泛型

23、请不要在新代码中使用原生态类型

(1) 泛型指声明中具有一个或者多个类型参数的类或者接口。定义一组参数化的类型，构成格式为：类或者接口的名称，接着用尖括号<>把对应于泛型形式类型参数的实际类型参数列表括起来。例如 List<String>，是一个参数化的类型，表示元素类型为 String 的列表，（String 是与形式类型参数 E 相对应的实际类型参数）。

(2) 每个泛型都定义一个原生态类型，即不带任何实际类型参数的泛型名称，如 List<E>对应的原生态类型是 List。

(3) 这句话的意思是就是要避免使用 List，要采用 List<String>之类的代码

(4) 必须使用原生态类型的例外：（源于泛型信息在运行时被擦除）

a.类文字，如 List.class，String[].class 是合法的，List<String>，List<?>.class 是不合法的。

b.instanceof 操作符

```
if (o instanceof Set) {
        Set<?> m = Set<?> o;
        ...
}
```

红色标记的部分不需要使用 Set<?>，这样显得多余。但是，一旦确定 o 是 Set，就将它转换为 Set<?>

24、消除非受检警告

例如：

```
private static void unsafeAdd(List<String> list ,String o){
   list.add(o);
 }
```

编译时 java 提示：The method unsafeAdd(List<String>, String) from the type Super is never used locally

增加

```
@SuppressWarnings("all")
 private static void unsafeAdd(List<String> list ,String o){
  list.add(o);
 }
```

可以消除 java 编译时的 warnings;

要尽可能小的范围使用 @SuppressWarnings，同时要用注释记录下来该警告的原因；

关键字	用途
deprecation	使用了不赞成使用的类或方法时的警告
unchecked	执行了未检查的转换时的警告，例如当使用集合时没有用泛型 (Generics) 来指定集合保存的类型。
fallthrough	当 Switch 程序块直接通往下一种情况而没有 Break 时的警告。
path	在类路径、源文件路径等中有不存在的路径时的警告。
serial	当在可序列化的类上缺少 serialVersionUID 定义时的警告。
finally	任何 finally 子句不能正常完成时的警告。
all	关于以上所有情况的警告。

25、列表优于数组

(1)列表和数组的区别

a.数组是协变的(convariant)，如果 Sub 是 Super 的子类型，那么数组类型 Sub[]就是 Super[]的子类型。泛型确实不可变的，List<Sub>不是 List<Super>的子类型。

b.数组是具体化的(reified)，因此数组在运行时才知道并检查它们的元素类型约束。泛型则是则是在编译时进行检查，所以使用列表能够更早地发现错误

26、优先考虑泛型

我的理解就是：比如：一些类使用泛型：public class Stack 改成 public class Stack<E> ，可以无需进行显示强制转换类型

27、优先考虑泛型方法

考虑如下的方法，它的作用是返回两个集合的联合：

```
public static Set union(Set s1, Set s2) {
    Set result = new HashSet(s1);
    result.addAll(s2);
    return result;
}
```

虽然这个方法可以编译，但是编译这段代码会产生警告，为了修正这些警告(在新代码中不应该直接使用原始类型，当前是为了举例子)要将方法声名修改为声明一个类型参数，表示这三个元素类型(两个参数及一个返回值)，并在方法中使用类型参数。

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> result = new HashSet<E>(s1);
    result.addAll(s2);
    return result;
}
```

28、利用有限制通配符来提升 API 的灵活性

修改前：

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

修改后：

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

pushAll 的输入参数类型不应该为“E 的 Iterable 接口”，而应该为“E 的某个子类型的 Iterable 接口”，有一个通配符类型正符全此意：Iterable<? Extends E>

29、优先考虑类型安全的异构容器

(1)一般情况下，泛型最通常应用于集合，如 set 和 Map,以及单元素的容器，在这些语法中一般情况下，这些容器都被充当被参数化了的容器，意味着每个容器只能有一个或者多个固定数目的类型参数。如一个 Set 只有一个类型参数，用于表示元素类型。一个 Map 有两个类型参数，表示它的键和值类型。（个人认为类型参数就是指定了类型的参数。这也是泛型的意义所在）

(2)但是有时候我们可能需要更多的灵活性，比如数据库行可以有任意多的列，每个列的类型可能不一样，如果能以类型安全的方式访问所有列就好了。目前，有一种方法可以实现，就是将键值（key）进行参数化而不是对整个容器进行参数化。然后将参数化的键值提交给容器，来插入或者获取值。

```
public class Favorite {
   /**
    * 注意这个地方 Map 的 key 进行了参数化，而非是 Map 本身
    */
   private Map<Class<?>,Object>   map = new HashMap<Class<?>, Object>();
   private Map<List<?>,Object>   map2 = new HashMap<List<?>, Object>();
   public <E> void add(Class<E> type,E instance) {
      if(type!=null) {
         map.put(type, instance);
      }
   }
   public <E> void add(List<E> type,E instance) {
      if(type!=null) {
         map2.put(type, instance);
      }
   }
   public <E> E get(Class<E> type) {
      Object object = map.get(type);
      return type.cast(object);
   }
   public  static void main(String[] args) {
      Favorite f = new Favorite();
      f.add(String.class, "string");//可以插入 String 类型
      f.add(Integer.class, Integer.valueOf(123));//可以插入 int 类型
      f.add(new ArrayList<String>(), "String");//可以插入 String 类型
      f.add(new ArrayList<Integer>(), 123);//可以插入 int 类型
      String s1 = f.get(String.class);
      Integer i = f.get(Integer.class);
      System.out.println(s1);
      System.out.println(i);
   }
}
```

上面的例子就是实现了一个可以指定多个类型的插入和读取，且是类型安全的。