## 创建和销毁对象

## 考虑用静态工厂方法代替构造器

构造器是创建一个对象实例最基本也最通用的方法，大部分开发者在使用某个class的时候，首先需要考虑的就是如何构造和初始化一个对象示例，而构造的方式首先考虑到的就是通过构造函数来完成，因此在看javadoc中的文档时首先关注的函数也是构造器。然而在有些时候构造器并非我们唯一的选择，通过反射也是可以轻松达到的。我们这里主要提到的方式是通过静态类工厂的方式来创建class的实例，如：

*Public static Boolean valueOf( boolean b ){*

*return b ? Boolean.TRUE:Boolean.FALSE*

*}*

使用静态方法代替构造器有个非常大的优势就是，在程序每次调用的时候不用生成一个新的对象。

## 遇到多个构造参数时考虑用构建器

如果一个class在构造初始化的时候存在非常多的参数，将会导致构造函数或者静态工厂函数带有大量的、类型相同的函数参数，特别是当一部分参数只是可选参数的时候，class的使用者不得不为这些可选参数也传入缺省值，有的时候会发现使用者传入的缺省值可能是有意义的，而并非class内部实现所认可的缺省值，比如某个整型可选参数，通常使用者会传入0，然后class内部的实现恰恰认为0是一种重要的状态，而该状态并不是该调用者关心的，但是该状态却间接导致其他状态的改变，因而带来了一些潜在的状态不一致问题。与此同时，过多的函数参数也给使用者的学习和使用带来很多不必要的麻烦，我相信任何使用者都希望看到class的接口是简单易用、函数功能清晰可见的。

## 泛型

**声明中具有一个或者多个类型参数的类或者接口，就是泛型类或接口**

## 避免使用原生类多使用泛型

声明中具有一个或者多个类型参数的类或者接口，就是泛型类或接口，如List<E>，这其中E表示List集合中元素的类型。在Java中，相对于每个泛型类都有一个原生类与之对应，即不带任何实际类型参数的泛型名称，如List<E>的原生类型List。他们之间最为明显的区别在于List<E>包含的元素必须是E(泛型)类型，如List<String>，那么他的元素一定是String，否则将产生编译错误。和泛型不同的是，原生类型List可以包含任何类型的元素，因此在向集合插入元素时，即使插入了不同类型的元素也不会引起编译期错误。那么在运行，当List的使用从List中取出元素时，将不得不针对类型作出判断，以保证在进行元素类型转换时不会抛出ClassCastException异常。由此可以看出，泛型集合List<E>不仅可以在编译期发现该类错误，而且在取出元素时不需要再进行类型判断，从而提高了程序的运行时效率。

## 列表优先数组

数组和泛型相比，有两个重要的不同点。首先就是数组是协变（我的理解就是引用对对象乡下兼容）的，如：Object[] objArray = new Long[10]是合法的，因为Long是Object的子类，与之相反，泛型是不可协变的，如List<Object> objList = new List<Long>()是非法的，将无法通过编译。因此泛型可以保证更为严格的类型安全性，一旦出现插入元素和容器声明时不匹配的现象是，将会在编译期报错。二者的另一个区别是数组是具体化的，因此数组会在运行时才知道并检查它们的元素类型约束。如将一个String对象存储在Long的数组中时，就会得到一个ArrayStoreException异常。相比之下，泛型则是通过擦除来实现的。因此泛型只是在编译时强化类型信息，并在运行时丢弃它们的元素类型信息。

## 枚举

**枚举类型是指由一组固定的常量组成合法值的类型**

## 用enum代替int常量

## 用实例域代替序数

## 方法

## 检查参数的有效性

绝大多数方法和构造器对于传递给它们的参数值都会有些限制。比如，索引值必须大于等于0，且不能超过其最大值，对象不能为null等。这样就可以在导致错误的源头将错误捕获，从而避免了该错误被延续到今后的某一时刻再被引发，这样就是加大了错误追查的难度。就如同编译期能够报出的错误总比在运行时才发现要更好一些。事实上，我们不仅仅需要在函数的内部开始出进行这些通用的参数有效性检查，还需要在函数的文档中给予明确的说明，如在参数非法的情况下，会抛出那些异常，或导致函数返回哪些错误值等

## 必要时进行保护性拷贝

如果你的对象没有做很好的隔离，那么对于调用者而言，则有机会破坏该对象的内部约束条件，因此我们需要保护性的设计程序。该破坏行为一般由两种情况引起，首先就是恶心的破坏，再有就是调用者无意识的误用，这两种条件下均有可能给你的类带来一定的破坏性

## 谨慎重载

## 通用程序设计

## 将程序变量的作用域最小化

## for-each循环优先于传统的for循环

## 如果需要精确的答案，请避免使用float和double

## 基本类型优先于基本装箱类型

Java的类型系统中主要包含两个部分，分别是基本类型，如int、double、long，还有就是引用类型，如String、List等。其中每个基本类型都对应着一种引用类型，被称为装箱基本类型，如分别和int、double、long对应的装箱类型Integer、Double和Long等。

## 通过接口引用对象（接口编程）

## 异常

## 只针对异常情况才使用异常

## 避免不必要的使用受检异常

受检异常是Java提供的一个很好的特征。与返回值不同，它们强迫程序员必须处理异常的条件，从而大大增强了程序的可靠性。然而，如果过分使用受检异常则会使API在使用时非常不方便，毕竟我们还是需要用一些额外的代码来处理这些抛出的异常，倘若在一个函数中，它所调用的五个API都会抛出异常，那么编写这样的函数代码将会是一项令人沮丧的工作。

      如果正确的使用API不能阻止这种异常条件的产生，并且一旦产生异常，使用API的程序员可以立即采用有用的动作，这种负担就被认为是正当的。

## 优先使用标准异常

使用标准异常，不仅可以更好的复用已有的代码，同时也使你设计的API更加容易学习和使用，因为它和程序员已经熟悉的习惯用法更为一致。另外一个优势是，代码的可读性更好，程序员在阅读时不会出现更多的不熟悉的代码。

## 抛出与抽象相对应的异常

## 并发

## 同步访问共享的可变数据

在Java中很多时候都是通过synchronized关键字来实现共享对象之间的同步的。事实上，对象同步并不仅限于当多个线程操作同一可变对象时，仍然能够保证该共享对象的状态始终保持一致。与此同时，他还可以保证进入同步方法或者同步代码块的每个线程，都看到由同一个锁保护的之前所有的修改效果。       Java的语言规范保证了读写一个变量是原子的，除非这个变量的类型为long或double。换句话说，读取一个非long或double类型的变量，可以保证返回的值是某个线程保存在该变量中的，即时多个线程在没有同步的情况下并发地修改这个变量也是如此。然而需要特别指出的是，这样的做法是非常危险的。即便这样做不会带来数据同步修改的问题，但是他会导致另外一个更为隐匿的错误发生。

## 系列化

## 考虑使用自定义的序列化形式

设计一个类的序列化形式和设计该类的API同样重要，因此在没有认真考虑好默认的序列化形式是否合适之前，不要贸然使用默认的序列化行为。在作出决定之前，你需要从灵活性、性能和正确性多个角度对这种编码形式进行考察。一般来讲，只有当你自行设计的自定义序列化形式与默认的形式基本相同时，才能接受默认的序列化形式。比如，当一个对象的物理表示法等同于它的逻辑内容，可能就适合于使用默认的序列化形式。