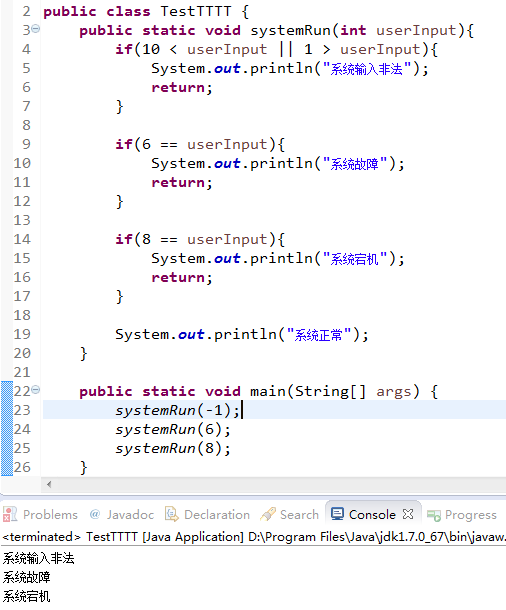
对异常的理解

\*\*异常是某一些高级语言出现的产物，但是 有一些语言 比如C语言就没有异常，是不是就不能完成比如Java中某些含有异常功能的逻辑了呢？

实际上不是的。

我们现在写一段代码：

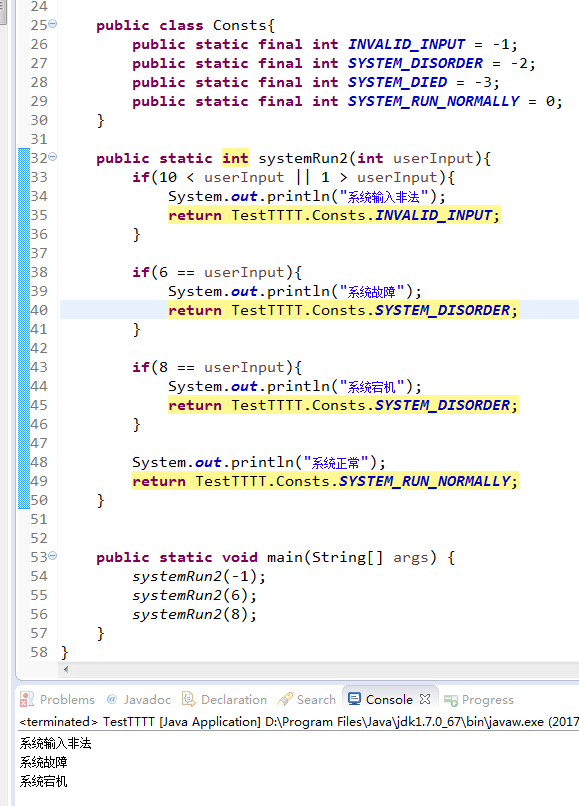
对用户的输入值进行判断，合法的用户输入的取值是1-10，但是遇到6，8的时候需要给出不同的提示 6表示系统故障 8表示系统宕机 以外的数值都是非法输入

这样 我的用户在调用这个接口的时候，由于输入的内容未知，没有限制，就可能随意输入。这样 调用了三次接口，返回的结果是不一样的。

结果就是 用户用了半天，也不知道这个接口里面有什么输入会有什么样子的返回。因此会觉得这个接口不好用。

需求变更：我的接口需要返回相应不同类型的值，告诉我的用户我的响应是什么意思。这样就要求开发者在文档中要详细描述这个接口有哪些返回值，每一个返回值是什么意思。

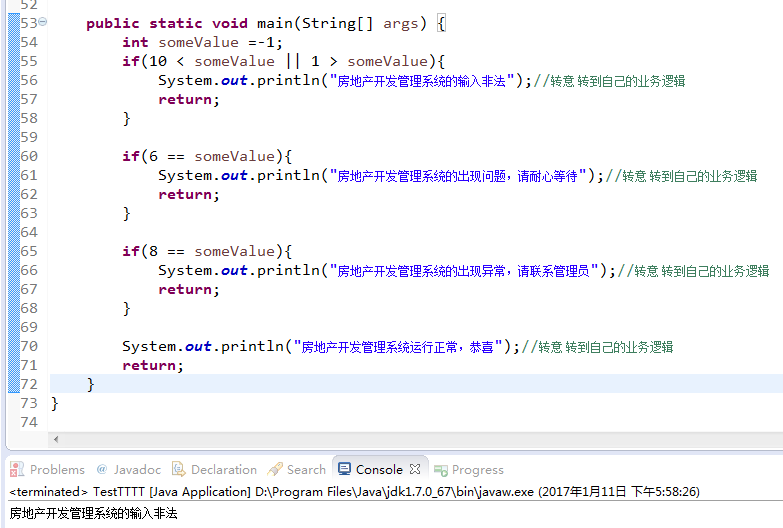
======== 第一次进化：把所有的功能直接写在用户的文档



用户使用的时候，由于运行过程，很可能触碰到了相关的临界值，导致又出现了上面的输出。如果用户没有看文档，或者看完文档忘记了输出的意思很可能就会出现和原来一样的情况。从而用户用起来非常不方便。

所以 提高不大。

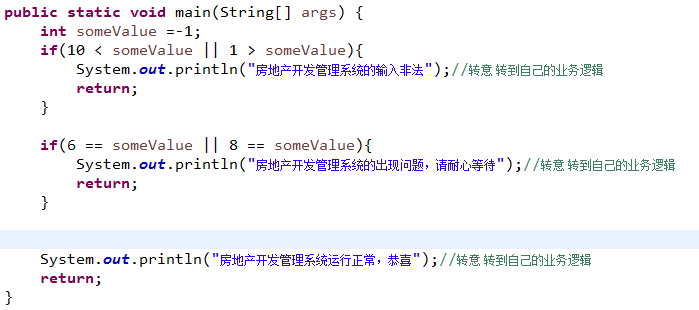
这时候用户做的代码改进如下：



**注意 这个时候 对于非法值的应对方式或者输出一定要根据自身业务才能给出合适的内容**

这里面虽然判断的逻辑和上面一样，但是给出的提示完全变了 这就是转义！ 业务逻辑含义的向上转义

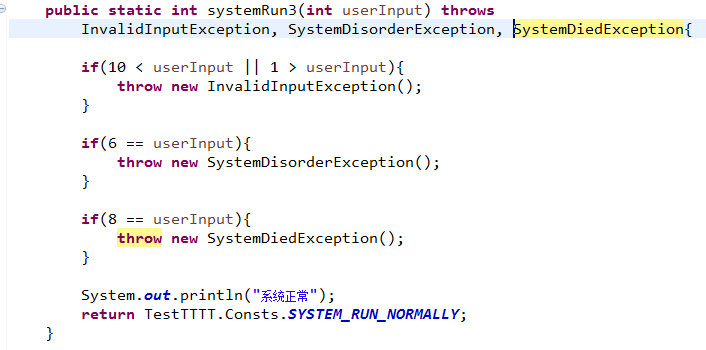
如果根据业务逻辑，认为6 和8 都要输出一样的内容，也就是这两个异常的输入引发的异常的处理方式一样，那么就如下：



这样的缺点已经描述 ----- 所有有没有异常对象的语言都可以做到任何业务逻辑无死角的覆盖

======== 第二次进化： 能不能把这样的不合法的输入值对应的结果以语法的形式强制让用户能看到呢？这样即使是用户没有阅读文档，一旦在编译代码调用了这个接口，如果没有处理可能有问题的情况，编译器也能提示用户有问题，要求用户早一点处理这样的输入呢？

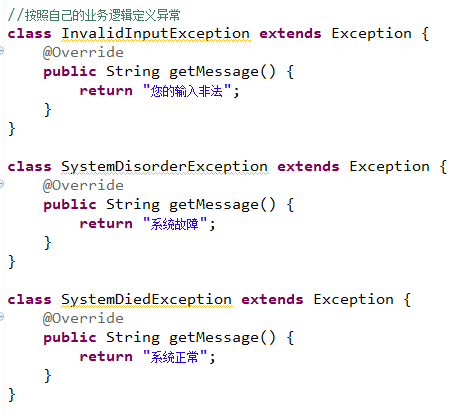
------ 这样 Java C++就引入了异常类



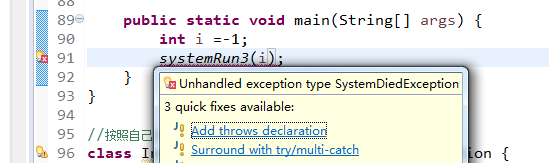
这样抛出的异常就是我根据我自己的业务的逻辑给出的返回值。

**异常就是一种对特定返回值的替代品**

虽然是三个不同的异常类的对象被抛出了 但是实际上就是业务逻辑的不同业务逻辑值的返回



**所以 以后你再看到任何框架定义的任何类型的异常，都不要怕，因为它是让你在编译代码的时候，就能看到这个方法或者接口可能出现哪些问题，提醒你是发生了某一种业务逻辑场景，需要你认认真真的去考虑，这种场景发生了，我该怎么处理。**

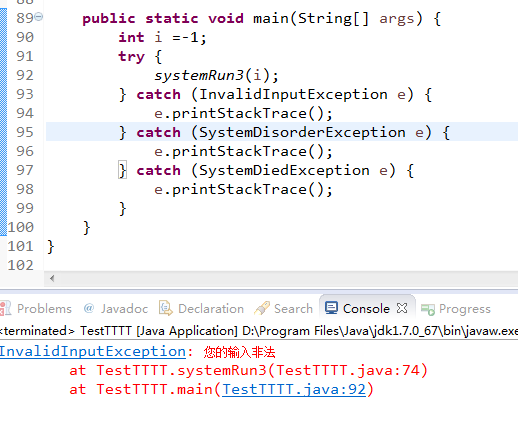


可以看到 由于异常的引入，导致我在调用这个接口的时候，没有运行的就会提示你有哪些有问题的情况可能发生，不处理是不行的！必须要处理！ ===== 就是**防患于未然**

**这个和Java中引入泛型是有异曲同工之妙**

这个确切的说是对检查类异常的理解

那么我这里面如何处理 那么处理的内容就需要按照自己的业务逻辑进行转义



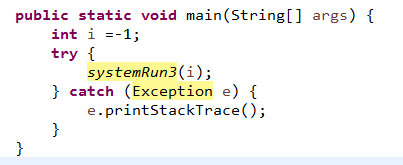
注意 这个eclipse生成的e.printStackTrace() 实际上就是一种eclipse默认的处理

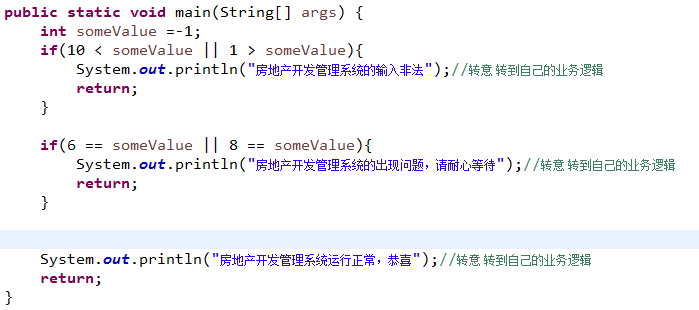
并没有做转义

======== 对异常是throws 还是try?

如果你也正在做一个接口，这个接口调用了基于某一个抛出异常的方法，注意 这个时候自己决定。如果你认为我这个异常可以在我接口使用这一层来决定 知道具体怎么处理，不需要通知出去 那么我就try ---否则 我就要抛

======= 通常对使用总的Exception的处理 相当于什么呢？



就类似于 

**也就是说 要求无论任何情况发生了，我都要做统一的处理。这种也好也不好，还是应该根据抛出的不同异常，做不同的处理或者转义再次抛出**

**----- Cactch的作用：相当于没有异常中的else 就是另一个场景**

也就是 catch完成 如果不向上转义抛出 我try之后的代码是可以通过catch的弥补 也就是相当于没有异常的else继续运行的

对于Java中的运行时异常RuntimeException

比如NullpointerException，没有throws 用户无法处理

Try也抓不到

这种运行时异常是为什么呢？

他就相当于 检查类异常的try{}catch{throw new Exception();}

也就是你catch到了 也停止工作了。

为什么？

比如空指针异常，明明对象不存在，是空的，你非要对空的对象获取什么属性或者方法 这不是开玩笑么？ 这个时候如果你后面的处理都是基于这个对象的话，好了 你都不用做了没法做了 直接结束程序运行。所以 这个时候运行时异常就出来了。

比如数组越界异常，你数组都没有那么大，你非要访问出界的内容，那不是错的么！即使访问到了又有什么用？ 程序退出吧！ 所以运行时异常的子类数组越界异常 就诞生了！

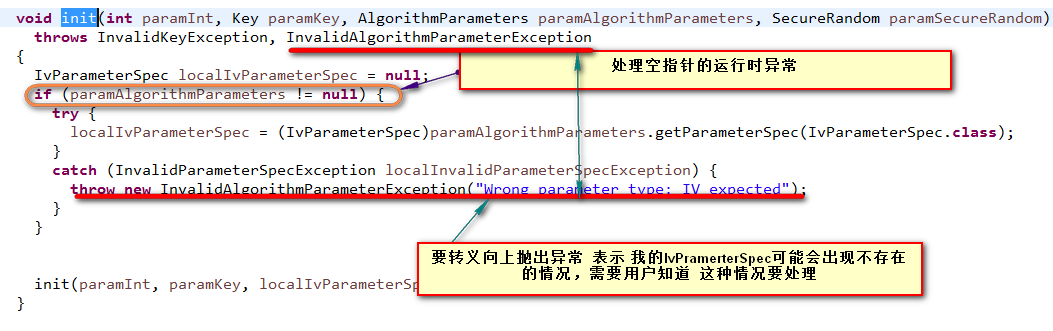
**====== 业务上如何处理运行时异常？=======**

至于 我们真的要不要达到运行时异常的效果，也就是应用发生运行时异常就停掉了，要根据我们的业务来确定。

**如果不想发生运行时异常，该怎么做呢？因为这个时候在编译的时候不会报错来强制你的对异常进行catch ----- 这个时候 就要回到最原始的情况，使用判断来找到运行时的异常的触发点，进行分支操作**

**所以 有的时候 代码中出现了try 还有if(null == obj )这个就是对检查类异常和运行时异常的不同处理！**

比如加密解密的时候，如果深入看到了源码：



这里面就抛出了一个InvalidaAlgorithmParameterException 的异常 你看到这个异常 就是见名知意，知道发生这个异常了，是我我的非法的算法参数导致的。

**所以 异常的另一个好处，以异常的名字在发生错误的时候，让用户能够见名知意！**

**====== 运行时异常向上转义为检查异常？=======**

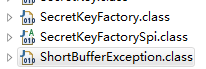
比如有些运行时异常在临街状态使用if检查出来的时候，觉得我不应该停掉程序 应该让使用者知道这个情况，可以直接抛出检查类异常 见名知意 抛给用户



**这里面好几处都是判断空指针的时候 给出了检查类异常 ---- 这就是根据业务逻辑来的**

所以 面对五花八门的异常类不要害怕 没有什么 主要在类名 发生的回收能够给你什么提示而已 里面都是Exception的子类

所以 既然异常和业务逻辑的某些异常的情景息息相关 ，那么很多开源框架的异常类都是放在相关的业务逻辑包中的。



可以看到 正常的类 和异常类 都放到一起了

**====== 异常的代价？=======**

异常虽然好，但是异常会额外产生很多对象，还有 对于异常要发生的地方，JVM在运行的时候 会给予足够的监视。所以 异常是耗费性能的。