## 创建和销毁对象

### Item1 考虑静态工场方法而不是构造器

优势：

1. 静态工厂方法的一个优点是，与构造函数不同，它们有名称。如果构造函数的参数本身并不能描述返回的对象，那么具有精心选择的名称的静态工厂更容易使用，生成的客户机代码也更容易阅读
2. 静态工厂方法的第二个优点是，与构造函数不同，它们不需要在每次调用时创建新对象。这允许不可变类使用预先构造的实例，或在构造实例时缓存实例，并重复分配它们以避免创建不必要的重复对象
3. 静态工厂方法的第三个优点是，与构造函数不同，它们可以返回返回类型的任何子类型的对象。
4. 静态工厂的第四个优点是，返回对象的类可以随输入参数的函数调用的不同而变化。声明的返回类型的任何子类型都是允许的。返回对象的类也可以因版本而异
5. 静态工厂的第五个优点是，当编写包含方法的类时，返回对象的类不需要存在。

劣势：

1. 仅提供静态工厂方法的主要限制是，没有公共或受保护构造函数的类不能被子类化。
2. 静态工厂方法的第二个缺点是程序员很难找到它们。

### Item2 在面对许多构造函数参数时，请考虑构建器

### Item3 使用私有构造函数或枚举类型执行单例属性

保证单例对象在序列化后还是同一个对象的方法，将属性设置成transient并提供一个readResolve方法。否则每次反序列化时都会创建一个对象

### Item4 使用私有构造函数强制非实例化

如有需要，可以通过包含私有构造函数，可以使类不可实例化，不要不写默认构造器，这样编译器会自动加上public修饰的构造器

### Item5 依赖注入优于硬连接资源p41

## 第四章 类与接口

### Item15 最小化类和成员的可访问性

最大程度的隐藏细节，只提供必要的api即可

经验法则很简单:

1、让每个类或成员尽可能不可访问

2、对于顶级(非嵌套)类和接口，只有两个可能的访问级别:包私有(默认情况下)和公共

3、如果包私有顶级类或接口仅由一个类使用，那么考虑将顶级类设置为使用它的唯一类的私有静态嵌套类

总之，应该尽可能减少程序元素的可访问性。在仔细设计了一个最小的公共API之后，

应该防止任何游离的类、接口或成员成为其中的一部分API。除了作为常量的公共静态final字段外，公共类应该没有公共字段。确保对象引用公共静态final字段是不可变的。

### Item16 在公共类中，使用访问器方法，而不是公共字段

### Item17 减少可变性

要使类不可变，请遵循以下5条规则：

1. 不要提供修改对象状态的方法
2. 确保类不能被扩展
3. 所有的属性都是final
4. 所有的属性都是private
5. 确保对任何可变组件的独占访问。如果您的类有任何引用可变对象的字段，请确保该类的客户端无法获得对这些对象的引用

扩展：

1. 泛函方法，方法返回对其操作数应用函数的结果，而不修改它。这强调了一个事实，方法不会改变对象的值，只会返回一个新值
2. 不可变对象很简单。不可变对象只能处于一种状态，即创建它的状态。不可变对象本质上是线程安全的;它们不需要同步
3. 关于可序列化性，应该添加一个注意事项。如果您选择让不可变类实现Serializable，并且它包含一个或多个引用可变对象的字段，那么您必须提供一个显式的readObject或readResolve方法。

或者使用ObjectOutputStream.writeUnshared和ObjectInputStream.readUnshared方法

### Item18 优先考虑组合而不是继承

### Item19 为继承而设计和记录或禁止

### Item20 与抽象类相比，更喜欢接口

### Item21 为后台设计接口

除非必要，否则应该避免使用默认方法向现有接口添加新方法，在这种情况下，您应该仔细考虑现有接口实现是否可能被默认方法实现破坏。然而，在创建接口时，默认方法对于提供标准方法实现非常有用，以减轻实现接口的任务

### Item22 只使用接口来定义类型

接口中定义常量是对接口的糟糕使用。类内部使用一些常量是实现细节而不应该暴露出来

总之，接口应该只用于定义类型。它们不应该仅仅用于导出常量

### Item23 更喜欢类层次结构而不是带标签的类

标记类很少合适。如果您想用显式标记字段编写类，请考虑是否可以消除标记并用层次结构替换类。当您遇到带有标记字段的现有类时，请考虑将其重构为层次结构

### Item24 支持静态成员类而不是非静态类

有四种嵌套类:

静态成员类、非静态成员类、匿名类和本地类。除了第一种，所有的类都被称为内部类

从语法上讲，静态成员类和非静态成员类之间的唯一区别是静态成员类在其声明中具有修饰符static。

尽管语法相似，但这两种嵌套类有很大不同。非静态成员类的每个实例都隐式地与包含类的封闭实例相关联。在非静态成员类的实例方法中，您可以调用封闭实例上的方法，或者使用限定的this构造获得对封闭实例的引用

***如果声明的成员类不需要访问封闭的实例，那么总是在声明中添加static修饰符，使其成为静态的而不是非静态的成员类，如果省略这个修饰符，每个实例都有一个隐藏的对其外部实例的无关引用。存储此引用需要时间和空间。更严重的是，它可能导致外部实例被保留，否则它将有资格进行垃圾收集。由于引用是不可见的，因此通常很难检测到***

私有静态成员类的一个常见用法是表示由其封闭类表示的对象的组件。例如，考虑一个映射实例，它将键与值关联起来。许多映射实现对于映射中的每个键-值对都有一个内部条目对象。虽然每个条目都与映射关联，但条目上的方法(getKey、getValue和setValue)不需要访问映射。因此，使用非静态成员类来表示条目是浪费:私有静态成员类是最好的

在lambdas被添加到Java之前，匿名类是动态创建小函数对象和进程对象的首选方法，但lambdas现在是首选方法。匿名类的另一个常见用法是实现静态工厂方法

总结：如果成员类的每个实例都需要引用其封闭实例，则使其非静态;否则，让它保持静态。

### Item25 不要在一个源文件中定义多个顶级类

永远不要将多个顶级类或接口放在一个源文件中。可以使用静态内部类的形式代替

## 泛型

### Item26 不要使用原始类型(尽量加上泛型)

如果您想使用泛型类型，但不知道或不关心实际的类型参数是什么，那么可以使用问号代替

注：泛型中T与？的区别

T：表示参数化类型，用于**类型形参**，比如**定义泛型类**、**泛型方法**或者**返回值**时使用

？：通配符，用于**类型实参**，在**使用泛型作为方法的参数时，就需要使用泛型通配符来解除传入参数的类型，类型参数就做不到**，如：

public void getClass1(TestA<?> test) {

}

//报错

public void getClass2(TestA<T> test) {

}

**不能将任何元素(除null之外)放入Collection<?>中，因为？通配符代表的是某一种实际参数类型**

对于不应该使用原始类型的规则，有一些小的例外：

1. List.class,String[].class和int.class都是合法的，但是List<String>.class和List <？>.class不合法
2. 第二个例外是instanceof运算符。由于泛型类型信息在运行时被删除，因此在非无界通配符类型之外的参数化类型上使用instanceof操作符是非法的

### Item27 消除未检查警告

### Item28 更喜欢List而不是Arrays

协变：如果Sub是Super的子类型，那么Sub[]就是Super[]的子类型

总之，数组和泛型具有非常不同的类型规则。数组是协变和具体化的;泛型是不变的并被擦除。因此，数组提供了运行时类型安全性，而不是编译时类型安全性，反之亦然。

### Item29 支持泛型类型

不能创建非可具体化类型的数组(泛型数组)，例如new E[]，每当您编写由数组支持的泛型类型时，都会出现这个问题。有两种合理的解决方法：

1. 第一种方法是直接创建Object[]，再将其转换为泛型类型，如:(E[])Object[]，但会有编译器警告
2. 创建Object[]，再获取元素的时候在转为E。

### Item30 支持泛型方法

public static <E> Set<E> union(Set<E> s1, Set<E> s2) {

Set<E> result = new HashSet<>(s1);

result.addAll(s2);

return result;

}

### Item31 使用有界通配符来增加API的灵活性

List<String>并不是List<Object>的子类型

例子：一个stack<E>类

public void pushAll(Iterable<E> src) {

for (E e : src)

push(e);

}

这样是有问题的，应该这样写：使用通配符

public void pushAll(Iterable<? extends E> src) {

for (E e : src)

push(e);

}

例子二：遍历栈中元素，放到集合dst中

public void popAll(Collection<E> dst) {

while (!isEmpty())

dst.add(pop());

}

这样的写法使用问题的，应该如下：使用通配符？和super，表示E是集合中类型的子类型(比如此处的？匹配Object)

public void popAll(Collection<? super E> dst) {

while (!isEmpty())

dst.add(pop());

}

为了获得最大的灵活性，在表示生产者或消费者的输入参数上使用通配符类型。如果输入参数既是生产者又是使用者，那么通配符类型就没有什么用处：需要一个精确的类型匹配，这是在没有通配符的情况下得到的结果

通配符(?)extends与super的使用环境：PECS启发式

如果一个参数化类型表示一个T生产者(或提供者)，使用<? extends T >，比如push(E e)

如果它表示一个T消费者，使用<? super T >，比如pop(Collection<? super T> dst)

如果类型参数在方法声明中只出现一次，则用通配符替换它

### Item32 明智地将泛型和参数变量(Object... object)组合起来

决定何时使用@SafeVarargs的规则很简单：

使用@SafeVarargs在每个方法上都有一个泛型或参数化类型的varargs参数

每当编译器警告，所控制的方法中的通用varargs（T... t）参数可能造成堆污染时，请检查该方法是否安全

提醒一下，如果:

1、它不会在varargs参数数组中存储任何东西

2、它不会使数组(或克隆)对不可信的代码可见。

如果违反了其中的任何一条禁令，请修复它

总之，varargs和泛型不能很好地交互，因为varargs工具是构建在数组之上的一个有漏洞的抽象，并且数组具有与泛型不同的类型规则。虽然它们是合法的，但是通用变量参数不是类型安全的

也即：在出现泛型的情况下，特别还有varargs(例如T... t)的情况下，切记方法不要直接返回t或者Object[]，需要将Object[]数组手动转换为List<T>返回，这样可以避免运行时发生ClassCastException.

### Item33 考虑类型安全异构容器

public class Favorites {

private Map<Class<?>, Object> favorites = new HashMap<>();

public <T> void putFavorite(Class<T> type, T instance) {

favorites.put(Objects.requireNonNull(type), instance);

}

public <T> T getFavorite(Class<T> type) {

return type.cast(favorites.get(type));

}

}

## 枚举和注解

### Item34 使用枚举而不是int常量

枚举本质上是int值

Java的enum类型的基本思想很简单:它们是通过公共静态final字段为每个枚举常量导出一个实例的类

enum类型还允许添加任意方法和字段，并实现任意接口

思考：如果每一种类型，我们只想根据属性来产生有限的几个实例(每个实例都可以看出是一种类型)，此时可以使用枚举，在枚举里面定义属性和方法，每一个枚举常量代表了一种类型，每种类型的属性都不一样！！！如太阳系八大行星：

public enum Planet {

MERCURY(3.302e+23, 2.439e6), //水星

VENUS (4.869e+24, 6.052e6), //金星

EARTH (5.975e+24, 6.378e6),

MARS (6.419e+23, 3.393e6),

JUPITER(1.899e+27, 7.149e7),

SATURN (5.685e+26, 6.027e7),

URANUS (8.683e+25, 2.556e7),

NEPTUNE(1.024e+26, 2.477e7);

private final double mass; // In kilograms 质量

private final double radius; // In meters 半径

private final double surfaceGravity; // In m / s^2

//方法略

}

八大行星的质量、半径及其他属性都不同，由此可以很好的建立模型！！！

**要将数据与enum常量关联起来，需要声明实例字段，并编写一个构造函数来获取数据并将其存储在字段中**

“如果从枚举类型中删除元素会发生什么?”

答案是，任何不引用已删除元素的客户端程序都能正常工作

每个行星常数都有不同的数据，但是有时候你需要将不同的行为与每个常数联系起来，可以使用swtich实现，如：

public enum Operation {

PLUS, MINUS, TIMES, DIVIDE;

// 每个枚举常量的行为都不同

public double apply(double x, double y) {

switch(this) {

case PLUS: return x + y;

case MINUS: return x - y;

case TIMES: return x \* y;

case DIVIDE: return x / y;

}

throw new AssertionError("Unknown op: " + this);

}

}

但是这种方式不好，代码脆弱且扩展性不好，用如下的方法：

**在enum类型中声明一个抽象的apply方法，并使用特定于常量的类主体中的每个常量的具体方法覆盖它。这些方法被称为常量特定的方法实现，如：**

public enum Operation {

PLUS (1){public double apply(double x, double y){return x + y;}},

MINUS (2){public double apply(double x, double y){return x - y;}},

TIMES (3){public double apply(double x, double y){return x \* y;}},

DIVIDE(4){public double apply(double x, double y){return x / y;}};

public abstract double apply(double x, double y);

Private Integer size;

}

总之，enum类型相对于int常量的优点是引人注目的。

枚举具有更强的可读性、安全性和功能。许多枚举不需要显式构造函数或成员，但其他枚举可以通过将数据与每个常量关联起来并提供行为受此数据影响的方法而获益。

更少的枚举受益于将多个行为与单个方法相关联。在这种相对少见的情况下，应该选择特定于常量的方法，而不是在自己的值上切换的枚举。如果某些枚举常量(但不是全部)共享公共行为，请考虑策略enum模式

### Item35 使用实例字段而不是序号

永远不要从枚举的序号中派生与枚举相关的值，如使用ordinal()方法；应该将其存储在实例字段中。

### Item36 使用EnumSet而不是位字段

### Item37 使用EnumMap而不是序号索引

### Item38 使用接口模拟可扩展的枚举

public interface Operation {

double apply(double x, double y);

}

public enum BasicOperation implements Operation {

PLUS("+") {

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

},

MINUS("-") {

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

},

TIMES("\*") {

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

},

DIVIDE("/") {

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

};

}

虽然您不能编写可扩展的枚举类型，但是您可以通过编写与实现该接口的基本枚举类型一起使用的接口来模拟它

### Item 39 与命名模式相比，更喜欢注释

处理可重复的注释需要小心，isAnnotationPresent明确表明，重复的注释不是注释类型，而是包含注释类型的注释。

### Item40 始终使用override注释

### Item41 使用标记接口来定义类型

标记接口是一个接口，它不包含方法声明，而只是指定(或“标记”)一个实现该接口的类具有某些属性

那么什么时候应该使用标记注释，什么时候应该使用标记接口?

显然，**如果标记应用于类或接口以外的任何程序元素，则必须使用注释**，因为只有类和接口才能实现或扩展接口；**标记类和接口应该使用标记接口**

**“如果您确实想定义类型，请使用接口。”**

## lambda和Stream

### Item42 喜欢lambdas而不是匿名类

### Item43 与lambdas相比，更喜欢方法引用

### Item44 喜欢使用标准的Functional Interface

既然Java已经有了lambdas，编写api的最佳实践已经发生了相当大的变化。

例如，模板方法模式就没有那么吸引人了，其中一个子类重写了一个基本的方法来专门化它的超类的行为。现代的替代方法是**提供一个接受函数对象的静态工厂或构造函数，以达到相同的效果**。更一般地说，您将编写更多**以函数对象为参数的构造函数和方法**。选择正确的函数参数类型需要谨慎

**java.util.Function包提供了大量的标准函数接口供您使用。**

**Interface Function Signature Example**

UnaryOperator<T> T apply(T t) String::toLowerCase

BinaryOperator<T> T apply(T t1, T t2) BigInteger::add

Predicate<T> boolean test(T t) Collection::isEmpty

Function<T,R> R apply(T t) Arrays::asList

Supplier<T> T get() Instant::now

Consumer<T> void accept(T t) System.out::println

### Item45 明智而审慎地使用Stream

1、流管道的评估是惰性的:评估直到终端操作被调用后才开始，而为了完成终端操作而不需要的数据元素永远不会被计算出来。这种懒惰的评估使无限的流工作成为可能。注意，没有终端操作的流管道是无声的不操作，所以不要忘记包含一个终端操作。

2、理想情况下，您应该避免使用流来处理char值

### Item46 更喜欢流中无副作用的函数

1、forEach操作是终端操作中最不强大的操作之一，也是最不支持流的操作之一。它是显式的迭代，因此不适合并行化。

2、forEach操作应该只用于报告流计算的结果，而不是执行计算

3、有时，将forEach用于其他目的是有意义的，例如将流计算结果添加到现有集合中

编程流管道的本质是无副作用的函数对象。这适用于传递给流和相关对象的所有函数对象。终端操作forEach只能用于报告由流执行的计算结果，而不是执行计算。为了正确使用流计算，您必须了解收集器

### Item47 更喜欢集合而不是作为返回类型的流

### Item48 在使流并行时要小心

1、不要不分青红皂白地并行化流管道。性能后果可能是灾难性的。

2、通常，并行性带来的性能收益最好在流上ArrayList、HashMap、HashSet和ConcurrentHashMap实例;数组;int Ranges;和Long ranges。这些数据结构的共同之处在于，它们都可以精确而廉价地分割成任意大小的子程序，这使得在并行线程之间划分工作变得很容易

3、并行性的最佳终端操作是reduce，即使用流的reduce方法组合管道中出现的所有元素

4、并行化一个流不仅会导致糟糕的性能，包括活性失败;它会导致不正确的结果和不可预知的行为

5、**总之，甚至不要尝试并行化流管道，除非您有充分的理由相信它将保持计算的正确性并提高其速度**

## 方法

### Item49 检查参数有效性

1. 对于public和protected的方法，使用Javadoc @throw标记来记录如果违反参数值限制将会引发的异常
2. Objects在Java 7中添加的requireNonNull方法灵活方便，因此不再需要手动执行null检查
3. 使用guava的Preconditions，Assert等断言工具
4. 非公共方法可以使用断言检查其参数
5. 在执行方法的计算之前，应该显式检查方法的参数

总结一下，每次编写方法或构造函数时，都应该考虑对其参数存在哪些限制。您应该记录这些限制，并在方法主体的开头使用显式检查强制执行它们。养成这样做的习惯是很重要的。在第一次有效性检查失败时，它所需要的少量工作将会得到利息回报。

### Item50 在需要的时候制作防御副本(不可变属性、对象时)

1. Date已过时，不应再用于新代码中
2. 所谓防御性副本，指的是：构造类时，类的属性不直接存储外部的引用(也即不与外部的指向同一个对象)，而是根据传入的引用从新构造一个新的对象；提供的访问接口也创建一个新的对象，这样可以保证对象内部的属性始终是不可变的，例如：

构造器：根据传入的对象引用创建新的对象，而不是直接应用旧的对象，这样即使旧的对象内容被更改也不会影响的类内部的不可变性

public Period(Date start, Date end) {

this.start = new Date(start.getTime());

this.end = new Date(end.getTime());

}

访问方法：提供新的对象而不是直接暴露内部属性，这样也保证了内部属性的不可变性

public Date start() {

return new Date(start.getTime());

}

public Date end() {

return new Date(end.getTime());

}

1. 参数的防御性复制不仅仅适用于不可变类。当您在内部数据结构中编写一个方法或构造函数来存储对客户机提供的对象的引用时，请考虑客户机提供的对象是否可能是可变的。如果是，那么考虑一下您的类在对象被输入到数据结构后是否能够容忍对象的更改。如果答案是否定的，那么您必须防御性地复制对象，并将副本输入到数据结构中，以代替原始数据结构

### Item51 仔细设计方法签名

### Item52 明智而审慎地使用重载

### Item53 明智而审慎地使用可变参数(Type... type)

1、varargs工具首先创建一个数组，其大小是在调用站点传递的参数数量，然后将参数值放入数组中，最后将数组传递给方法

2、需要用可变数量的参数定义方法时，varargs是非常宝贵的。在varargs参数前面加上任何需要的参数，并注意使用varargs的性能后果

### Item54 返回空集合或数组，而不是null

永远不要在空数组或集合的位置返回null。它使您的API更难使用，更容易出错，而且没有性能优势

### Item55 返回Optional<T>

1、不是所有的返回类型都能从可选的处理中获益。容器类型(包括集合、映射、流、数组和选项)不应该包装在选项中

如果您发现自己编写的方法不能总是返回值，并且您认为方法的用户每次调用该方法时都考虑这种可能性是很重要的，那么您可能应该返回一个可选的。

然而，您应该意识到返回选项会带来真正的性能后果;对于性能关键型方法，最好返回null或抛出异常。最后，除了作为返回值之外，您几乎不应该以任何其他方式使用可选值

### Item56 为所有公开的API元素编写文档注释

1. 要正确记录API，必须在每个导出的类、接口、构造函数、方法和字段声明之前加上doc注释。
2. @param、@return、@throws(未检查时异常也要标记出来)
3. 使用 {@code}包裹代码片段，如{@code index < 0 || index >= this.size()}
4. 每个doc注释的第一个“句子”(定义如下)成为注释所属元素的摘要描述
5. 泛型、枚举和注释在文档注释中需要特别小心。在记录泛型类型或方法时，一定要记录所有类型参数，如@param <K> 等

## 编程通用

### Iten57 最小化局部变量的范围

### Item58 与传统的for循环相比，更喜欢for-each循环

### Item59 了解和使用库

1. 在Java 7中，您不应该再使用Random。对于大多数使用，选择的随机数生成器现在是ThreadLocalRandom
2. 对于fork连接池和并行流，使用SplittableRandom

### Item60 避免float和double，如果需要准确的答案

使用BigDecimal、int或long来进行货币计算，

### Item61 与装箱的原语相比，更喜欢原始类型

对boxed原语应用==运算符几乎总是错误的

只要有选择，优先使用原语而不是盒装原语

### Item62 其他类型更合适的情况下避免使用String

1. 字符串不能很好地替代enum类型

### Item63 注意字符串连接的性能

为了获得可接受的性能，使用StringBuilder代替字符串来存储正在构建中的语句

### Item64 通过对象的接口引用对象

如果存在适当的接口类型，那么应该使用接口类型声明参数、返回值、变量和字段。真正需要引用对象类的唯一时间是使用构造函数创建它的时候

### Item65 选择接口而不是反射

### Item66 慎重使用本地方法

### Item67 明智的优化

### Item68 遵守公认的命名约定

## 异常处理

### Item69 仅在异常情况下使用异常

异常是为特殊情况而设计的。不要将它们用于普通的控制流，也不要编写迫使其他人这样做的api

### Item70 对于可恢复的条件使用检查异常，对于编程错误使 用运行时异常

决定使用检查时异常还是非检查异常的基本规则：

对于可以合理地期望调用者从中恢复的条件，使用检查时异常。通过抛出检查时异常，可以强制调用者在catch子句中处理该异常或将其传播出去

使用非检查时异常或错误来指示程序错误。如果程序抛出未检查(Runtime)的异常或错误，通常情况下恢复是不可能的，继续执行弊大于利

如果不清楚异常出现后恢复是否可能，您最好使用未检查异常

虽然Java语言规范并不要求这样做，但是有一个很好的约定，即错误(error)被保留给JVM使用，以指示资源不足、不变的故障或其他不可能继续执行的条件。考虑到这个约定几乎被普遍接受，最好不要实现任何新的错误子类

由于受控异常通常表示可恢复的条件，因此提供方法以帮助调用者从异常条件中恢复显得尤为重要。例如，假设当试图用礼品卡购买时由于资金不足而失败时，引发了一个检查异常。异常应该提供一个访问器方法来查询不足的数量。这将使调用者能够将金额传递给购物者

总结一下，对于可恢复的条件抛出检查异常，对于编程错误抛出未检查的异常。当有疑问时，抛出未检查的异常。不要定义任何既非检查异常也非运行时异常的可抛弃项。提供关于受控异常的方法，以帮助恢复

### Item71 避免使用不必要的检查时(Exception)异常

如果一个方法抛出一个检查过的异常，这个异常是该方法必须出现在try块中的唯一原因。如果方法没有抛出异常，则不应该将方法放在try块中

消除检查异常的最简单方法是返回所需结果类型的可选值optional，但缺点是不能够详细的描述异常发生的细节

### Item72 支持使用标准异常

最常用的重用异常类型是IllegalArgumentException

另一些经常重用的异常是IllegalStateException、ConcurrentModificationException、UnsupportedOperationException、NullPointerException、IndexOutOfBoundsException

ArithmeticException、NumberFormatException

不直接重用Exception、RuntimeException、Throwable或Error，应该将这些类视为抽象类

重用必须基于文档化的语义，而不仅仅是基于名称

另外，如果您想要添加更多的细节，可以随意子类化标准异常，但是要记住异常是可序列化的

如果没有参数值，抛出IllegalStateException，否则抛出IllegalArgumentException

### Item73 抛出适合于抽象的异常

**更高的层应该捕获较低级别的异常，并在它们的位置抛出可以用更高级别抽象解释的异常，这叫做异常转化**，例如：

// Exception Translation

try {

... // Use lower-level abstraction to do our bidding

} catch (LowerLevelException e) {

throw new HigherLevelException(...);

}

也即不应该在高层中抛出低层的异常，应该抛出抽象的异常

由于低级别的异常引发的高级别异常，可以通过高级别的异常的getCause方法获取低级别的异常

高级异常的构造函数将原因传递给感知链的超类构造函数，因此它最终被传递给一个感知链的构造函数，比如Throwable

// Exception with chaining-aware constructor

class HigherLevelException extends Exception {

HigherLevelException(Throwable cause) {

super(cause);

}

}

**尽管异常转换优于低级层中盲目传播异常，但它不应该被过度使用：**

在可能的情况下，处理来自较低层的异常的最佳方法是通过确保较低层方法成功来避免它们。这可以通过在将高级方法的参数传递到较低的层之前检查它们的有效性来做到这一点

如果不可能阻止来自较低层的异常，那么下一个最好的方法是让较高层安静地处理这些异常，将较高层方法的调用者与较低层问题隔离开来

**总之，如果无法防止或处理来自较低层的异常，则使用异常转换。如果较低层方法保证其所有异常都适用于较高层，则直接抛出**

### Item74 记录每个方法抛出的所有异常

始终单独声明已检查的异常，并使用Javadoc @throw标记精确记录每个异常的抛出条件

### Item75 在详细消息中包含故障捕获信息

### Item76 尽量维持失败的原子性

### Item77 不要忽略异常

## 并发

### Item78 同步访问共享可变数据

Volatile保证线程间的内存可见性，但不保证修改变量的原子性

### Item79 避免过度同步

为了避免死锁和数据损坏，永远不要从同步区域内调用外来方法。更一般地，将同步区域内的工作量保持在最小

### Item80 优先选择executors、tasks和stream，而不是线程

### Item81 更喜欢并发utils而不是wait和notify

与java.util.concurrent提供的高级语言相比，直接使用wait和notify类似于使用“并发汇编语言”进行编程。在新代码中使用wait和notify很少有(如果有的话)的理由。如果您维护使用wait和notify的代码，请确保它总是使用标准习惯用法从while循环中调用wait。

notifyAll方法通常应该优先用于notify。如果使用notify，必须非常小心以确保其活性

### Item82 文档说明线程安全性

### Item83 明智地使用延迟初始化

### Item84 不要依赖线程调度程序

## 序列化

### Item85 喜欢Java序列化之外的其他选择

1. 不可信数据的反序列化本质上是危险的，应该避免
2. 使用其他序列化框架，如json
3. 如果无法避免序列化(二进制)，并且不能绝对确定反序列化数据的安全性，请使用java9中的java.io.ObjectInputFilter，该工具允许您指定一个过滤器，该过滤器作用于反序列化之前的数据流。它在类粒度上运行，允许您接受或拒绝某些类。默认情况下接受类并拒绝潜在危险类的列表称为黑名单;默认情况下拒绝类并接受假定安全的类列表称为白名单。与黑名单相比，更喜欢白名单，因为黑名单只保护您免受已知的威胁

序列化是危险的，应该避免。如果您从头开始设计系统，那么可以使用跨平台的结构化数据表示，例如JSON或protobuf。不要反序列化不可信的数据。如果必须这样做，请使用java.io.ObjectInputFilter，但要注意，它不能保证阻止所有攻击。避免编写可序列化的类

### Item86 非常谨慎地实现Serializable

### Item87 考虑使用定制的序列化表单

1. 在没有考虑是否合适之前，不要接受默认的序列化表单。一般来说，只有当默认的序列化表单与您在设计自定义序列化表单时所选择的编码基本相同时，您才应该接受它
2. 默认的序列化方式是所有的非transient字段都进行序列化
3. 对象的默认序列化形式是基于对象的对象图的物理表示的合理有效编码。换句话说，它描述了对象中包含的数据以及从该对象可以访问的每个对象中的数据
4. 即使您认为默认的序列化表单是合适的，通常也必须提供readObject方法来确保不变量和安全性
5. 当对象的物理表示形式与其逻辑数据内容有很大差异时，使用默认的序列化形式有四个缺点:
   1. 它将导出的API永久地绑定到当前的内部表示
   2. 它会占用过多的空间
   3. 它会消耗过多的时间
   4. 它会导致堆栈溢出
6. 使用writeObject和readObject方法实现这个序列化表单。作为提醒，transient修饰符表示实例字段将从类的默认序列化表单中删除，例如：

public final class StringList implements Serializable {

private **transient** int size = 0;

private **transient** Entry head = null;

// No longer Serializable!

private static class Entry {

String data;

Entry next;

Entry previous;

}

//序列化方法

private void writeObject(ObjectOutputStream s) throws IOException {

s.defaultWriteObject();

s.writeInt(size);

// Write out all elements in the proper order.

for (Entry e = head; e != null; e = e.next)

s.writeObject(e.data);

}

//反序列化方法

private void readObject(ObjectInputStream s) throws IOException, ClassNotFoundException{

s.defaultReadObject();

int numElements = s.readInt();

// Read in all elements and insert them in list

for (int i = 0; i < numElements; i++)

add((String) s.readObject());

}

}

1. writeObject做的第一件事是调用defaultWriteObject，而readObject做的第一件事是调用defaultReadObject，即使所有StringList的字段是transient的
2. 如果类的所有实例字段都是瞬态的，那么可以不调用defaultWriteObject和defaultReadObject，但是序列化规范要求无论如何都要调用它们(这两个方法相当于**序列化类中的非transient字段**)，这些调用的存在使得在以后的版本中添加非transient实例字段成为可能，同时保持向后和向前兼容性
3. 接受哈希表的默认序列化表单将构成严重的错误。序列化和反序列化哈希表可能会产生一个不变量严重损坏的对象
4. 无论您是否接受默认的序列化表单，在调用defaultWriteObject方法时，未标记为transient的每个实例字段都将被序列化
5. 在决定使字段非瞬态之前，请确信它的值是对象逻辑状态的一部分。如果使用定制的序列化表单，大多数或所有实例字段都应该标记为transient
6. 如果使用默认的序列化表单，并且标记了一个或多个字段为瞬态，请记住，在反序列化实例时，这些字段将初始化为它们的默认值:对象引用字段为null，数值原语字段为0，布尔字段为false 。如果这些值对于任何暂态字段都是不可接受的，则必须提供一个readObject方法，该方法调用defaultReadObject方法，然后将暂态字段恢复到可接受的值。或者，这些字段可以在第一次惰性初始化
7. 无论您选择哪种序列化形式，在您编写的每个可序列化类中声明一个显式的串行版本UID：

private static final long serialVersionUID = randomLongValue;

### Item88 防御性地编写readObject方法

1. 简单地说，readObject是一个构造函数，它接受一个字节流作为惟一的参数
2. readObject方法，带有防御性复制和有效性检查：

private void readObject(ObjectInputStream s)throws IOException, ClassNotFoundException {

s.defaultReadObject();

// 防御性地复制可变组件

start = new Date(start.getTime());

end = new Date(end.getTime());

// 检查我们的不变量是否满足

if (start.compareTo(end) > 0)

throw new InvalidObjectException(start +" after "+ end);

}

### Item89 与readResolve相比，更喜欢枚举类型，例如控件

1. 任何readObject方法，无论是显式的还是默认的，都会返回一个新创建的实例，这个实例与在类初始化时创建的实例不同，这在一定程度上破坏了单例模式(如果实现了Serializable)
2. readResolve特性允许您用另一个实例替换readObject创建的实例。如果被反序列化的对象的类使用适当的声明定义了readResolve方法，则在新创建的对象被反序列化后调用该方法，该方法返回的对象引用将替代新创建的对象：

public class Elvis implements Serializable {

public static final Elvis INSTANCE = new Elvis();

private Elvis() { }

**private Object readResolve() {**

**return INSTANCE; //直接返回以前的实例 而不是新创建的实例 保护了单例模式**

**}**

}

1. 如果您依赖readResolve控制反序列化的实例，具有对象引用类型的所有实例字段都必须声明为瞬态的

总结一下，尽可能使用枚举类型来强制实例控制不变量。如果这是不可能的，并且您需要一个既可序列化又可实例控制的类，那么您必须提供一个readResolve方法，并确保类的所有实例字段要么是基本的要么是transient的

### Item90 考虑序列化代理，而不是序列化实例

实现Serializable的决定增加了bug和安全问题的可能性，因为它允许使用语言外机制代替普通构造函数创建实例。然而，有一种技术可以大大降低这些风险。这种技术称为**序列化代理模式**

**略！！！**