field1(c).field2(d).build();

其中 Builder为静态内部类，field方法都返回Builder(return this)，build方法返回Component对象。

缺点：创建对象需要先创建构造器，因此它只有在有很多参数的时候才使用。

## 3 用私有构造器或者枚举类型强化Singleton属性

用私有构造方法或枚举型来强化单例模式

单例模式：

枚举型：

public enum Elvis{

INSTANCE;

public void leaveTheBuilding(){...}

}

## 4 通过私有构造器强化不可实例化的能力

对于一些只提供静态方法的工具类，我们不希望它被实例化，则可以用私有构造器来避免。如：

public class ToolUtils{

private ToolUtils(){

throw new Exception(“本类内部实例化抛异常”);

}

// other static method.

}

## 5 避免创建不必要的对象

对于在类的方法中每次调用都需要创建相同的对象，那么就可以使用静态属性及静态代码块来初始化类时创建一次,例如A类中有个方法a要比较传入的日期是否在两个固定的日期之间，一种方法是在a中构建两个日期，再比较。这样每次比较都会创建两个日期对象。简化的方法就是在类中定义两个静态成员变量。

public class Person{

private static final Date START;

private static final Date END;

//other fields

static {

//initial code can move to method isInDateArea() and initialize for the first calling.

SimpleDateFormat sdf=new SimpleDateFormat(“yyyy-MM-dd HH:mm:ss”);

START=sdf.parse(“1946-01-01 00:00:01”);

END=sdf.parse(“1965-01-01 00:00:00”);

}

public boolean isInDateArea(Date birthday){

return birthday.compareTo(START) >=0 &&birthday.compareTo(END)<0;

}

}

基本类型优先于包装类型使用。当重用对象的代价大于创建对象时不建议重用对象，例如自己的不必要的对象池，维护对象池的开销大于创建对象。除非像数据库连接池那样，创建连接对象的代价非常大，则可以用连接池来复用连接对象。

## 6 消除过期的对象引用

异常内存泄漏的原因：

1. 自定义栈中过期元素没有清除。——不用的元素要设为null
2. 过期不用的缓存。
3. 无需长久存在的监听器和回调没有显示取消。

## 7 避免使用终结方法

终结方法 super.finalize()：不一定及时执行，也不一定会执行。可用于终止非关键的本地资源。如果用终结方法作为安全网，要记得记录终结方法的非法用法。

## 8 覆盖equals方法时遵守通用约定

约定：

1. 自反性：对于任何非null的引用值x, x.equals(x)必须返回true。
2. 对称性：对于任何非null的引用值x,y。当y.equals(x)=true时，x.equals(y)=true。
3. 传递性：非null引用值x,y,z 当x.equals(y)=true,y.equals(z)=true,则x.equals(z)=true。
4. 一致性：非null引用值x,y，不管怎样操作，只要x,y信息没修改，则多次调用x.equals(y)会一致返回true或false。
5. 非空性：所有引用不能为null。

告诫：

1. 覆盖equals时总要覆盖hashCode。
2. 不要企图让equals方法过于智能。
3. 不要将equals声明中的Object对象替换为其它类型方法参数类型要为Object。
4. 在每次覆盖equals方法都逐一审查equals 的五个约定

## 9 覆盖equals时总要覆盖hashCode

覆盖equals时如果不覆盖hashCode，会导致该类无法结合所有基于散列的集合一起正常运作，这样的集合包括HashMap,HashSet和HashTable。

覆盖hashCode约定：

1. 在应用程序执行期间，只要对象equals方法比较操作所用到的信息没有被修改，那么对这同一个对象调用多次，hashCode方法都必须始终如一地返回同一个整数。同一个应用程序多次执行过程中，每次执行所返回的整数可以不一致。
2. 如果两个对象equals比较是相等的，那么这两个对象任意一个的hashCode方法都必须产生同样的整数结果。
3. 如果两个对象equals比较是不相等的，那么这两个对象任意一个的hashCode方法都必须产生不同的整数结果。这样有可能提高散列表的性能。

利用equals比较的每个域来生成hashCode：

a 计算域f的散列值c

1. 如果该域为boolean类型，则计算(f?1:0)。
2. 如果该域为byte ,char,shot或者int类型，则计算(int)f。
3. 如果该域为long类型，则计算(int)(f^(f>>>32))。
4. 如果该域为float类型，则计算Float.floatToIntBits(f)。
5. 如果该域为double类型，则计算Double.doubleToLongBits(f)，然后按3为 得到的long类型值计算散列值。
6. 如果该域为一个对象引用，且该类的equals方法通过递归的调用equals方 法来比较这个域，则同样为这个域递归调用hashCode。
7. 如果该域为一个数组，则把每个元素当作一个单独的域来处理，每个域计 算一个散列码，然后根据b中的方法把这些散列值组合起来。

b 按下面的公式把a每个域得到的散列码c合并到result中

result=31\*reuslt+c

c 返回result。

## 10 始终要覆盖toString

注意：避免依赖于toString数据格式的细节进行编程。

## 11 谨慎地覆盖clone

## 12 考虑实现Comparable接口

对象会参与顺序排序的功能时，类可以考虑实现comparable接口，覆盖compareTo方法。参数类型与类一致。

@override

int compareTo(T t){

//比较整数型基本类型的域，可以使用关系操作符 < 和 >

//浮点型数据比较 用 Double.compare 或者 Float.compare

//如果有多个域参与比较，分清域的比较优先级，依次比较，如一个带有两个域(a,b,a比较优先级大于b)的类对象作比较时：

if(a<t.a)return -1;if(a>t.a)return 1;

if(b<t.b)return -1;if(b>b.b)return 1;

return 0;

}

## 13 使类和成员的可访问性最小化

对于顶层的（非嵌套的）类和接口，只有两种可能的访问级别：private，public

如果用public修饰，那它就是公有的。如果是private，那它就是包级私有的，它实际上成了包的一部分，而不是导出API的一部分，在以后的发行版本中，可以对它进行修改，替换或删除，而无需担心会影响到现有的客户端程序。如果做成公有的，那你就有责任永远支持它，以保证它们的兼容性。

如果一个包级私有的顶层类或接口只是在某一个类内部被用到，就应该考虑使它成为那个类的私有嵌套类，将可访问性由某个包缩小到到某个类。

类具有公有的静态final（不希望被修改）数据域或对象域，或者返回这种域（不希望调用后的使用过程中不会改变内部静态final属性的值）的访问方法，这几乎总是错误的，客户端能够修改数组或对象中的内容。可以使用以下两种方法：

1.数组变私有，增加公有的不可变列表

private static final Thing[] PRIVATE\_VALUES={...}

public static final List<Thing> VALUES=

Collections.unmodifiableList(Arrays.asList(PRIVATE\_VALUES));//add操作报错

此法对获得的VALUES不可作add,set操作，只能操作get后得到的对象。

1. 数组变私有，增加公有方法返回数组的备份

private static final Thing[] PRIVATE\_VALUES={...}

public static final Thing[] values(){return PRIVATE\_VALUES.clone();}

此法获得的values，长度不可变，但values[i]可以重新赋值。

除了公有静态final域的特殊情形之外，公有类都不应该包含公有域。并且要确保公有静态final域所引用的对象都不是可变的。

## 14 在公有类中使用访问方法而非公有域

定义私有域，提供公开getter/setter方法.

优点：保留将来改变该类的内部表示法的灵活性；可以对数据采取辅助行动或强加约束条件（校验get/set数据是否合法等）。

## 15 使可变性最小化

使用理由：不可变的类比可变类更加易于设计、实现和使用。它们不容易出错，且更加安全。

使用不可变类，需遵循以下5条规则：

1. 不要提供任何会修改对象状态的方法。
2. 保证类不会被扩展。一般定义成final的，后面还有其它方法。
3. 使所有的域都是final的。
4. 使所有的域都成为私有的。
5. 确保对于任何可变组件的互斥访问。如果类具有指向可变对象的域，则必须确保该类的客户端无法获得指向这些对象的引用。

优点：

不可变对象本质上是线程安全的，它们不要求同步；

不仅可以共享不可变对象，甚至也可以共享它们的内部信息；

不可变对象为其它对象提供了大量的构件。

缺点：对于每个不同的值都需要一个单独的对象。

除了定义final类之外，另一种不可变类的实现方法是让类的所有构造器都变成私有的或者包级私有的，并添加公有的静态工厂来代替共有的构造器。

public class Complex{

private final double re,im;

private Complex(double re,double im){this.re=re;this.im=im;}

public static Complex valueOf(double re,double im){return new Complex(re,im);}

}

## 16 复合优先于继承

继承的功能非常强大，但是也存在诸多问题，因为它违背了封装原则。只有当子类和超类之间确实存在子类型关系时，使用继承才是恰当的。如果子类和超类处在不同的包中，并且超类不是为了继承而设计的，那么继承将会导致脆弱性。为了避免这种脆弱性，可以用复合和转发机制来代替继承。

## 17 要么为继承而设计并提供文档说明，要么就禁止继承

超类中的方法a调用了b方法，那么子类覆盖b方法时要注意了，覆盖b方法是否会影响a方法的功能。所以超类需要提供说明文档。

当超类的方法实现发生改变时，需要编写子类对类进行测试，以确保对子类不会产生影响。

## 18 接口优先于抽象类

现有的类可以很容易被更新，以实现新的接口。

接口是定义mixin(混合类型)的理想选择。

接口允许我们构造非层次结构的类型框架。

骨架实现类：简单实现接口的所有方法。可以直接使用，也可以被继承后用户覆盖自己想要用到的方法。

接口通常是定义允许多个实现的类型的最佳途径。这条规则有个例外，即当演变的容易性比灵活性和功能更为重要的时候。在这种情况下，应该使用抽象类来定义类型，但前提是必须理解并且可以接受这些局限性。如果你导出了一个重要的接口，就应该坚决考虑同时提供骨架实现类。最后，应该尽可能谨慎的设计所有的公有接口，并通过编写多个实现来对它们进行全面的测试。

## 19 接口只用于定义类型

当类实现接口时，接口就充当可以引用这个类的实例的类型。

常量接口模式（只有static final修饰的常量）是对接口的不良使用。

## 20 类层次优于标签类

如(反例)

class Figure{

enum Shape{RECTANGLE, CIRCLE};

final Shape shape;

Figure(double radius){shape=CIRCLE;this.radius=radius;}

Figure(double length,double width){...}

double area(){

switch(shape){

case RECTANGLE:...

case CIRCLE: ...

}

}

}

更好的方法是用继承法

abstract class Figure{ abstract double area();}

calss Circle extends Figure{...}

class Rectangle extends Figure{...}

## 21 用函数对象表示策略

函数指针的主要用途就是实现策略模式。为了在JAVA中实现这种模式，①要声明一个接口来表示该策略，并且为每个具体策略声明一个实现了该接口的类。②当一个具体策略只被使用一次时，通常使用匿名类来声明和实例化这个具体策略类。③当一个策略是设计用来重复使用的时候，他的类通常就要被实现为私有的静态成员类，并通过共有的静态final域被导出，其类型为该策略接口。

① class StringLengthComparator implements Comparator<String>{...}

② Arrays.sort(stringArray,new Comparator<String>(){

public int compare(String s1,String s2){return s1.length()-s2.length();}

});

③ class Host{

private static class StrLenCmp implements Comparator<String>,Serializable{

public int compare(String s1,String s2){return s1.length()-s2.length();}

}

public static final Comparator<String>

STRING\_LENGTH\_COMPARATOR=new StrLenCmp();

}

## 22 优先考虑静态成员类

嵌套类是指被定义在另一个类的内部的类。嵌套类有四种：

①静态成员类 ②非静态成员类 ③匿名类 ④局部类

其中②③④被称为内部类。

如果声明成员类不要求访问外围实例，就要始终把static修饰符放在它的声明中，使它成为静态成员类，否则它的实例就会包含一个额外的指向外围对象的引用。保存这份引用要消耗时间和空间，并且会导致外围实例在符合垃圾回收时却仍然得以保留。并且使用时不需要有外围类的实例，直接静态引用就可以使用该类的实例。

匿名类除了在声明时之外，是无法实例化的。且不能执行instanceof，不能实现多个接口或者扩展一个类的同时实现接口，不能调用任何成员。它们必须简短，大约10行内，否则可读性差。常见用法：如21条创建函数对象，或者创建过程对象如Runnable,Thread,TimerTask实例，或者用在静态工厂方法内部(18条)。

简而言之，共有四种不同的嵌套类，每一种都有自己的用途。如果一个嵌套类需要在单个方法之外仍然是可见的，或者它太长了，不适合于放在方法内部，就应该使用成员类。如果成员类的每个实例都需要一个指向其外围实例的引用，就要把成员类做出非静态的；否则，就做成静态的。假设这个嵌套类属于一个方法的内部，如果你只需要在一个地方创建实例，并且已经有了一个预置的类型可以说明这个类的特征，就要把它做成匿名类；否则，就做成局部类。

## 23 请不要在新代码中使用原生态类型

声明中具有一个或者多个类型参数的类或者接口，就是泛型类或者接口。统称为泛型。

List<E> （E的列表）是泛型，List是原生态类型。

使用泛型，使类型规范化，使用时不必拆箱。

List<String> 是List的子类，但不是List<Object>的子类。所以List<String> 可以作为方法m(List list)的参数，而不能作为m(List<Object> list)的参数。

原生态类型不安全，可以使用List<Object> , List<?>等。

if(o instantceof Set){ Set<?> m = Set<?> o;}

不要再新代码中使用原生态类型，但有例外：在类文字中必须使用原生态类型。如List.class ,Stirng[].class , int.class 都合法,但是List<String.class>, List<?>.class都不合法。

总之，使用原生态类型会在运行时导致异常，因此不要在新代码中使用。原生态类型只是为了与引入反省之前的遗留代码进行兼容和互用而提供的。回顾：Set<Object>是个参数化类型，表示可以包含任何对象类型的一个集合；Set<?>则是一个通配符类型，表示只能包含某种未知对象类型的一个集合；Set则是一个原生态类型，它脱离了泛型系统。前两种是安全的，最后一种不安全。

|  |  |  |
| --- | --- | --- |
| 术语 | 实例 | 所在条例 |
| 参数化的类型 | List<String> | 23 |
| 实际类型参数 | Stirng | 23 |
| 泛型 | List<E> | 23,26 |
| 形式类型参数 | E | 23 |
| 无限制通配符类型 | List<?> | 23 |
| 原生态类型 | List | 23 |
| 有限制类型参数 | <E extends Number> | 26 |
| 递归类型限制 | <T extends Comparable<T>> | 27 |
| 有限制通配符类型 | List<? extends Number> | 28 |
| 泛型方法 | static <E> List<E> asList(E[] a) | 27 |
| 类型令牌 | String.class | 29 |

## 24 消除非受检警告

@SuppressWarnings注解可以用在任何粒度的级别中，变量、方法或类上。应该在尽可能小的范围内使用，以免掩盖了重要的警告。同时，你必须先消除一些经过，如泛型安全警告，然后证实代码警告是安全的，则可以使用此注解，并注释说明能忽略此警告的原因。

**25 列表优先于数组**

Object[] o=new Long[1];o[0]=”string”;//编译不报错，运行报错

List<Object> o1=new ArrayList<Long>();//编译报错

利用数组，运行时才会报错，而使用列表，编译时就可以发现错误。数组与泛型之间的第二大区别在于，数组是具体化的，因此只有在运行时才知道并检查它们的元素类型约束。而泛型则是通过擦除来实现的，因此泛型只在编译时强化它们的类型信息，并在运行时擦除它们的元素类型信息。擦除就是使反省可以与没有使用泛型的代码随意进行互用。

## 26 优先考虑泛型类

将集合声明参数化，或者使用JDK所提供的泛型和泛型方法，这些都不困难。但怎样编写自己的泛型？

将第6条的Stack存储数组进行改造成使用泛型：

public class Stack<E>{

private Object[] elements;

private int size=0;

private static final int DEFALUT\_INITIAL\_CAPACITY=16;

public Static(){

elements=new Object[DEFALUT\_INITIAL\_CAPACITY];

}

public void push(E e){

ensureCapacity();

elements[size++] =e;

}

@SuppressWarnings(“uncheked”)

public E pop(){

if(size==0) throw new EmptyStatckException();

Object result=elements[--size];

elements[size]=null;

return (E) result;

}

}

使用泛型比使用需要在客户端代码中进行转换的类型来得更加安全，也更加容易。在设计新类型的时候，要确保它们不需要这种转换就可以使用。这通常意味着要把类做成是泛型的。只要时间允许，就把现有的类型都泛型化。这对于这些类型的新用户来说会变得更加轻松，又不会破坏现有的客户端。

## 27 优先考虑泛型方法

静态工具方法尤其适合于泛型化。编写泛型方法与编写泛型类相类似。

泛型方法的类型限制必须放在返回类型前面

public static <T extends Comparable<T>> T max(List<T> list){...}

public static <T> T max1(List list){ ...}

泛型方法就像泛型一样，使用起来比要求客户端转换输入参数并返回值的方法来得更加安全，也更加容易。就像类型一样，你应该确保新方法可以不用转换就能使用，这通常意味着要将它们泛型化。并且就像类型一样，还应该将现有的方法都泛型化，使新用户使用起来更加轻松，又不会破坏现有的客户端。

**28 利用有限制通配符来提升API的灵活性**

借助地26条的例子，添加方法：

public void pushAll(Iterable<E> src){for(E e:src){push(e);}}

但是使用时

Stack<Number> numberStack=new Stack<>();  
 Iterable<Integer> integers=new HashSet<>();  
 numberStack.pushAll(integers);

警告：pushAll要求使用Iterator<Number> 参数，不能用Iterator<Integer>类型参数。原因是Iterator<Integer> 不是Iterator<Number>的子类

解决问题：

使用有限制通配符来定义方法

public void pushAll(Iterable<? extends E> src){for(E e:src){push(e);}}

要把元素放入，要确保元素是原本类型或子类。

同理加入我们要把所有元素取出来放到指定的集合中，则需要原本类型或其父类的泛型集合来接收。

public void popAll(Collection<? super E> dst){while (size>0){dst.add(pop()); }}

助记：PECS -- producer-extends,consumer-super ，所有的comparable和comparator都是消费者（用 <? super E>）。

第27条有个方法public static <T extends Comparable<T>> T max(List<T> list){...}

使用通配符更好public static <T extends Comparable<? super T> T max(List<T> list){...}

如果类型参数只在方法声明中出现一次，就可以用通配符取代它：如果是无限制的类型参数，就用无限制的通配符取代它；如果是有限制的类型参数，就用有限制的通配符取代它。

public static void swap(List<?> list,int i,int j){。。。}  
 //public static <E> void swap(List<E> list,int i,int j){。。。 }

## 29 优先考虑类型安全的异构容器(((φ(◎ロ◎;)φ)))

。。。总而言之，集合API说明了泛型的一般用法，限制你每个容器只能有固定数目的类型参数。你可以通过将类型参数放在键上而不是容器上来避开这一限制。对于这种类型安全的异构容器，可以用Class对象作为键。以这种方式使用的Class对象称作类型令牌。你也可以使用定制的键类型。例如，用一个DatabaseRow类型表示一个数据库行（容器），用泛型Column<T>作为它的键。

# 枚举和注解

## 30 用enum代替int常量

枚举类型：是指由一组固定的常量组成合法值的类型。java的枚举本质上是int值。

public enum Apple {FUJI,PIPPIN}

public enum Orange {NAVEL,TEMPLE,BLOOD}

对于不同枚举类型分别要覆盖枚举类型中的抽象方法，被称作特定于常量的方法实现。

public enum Operation{

PLUS { double apply(double x,double y){return x+y;} },

MINUS { double apply(double x,double y){return x-y;} },

TIMES { double apply(double x,double y){return x\*y;} },

DIVIDE { double apply(double x,double y){return x/y;} };

abstract double apply(double x,double y);

}

把每个类型与特定常量结合起来，并覆盖toString方法

public enum Operation{

PLUS(“+”) { double apply(double x,double y){return x+y;} },

MINUS(“-”) { double apply(double x,double y){return x-y;} },

TIMES(“\*”) { double apply(double x,double y){return x\*y;} },

DIVIDE(“/”) { double apply(double x,double y){return x/y;} };

private final String symbol;//枚举类型不可变，用final修饰，一旦赋值，不可改变。

Operation(String symbol){this.symbol=symbol;}

@Override public String toString(){return symbol;}

public static Operation fromString(String symbol){  
 for(Operation o:values()){ if (o.toString().equals(symbol)){return o;}}  
 throw new ClassCastException("未找到符号对应的枚举类型："+symbol);  
 }

abstract double apply(double x,double y);

}

什么时候使用枚举呢？每当需要一组固定常量（如菜单选项，操作代码，命令标记等）的时候，包括天然的枚举类型（如行星，四季，星期几等）。

## 31 用实例域代替序数