## 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

## 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