



主讲教师 张 智
计算机学院软件工程系
课程群: 421694618

9 Java异常处理机制

9.1 异常机制

9.2 异常处理

9.3 异常分类

9.4 异常抛出

9.5 自定义异常类

【附录1】 try/catch/finally中的return

【附录2】 日志

【附录3】 断言

9.1 异常机制

- 异常机制是指当程序出现错误后，程序如何处理。
- 在异常引发后，应用程序应该能够转移到一个安全状态，使得系统能够恢复控制权或降级运行或正常结束程序运行，不至于使系统崩溃或死机，并且尽可能地保存数据、避免损失。

没有异常处理示例：

```
public class Test {  
    public static void main (String args[]) {  
        int i = 0;  
        String[ ] greetings = { "Hello world!", "No, I mean it!", "HELLO WORLD!!" };  
        while ( i < 4 ) {  
            System.out.println ( greetings[i] );  
            i++;  
        }  
        System.out.println ("运行完毕");  
    }  
}
```

运行结果

```
Hello world!  
No, I mean it!  
HELLO WORLD!!
```

} 正常
输出

程序出现异常导致自动终止(崩溃)
输出错误信息，后续代码无法继续运行

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

添加异常处理

```
...  
try {  
    while ( i < 4 ) {  
        System.out.println( greetings[i] );  
        i++;  
    }  
} catch ( Exception e ) {  
    System.out.println( "有异常!" );    //异常处理  
}  
//end try...catch  
System.out.println ( "运行完毕" );
```

← 可能会产生异常的代码(块)

Hello world!
No, I mean it!
HELLO WORLD!!
有异常!
运行完毕

Java异常处理允许程序捕获异常并处理，异常处理代码与正常流程代码分离，更易识别和管理

更具体的异常处理（异常分类）

```
try {  
    while (i < 4) {  
        System.out.println(greetings[i]);  
        i++;  
    }  
} catch ( ArrayIndexOutOfBoundsException e ) {  
    System.out.println( "数组下标" + i + "越界啰！ "); //异常处理  
}  
System.out.println ("运行完毕");
```

```
Hello world!  
No, I mean it!  
HELLO WORLD!!  
数组下标3越界啰!  
运行完毕
```

异常产生的主要原因

- Java 内部错误发生异常，Java 虚拟机产生的异常
- 编写的程序代码中的错误所产生的异常，例如空指针异常、数组越界异常等
- 通过 throw 语句手动生成的异常，一般用来告知该方法的调用者一些必要信息

[【返回】](#)

9.2 异常处理

■ Java异常处理涉及五个关键字：

try、catch、finally

throws、throw (这两个见9.4节)

try和catch语句

用法：

```
try {
```

```
    // 可能会产生异常的代码
```

```
}
```

```
catch ( 异常类型 异常变量名 ) {
```

```
    // 某个特定类型的异常处理代码
```

```
    // 即使catch块是空的，也算是处理情况
```

```
}
```

通用异常处理方法；
catch (**Exception e**) { }

注意：一个try块可搭配多个catch块，每一catch块可处理不同的异常

多个catch块示例

多个catch匹配原则：由上到下只匹配其中一个异常，如匹配则不会再执行其他catch块。

```
class Test {  
    public static int reciprocal(int a) {  
        return 1/a;  
    }  
    public static void main(String[] args) {  
        int i = 0;  
        String[] data = {"5", "0", "aaa"};  
        while (i < 4) {  
            try {  
                int a = Integer.parseInt(data[i]);  
                System.out.println(reciprocal(a));  
            } catch (ArrayIndexOutOfBoundsException e) {  
                System.out.println("数组下标越界:" + i);  
            } catch (NumberFormatException e) {  
                System.out.println("数据格式不正确");  
            } catch (ArithmeticException e) {  
                System.out.println("除0异常");  
            } catch (Exception e) {  
                System.out.println("有异常");  
            }  
            i++;  
        }  
        System.out.println("运行完毕");  
    }  
}
```

注意：catch(Exception e) 位置存在多个catch的时候,异常的子类必须写在前面(否则编译报错)

运行结果

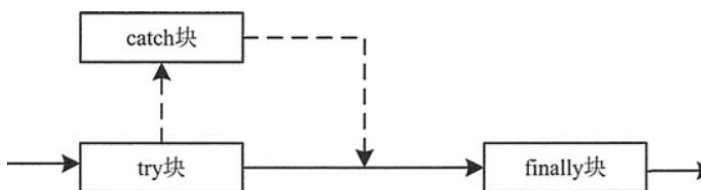
```
0  
除0异常  
数据格式不正确  
数组下标3越界  
运行完毕
```

finally语句

- finally语句定义一个总是要执行的代码块，而不考虑异常是否被捕获。
- 基本框架：

```
try {  
    ...  
}  
catch ( ... ) {  
    // 异常处理  
}  
finally {  
    // 不管是否有异常发生，这里代码都要执行  
}
```

非必须模块



```
class Test {  
    public static void main(String[] args) {  
        Scanner input = new Scanner(System.in);  
        System.out.println("Windows 系统已启动! ");  
        String[] pros = { "记事本", "计算器", "浏览器" };  
        try {  
            for (int i = 0; i < pros.length; i++) {  
                System.out.println(i + 1 + ": " + pros[i]);  
            }  
            System.out.print("是否运行程序(y/n): ");  
            String answer = input.next();  
            if (answer.equals("y")) {  
                System.out.println("请输入程序编号: ");  
                int no = input.nextInt();  
                System.out.println("正在运行程序[" + pros[no - 1] + "]");  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            System.out.println("谢谢使用!");  
        }  
    }  
}
```

运行结果

```
Windows 系统已启动!  
1: 记事本  
2: 计算器  
3: 浏览器  
是否运行程序(y/n): y  
请输入程序编号:  
3  
正在运行程序[浏览器]  
谢谢使用!
```

注意

- try、catch、finally三个语句块均不能单独使用，三者可以组成 try...catch、try...catch...finally、try...finally 三种结构，catch语句可以有一个或多个，finally语句最多一个。

说明：try、catch、finally三个代码块中变量的作用域为代码块内部，分别独立而不能相互访问。如果要在三个块中都能访问，则需要将变量定义到这些块的外面。

获取具体异常的名称

打印出所有与之相关的异常出处

关于getMessage()和printStackTrace()

■ 例如某个异常信息：

For input string: "a"

java.lang.NumberFormatException: For input string: "a"

at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)

at java.lang.Integer.parseInt(Integer.java:447)

at java.lang.Integer.parseInt(Integer.java:497)

at test.main(test.java:15)

e.getMessage()

得到异常信息(蓝色部分)

e.printStackTrace()

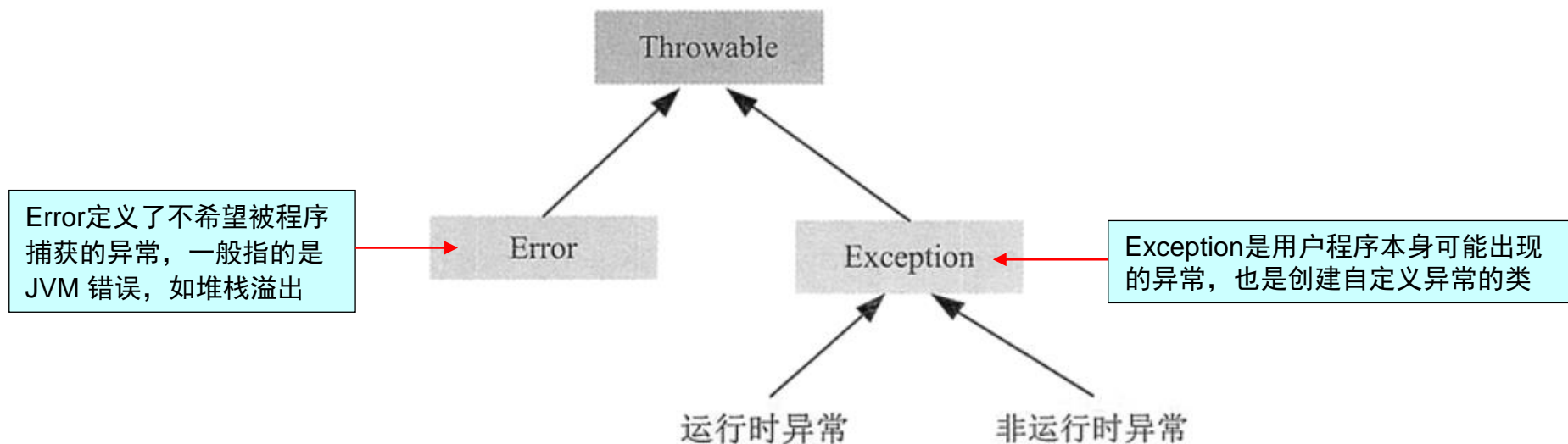
输出详细异常信息(红色部分)

```
try {  
    int k = Integer.parseInt("a");  
}  
catch( Exception e ){  
    System.out.println( e.getMessage() );  
    e.printStackTrace();  
}
```

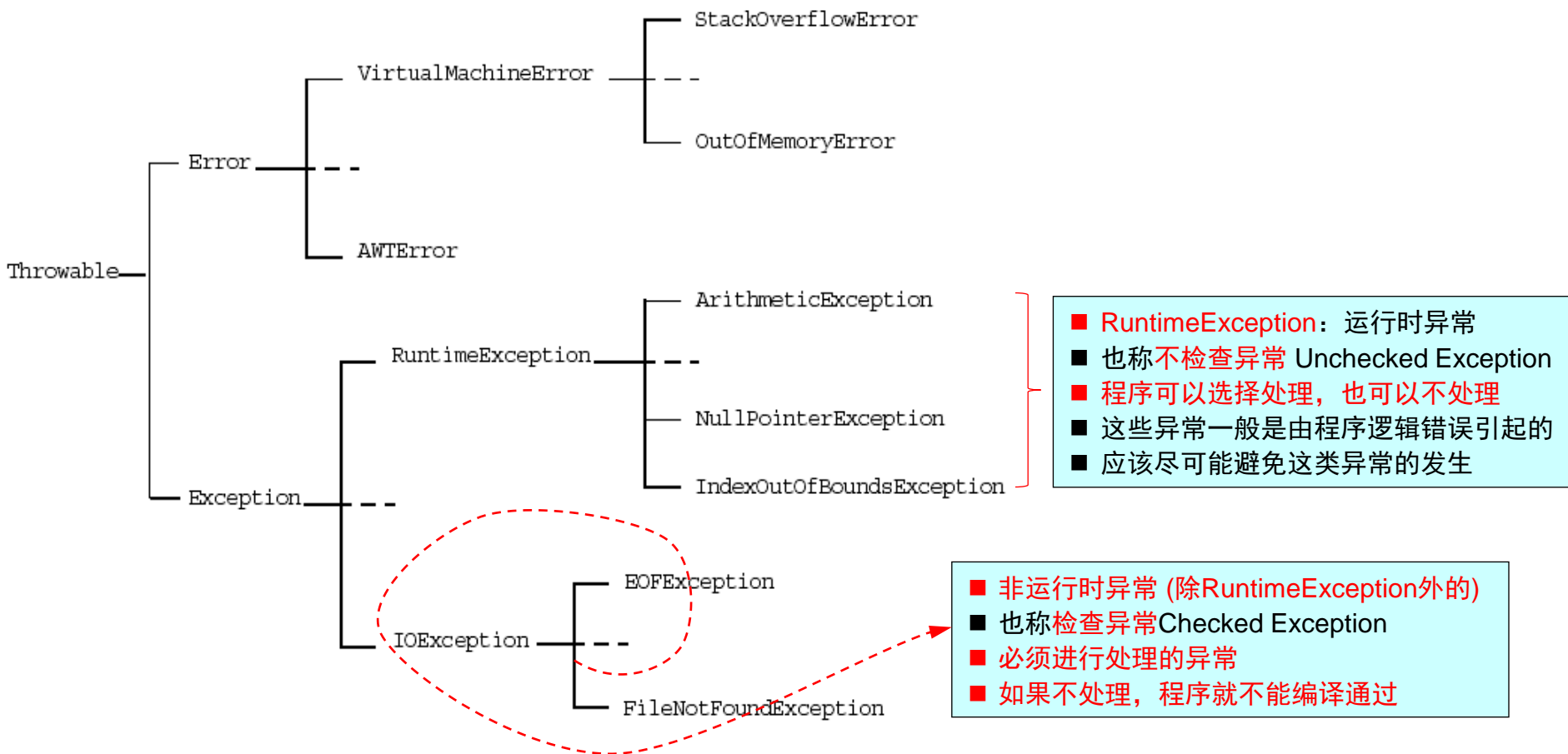
【返回】

9.3 异常分类

- Java异常分为两大类，**错误类Error**和**异常类Exception**
- `java.lang.Throwable`作为所有异常的超类



异常分类



常见的RuntimeException异常类

异常类型	说明
ArithmeticException	算术错误异常，如以零做除数
ArrayIndexOutOfBoundsException	数组索引越界
ArrayStoreException	向类型不兼容的数组元素赋值
ClassCastException	类型转换异常
IllegalArgumentException	使用非法实参调用方法
IllegalStateException	环境或应用程序处于不正确的状态
IllegalThreadStateException	被请求的操作与当前线程状态不兼容
IndexOutOfBoundsException	某种类型的索引越界
NullPointerException	尝试访问 null 对象成员，空指针异常
NegativeArraySizeException	再负数范围内创建的数组
NumberFormatException	数字转化格式异常，比如字符串到 float 型数字的转换无效
TypeNotPresentException	类型未找到

常见非运行时异常

异常类型	说明
ClassNotFoundException	类未找到异常
NoSuchFieldException	字段未找到异常
NoSuchMethodException	方法未找到异常
InstantiationException	实例化异常
IllegalAccessException	没有访问权限
InterruptedException	线程被另一个线程中断的异常
IOException	输入输出异常
SQLException	操作数据库sql异常
FileNotFoundException	文件未找到异常

示例：常见的运行时异常

■ ArithmeticException: 算术异常

■ 如: `int k = 1/0;` 整数除0

■ NullPointerException: 空指针异常

■ 如: `String s = null; boolean eq = s.equals("abc");`

■ NumberFormatException: 数字格式异常

■ 如: `int x = Integer.parseInt("12.34");`

■ ArrayIndexOutOfBoundsException: 数组索引越界异常

■ 如: `int[] a=new int[3]; int x=a[-1] 或 a[3];`

常见的运行时异常

- **StringIndexOutOfBoundsException**: 字符串索引越界异常

- 如: `String s = "hello"; char c = s.charAt(5);`

- **ClassCastException**: 类型转换异常

- 如: A不是B的父类或子类, `A a = new A(); B b=(b)a;`

- **IllegalArgumentException**: 传递非法参数异常

- 如: `Color c = new Color(0,256,255);` 颜色值只能0-255

[【返回】](#)

9.4 异常抛出

- 非RuntimeException异常，是必须进行处理的异常，如果不处理，程序就**不能编译通过**
- 对于有些异常是**当前层不能或不想解决的**，则可**抛出异常**，由上层调用者来处理（被上层某个try ... catch捕获）

数据库程序示例

```
import java.io.*;  import java.sql.*;
public class Test {
    public static void main(String[] args) {
        try{
            String url ="jdbc:mysql://localhost:3306/school"; //数据库连接字符串
            Class.forName("org.gjt.mm.mysql.Driver").newInstance(); //加载驱动程序
            Connection conn= DriverManager.getConnection(url,"root","dba"); //建立连接
            Statement stmt=conn.createStatement(); //创建SQL容器
            String sql="select * from teacher"; //表为teacher
            ResultSet rs=stmt.executeQuery(sql); //获得结果集
            while( rs.next() ) { //处理结果集
                System.out.print(rs.getString("id")+" ");
                System.out.print(rs.getString("name")+" ");
                System.out.print(rs.getString("address")+" ");
                System.out.print(rs.getString("year")+"\n");
            }
            rs.close();  stmt.close();  conn.close(); //依次关闭
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

抛出异常

```
public class Test {  
    public static void main(String[] args) throws Exception {  
        String url ="jdbc:mysql://localhost:3306/school"; //数据库连接字符串  
        Class.forName("org.gjt.mm.mysql.Driver").newInstance(); //加载驱动程序  
        Connection conn= DriverManager.getConnection(url,"root","dba"); //建立连接  
        Statement stmt=conn.createStatement(); //创建SQL容器  
        ...  
    }  
}
```

当前方法不处理异常
向上层抛出(谁调用谁处理)

代码结构看上去更完整

throws关键字 → throws抛出异常，谁调用谁处理

■ throws关键字：

- 出现在方法的声明中，表示该方法可能会抛出的异常
- 允许throws后面跟着多个异常类型(用逗号分隔)

- 这些异常类可以是方法中调用了可能抛出异常的方法而产生的异常，也可以是方法体中生成并抛出的异常
- 如果 main 方法也不知道如何处理这种类型的异常，也可使用 throws 抛出异常，该异常将交给 JVM 处理

注意：方法重写时声明抛出异常的限制

- 方法重写规则：重写方法一定不能抛出新的非运行时异常或者比被重写方法声明更加宽泛的非运行时异常
- 解释：子类方法声明抛出的异常类型应该是父类方法声明抛出的异常类型的子类或相同，子类方法声明抛出的异常不允许比父类方法声明抛出的异常多。

示例1

```
class Father {  
    public void test() throws IOException {  
    }  
}
```

```
class Son extends Father {  
    @Override  
    public void test() throws Exception {  
    }  
}
```

编译报错：子类方法声明抛出了比父类方法更大的异常




补充2

```
public class A {  
    public void foo() throws EOFException {  
    }  
}
```

```
class B extends A {  
    @Override  
    public void foo() throws FileNotFoundException {  
    }  
}
```

报错：重写方法不能抛出新的非运行时异常



throw关键字

■ throw关键字：

- throw出现在方法体中，用于明确抛出一个具体的异常对象
- throw关键字不会单独使用 (通常要配合一些条件)

■ throw用法：


- if (异常条件) throw 异常对象；
- try {...}
 catch { throw 异常对象; }

说明：当 throw 语句执行时，它后面的语句将不执行，此时程序转向调用者程序，寻找与之相匹配的 catch 语句，执行相应的异常处理程序。如果没有找到相匹配的 catch 语句，则再转向上一层的调用程序。这样逐层向上，直到最外层的异常处理程序终止程序并打印出调用栈情况。

throw示例

- 在某仓库管理系统中，要求管理员的用户名需要由 8 位以上的字母或者数字组成，不能含有其他的字符。
- 当长度在 8 位以下时抛出异常，并显示异常信息；
- 当字符含有非字母或者数字时，同样抛出异常，显示异常信息。

```
class Test {  
    public static boolean validateUserName(String username) {  
        boolean con = false;  
        if (username.length() > 8) {  
            // 判断用户名长度是否大于8位  
            for (int i = 0; i < username.length(); i++) {  
                char ch = username.charAt(i); // 获取每一位字符  
                if ((ch >= '0' && ch <= '9') || (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')) {  
                    con = true;  
                } else {  
                    con = false;  
                    throw new IllegalArgumentException("用户名只能由字母和数字组成！");  
                }  
            }  
        } else {  
            throw new IllegalArgumentException("用户名长度必须大于 8 位！");  
        }  
        return con;  
    }  
}
```



抛出一个IllegalArgumentException异常对象(自定义异常信息)

注：当异常被抛出时,程序会跳出方法

```
public static void main(String[] args) {  
    Scanner input = new Scanner(System.in);  
    System.out.print("请输入用户名: ");  
    String username = input.next();  
    try {  
        boolean con = validateUserName(username);  
        if (con) {  
            System.out.println("用户名输入正确!");  
        }  
    } catch (IllegalArgumentException e) {  
        System.out.println(e.getMessage());  
    }  
}
```

运行结果

请输入用户名: **zz**
用户名长度必须大于 8 位!

请输入用户名: **wustzz@wust**
用户名只能由字母和数字组成!

请输入用户名: **wustzz168**
用户名输入正确!

throws和throw编程示例

```
public class Test {  
    public static void main(String[] args) {  
        try {  
            System.out.println( Math.sqrt(-144) );  
        } catch( Exception e ) {  
            System.out.println("根号下不能负数");  
        }  
    }  
}
```

运行结果：
NaN

无法捕捉异常

自定义MyMath.sqrt()来改进Math.sqrt(), 并对异常进行重新封装

添加异常处理重新封装

```
class MyMath {  
    public static double sqrt(String nStr) throws Exception {  
        if (nStr == null) {  
            throw new Exception("输入的字符不能为空!");  
        }  
        double n = 0;  
        try {  
            n = Double.parseDouble(nStr);  
        }  
        catch( NumberFormatException e ) {  
            throw new Exception("输入的字符串必须能够转化成数字!");  
        }  
        if ( n < 0 ){  
            throw new Exception("输入的字符串转化成的数字必须大于0!");  
        }  
        return Math.sqrt(n);  
    }  
}
```

如果在函数体内用 throw 抛出了某种异常，最好在函数名中加 throws 抛出异常(交给调用它的上层方法进行处理)

测试

```
public static void main(String[] args) {  
    boolean f = true;  
    while( f ) { ← 输入数据直至输入正确格式为止  
        try{  
            System.out.print("请输入值: ");  
            String s=new Scanner(System.in).nextLine();  
            System.out.println( MyMath.sqrt(s) );  
            f=false;  
        } catch( Exception e ) {  
            System.out.println( e.getMessage() );  
            f=true;  
        }  
    }  
}
```

运行结果

```
请输入值: -100  
输入的字符串转化成的数字必须大于0!  
请输入值: abc  
输入的字符串必须能够转化成数字!  
请输入值: 100  
10.0
```

throws和throw几点区别

- throws用在方法的声明处，而throw用在方法内部通过
- throws用来声明方法可能抛出是哪种类型的异常，throw则是抛出的一个具体的异常实例
- throws抛出的异常是一种可能出现的异常(并不一定会发生)，throw如果执行了，则一定是抛出了某种异常
- 两种抛出都不会由方法自身去处理，真正的处理由上层调用者处理

[【返回】](#)

9.5 自定义异常类

- 用户自定义异常类是通过扩展Exception类来创建的
- 这种异常类可以包含一个“普通类”所包含的任何东西

示例

- 编写一个程序，对会员注册时的年龄进行验证，即检测是否在 0~100 岁

自定义异常类

```
class AgeRangeException extends Exception {  
    public AgeRangeException() {  
    }  
    public AgeRangeException(String msg) {  
        super(msg);  
    }  
}
```

自定义异常类一般包含两个构造方法：

- ① 无参的默认构造函数
- ② 以字符串的形式接收一个定制的异常消息并将该消息传递给超类的构造函数

```
class Test {  
    public static void main(String[] args) {  
        int age;  
        Scanner input = new Scanner(System.in);  
        System.out.print("请输入年龄: ");  
        try {  
            age = input.nextInt();    // 获取年龄  
            if (age < 0) {  
                throw new AgeRangeException("输入的年龄为负数! 输入有误!");  
            } else if (age > 100) {  
                throw new AgeRangeException("输入的年龄大于100! 输入有误!");  
            } else {  
                System.out.println("您的年龄为: " + age);  
            }  
        } catch ( AgeRangeException e ) {  
            System.out.println( e.getMessage() );  
        } catch ( InputMismatchException e ){  
            System.out.println("输入的年龄不是数字!");  
        }  
    }  
}
```

运行结果

```
请输入年龄: 101  
输入的年龄大于100! 输入有误!  
  
请输入年龄: -2  
输入的年龄为负数! 输入有误!  
  
请输入年龄: abc  
输入的年龄不是数字!  
  
请输入年龄: 19  
您的年龄为: 19
```

[【返回】](#)

【附录1】try/catch/finally中的return

- 在try块、catch块遇到return语句时，当执行完return语句之后，并不会立即结束该方法，而是去寻找该是否包含finally块
- 如果没有finally块，方法终止，返回相应的返回值。如果有finally块，则开始执行finally块；
- 只有当finally块执行完成后，系统才会再次跳回来根据return语句结束方法；
- 如果在finally块里使用了return语句来导致结束方法，则finally块已经结束了方法，系统不会跳回去执行try、catch块里的任何代码。

示例1：

```
public class Test {  
    public static boolean foo() {  
        try {  
            int i = 10 / 0;  
            System.out.println("i = " + i);  
            return true;  
        } catch (Exception e) {  
            System.out.println(" -- catch --");  
            return false;  
        } finally {  
            System.out.println(" -- finally --");  
        }  
    }  
    public static void main(String[] args) {  
        System.out.println( foo() );  
    }  
}
```

catch中有return

运行结果

```
-- catch --  
-- finally --  
false
```


示例2:

```
public class Test {  
    public static boolean foo () {  
        boolean b = true;  
        try {  
            int i = 10 / 0;  
            System.out.println("i = " + i);  
            return true;  
        } catch (Exception e) {  
            System.out.println("-- catch --");  
            System.out.println("b:" + b);  
            return b = false;  
        } finally {  
            System.out.println("-- finally --");  
            System.out.println("b:" + b);  
        }  
    }  
    public static void main(String[] args) {  
        System.out.println( foo() );  
    }  
}
```

catch中有return

运行结果

```
-- catch --  
b:true  
-- finally --  
b:false  
false
```

示例3:

```
public class Test {  
    public static boolean foo() {  
        try {  
            int i = 10 / 0;  
            System.out.println("i = " + i);  
            return true;  
        } catch (Exception e) {  
            System.out.println(" -- catch --");  
            return false;  
        } finally {  
            System.out.println(" -- finally --");  
            return true;  
        }  
    }  
    public static void main(String[] args) {  
        System.out.println( foo() );  
    }  
}
```

catch中有return

finally中也有return

运行结果

```
-- catch --  
-- finally --  
true
```

示例4:

```
public class Test {  
    public static int foo() {  
        int count = 5;  
        try {  
            return ++count;  
        } catch (Exception e) {  
            // TODO: handle exception  
        } finally {  
            System.out.println("finally()执行");  
            return count++;  
        }  
    }  
    public static void main(String[] args) {  
        System.out.println( foo() );  
    }  
}
```

try中有return

finally中也有return

运行结果

```
finally()执行  
6
```

【返回】

【附录2】日志

- 日志用来记录程序的运行轨迹，方便查找关键信息，也方便快速定位解决问题。(比使用System.out.print方法好)
- Java 自带的日志工具类：`java.util.logging`

JDK Logging日志级别

JDK Logging 把日志分为如下表 7 个级别，等级依次降低：

级别	SEVERE	WARNING	INFO	CONFIG	FINE	FINER	FINEST
调用方法	severe()	warning()	info()	config()	fine()	finer()	finest()
含义	严重	警告	信息	配置	良好	较好	最好

Logger的默认级别是 INFO
在默认情况下，日志只显示info、warning、severe三个级别

日志用法示例：

```
public class Test {  
    private static Logger log = Logger.getLogger( Test.class.toString() );  
  
    public static void main(String[] args) {  
        log.info("这是info级信息");  
        log.warning("这是warning级信息");  
        log.severe("这是server级信息");  
    }  
}
```

getLogger方法用于查找或创建记录器

getLogger参数：日志器名称串，一般使用类名

使用 info()等方法创建日志信息

运行结果

```
9月 10, 2021 4:52:04 下午 edu.wust.examples.Test main  
信息： 这是info级信息  
9月 10, 2021 4:52:04 下午 edu.wust.examples.Test main  
警告： 这是warning级信息  
9月 10, 2021 4:52:04 下午 edu.wust.examples.Test main  
严重： 这是server级信息
```

[【返回】](#)

【附录3】断言

- 断言是一种调试方式，断言失败会抛出AssertionError，只能在开发和测试阶段启用断言
- 基本用法：

assert boolean表达式：错误信息

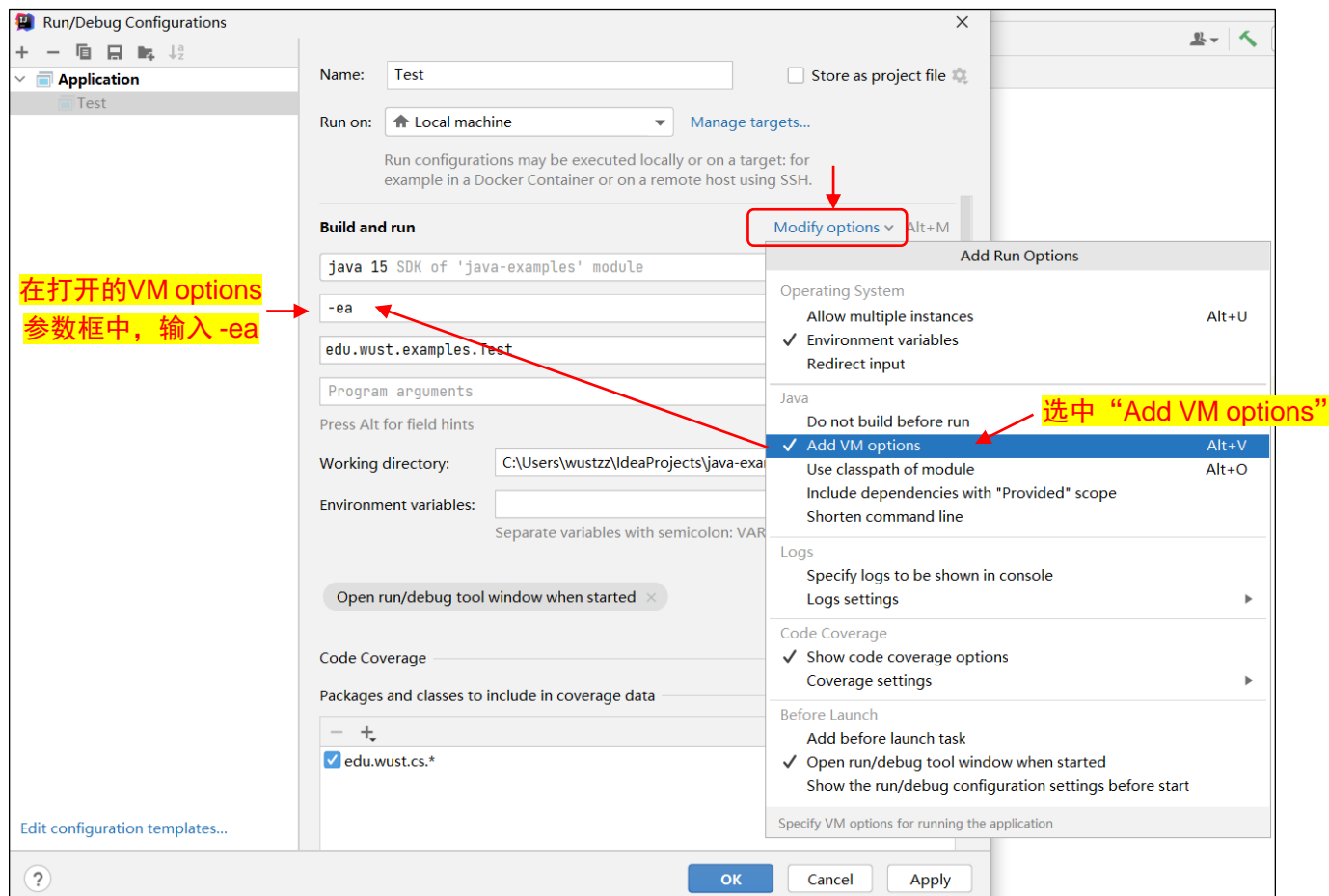
功能：

当表达式为真时，程序继续执行；

当表达式为假时，程序中断，显示AssertionError异常和错误信息

如何开启断言

- assert默认是关闭的
- 开启方法：在运行配置中添加VM参数：-ea



断言示例

```
public class Test {  
    public static void main(String[] args) {  
        double x = 100;  
        assert x > 200 : "x must > 200";  
        System.out.println(x);  
    }  
}
```

运行结果(开启断言时)

```
Exception in thread "main" java.lang.AssertionError Create breakpoint : x must>200  
    at edu.wust.examples.Test.main(Test.java:6)
```

【完】