**一下皆为官方文档的个人翻译，见：https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html**

**What Is an Exception?**

The term *exception* is shorthand for the phrase "exceptional event."

**Definition:** An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

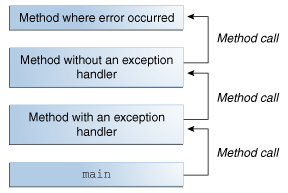
异常：打乱正常运行的事件

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

当问题出现的时候，将会**为这个问题创造一个对象**并且将它传递给“runtime system”。这个对象叫做异常对象，包含错误的信息（类型，当问题出现时程序的状态）。这个创造并传递的过程称为“抛出异常”。

After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack* (see the next figure).

方法被抛出后，运行系统就会按下图流程去找能处理它的东西，即被调用的一列有先后顺序的方法，他们叫做调用栈。



The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*.

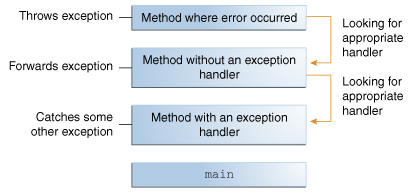
运行系统访问这个栈并找到能处理这个异常的代码块，这个代码块叫做异常处理者

/\*The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.\*/

The exception handler chosen is said to *catch the exception*.

代码处理者被选择了，就叫捕获异常

If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.



Searching the call stack for the exception handler.

Using exceptions to manage errors has some advantages over traditional error-management techniques. You can learn more in the [Advantages of Exceptions](https://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html) section.

**The Catch or Specify Requirement**

Valid Java programming language code must honor the *Catch or Specify Requirement*. This means that code that might throw certain exceptions must be enclosed by either of the following:

有意义的JAVA代码必须得遵守*捕获或提出 原则*：

* A try statement that catches the exception. The try must provide a handler for the exception, as described in [Catching and Handling Exceptions](https://docs.oracle.com/javase/tutorial/essential/exceptions/handling.html).
* A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception, as described in [Specifying the Exceptions Thrown by a Method](https://docs.oracle.com/javase/tutorial/essential/exceptions/declaring.html).
* Try语句来捕获异常，try必须一个处理者
* 一个方法声明了说，我能抛出这个异常。那么这个方法得有一个throws语句。

Code that fails to honor the Catch or Specify Requirement will not compile.

不遵守规则的代码不会被编译

Not all exceptions are subject to the Catch or Specify Requirement. To understand why, we need to look at the three basic categories of exceptions, only one of which is subject to the Requirement.

但是并不是所有的异常都遵守这个要求，为了理解为什么，我们需要了解异常的三大基本类型。

**The Three Kinds of Exceptions**

The first kind of exception is the *checked exception*. These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for java.io.FileReader. Normally, the user provides the name of an existing, readable file, so the construction of the FileReader object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws java.io.FileNotFoundException. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.

第一类叫做已知异常。这些意外情况往往是一个好程序都会预期并恢复的。比如常见的文件读取异常。

Checked exceptions *are subject* to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by Error, RuntimeException, and their subclasses.

已知异常是遵守*捕获或提出原则*的。所有的异常都是已知异常，除了那些由**Error，RuntimeException**或者它们的子类产生的

The second kind of exception is the *error*. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw java.io.IOError. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit.

第二类叫做error（错误）。这些意外情况是在应用外部的，所以一般程序都不能预期以及恢复。比如说，假如一个程序成功的打开一个文件作为输入，但之后发现无法读取文件因为硬件或者系统的鼓掌。这样会抛出IOerror。一个程序可能选择捕获这个程序，因此用户可以清楚这个问题的出现。但也可以打印出问题出错的位置然后退出。

Errors *are not subject* to the Catch or Specify Requirement. Errors are those exceptions indicated by Error and its subclasses.

Error不必要遵循这个原则。

The third kind of exception is the *runtime exception*. These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. For example, consider the application described previously that passes a file name to the constructor for FileReader. If a logic error causes a null to be passed to the constructor, the constructor will throw NullPointerException. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

第三类叫做运行时异常。这些意外情况是在应用内部的，所以也难以预期及恢复。这些问题一般意味着程序的bug。如果一个逻辑上的问题导致了null被使用了，那么将会造成NullPointerException异常。依然，程序可以选择捕获或者抛出它。但可能更有意义的是去发现它的bug。

Runtime exceptions *are not subject* to the Catch or Specify Requirement. Runtime exceptions are those indicated by RuntimeException and its subclasses.

Runtime异常不需要遵守这个原则。

Errors and runtime exceptions are collectively known as *unchecked exceptions*.

未知异常由error和runtime exception组成。

**Bypassing Catch or Specify**

Some programmers consider the Catch or Specify Requirement a serious flaw in the exception mechanism and bypass it by using unchecked exceptions in place of checked exceptions. In general, this is not recommended. The section [Unchecked Exceptions — The Controversy](https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) talks about when it is appropriate to use unchecked exceptions.

一些程序员认为这个原则有一些很严重的瑕疵在异常机制所以他们绕过它用未知异常来代替已知异常。总的来说，这是不好的。（我知道这是在说Kotlin，:p）详情见 [Unchecked Exceptions — The Controversy](https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html)。

**Catching and Handling Exceptions**

This section describes how to use the three exception handler components — the try, catch, and finally blocks — to write an exception handler. Then, the try-with-resources statement, introduced in Java SE 7, is explained. The try-with-resources statement is particularly suited to situations that use Closeable resources, such as streams.

这一节描述如何使用try catch finally.然后再是有 try-with-resources（尤其适用于closeable）

The try-with-resources statement is a try statement that declares one or more resources. A ***resource*** is an object that must be **closed** after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

**The try-with-resources Statement**

The try-with-resources statement is a try statement that declares one or more resources. A *resource* is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statemen**t. Any object that implements java.lang.AutoCloseable**, which includes all objects which implement java.io.**Closeable**, can be used as a resource.

The following example reads the first line from a file. It uses an instance of BufferedReader to read data from the file. BufferedReader is a resource that must be closed after the program is finished with it:

static String readFirstLineFromFile(String path) throws IOException {

**try (BufferedReader br =**

**new BufferedReader(new FileReader(path)))** {

return br.readLine();

}

}

In this example, the resource declared in the try-with-resources statement is a BufferedReader. The declaration statement appears within parentheses immediately after the try keyword. The class BufferedReader, in Java SE 7 and later, implements the interface java.lang.AutoCloseable. Because the BufferedReader instance is declared in a try-with-resource statement, it will be closed regardless of whether the try statement completes normally or abruptly (as a result of the method BufferedReader.readLine throwing an IOException).

Prior to Java SE 7, you can use a finally block to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly. The following example uses a finally block instead of a try-with-resources statement:

static String readFirstLineFromFileWithFinallyBlock(String path)

throws IOException {

BufferedReader br = new BufferedReader(new FileReader(path));

try {

return br.readLine();

} finally {

if (br != null) br.close();

}

}

However, in this example, if the methods readLine and close both throw exceptions, then the method readFirstLineFromFileWithFinallyBlock throws the exception thrown from the finally block; the exception thrown from the try block is suppressed. In contrast, in the example readFirstLineFromFile, if exceptions are thrown from both the try block and the try-with-resources statement, then the method readFirstLineFromFile throws the exception thrown from the try block; **the exception thrown from the try-with-resources block is suppressed**. In Java SE 7 and later, you can retrieve suppressed exceptions; see the section [Suppressed Exceptions](https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html#suppressed-exceptions) for more information.

**Specifying the Exceptions Thrown by a Method**

The previous section showed how to write an exception handler for the writeList method in the ListOfNumbers class. Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method **further up the call stack handle the exception**. For example, if you were providing the ListOfNumbers class as part of a package of classes, you probably couldn't anticipate the needs of all the users of your package. In this case, it's better to *not* catch the exception and to allow a method further up the call stack to handle it.

If the writeList method doesn't catch the checked exceptions that can occur within it, the writeList method must specify that it can throw these exceptions. Let's modify the original writeList method to specify the exceptions it can throw instead of catching them. To remind you, here's the original version of the writeList method that won't compile.

public void writeList() {

PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));

for (int i = 0; i < SIZE; i++) {

out.println("Value at: " + i + " = " + list.get(i));

}

out.close();

}

Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the throw statement.

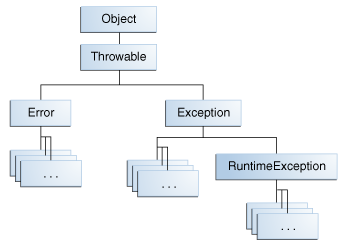
**The throw Statement**

All methods use the throw statement to throw an exception. The throw statement requires a single argument: **a throwable object**. Throwable objects are instances of any subclass of the Throwable class. Here's an example of a throw statement.

throw *someThrowableObject*;

**Throwable Class and Its Subclasses**

The objects that inherit from the Throwable class include direct descendants (objects that inherit directly from the Throwable class) and indirect descendants (objects that inherit from children or grandchildren of the Throwable class). The figure below illustrates the class hierarchy of the Throwable class and its most significant subclasses. As you can see, Throwable has two direct descendants: [Error](https://docs.oracle.com/javase/8/docs/api/java/lang/Error.html) and [Exception](https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html).



The Throwable class.

**Error Class**

When a dynamic linking failure or other hard failure in the Java virtual machine occurs, the virtual machine throws an Error. Simple programs typically do *not* catch or throw Errors.

**Exception Class**

Most programs throw and catch objects that derive from the Exception class. An Exception indicates that a problem occurred, but it is not a serious system problem. Most programs you write will throw and catch Exceptions as opposed to Errors.

The Java platform defines the many descendants of the Exception class. These descendants indicate various types of exceptions that can occur. For example, IllegalAccessException signals that a particular method could not be found, and NegativeArraySizeException indicates that a program attempted to create an array with a negative size.

One Exception subclass, RuntimeException, is reserved for exceptions that indicate incorrect use of an API. An example of a runtime exception is NullPointerException, which occurs when a method tries to access a member of an object through a null reference. The section [Unchecked Exceptions — The Controversy](https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) discusses why most applications shouldn't throw runtime exceptions or subclass RuntimeException.

**Unchecked Exceptions — The Controversy**

Because the Java programming language **does not require** methods to catch or to specify unchecked exceptions (RuntimeException, Error, and their subclasses), programmers may be tempted to write code that throws only unchecked exceptions or to make all their exception subclasses inherit from RuntimeException. Both of these shortcuts(捷径) allow programmers to write code without bothering with compiler errors and without bothering to specify or to catch any exceptions. Although this may seem convenient to the programmer, it sidesteps（回避避免） the intent（aim） of the catch or specify requirement and can cause problems for others using your classes.

Why did the designers decide to force a method to specify all uncaught checked exceptions that can be thrown within its scope? Any Exception that can be thrown by a method is part of the method's public programming interface. Those who call a method must know about the exceptions that a method can throw so that they can decide what to do about them. These exceptions are as much a part of that method's programming interface as its parameters and return value.

The next question might be: "If it's so good to document a method's API, including the exceptions it can throw, why not specify runtime exceptions too?" Runtime exceptions represent problems that are the result of a programming problem, and as such, the API client code cannot reasonably be expected to recover from them or to handle them in any way. Such problems include arithmetic exceptions, such as dividing by zero; pointer exceptions, such as trying to access an object through a null reference; and indexing exceptions, such as attempting to access an array element through an index that is too large or too small.

**Runtime exceptions can occur anywhere in a program**, and in a typical one they can be very numerous. Having to add runtime exceptions in every method declaration would reduce a program's clarity. Thus, the compiler does not require that you catch or specify runtime exceptions (although you can).

One case where it is common practice to throw a RuntimeException is when the user calls a method incorrectly. For example, a method can check if one of its arguments is incorrectly null. If an argument is null, the method might throw a NullPointerException, which is an *unchecked* exception.

**Generally speaking, do not throw a RuntimeException or create a subclass of RuntimeException simply** because you don't want to be bothered with specifying the exceptions your methods can throw.

Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

**Advantages of Exceptions**

* **Advantage 1: Separating Error-Handling Code from "Regular" Code**

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code.

* **Advantage 2:** **Propagating(spread) Errors Up the Call Stack**

A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. Suppose that the readFile method is the fourth method in a series of nested method calls made by the main program: method1 calls method2, which calls method3, which finally calls readFile.

* **Advantage 3: Grouping and** **Differentiating Error Types**

Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy.