异常和常用类和枚举

一 内容回顾（列举前一天重点难点内容）

1.1 教学重点:

```
1.掌握接口的基本概念
2.掌握接口的基本使用
3.掌握接口中的多态
4.掌握内部类的分类
5.掌握基本内部类的使用
```

1.2 教学难点:

```
1.接口新特性
这里是接口的知识扩展,可以作为了解内容,有额外精力的小伙伴可以学习一下
2.匿名内部类
内部类是java中很重要的一个知识点,匿名内部类作为内部类的一个特殊形式,也有大量的应用,不过刚开始理解起来会有一些困难,我们只需要学习一些基础知识即可,有额外精力的小伙伴可以学习一下
```

二 教学目标

```
1.掌握包装类与字符串和基本数据类型的转换
2.掌握常用类的基本使用
3.掌握枚举的基本使用
4.掌握异常的分类
5.掌握异常的常用结构
6.熟练使用自定义异常
```

三 教学导读

3.1. 包装类

> 包装类， 就是在基本数据类型的基础上， 做一层包装。 每一个包装类的内部都维护了一个对应的基本数据类型的属性， 用来存储管理一个基本数据类型的数据。
>
> 包装类是一种引用数据类型， 使用包装类， 可以使得基本数据类型数据有着引用类型的特性。 例如， 可以存储在集合中。 同时， 包装类还添加了若干个特殊的方法。

3.2. 常用类

> 在开发过程中,很多功能是大家很常用的,系统为了方法大家使用,提高开发效率,将很多功能提前封装成了常用类.

3.3. 枚举

枚举也是一种自定义的数据类型， 是一个引用数据类型。 枚举经常用来被描述一些取值范围有限的数据。

例如:

- 性别: 只有两个值， 此时可以用枚举来表示
- 月份: 只有12个值， 此时可以用枚举来表示
- 星期: 只有七个值， 此时可以用枚举来表示

3.4. 异常

异常，是对程序在运行过程中的种种不正常的情况的描述。

如果程序遇到了未经处理的异常，会导致这个程序无法进行编译或者运行。

例如：

- ArrayIndexOutOfBoundsException: 数组下标越界异常， 会导致程序无法继续运行。
- NullPointerException: 空指针异常， 会导致程序无法继续执行。
- ParseException: 解析日期异常， 会导致程序无法继续编译。

四 教学内容

4.1. 包装类(会)

4.1.1. 基本数据类型与包装类型

定义:专门将简单数据类型的数据进行封装,形成的对应的类.

基本数据类型	包装类型
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

4.1.2. 装箱拆箱

4.1.2.1. 手动装箱

概念：由基本数据类型， 完成向对应的包装类型进行转换。

作用: 为了更好的存储

方式1： 可以通过每一个包装类的构造方法来完成。 在每一个包装类的构造方法中，都有一个与之对应的基本数据类型作为参数的重载方法。 此时， 直接将需要包装起来的基本数据类型， 写到构造方法的参数中即可完成装箱。

```
Byte n1 = new Byte((byte)1);
Short n2 = new Short((short)2);
Integer n3 = new Integer(3);
Long n4 = new Long(4L);
Float n5 = new Float(3.14f);
Double n6 = new Double(3.14);
Character n7 = new Character('a');
Boolean n8 = new Boolean(false);
```

【推荐使用】方式2： 可以通过包装类的静态方法 `valueOf()` 完成装箱。每一个包装类中， 都有一个静态方法 `valueOf`， 这个方法的参数是包装类型对应的基本数据类型的参数。 直接将需要包装起来的基本数据类型的数据， 写到这个方法的参数中， 即可完成对这个基本数据类型数据的装箱。

```
Byte n1 = Byte.valueOf((byte)1);
Short n2 = Short.valueOf((short)2);
Integer n3 = Integer.valueOf(3);
Long n4 = Long.valueOf(4);
Float n5 = Float.valueOf(3.14f);
Double n6 = Double.valueOf(3.14);
Character n7 = Character.valueOf('a');
Boolean n8 = Boolean.valueOf(true);
```

4.1.2.2. 手动拆箱

概念: 由包装类型， 完成向对应的基本数据类型进行转换。

作用: 为了方便进行运算

方式： 使用每一个包装类对象的 `xxxValue` 可以实现。 这里的 `xxx` 就是需要转型的基本数据类型。 例如， 如果需要转型为int类型， 则直接调用 `intValue` 即可。

```java
Byte i1 = Byte.valueOf((byte) 100);
byte n1 = i1.byteValue();

Short i2 = Short.valueOf((short) 100);
short n2 = i2.shortValue();

Integer i3 = Integer.valueOf(100);
int n3 = i3.intValue();

Long i4 = Long.valueOf(100);
long n4 = i4.longValue();

Float i5 = Float.valueOf(3.14f);
float n5 = i5.floatValue();

Double i6 = Double.valueOf(3.14);
double n6 = i6.doubleValue();

Character i7 = Character.valueOf('a');
char n7 = i7.charValue();

Boolean i8 = Boolean.valueOf(true);
boolean n8 = i8.booleanValue();
```

备注： 某些包装类对象， 除了可以拆箱成对应的基本数据类型的数据之外。 还可以将包装起来的数字转成其他的基本数据类型的数据。 例如，Integer，除了有 `intValue` 之外， 还有 `byteValue` 等方法。 其实， 就是将包装类中包装起来的int数据， 强转成byte类型返回结果。 在使用的时候， 找自己需要的方法去转型即可。

4.1.2.3. 自动装箱拆箱

概念： 所谓的自动装箱和自动拆箱， 指的是在进行装箱和拆箱的时候， 不用再使用上面的方法完成装箱和拆箱的操作。 在JDK1.5之后， 装箱和拆箱是可以自动完成的！ 只需要一个赋值语句即可!

方式: 没有什么特殊语法， 直接去进行赋值即可。

```
// 自动装箱：由一个基本数据类型，到对应的包装类型的转换。只需要一个赋值语句即可完成。
Integer i1 = 10;
// 自动拆箱：由一个包装类型，到对应的基本数据类型的转换。只需要一个赋值语句即可完成。
int a = i1;
```

注意： 既然已经有了自动的装箱和拆箱， 为什么还要掌握手动的装箱和拆箱。 因为， 在有些情况下， 自动的装箱和拆箱是不能使用的。

示例: 如果在一个类的重载方法中， 有两个方法的参数类型， 一个是基本数据类型， 一个是对应的包装类型。 此时， 将无法使用自动装箱和拆箱。 必须通过手动的装箱和拆箱完成对应的方法的调用。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/9
 * @Description
 *       自动的装箱和拆箱不能完成的逻辑：
 */
public class Program2 {
    public static void main(String[] args) {
        // 此时，10会最优先匹配到int类型的参数
        show(10);
        show(Integer.valueOf(10));
```

```
    }
    public static void show(int a) {
        System.out.println(a);
    }
    public static void show(Integer a) {
        System.out.println(a);
    }
}
```

4.1.2.4. 享元原则

概念： 是程序设计的一个基本原则。 当我们需要在程序中频繁的用到一些元数据的时候， 此时， 我们可以提前将这些元数据准备好， 当需要的时候， 直接拿过来使用即可。 使用完成之后， 也不进行销毁， 以便下次继续使用。

包装类中的享元： 将常用到的基本数据类型对应的包装类对象，预先存储起来。 当使用到这些基本数据类型对应的包装类对象的时候， 可以直接拿过来使用， 不用再实例化新的对象了。

示例: Integer类中， 将 [-128, 127] 范围内的数字对应的包装类对象预存到了一个 Integer.cache 数组中， 每当我们用到这个范围内的数字的时候，可以直接从这个数组中获取到元素。 如果用到了不在这个范围内的数字，再去进行新的包装类对象的实例化。 这样， 不用频繁的开辟空间、销毁空间， 节省了CPU资源。

```java
Integer i1 = Integer.valueOf(10);
Integer i2 = Integer.valueOf(10);
System.out.println(i1 == i2);   // 此时， 由于10在缓存范围内， 因此可以直接从数组中获取包装类对象。 true。

Integer i3 = Integer.valueOf(200);
Integer i4 = Integer.valueOf(200);
System.out.println(i3 == i4);   // 此时， 由于200不在缓存范围内， 因此这个方法会返回一个新的包装类对象。 false。
```

4.1.3. 字符串与基本数据类型转换

4.1.3.1. 基本数据类型转型字符串类型

概念： 基本数据类型， 转成字符串， 希望得到的结果是这个数值转成字符串的样式。 其实， 就是直接给这个数值添加上双引号。

方式1: 可以利用字符串拼接运算符完成。 当加号两端有任意一方是字符串的时候， 此时都会自动的把另一方也转成字符串， 完成字符串的拼接。 所以， 当需要把一个基本数据类型的数据转成字符串的时候， 只需要在另一端拼接上一个空的字符串即可。

```java
int a = 10;
String str = a + "";
```

【推荐使用】方式2： 使用字符串的静态方法 `valueOf` 完成。

```java
String str = String.valueOf(10);
```

方式3： 借助包装类的实例方法 `toString` 方法。

```java
String str = Integer.valueOf(10).toString();
```

方式4： 借助包装类的静态方法 `toString` 方法。

```
String str = Integer.toString(10);
```

4.1.3.2. 字符串类型转型基本数据类型

概念： 字符串类型转基本数据类型， 其实就是解析出这个字符串中的内容，转型成对应的基本数据类型的表示。

注意事项1： 基本数据类型转字符串肯定是没问题的， 但是由字符串类型转到基本数据类型的时候， 可能会出现问题。字符串中的内容， 不一定能够转成希望转换的基本数据类型。 如果转换失败， 会出现 `NumberFormatException` 异常。

注意事项2: 对于整数来说，字符串中如果出现了其他的非数字的字符， 都会导致转整数失败， 即便是小数点， 也不可以转。 这里并没有转成浮点数字， 再转整数的过程。

方式1: 使用包装类的静态方法 `valueOf` 方法

```
Integer num = Integer.valueOf("123");
```

方式2： 使用包装类的静态方法 `parseXXX` 方法。 这里的XXX就是要转换的基本数据类型。

```
int number = Integer.parseInt("123");
```

备注： 以上两种方式，都可以完成字符串类型到基本数据类型之间的转换。 如果希望直接转成基本数据类型， 推荐使用方式2； 如果希望转成包装类型， 推荐使用方式1。

关于字符类型的特殊说明：

字符串类型， 没有类似于上面的方式， 可以直接转成字符类型。 如果一个字符串， 要转成字符， 需要使用字符串的一个实例方法 `charAt()` 方法。 使用这个方法， 获取字符串中的指定下标位的字符。

4.1.3.3. 实现进制间的转换

- 把十进制转成其它进制
  - Integer.toHexString() 转十六进制
  - Integer.toOctalString() 转八进制
  - Integer.toBinaryString() 转二进制

```
String hex = Integer.toHexString(1234);
System.out.println(hex);
```

- 把其它进制转十进制

```
//第一个参数:数据-以字符串形式存储
//第二个参数:进制-转之前的进制
Integer.parseInt(数据,进制)
int i2 = Integer.parseInt("10100011",2);
System.out.println(i2);
```

课上练习

```
//创建Integer类对象，并以int型返回
Integer intAb = new Integer("123");
System.out.println(intAb.intValue());

//创建两个Character对象，相应转换后判断是否相等
Character charA = new Character('a');
Character charB = new Character('A');
```

```
System.out.println("charA = "+charA + "  "+"charB = "+charB);
System.out.println(charA.equals(charB));
charA = Character.toLowerCase(charA);
charB = Character.toLowerCase(charB);
System.out.println(charA.equals(charB));

//建立两个Boolean型变量，注意输出
Boolean boolA = new Boolean("true");
Boolean boolB = new Boolean("asd");
Boolean boolC = new Boolean("True");
System.out.println(boolA);
System.out.println(boolB);
System.out.println(boolC);
```

4.2. 常用类(会)

4.2.1. 常用类Math

4.2.1.1. 概念

是一个数学类， 这个类中封装了很多用来做数学计算的方法。 当我们需要使用到数学计算的时候， 要能够想到这个类。 这个类中有很多封装好的数学公式， 而且， 都是静态方法， 方便调用。

4.2.1.2. 常用静态常量

属性	描述	值
PI	圆周率	3.14159265358979323846
E	自然对数	2.71828182845904523354

4.2.1.3. 常用方法

方法	参数	描述
abs	int/long/float/double	计算一个数字的绝对值
max	(int, int)/(long, long)/(float, float)/(double, double)	计算两个数字的最大值
min	(int, int)/(long, long)/(float, float)/(double, double)	计算两个数字的最小值
round	float/double	计算一个数字的四舍五入
floor	float/double	计算一个数字的向下取整
ceil	float/double	计算一个数字的向上取整
pow	(double, double)	计算一个数字的指定次幂
sqrt	double	计算一个数字的平方根
random	-	获取一个 [0,1) 范围内的浮点型随机数

4.2.1.4. 示例代码

```java
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description Math类
 */
public class MathUsage {
    public static void main(String[] args) {
        System.out.println(Math.abs(-3));   // 计算一个数字的绝对值
        System.out.println(Math.max(10, 20)); // 计算两个数字的最大值
        System.out.println(Math.min(10, 20)); // 计算两个数字的最小值

        System.out.println(Math.round(3.14));   // 四舍五入
        System.out.println(Math.floor(3.14));   // 向下取整，找到比这个数字小的第一个整数
        System.out.println(Math.ceil(3.14));    // 向上取整，找到比这个数字大的第一个整数

        System.out.println(Math.pow(2, 3));     // 计算2的3次方
        System.out.println(Math.sqrt(4));       // 计算4开平方
        // 需求：计算27的立方根
        System.out.println(Math.pow(27, 1/3.0));

        System.out.println(Math.random());
// [0, 1)
```

```
22          System.out.println((int)(Math.random() * 100));
    // [0, 100) 整型随机数
23      }
24  }
```

4.2.2. 常用类Random

4.2.2.1. 概念

是一个专门负责产生随机数的类。 在Java中， Random类在 java.util 包中。 在使用之前， 需要先导包。

> 其实， 随机数的产生， 是有一个固定的随机数算法的。 代入一个随机数种子， 能够生成一个随机数列。 但是由于算法是固定的， 因此会有一个“BUG”： 如果随机数的种子相同， 则生成的随机数列也完全相同。

4.2.2.2. 常用方法

方法	参数	描述
Random	**无**	**通过将系统时间作为随机数种子， 实例化一个Random对象**
Random	int	通过一个指定的随机数种子，实例化一个Random对象
nextInt	**int**	**生成一个 [0, bounds) 范围内的整型随机数**
nextInt	无	生成一个int范围内的随机数
nextFloat	无	生成一个 [0, 1) 范围内的float类型的随机数
nextDouble	无	生成一个 [0, 1) 范围的double类型的随机数
nextBoolean	无	随机生成一个boolean数值

4.2.2.3. 示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description Math类
 */
public class RandomUsage {
    public static void main(String[] args) {
        // 1. 实例化一个Random对象
        Random random = new Random(1);
        // 2. 产生随机数
        for (int i = 0; i < 20; i++) {
            // 产生 [0, 50) 范围内的随机数
```

```
13                 System.out.print(random.nextInt(50) + ",
   ");
14             }
15         }
16     }
```

课上练习

求[0,10)之间的整数

```
1  Random random = new  Random();
2  System.out.println(Math.abs(random.nextInt()%10));
3  System.out.println(random.nextInt(10));//获取的是[0,10)之
   间的整数
```

4.2.3. 常用类BigInteger、BigDecimal

4.2.3.1. 概念

这两个类，都是用来表示数字的类。 BigInteger表示整型数字，BigDecimal表示浮点型数字。 这两个类， 可以用来描述非常、非常、非常大的数字。 例如整数， long是最大的表示范围， 但是即便是long型， 也有它表示不了的情况。 BigInteger就是可以表示任意大小的数字。

BigInteger: 表示整型数字， 不限范围。

BigDecimal: 表示浮点型数字，不限范围， 不限小数点后面的位数。

4.2.3.2. 常用方法

方法	参数	描述
构造方法	String	通过一个数字字符串，实例化一个对象
add	BigInteger/BigDecimal	加
subtract	BigInteger/BigDecimal	减
multipy	BigInteger/BigDecimal	乘
divide	BigInteger/BigDecimal	除
divideAndRemainder	BigInteger/BigDecimal	除, 保留商和余数 将商存到结果数组的第0位 将余数存到结果数组的第1位
xxxValue	-	转成指定的基本数据类型的结果（可能会溢出）

4.2.3.3. 示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description BigInteger、BigDecimal
 */
public class BigIntegerAndBigDecimal {
    public static void main(String[] args) {
        // 1. BigInteger类
        BigInteger n1 = new
BigInteger("12347328461827364812736481726348712643872634871263293413648273684716238746");
```

```
        BigInteger n2 = new BigInteger("382423748712384719873498723176238647162375912638756287643812398471987387 63");

        // 2. 四则运算
        BigInteger add = n1.add(n2); // 加法
        System.out.println(add);

        BigInteger subtract = n1.subtract(n2); // 减法
        System.out.println(subtract);

        BigInteger multiply = n1.multiply(n2); // 乘法
        System.out.println(multiply);

        BigInteger divide = n1.divide(n2); // 除法
        System.out.println(divide);

        // 用n1除n2, 保留商和余数
        // 将商存到结果数组的第0位
        // 将余数存到结果数组的第1位
        BigInteger[] bigIntegers = n1.divideAndRemainder(n2);
        System.out.println(bigIntegers[0]);     // 输出商
        System.out.println(bigIntegers[1]);     // 输出余数

        long ret = bigIntegers[0].longValue();
    }
```

```
34  }
```

4.2.4. 常用类Date

4.2.4.1. 概念

是一个用来描述时间、日期的类。 **在 java.util 包中！！！**

4.2.4.2. 注意点

- 比较Date和Data类

```
Date:日期类
Data:数据类,装的是二进制的数据
```

- 比较java.util.date和java.sql.date包

```
java.util.date    对应的是java的日期类型,包括年月日   时分秒
java.sql.date     对应的是数据库的日期类型 ,只包括 年月日

如果需要数据类型转换,从java.sql.date转成java.util.date是自动类型转换,反之是强制类型转换
```

4.2.4.3. 常用方法

方法	参数	描述
Date	-	实例化一个Date对象，来描述系统当前时间。
Date	long	通过一个指定的时间戳，实例化一个Date对象，描述指定的时间。
getTime	-	获取一个日期对应的时间戳，从1970年1月1日0时0分0秒开始计算的毫秒数。
setTime	long	通过修改一个时间的时间戳，修改这个时间对象描述的时间。
equals	Date	判断两个时间是否相同
before	Date	判断一个时间是否在另一个时间之前
after	Date	判断一个时间是否在另一个时间之后

4.2.4.4. 示例代码

```
/*
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description Date日期类
 */
public class DateUsage {
    public static void main(String[] args) {
        // 1. 实例化一个Date对象
        Date date = new Date();

        // 2. 获取一个日期的对应的时间戳 (从 1970年 1月 1日 0时开始的毫秒数)
```

```
12          long timestamp = date.getTime();
13
14          // 3. 实例化一个Date对象
15          Date date1 = new Date(1586587414273L);
16          System.out.println(date1);
17
18          // 4. 通过设置一个时间戳，修改这个对象描述的时间
19          date1.setTime(1586587414273L);
20
21          System.out.println(date.equals(date1));      //
    判断两个时间是否相同
22          System.out.println(date.before(date1));      //
    判断一个时间是否在另一个时间之前
23          System.out.println(date.after(date1));       //
    判断一个时间是否在另一个时间之后
24      }
25  }
```

4.2.5. 常用类SimpleDateFormat

4.2.5.1. 概念

是一个用来格式化时间的类。 使用这个类， 一般有两种操作：

- 将一个 Date 对象， 转成指定格式的时间字符串。
- 将一个指定格式的时间字符串， 转成 Date 对象。

4.2.5.2. 常用时间格式

在时间格式中， 有几个常见的时间占位符。

占位符	描述
y	表示年。 常用 `yyyy` 表示长年分。 `yy` 表示短年份。
M	表示月。 常用 `MM` 表示两位占位， 如果月份不够两位， 往前补零。
d	表示日。 常用 `dd` 表示两位占位， 如果日期不够两位， 往前补零。
H	表示时， 24小时制。 常用 `HH` 表示两位占位， 如果时不够两位， 往前补零。
h	表示时， 12小时制。 常用 `hh` 表示两位占位， 如果时不够两位， 往前补零。
m	表示分。 常用 `mm` 表示两位占位， 如果分不够两位， 往前补零。
s	表示秒。 常用 `ss` 表示两位占位， 如果秒不够两位， 往前补零。
S	表示毫秒。 常用 `sss` 表示三位占位， 如果毫秒不够三位， 往前补零。

4.2.5.3. 常用方法

方法	参数	描述
SimpleDateFormat	String	通过一个指定的时间格式， 实例化一个对象。
format	Date	将一个Date对象转成指定格式的字符串。
parse	String	将一个指定格式的时间字符串，解析成一个Date对象。

> **parse** 方法
>
> 会抛出一个编译时的异常。 在使用的时候， 目前， 直接使用一键修复(alt + Enter)， 用 try-catch 包围即可。
>
> 将一个字符串， 按照指定的格式进行解析。 如果字符串中的时间格式， 和对象实例化的时候给定的格式不同， 此时会出现异常。

4.2.5.4. 示例代码

```
1  import java.text.ParseException;
2  import java.text.SimpleDateFormat;
3  import java.util.Date;
4
5  /**
6   * @Author 千锋大数据教学团队
7   * @Company 千锋好程序员大数据
8   * @Description
9   */
10 public class SimpleDateFormatUsage {
```

```
11      public static void main(String[] args) {
12          // format();
13          // parse();
14          System.out.println(getDeltaDays("2002-09-28",
    "2020-04-11"));
15      }
16
17      // 将一个时间对象，转成指定格式的字符串
18      private static void format() {
19          // 1. 获取系统当前时间
20          Date now = new Date();
21          // 2. 指定一个时间格式，例如： 2020年4月11日
    18:09:49
22          String format = "yyyy年M月d日 HH:mm:ss";
23          // 3. 通过一个时间格式，实例化一个SimpleDateFormat对
    象
24          SimpleDateFormat sdf = new
    SimpleDateFormat(format);
25          // 4. 转换成指定格式的字符串
26          String str = sdf.format(now);
27          System.out.println(str);
28      }
29
30      // 将一个指定格式的字符串，转成时间对象
31      private static void parse() {
32          // 1. 通过一个时间格式，实例化一个对象
33          SimpleDateFormat sdf = new
    SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
34          // 2. 将一个指定格式的字符串，解析成Date对象
35          try {
36              Date date = sdf.parse("2019-09-27
    22:18:05");
37              System.out.println(date);
```

```
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

课上练习

设计方法，计算两个日期之间相差多少天

```
 private static int getDeltaDays(String from, String to) {
        // 1. 限定一个时间格式
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        // 2. 将两个时间转成Date对象
        try {
            Date fromDate = sdf.parse(from);
            Date toDate = sdf.parse(to);

            // 3. 计算相差多少天
            long days = (toDate.getTime() - fromDate.getTime()) / 1000 / 60 / 60 / 24;
            return (int)Math.abs(days);
        } catch (ParseException e) {
            e.printStackTrace();
        }

        return 0;
    }
```

4.2.6. 常用类Calendar

4.2.6.1. 概念

是一个用来描述时间、日期的类。 比Date的功能更加完善。 在Date类中， 有很方法都已经被废弃了。 用Caleendar类中的某些方法来替代。

4.2.6.2. 常用方法

方法	参数	描述
getInstance	-	获取用来描述当前时间的Calendar子类对象， 并向上转型为了Calendar类型。 Calendar类是一个抽象类， 无法实例化对象。
get	int	通过指定的字段，获取字段对应的值。 字段的信息， 通过Calendar类中定义的一些静态常量来获取。 例如： Calendar.YEAR、Calendar.MONTH、Calendar.DAY_OF_MONTH...
set	int， int	通过指定的字段，设置字段对应的值。 字段的信息， 通过Calendar类中定义的一些静态常量来获取。 例如： Calendar.YEAR、Calendar.MONTH、Calendar.DAY_OF_MONTH...
set	int, int, int int, int, int, int, int int, int, int, int, int	同时设置年月日信息 同时设置年月日时分信息 同时设置年月日时分秒信息
getTime	-	获取指定时间对应的Date对象
setTime	Date	通过一个Date对象，设置时间
getTimeInMillis	-	获取当前时间戳
setTimeInMillis	long	通过时间戳设置时间
equals	Calendar	判断两个日期是否相同
before	Calendar	判断一个日期是否在另外一个日期之前
after	Calendar	判断一个日期是否在另外一个日期之后
add	int, int	对指定的字段进行加法， 增加指定的时间偏移量

4.2.6.3. 示例代码

```
import java.util.Calendar;
import java.util.Date;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description Calendar类
 */
public class CalendarUsage {
    public static void main(String[] args) {
        // 1. Calendar是一个抽象类，无法直接进行实例化
        Calendar calendar = Calendar.getInstance();

        // 2. 通过指定的字段，获取对应的值。
        //    在 Calendar 类中，已经封装好了若干个静态常量，来表示不同的字段。

 System.out.println(calendar.get(Calendar.YEAR));

 System.out.println(calendar.get(Calendar.MONTH));
     // 在Calendar中，月份是从0开始的。

 System.out.println(calendar.get(Calendar.DAY_OF_MONTH));

 System.out.println(calendar.get(Calendar.HOUR_OF_DAY));

 System.out.println(calendar.get(Calendar.MINUTE));

 System.out.println(calendar.get(Calendar.SECOND));
```

```java

        // 3. 通过指定的字段，设置对应的值
        calendar.set(Calendar.YEAR, 2022);
        calendar.set(Calendar.DAY_OF_MONTH, 29);

        // 4. 同时设置年月日
        calendar.set(2021, Calendar.SEPTEMBER, 7);
        //    同时设置年月日时分
        calendar.set(2022, Calendar.NOVEMBER, 12, 23, 59);
        //    同时设置年月日时分秒
        calendar.set(2022, Calendar.NOVEMBER, 12, 23, 59, 59);

        // 5. 获取日期(Date对象)
        Date date = calendar.getTime();
        // 6. 设置日期(Date对象)
        calendar.setTime(new Date());
        // 7. 获取时间戳
        long timestamp = calendar.getTimeInMillis();
        // 8. 设置时间戳
        calendar.setTimeInMillis(timestamp);

        // 9. 判断一个日期是否在另外一个日期之前
        //    类似的方法还有 equals、after
        calendar.before(Calendar.getInstance());

        // 10. 对一个日期进行加法操作
        calendar.add(Calendar.MONTH, 3);
        calendar.add(Calendar.DAY_OF_MONTH, 21);

        System.out.println(calendar);
    }
```

```
53  }
```

4.2.7. System

- 概念

`System` 类包含一些有用的类字段和方法。它不能被实例化。

在 `System` 类提供的设施中，有标准输入、标准输出和错误输出流；对外部定义的属性和环境变量的访问；加载文件和库的方法；还有快速复制数组的一部分的实用方法。

- 常用字段

字段	详情描述
err	“标准”错误输出流。
in	“标准”输入流。
out	“标准”输出流。

- 常用方法

常用方法	详情描述
currentTimeMillis()	返回以毫秒为单位的当前时间。
exit(int status)	终止当前正在运行的 Java 虚拟机。
gc()	运行垃圾回收器。
getProperties()	确定当前的系统属性。
nanoTime()	返回最准确的可用系统计时器的当前值，以毫微秒为单位。
setIn(InputStream in)	重新分配“标准”输入流。
setOut(PrintStream out)	重新分配“标准”输出流。

- 示例代码

```java
import java.io.InputStream;
import java.io.PrintStream;
import java.util.Properties;
import java.util.Scanner;

public class SystemDemo {
    public static void main(String[] args) {
        //获取当前系统时间--单位毫秒
        long time1 = System.currentTimeMillis();
        System.out.println("系统时间(毫秒):"+time1);
        //最准确的可用系统计时器的当前值
        long time2 = System.nanoTime();
        System.out.println("系统时间(单位毫微秒):"+time2);
```

```
14          //垃圾回收器
15          // 调用 gc 方法暗示着 Java 虚拟机做了一些努力来回收未用对象，以便能够快速地重用这些对象当前占用的内存。
16          // 当控制权从方法调用中返回时，虚拟机已经尽最大努力从所有丢弃的对象中回收了空间。
17          // 作用跟Runtime.getRuntime().gc();是一样的
18
19          //注意:我们在讲解多线程时会稍微使用一下.
20          //System.gc();
21
22          //获取当前的系统属性。
23          Properties properties = System.getProperties();
24          System.out.println(properties);
25
26          //获取标准输入流
27          InputStream in = System.in;
28          //例如
29          new Scanner(System.in);
30          //获取标准输出流
31          PrintStream out = System.out;
32          //例如
33          System.out.println();
34      }
35  }
```

4.2.8. Runtime

- 概念

  运行时类,每个 Java 应用程序都有一个 Runtime 类实例，使应用程序能够与其运行的环境相连接。可以通过 getRuntime 方法获取当前运行时。应用程序不能创建自己的 Runtime 类实例。

- 示例代码

```
public class Demo3 {
    public static void main(String[] args) {
        Runtime runtime = Runtime.getRuntime();//通过一个公共的方法获取Runtime对象

        //单位默认是字节

 System.out.println(runtime.totalMemory()/1024./1024);// 返回 Java 虚拟机中的内存总量。

 System.out.println(runtime.freeMemory()/1024./1024);//返回 Java 虚拟机中的空闲内存量。

 System.out.println(runtime.maxMemory()/1024./1024);//返回 Java 虚拟机试图使用的最大内存
    }
}
```

课上练习一

首先询问顾客是否是会员（1代表是0代表不是），再请用户输入购物金额。非会员购物统一打九折；会员如果购物200元以下打8折，如果购物上200则可以打七五折。使用嵌套if输出最后用户实际需要支付的钱数

```
import java.util.Scanner;

public class hwork1 {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("是否是会员（1代表是0代表不是）?");
    int state = sc.nextInt();
    System.out.println("请输入购物金额");
```

```
9       int money = sc.nextInt();
10
11      double discountMoney = discount(state,money);//调用
    打折的函数，返回打折后的金额
12      if(discountMoney!=-1) {
13        System.out.println("打折后为：" + discountMoney);
14      }else {
15        //System.out.println("输入不合法！");
16      }
17    }
18
19    public static double discount(int state,int money) {
20      if(state == 0) {//如果不是会员直接打9折
21        System.out.println(money*0.9);
22        return money*0.9;
23      }else if(state == 1) {//如果是会员大于等于200打7.5折，
    否则打8折
24        if(money >= 200) {
25          //System.out.println(money*0.75);
26          return money*0.75;
27        }else if(money > 0 && money <200) {
28          //System.out.println(money*0.8);
29          return money*0.8;
30        }else {
31          System.out.println("输入金额不合法！");
32          return -1;
33        }
34      }else {
35        System.out.println("会员输入不合法！");
36        return -1;
37      }
38    }
39  }
```

课上练习二

现在有一个银行保险柜，有两道密码。想拿到里面的钱必须两次输入的密码都要正确。如果第一道密码都不正确，那直接把你拦在外面；如果第一道密码输入正确，才能有权输入第二道密码。只有当第二道密码也输入正确，才能拿到钱！（第一道密码666666，第二道密码888888)(嵌套if)

```
import java.util.Scanner;

public class hwork2 {
  public static void main(String[] args) {
    if(checkPassword()) {//checkPassword:检测密码函数
      System.out.println("密码输入正确，大门已开！");
    }
  }

  public static boolean checkPassword() {
    Scanner sc = new Scanner(System.in);
    String pwd1 = "666666";//第一个密码
    String pwd2 = "888888";//第二个密码

    System.out.println("请输入第一个密码：");
    String password = sc.nextLine();
    if(!pwd1.equals(password)) {
      System.out.println("密码输入错误！");
      return false;
    }else {
      System.out.println("请输入第二个密码：");
      password = sc.nextLine();
      if(!pwd2.equals(password)) {
        System.out.println("第二个密码输入错误！");
        return false;
      }
```

```
27
28      }
29      return true;
30    }
31  }
```

课上练习三

随机生成一个1到13的整数，如果生成的是1到10之间的数，就输出“电脑出了一张几”，比如产生了一个5就输出“电脑出了一张5”，如果生成的是11就输出“电脑出了一张J”，如果生成的是12就输出“电脑出了一张Q”，如果生成是13就输出“电脑出了一张K”

```
1  import java.util.Random;
2
3  public class hwork3 {
4    public static void main(String[] args) {
5      computerPlayCard();//调用电脑出牌的函数
6    }
7
8    public static void computerPlayCard() {
9      Random random =new Random();
10     int count = 0;//定义了一个count没啥卵用，只是单纯的让循环终止，否则CPU会直接飙到100
11     while(true) {
12       int num = random.nextInt(13) + 1;//随机产生1~13的数
13       if(num<11&&num>1) {
14         System.out.println("电脑出啦一张" + num);
15       }else if(num == 11) {
16         System.out.println("电脑出啦一张J");
17       }else if(num == 12) {
18         System.out.println("电脑出啦一张Q");
```

```
19          }else if(num==1) {
20            System.out.println("电脑出啦一张A");
21          }else {
22            System.out.println("电脑出啦一张K");
23          }
24          count++;
25          if(count==100) {//这里和上面的作用一样，就是没啥用
26            break;
27          }
28        }
29    }
30  }
```

4.3. 枚举

4.3.1. 枚举的基本定义和使用(会)

4.3.1.1. 枚举的定义

定义枚举类型， 需要使用到关键字 **enum** 。 枚举的名字是一个标识符，遵循大驼峰命名法。

```java
public enum Gender {
    // 将这个枚举对象所有可能取到的值，都列出来
    // 枚举中的元素，也是标识符，遵循大驼峰命名法
    Male, Female
}
public enum Month {
    Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
}
public enum Week {
    Mon, Tue, Wed, Thu, Fri, Sat, Sun
}
```

4.3.1.2. 枚举的使用

枚举是一种自定义的数据类型， 可以声明变量。 在使用的时候， 直接使用枚举类型.枚举值这样的形式进行枚举值的获取。

```java
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Test {
    public static void main(String[] args) {
        // 枚举的使用
        Gender gender1 = Gender.Male;
        Gender gender2 = Gender.Female;

        Month m1 = Month.Jan;
        Month m2 = Month.Nov;

```

```
        Week week1 = Week.Sat;
    }
}
```

4.3.2. 枚举中的成员定义(了解)

4.3.2.1. 枚举的分析

枚举，其实可以认为是Object类的一个最终子类。 不能被其他的类、枚举继承。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class EnumTest {
    public static void main(String[] args) {
        // 1. 获取一个枚举对象
        Gender gender = Gender.Male;
        // 1.1. 证明方式1：枚举对象，可以调用Object类中的方法，说明这些方法是从Object类中继承到的。
        String str = gender.toString();
        // 1.2. 证明方式2：可以向上转型为 Object 类型。
        Object obj = gender;
    }
}
enum Gender {
    Male, Female
}
```

4.3.2.2. 枚举中的属性定义

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public enum Gender {
    Male, Female;
    // 1. 在枚举中定义属性、方法、构造方法... 是需要写在枚举元素的下方!
    //    如果需要在枚举中定义成员，需要在最后一个枚举元素后面添加一个分号。
    public String desc;
}
```

4.3.2.3. 枚举中的构造方法定义

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Test2 {
    // 为了防止和当前包中的Gender枚举重复，在这里写成了静态内部枚举
    private static enum Gender {
        // 其实，所谓枚举中的元素，其实就是一个静态的、当前类的对象。
        Male("男"), Female("女");
        // 添加属性
        private String desc;
```

```
        // 添加构造方法，为这个属性赋值
        // 在枚举中定义构造方法，一般情况下，只是在当前的枚举中使用
        // 所以，枚举的构造方法，一般情况下，权限都是私有的
        Gender(String desc) {
            this.desc = desc;
        }
    }
    public static void main(String[] args) {
        // 1. 枚举对象的获取
        Gender gender = Gender.Male;
    }
}
```

4.3.2.4. 枚举中的方法定义

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public enum Gender {
    Male, Female;
    // 1. 在枚举中定义属性、方法、构造方法... 是需要写在枚举元素的下方!
    //    如果需要在枚举中定义成员，需要在最后一个枚举元素后面添加一个分号。
    public String desc;

    // 2. 定义方法
    public void show() {
        System.out.println("枚举中的方法定义");
```

```
    }
    public static void display() {
        System.out.println("枚举中的静态方法的定义");
    }
}
```

4.3.2.5. 枚举中的方法重写

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
private static enum Gender {
    // 其实，所谓枚举中的元素，其实就是一个静态的、当前类的对象。
    Male("男"), Female("女");
    // 添加属性
    private String desc;
    // 添加构造方法，为这个属性赋值
    // 在枚举中定义构造方法，一般情况下，只是在当前的枚举中使用
    // 所以，枚举的构造方法，一般情况下，权限都是私有的
    Gender(String desc) {
        this.desc = desc;
    }

    @Override
    public String toString() {
        return this.desc;
    }
 }
```

4.3.2.6. 枚举实现接口

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
enum Gender implements MyInterface {
    @Override
    public void test() {
        System.out.println("接口中的方法");
    }
}

interface MyInterface {
      void test();
}
```

4.3.2.7. 枚举值

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
enum Gender implements MyInterface {
    // 其实，所谓枚举中的元素，其实就是一个静态的、当前类的对象。
    Male("男") {
        // 这里，其实就相当于是一个匿名内部类
        // 在这里，可以重写Gender中的方法
```

```
        @Override
        public void test() {
            System.out.println("Male的重写实现");
        }
    },Female("女");
    // 添加属性
    private String desc;
    // 添加构造方法，为这个属性赋值
    // 在枚举中定义构造方法，一般情况下，只是在当前的枚举中使用
    // 所以，枚举的构造方法，一般情况下，权限都是私有的
    Gender(String desc) {
        this.desc = desc;
    }

    @Override
    public String toString() {
        return this.desc;
    }

    @Override
    public void test() {
        System.out.println("接口中的方法");
    }
}
```

4.4. 异常(会)

4.4.1. 异常的结构和分类

4.4.1.1. 异常的结构

在Java中，用`Throwable`类来描述所有的不正常的情况。 `Throwable` 有两个子类： `Exception` 和 `Error` 。

Error: 描述发生在JVM虚拟机级别的错误信息， 这些错误无法被处理， 不做为现在的重点。

> StackOverflowError: 栈溢出错误。

Exception： 描述程序遇到的异常。 异常是可以被捕获处理的， 是现在考虑的重点内容。

> 未经处理的异常，会导致程序无法进行编译或者运行。 因此在异常部分， 重点内容是： **异常该如何捕获处理。**

> Java中的异常的继承体系：
>
> - 根类: Throwable
>   - 错误： Error
>   - 异常: Exception
>   - 运行时异常: RuntimeException

4.4.1.2. 异常的分类

- 根据异常发生的位置

第一:普通的异常,会导致程序无法完成编译。 这样的异常被称为 -- **编译时异常**。 （Non-Runtime Exception: 非运行时异常， 但是由于异常是发生在编译时期的，因此，常常称为编译时异常。）

第二:Exception有一个子类-RuntimeException类， 在这个类中， 对异常进行了自动的处理。 这种异常不会影响程序的编译， 但是在运行中如果遇到了这种异常， 会导致程序执行的强制停止。 这样的异常被称为 -- **运行时异常**。

- 根据创建异常类的主体

第一:**系统异常**,系统提前定义好的,我们直接使用

第二:**自定义异常**,需要我们自己定义.

4.4.1.3. 异常的工作原理

示例代码

```
public class Demo7 {
    public static void main(String[] args) {//4
        Math math = new Math();
        math.div(3,0);//3
    }
}

class  Math{
    public int div(int a,int b){//2
        return  a/b;//1
    }
}
```

原理说明

- 1.在异常最初发生的位置创建一个异常的对象(new ArithmeticException()) 因为这里没有处理异常的能力,所以会将异常往上抛,抛给他所在的方法
- 2.div()方法也没有处理异常的能力,所以会继续往上抛,抛给调用这个方法

的位置

- 3.调用div()方法的位置也没有处理异常的能力,所以会继续往上抛,抛给他所在的方法
- 4.main方法也没有处理异常的能力,所以会继续往上抛,抛给JVM,JVM会调用异常对象的打印方法,将异常信息打印到控制台

4.4.1.4. 异常的特点

程序出现异常的时候,会打印异常的信息并中断程序,所以当有多个异常同时出现的时候,默认只能执行第一个.

```
public class Demo8 {
    public static void main(String[] args) {
        int[] arr = new int[] {4,5,6,6};

        //会报NullPointerException异常:空指针异常
        arr = null;

        //会报ArrayIndexOutOfBoundsException异常:数组下标越界异常
        //当前的情况下,这个异常不会执行,执行空指针异常时,程序中断
        System.out.println(arr[10]);
    }
}
```

4.4.2. 异常的捕获处理

4.4.2.1. try-catch

如果一个异常不去处理， 会导致程序无法编译或者运行。

- 语法

```
try {
  // 将可能出现异常的代码写到这里
  // 如果这里的代码出现了异常， 从出现异常的位置开始， 到try代码段结束， 所有的代码不执行。
}
catch （异常类型 标识符） {//捕获异常
  // 如果try中的代码出现了异常，并且异常的类型和catch的异常的类型是可以匹配上的，就会执行这里的逻辑
}
```

catch会对try里面的代码进行监听,如果try里面的代码没有发生异常,catch不会执行,会直接执行后面的代码.如果try里面的代码发生了异常,catch会立刻捕获(效果:try里面的代码会立刻中断,直接执行catch)

- 示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 异常的基本的捕获处理
 */
public class Demo9 {
    public static void main(String[] args) {
        Math1 math = new Math1();
        try {//可能发生异常的代码

            math.div(3,0);
            //只有try里面的代码没有发生异常,这里的代码才能执行
```

```
13              System.out.println("try");
14          }catch (ArithmeticException e){// catch的异常类型，一定要和try中实际出现的异常类型一致
15              //e.printStackTrace();  获取异常的位置,原因,名字
16              System.out.println(e.getMessage());//原因
17              System.out.println("catch");
18          }
19
20          System.out.println("go on");
21      }
22  }
23
24  class  Math1{
25      public int div(int a,int b){
26          return  a/b;
27      }
28  }
```

- 注意事项

catch中捕获的异常类型， 一定要和try中实际出现的异常类型一致。 否则将捕获不到异常， 会导致try中实际出现的异常没有被捕获处理， 依然可以终止程序的编译或运行。

4.4.2.2. 多个catch子句

- 使用场景

如果在try代码段中， 出现了多种类型的异常， 此时如果需要对这些异常进行不同的处理， 可以写多个catch子句。

> 在实际使用中:

> - 如果要针对不同的异常，进行不同的处理，可以用多个catch。
> - 如果要针对每一种异常，进行的处理方式相同，直接catch父类异常即可。

- 语法

```
try{
    可能发生异常的代码
}catch(异常一  e){ //捕获异常     e就是要捕获的异常
    对当前异常的处理
}catch(异常二  e){ //捕获异常     e就是要捕获的异常
    对当前异常的处理
}catch(Exception  e){ //捕获异常     e就是要捕获的异常
    对当前异常的处理
}

 go on
```

- 示例代码

特点:结构清晰,易于编写代码,推荐

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Demo10 {
    public static void main(String[] args) {
        Math2 math2 = new Math2();
        try {
            math2.div(3, 0);
```

```
        } catch (ArithmeticException e) {//除数为零异常
            e.printStackTrace();
        } catch (NullPointerException e) {
            e.printStackTrace();
        } catch (Exception e) {//注意:Exception异常必须放在最后一个catch
            e.printStackTrace();
        }
    }
}

class Math2{
    public int div(int a,int b) {//第二:这里也没有处理异常的能力,继续抛,抛给调用这个方法的位置
        int[] arr = null;
        System.out.println(arr[0]);
        return a/b;//第一:会首先自动生成除数为零的异常对象(new ArithmeticException()),
        //这里没有处理异常的能力,会将异常对象抛给它所在的方法
    }
}
```

- 注意事项

```
多个catch书写的先后顺序，  对异常捕获有影响吗?

- 如果多个catch捕获的异常类型之间，  没有继承关系存在，  此时先后顺序无所谓。
- 如果多个catch捕获的异常类型之间，  存在继承关系，  则必须保证父类异常在后，  子类异常在前。
```

4.4.2.3. 一个catch捕获多种异常

- 使用场景

如果try中出现了多种异常，并且某些类型的异常，处理方式相同。 并且与其他类型的处理方式不同。 此时， 可以使用一个catch捕获处理多种异常。区分一个catch里面的多个异常时通过instanceof.

缺点:这种分类方式代码很混乱,所以不推荐.

- 示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Handle3 {
    public static void main(String[] args) {
        // 需求:
        // NullPointerException ArrayIndexOutOfBoundsException 这两种异常处理方式相同， 输出 “数组相关异常”
        // ArithmeticException NumberFormatException 这两种异常处理方式相同, 输出 “格式异常”
        try {
            nullPointerTest();      // NullPointerException
            outOfBoundsTest();      // ArrayIndexOutOfBoundsException
            arithmeticTest();       // ArithmeticException
            formatException();      // NumberFormatException
```

```
        }
        catch (NullPointerException | ArrayIndexOutOfBoundsException e) {
            System.out.println("数组相关异常");

                if (e instanceof NullPointerException) {

System.out.println("NullPointerException");

                }else if (e instanceof ArrayIndexOutOfBoundsException) {


 System.out.println("ArrayIndexOutOfBoundsException");
                }
        }
        catch (ArithmeticException | NumberFormatException e) {
            System.out.println("格式异常");
        }
    }

    // NullPointerException
    private static void nullPointerTest() {
        int[] array = null;
        array[0] = 10;
    }

    // ArrayIndexOutOfBoundsException
    private static void outOfBoundsTest() {
        int[] array = new int[5];
        array[5] = 10;
```

```
42      }
43
44      // ArithmeticException
45      private static void arithmeticTest() {
46          int a = 10 / 0;
47      }
48
49      // NumberFormatException
50      private static void formatException() {
51          Integer i = Integer.valueOf("123a");
52      }
53  }
```

- 注意事项

在一个catch子句中捕获的多种类型的异常中，不允许出现有继承关系的异常类型。

4.4.2.4. finally子句

- 概念

finally出现在try-catch子句的结尾， finally代码段中的内容， 始终会执行。

- 语法

```
try{
   可能发生异常的代码
}catch(Exception  e){ //捕获异常      e就是要捕获的异常
   对当前异常的处理
}finally{
   必须执行的代码:主要用于资源的释放:比如关闭数据库,关闭流,关闭锁等
}
```

- 特点

无论try代码段中有没有异常出现，无论try里面出现的异常没有被捕获处理，finally中的代码始终会执行。基于这个特点，常常在finally中进行资源释放、流的关闭等操作

- 作用范围

我们发现在catch中执行return后main方法结束,finally还能正常执行

catch中执行System.exit(0)后,程序退出,finally不能执行了

结论:只要当前的程序在运行,finally代码就能执行.

- 示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description finally的使用
 */
public class Handle4 {
    public static void main(String[] args) {
        try {
            System.out.println(10 / 0);
```

```
        }
        catch (ArithmeticException e) {
            System.out.println("出现了算术异常");
            //return;//结束当前的函数,finally还能执行
            System.exit(0);//退出程序,finally不能执行了
        }
        finally {
            System.out.println("finally代码段中的内容执行了");
        }
        System.out.println("end");
    }
}
```

4.4.2.5. try-finally语句

- 语法

```
try{
  获取资源
}finally{
  释放资源
}
```

- 特点

这个结构跟异常没有关系,主要用于资源的管理.

- 示例代码

```
public class Demo11 {
    public static void main(String[] args) {
        //创建锁对象
        Lock lock;
        try {//获取锁
            lock.lock();
        } finally {//释放锁
            lock.unlock();
        }
      System.out.println("go on");
    }
}
```

4.4.3. 两个关键字

4.4.3.1. throw

- 概念/使用场景

```
一个异常对象，被实例化完成后，没有任何意义。不会影响到程序的编译或者运行。
如果希望某一个异常对象生效（可以影响程序的编译、运行），需要使用关键字 throw 进行异常的抛出。
```

- 示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description throw关键字
 */
public class Handle5 {
```

```
    public static void main(String[] args) {
        int ret = calculate(10, 20);
        System.out.println(ret);
    }

    private static int calculate(int a, int b) {
        if (a > b) {
            return a - b;
        }
        // 否则，视为实参有逻辑错误，抛出一个异常
        RuntimeException exception = new RuntimeException();
        // 让当前的exception异常生效，使其可以终止程序的运行。
        // 而且，在一个方法中抛出了异常，从这个位置开始，向后所有的代码都不执行了。
        throw exception;
    }
}
```

4.4.3.2. throws

- 概念/使用场景

```
用在方法的声明部分，写在参数列表后面，方法体前面。
定义在方法中，表示这个方法过程结束中，可能会遇到什么异常。
定义了throws异常抛出类型的方法，在当前的方法中，可以不处理这个异常，由调用方处理。
```

- 示例代码

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
```

```java
import java.util.Date;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Handle6 {
    public static void main(String[] args) {
        try {
            test2();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    private static void test2() throws ParseException {
        test();
    }

    // throws ParseException:
    // 1. 告诉调用方，这个方法有一个异常，在使用的时候，需要注意。
    // 2. 在这个方法中，如果遇到了ParseException异常，可以不去处理，谁调用这个方法谁处理。
    private static void test() throws ParseException {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        // 将一个指定格式的时间字符串，解析为一个Date对象
        Date date = sdf.parse("2000-01-01");
        System.out.println(date);
    }
}
```

4.4.4. 自定义异常

4.4.4.1. 为什么要自定义异常

使用异常， 是为了处理一些重大的逻辑BUG。 这些逻辑BUG可能会导致程序的崩溃。 此时， 可以使用异常机制， 强迫修改这个BUG。

系统中， 提供很多很多的异常类型。 但是， 异常类型提供的再多， 也无法满足我们所有的需求。 当我们需要的异常类型， 系统没有提供的时候，此时我们就需要自定义异常了。

4.4.4.2. 如何自定义异常

其实，系统提供的每一种异常，都是一个类。所以，自定义异常，其实就是写一个自定义的异常类。

- 如果要自定义一个编译时的异常，需要继承自 Exception 类。

特点:对异常进行处理的所有工作都要我们手动完成

- 如果要自定义一个运行时的异常，需要继承自 RuntimeException 类。

特点:所有的工作我们都可以不管

> 规范:
>
> 自定义的异常类， 理论上来讲， 类名可以任意定义。 但是出于规范，一般都会以 Exception 作为结尾。
>
> 例如： ArrayIndexOutOfBoundsException、NullPointerException、 ArithmeticException...

> 常见的异常:订单异常 用户异常 负数异常

4.4.4.3. 示例代码

以负数异常为例：

对于编译异常需要我们进行处理的有：
异常类的创建----FuShuException
异常对象的创建---应该是在发生异常的位置
异常对象的抛出---throw
异常的声明(我们要给可能发生异常的方法进行异常的声明)----throws
    作用:告诉别人我有可能发生异常

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 自定义的异常类
 */
class  FuShuException extends  Exception{
      // 异常的描述信息
    // 在根类 Throwable 中，有一个私有的属性 detailMessage，存储异常的描述信息。
    // 在自定义异常描述信息的时候，只需要添加一个有参的构造方法即可完成
    public FuShuException(){}
    public FuShuException(String message){
        //这行代码必须写
        // 调用父类中的构造方法，
        // 在父类中，再调用它的父类中的构造方法，一层层向上调用，最终可以调用到Throwable类中的有参构造
```

```
15          // 实现对 detailMessage 属性的赋值。
16          super(message);
17      }
18  }
19  public class Demo2 {
20      public static void main(String[] args) //throws
    FuShuException
21      {
22          Math math = new Math();
23          try {
24              math.div(2,-3);
25          }catch (FuShuException e){
26              //异常的解决方案
27              e.printStackTrace();
28          }
29
30      }
31  }
32
33  class  Math{
34      //异常的声明(我们要给可能发生异常的方法进行异常的声明)----
    throws       作用:告诉别人我有可能发生异常
35      public int div(int a,int b)throws FuShuException
36      {
37          if (b < 0){
38              //创建异常对象并抛出
39              throw  new FuShuException("除数为负数了");
40          }
41          return a/b;
42      }
43  }
```

提问:自定义异常构造方法中的参数是如何在打印方法中显示的?

说明案例:老师上课

```
public class Demo3 {
    public static void main(String[] args) {
        Teacher teacher = new Teacher("除数为负数了");
        teacher.printStackTrace();
    }
}

class Person{//相当于Exception
    private String message;
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public Person() {
        super();
    }
    public Person(String message) {
        super();
        this.message = message;
    }
    public void printStackTrace() {
        System.out.println(this.getMessage());
    }
}

//相当于FuShuException
class Teacher extends Person{
    public Teacher() {}
    public Teacher(String message) {
```

```
        super(message);
    }
}
```

4.4.4.4. 在重写方法中使用异常

注意点:

- 子类的同名方法中声明的异常等级要<=父类的.
- 如果子类同名方法声明了异常,父类必须声明异常.
- 父类抛出了异常,子类在重写方法的时候，可以不抛出异常

```
//父类
class Teacher {
    public Teacher() {}
    public void show(int a) throws Exception1,Exception2
    {
        if (a >4) {
            throw new Exception1();
        }else {
            throw new Exception2();
        }
    }
}
//子类
class GoodTeacher extends Teacher{
    @Override
    public void show(int a) throws Exception1, Exception2 {
    }
}
//两个异常对象
class Exception1 extends Exception{
```

```
}
class Exception2 extends Exception{
}
```

- 如果父类方法抛出的异常是 **编译时异常**
  - 子类重写方法的时候，可以抛出相同的异常，或子类异常

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

class Child1 extends Father {
    /**
     * 此时抛出的异常类型与父类的方法一致
     */
    @Override
    public void test() throws IOException {}
}

class Child2 extends Father {
    /**
     * 此时抛出的异常类型是父类方法抛出的异常类型的子类型
     * FileNotFoundException是IOException的子类
     */
    @Override
    public void test() throws FileNotFoundException {}
}
```

- 子类重写方法的时候，可以抛出运行时异常

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

```
}

class Child1 extends Father {
    @Override
    public void test() throws NullPointerException {}
}

class Child2 extends Father {
    @Override
    public void test() throws ArithmeticException {}
}

class Child3 extends Father {
    @Override
    public void test() throws
ArrayIndexOutOfBoundsException {}
}

class Child extends Father {
    @Override
    public void test() throws RuntimeException {}
}
```

1. 注意事项

> 除了上述两种情况外，其他的都是错误的。
>
> 例如：

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

class Child extends Father {
@Override
public void test() throws ParseException {}
}
```

上述代码中，**IOException**是编译时异常，**ParseException**也是编译时异常。但是两者没有继承关系存在，因此这是错误的。

例如:

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

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

上述代码中，**IOException**是编译时异常，**IOException**也是编译时异常。但是**Exception**不是**IOException**的子类，因此这是错误的。

- 如果父类方法抛出的异常是 **运行时异常**
  - 子类在重写的时候，可以是任意的运行时异常，不能是编译时异常

```java
class Father {
    public void test() throws NullPointerException {}
}

class Child1 extends Father {
    @Override
    public void test() throws ArithmeticException {}
}

class Child2 extends Father {
    @Override
    public void test() throws RuntimeException {}
}
```