TryCatchDemo

```
package exception;

/*
java异常处理机制中的try-catch:
语法:
try{
    可能出现异常的代码片段
}catch(XXXException e){
    当try中代码出现XXXException后的处理代码（B计划）
}
 */
public class TryCatchDemo {
    public static void main(String[] args) {
        System.out.println("start");

        try {
            String line = null; //异常只处理一次
            System.out.println(line.length());
            System.out.println(line.charAt(0));
            System.out.println(Integer.parseInt(line));

            //try语句块中出错，出错代码下面的内容都不会被执行
            System.out.println("! ! ! ");
```

```
        /*} catch (NullPointerException e) {
            System.out.println("空指针，解决了");
            //当我们在try中针对可能出现的不同异常有不同处理方式时，我们可以定义多个catch来分别处理
        } catch (StringIndexOutOfBoundsException e) {
            System.out.println("下标越界，解决了");*/
        } catch (NullPointerException | StringIndexOutOfBoundsException e) { //注意：一个“或”符
号!
            System.out.println("出现空指针或下标越界，解决了");
            /*
            如果catch的是一个超类型异常，那么在try中出现的任何它的子类型异常都可以被它捕获。
            如果catch有多个，那么catch的异常遵循从小到大的先后捕获原则。
            即：子类型异常在上先catch，超类型异常在下后catch
             */
        } catch (Exception e) { //超类的异常
            System.out.println("反正就是出错了啦");
        }

        System.out.println("end");
    }
}
```

FinallyDemo

```
package exception;

/*
finally块：
finally块是异常处理机制的最后一块，它可以直接跟在try语句块之后或者最后一个catch之后。

finally的特点：
只要程序执行到try语句块中，无论try中语句是否出现异常，finally最后都必然执行。
通常我们会将释放资源这类操作放在finally中，确保最后一定会被执行，例如IO操作后的close()。
 */
public class FinallyDemo {
    public static void main(String[] args) {
        System.out.println("程序开始了");

        /*
        快捷键：ctrl+alt+T
        可以将选中的代码用指定的代码包围（比如for，while，try...catch等）
         */
        try {
            String line = null;
            System.out.println(line.length());
            return; //return也会执行finally中的代码
        } catch (Exception e) {
            System.out.println("出错啦");
```

```
        } finally {
            System.out.println("finally中的代码执行啦");
        }

        System.out.println("程序结束了");
    }
}
```

FinallyDemo2

```
package exception;

import java.io.FileOutputStream;
import java.io.IOException;

//异常处理机制在IO中的实际应用
public class FinallyDemo2 {
    public static void main(String[] args) {
        FileOutputStream fos = null; //局部变量使用前必须初始化
        try {
            fos = new FileOutputStream("fos.dat");
            //这里可能路径不对，new不成功，所以上面必须初始化为null
            fos.write(1);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (fos != null) {
                    fos.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

```
    }
}
```

AutoCloseableDemo

```java
package exception;

import java.io.FileOutputStream;
import java.io.IOException;

//JDK7时，java推出了一个新的特性：Autocloseable，自动关闭特性。旨在解决异常处理机制中繁琐的关闭流操作。
public class AutoCloseableDemo {
    public static void main(String[] args) {
        /*
        新语法特性要求我们将需要在finally中调用close()方法关闭的那些对象在try的()中定义并初始化即可。
        编译器在编译时会自动补充finally并调用close()方法关闭它们，还原后的代码可参考FinallyDemo2。
        try的()中只能定义并初始化那些实现了Autocloseable接口的类。所有的流都实现了该接口。
         */
        try (
                FileOutputStream fos = new FileOutputStream("fos.dat");
        ) {
            fos.write(123);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

FinallyDemo3

```java
package exception;

/*
finally常见面试题：
请分别说明：final，finally，finalize是什么？
finalize是Object中定义的一个方法，这意味着java中所有类都有该方法。
该方法会被GC（垃圾回收）调用，当GC扫描时发现一个对象不再被引用，则会将其释放，
在释放前会调用finalize，一旦该方法执行完毕，则对象被释放。
API手册明确说明：该方法可以被重写，但是不应当做耗时的操作，否则可能影响GC的工作。
 */
public class FinallyDemo3 {
    public static void main(String[] args) {
        //null不能调方法，会空指针异常
        //""空字符串没有第一个元素，会下标越界异常
        System.out.println(dosome("0") + "," + dosome(null) + "," + dosome(""));
    }

    public static int dosome(String str) {
        try {
            return str.charAt(0) - '0'; //str.charAt(0)返回str下标为0的字符
        } catch (NullPointerException e) {
            return 1;
        } catch (ArrayIndexOutOfBoundsException e) {
            return 2;
```

```
        } finally {
            return 3;
        }
    }
}
```

Person

```
package exception;

/*
使用当前类测试异常的抛出
throw关键字可以主动对外抛出异常，通常下列两种场景会这样：
1、程序出现了异常，但是该异常不应当被当前代码片段处理时，可以主动将其抛出给调用者。
2、程序可以执行，但是不满足业务需求时，可以主动对外抛出一个异常。
本案例就是第2种情况
 */
public class Person {
    private int age;

    public int getAge() {
        return age;
    }

    public void setAge(int age) throws IllegalAgeException {
        if (age < 0 || age > 100) {
            //throw new RuntimeException("年龄超出范围");
            /*
            当我们在一个方法中使用throw主动对外抛出一个异常时，
            除了RuntimeException这类异常之外，
            其它异常都应当使用throws在方法上声明该异常的抛出，
            来通知上层调用者处理该异常。
```

```
             */
            throw new IllegalAgeException("年龄超出范围");
        }
        this.age = age;
    }
}
```

ThrowDemo

```
package exception;

//异常的抛出
public class ThrowDemo {
    public static void main(String[] args) {
        Person p = new Person();

        try {
            /*
            当我们调用一个含有throws声明异常抛出的方法时，
            编译器要求我们必须处理这个异常，处理手段有两种：
            1、使用try-catch捕获并处理该异常
            2、在当前方法上继续使用throws声明这个异常的抛出通知上层调用者解决
            具体使用哪种要视异常处理的责任来确定。
             */
            p.setAge(2000); //满足语法，但是不满足业务
        } catch (IllegalAgeException e) {
            e.printStackTrace();
        }

        System.out.println("此人年龄：" + p.getAge());
    }
}
```

ThrowsDemo

```java
package exception;

import java.awt.AWTException;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.sql.SQLException;

//当子类重写超类一个含有throws声明异常抛出的方法时，对throws的重写规则
public class ThrowsDemo {
    public void dosome() throws IOException, AWTException {
    }
}

class SubClass extends ThrowsDemo {
    /*
    public void dosome() throws IOException, AWTException {
    }

    //允许仅抛出部分异常
    public void dosome() throws IOException {
    }

    //允许不再抛出任何异常
    public void dosome(){
```

```
    }

    //允许抛出超类方法抛出的异常的子类型异常
    public void dosome() throws FileNotFoundException {
    }

    //不允许抛出额外异常
    public void dosome() throws SQLException {
    }

    //不允许抛出超类方法抛出的异常的超类型异常
    public void dosome() throws Exception {
    }
     */
}
```

ExceptionAPIDemo

```java
package exception;

//异常常用方法
public class ExceptionAPIDemo {
    public static void main(String[] args) {
        try {
            String str = "abc";
            System.out.println(Integer.parseInt(str));
        } catch (NumberFormatException e) {
            e.printStackTrace(); //输出错误的堆栈信息，便于定位错误进行修复

            String message = e.getMessage(); //获取错误信息，给用户提示或记录日志
            System.out.println(message);
        }
    }
}
```

IllegalAgeException

```
package exception;

/*
illegal：非法的
自定义异常：
自定义异常通常用于那些满足语法但是不满足业务的场景。
自定义异常要做到：
1、类名见名知义
2、继承自Exception（直接或间接继承）
3、提供所有构造器（alt+insert生成即可）
 */
public class IllegalAgeException extends Exception {
    public IllegalAgeException() {
    }

    public IllegalAgeException(String message) {
        super(message);
    }

    public IllegalAgeException(String message, Throwable cause) {
        super(message, cause);
    }

    public IllegalAgeException(Throwable cause) {
```

```
        super(cause);
    }

    public IllegalAgeException(String message, Throwable cause, boolean enableSuppression,
boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}
```