Java 与异常

什么是异常？

异常发生的原因是 Java 执行过程中发生干扰程序正常执行流的事件，程序通过生成并且抛出 Exception 对象告知系统出现意外事件。对象中的方法生成并且抛出异常，而系统则负责捕捉并且将其交与特定的 Exception handler 处理。如果没有合理的 Exception handler 来处理该异常，则程序终止，如下带有除零操作的类：

ZeroDiv.java：

```
public class ZeroDiv{
    public void div(int a, int b){
        int c = a/b;
        System.out.println("the result = "+c);
    }
    public static void main(String[] args){
        ZeroDiv zd = new ZeroDiv();
        zd.div(10, 0);
    }
}
```

ZeroDiv.java 编译正常通过，但执行时由于异常没有得到处理而终止：

```
C:\Users\tfzhang\Desktop\Exception>java ZeroDiv
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ZeroDiv.div(ZeroDiv.java:5)
        at ZeroDiv.main(ZeroDiv.java:11)
```

Java 异常机制正常工作的两个方面：

1. 将可能发生异常的操作放置在 try{}语句中，并且提供对应的 Exception handler（异常处理方法）；
2. 将可能产生异常的方法用关键词 throw 声明，明确抛出何种类型的异常；

Exception Handler 由三个代码块构成：try, catch 和 finally。try 块之后一定要跟上 catch 块。try 块限定异常捕捉的范围，catch 则是捕捉并且处理异常。

```
try {

} catch (ExceptionType1 name) {

} catch (ExceptionType2 name) {

}
```

try 之后可以跟多个的 catch 块，并且一般而言，ExceptionType2 包含 ExceptionType1，也即 ExceptionType1 定位更具体更精确。上述两个 catch 块也可以使用或符号合并：

```
try {
} catch (ExceptionType1 name | ExceptionType2 name) {
}
```

finally 块跟在 catch 块之后，而且 try 块执行时发生异常不能完全执行时，finally 块一定会被

执行。所以，回收资源的事务都放在 finally 块中被执行。
finally 不被执行的例外情况有两种：
1. java 虚拟机在执行 try 或者 catch 块时退出，则 finally 块不定被执行；
2. 当线程在执行 try 或者 catch 块时被挂起或者杀死，则 finally 块不一定被执行；

我们将上述的 try-catch-finally 语句应用于 ZeroDiv 案例：

```
public class ZeroDiv{
	public void div(int a, int b){
		int c = 0;
		try{
			c = a/b;
		}catch(ArithmeticException e){
			System.out.println("can not divide zero");
		}catch(Exception e){
			System.out.println("some other errors than dividing zero");
		}finally{
			System.out.println("please try again");
		}
		System.out.println("the result = "+c);
	}

	public static void main(String[] args){
		ZeroDiv zd = new ZeroDiv();
		zd.div(10, 0);
		System.out.println("---------------");
		zd.div(10, 1);
	}
}
```

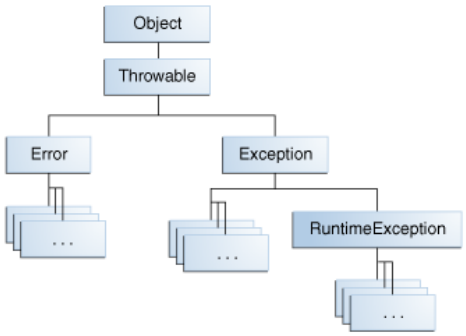
接下来介绍如何让方法申明和抛出异常？
方法声明异常的格式如下：

```
public void  函数名  throws ExceptionType1, ExceptionType2{
...
}
```

方法执行过程中，扔出异常的代码如下：

```
throw someThrowableObject;
```

任何一个被抛出的异常类都继承自 Throwable class。



如下是摘自 java.util.Stack 中的示例代码：

```
public synchronized E pop() {
        E             obj;
        int           len = size();
        obj = peek();
        removeElementAt(len - 1);
        return obj;
    }

    public synchronized E peek() {
        int           len = size();
        if (len == 0)
            throw new EmptyStackException();
        return elementAt(len - 1);
    }
```

语句"throw new EmptyStackException()"，创建一个 EmptyStackException 的异常，并且抛出；其中 EmptyStackException 是 java.util 包中的一个类。

如何自定义异常？

根据 Throwable 的类层次图，所有异常类的祖先是 Exception，所以，自定义的类也需要继承自 Exception。示例程序要求用户，从终端输入学生的成绩 0~100，如果出现超越范围的数据，抛出异常。

Score.java:

```
import java.util.Scanner;
public class Score{
    public static void main(String[] args){
        int scores[] = new int[10];
        Scanner scan = new Scanner(System.in);

        for(int i=0; i<scores.length; i++)
        {
            System.out.print("the score of "+(i+1)+"th student:");
            scores[i] = scan.nextInt();
        }
```

```
    }
}
```
当用户输入学生的分值出错，比如大于 100，程序就应该抛出异常，而异常只要说明自身即可，所以只要携带一条信息内容。

自定义的 Exception 类如下：
```
class MyException extends Exception{
    public MyException(String s){
        super(s);
    }
}
```
修改后的 Score.java 如下：
```
import java.util.Scanner;

public class Score{
    public static void main(String[] args){
        int scores[] = new int[10];
        Scanner scan = new Scanner(System.in);

        try{
            for(int i=0; i<scores.length; i++)
            {
                System.out.print("the score of "+(i+1)+"th student:");
                scores[i] = scan.nextInt();
                if(scores[i] <0 || scores[i]>100){
                    throw new MyException("score <0 or score > 100");
                }
            }
        }catch(MyException e){
            System.out.println(e);
        }
    }
}
```

为什么要使用 Exception？一、保持代码的整洁：
如下关于 readFile 的伪代码:
```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```
上述的 1~5 步，每步都容易出现错误，如果使用传统的 C 语言来定位错误，则代码如下：

```
errorCodeType readFile {
    initialize errorCode = 0;
      open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

代码看上去十分的繁复，通过例外得到的代码如下，既能定位错误的出处，也能保持代码整洁：

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
```

```
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

例外的好处二：方便地将错误码上传到目标方法。

```
method1 {
    call method2;
}
method2 {
    call method3;
}
method3 {
    call readFile;
}
```

Method1 调用 Method2，method2 调用 method3，method3 中调用方法 readFile；Method1 对 readFile 中发生的错误感兴趣。C 语言的处理方式：

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}
errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}
errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

method2 和 method3 虽然对 readFile 可能发生的错误不感兴趣，但还是要将 errorCode 往上传递，十分麻烦。而采用例外的处理方式：

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

method2 和 method3 只需要申明会抛出异常即可。

异常的好处三、以不同的粒度灵活处理错误。

就文件操作而言，只想处理文件打开错误的话，那么对应的语句如下：

```
catch (FileNotFoundException e) {
    ...
}
```

因为 FileNotFoundException 类没有被其他类继承，所以此 Exception Handler 只能处理文件没找到的异常。而如果想处理文件有关的错误，则只要捕捉的是 IOException 即可。

```
catch (IOException e) {
    ...
}
```

通过捕捉同一类错误的不同对象，达到灵活处理错误的目的。

练习 1：

考察下面两段代码，确定每段代码中的 finally 语句是否执行？

代码 1：

```
public class ExceptionTest01{
    public static void main(String[] args){
        try{
            System.out.println("in try");
            int i = 1/0;
        }catch(ArithmeticException e){
            System.exit(0);
            e.printStackTrace();
        }finally{
            System.out.println("in finally");
```

```
        }
    }
}
```

代码 2：

```
public class ExceptionTest02{
    public static void main(String[] args){
        int n = fun(2);
        System.out.println(n);
    }
    public static int fun(int i){
        try{
            int m = i/0;
            return (i+1);
        }catch(ArithmeticException e){
            return (i+2);
        }finally{
            return (i+4);
        }
    }
}
```

练习 2：

如下代码中有三个 print 语句，请确定哪几个会被打印以及打印的先后顺序？

```
public class ThrowTest{
    public static void main(String[] args){
        try{
            System.out.println("1: before throw_do");
            throw_do();
            System.out.println("2: after throw_do");
        }finally{
            System.out.println("3:finally executing");
        }
    }
    public static void throw_do(){
        throw new ArithmeticException();
    }
}
```