JAVA

一、第一章

1.1 特点：

- 简单性：在Java语言当中真正操作内存的是：JVM（Java虚拟机）所有的java程序都是运行在Java虚拟机当中的。
  而Java虚拟机执行过程中再去操作内存。Java语言屏蔽了指针概念，程序员不能直接操作指针，或者说程序员不能直接操作内存。
- 健壮性：自动垃圾回收机制（GC机制）
- 可移植性/跨平台：JVM屏蔽了各操作系统之间的差异

1.2 JDK、JRE、JVM

- JDK:Java开发工具箱；JRE:java运行环境；JVM:java虚拟机
- JDK包括JRE，JRE包括JVM
- JDK和JRE有独立安装包

```
生成帮助文档
 javadoc -d javaapi -author -version java文件名
注意：只有/**     */ 注释的才能被javadocs
```

1.3

- 一个java源文件中可以定义多个类，但公共类只能有一个，且类名必须和Java源文件名保持一致。任何一个class中都可以编写main方法。

二、数据类型

> 字符和字符串按照单双引号区别，Java中字符只能是单个字符

1、整数型

> 任何情况下Java中整数型字面量被默认当作int类型处理，如果想表示long类型需要在字面量添加L/l，建议写L

- java中小容量可以直接赋值给大容量，这个过程称为自动类型转换
- 大容量不允许“直接”赋值给小容量，需要使用强制类型转换符(int)(short)(char)，但运行时可能会损失精度

```
int a=100           //不存在类型转换
long b=200          //存在类型转换，小容量转大容量
long c=300L         //不存在类型转换    300L当作long类型
long d =2147483648   //编译器会报错，因为2147483648被当作int类型，而其已经超过了int最大范围
byte m = 300         //编译错误，大容量转小容量
```

- 规则：当整数型字面量没有超过byte/short/char的取值范围，那么这个整数型字面量可以直接赋值给byte/short/char类型的变量

```
byte x = 1
byte x = 127      //不会报错
byte y = 128      //会报错
char x = 97       //输出为a,当一个整数赋值给char类型变量的时候，会自动转换成char字符型，最终的结果是一个字符。
```

1.1二进制原码、反码、补码

```
1、计算机在任何情况下都只能识别二进制
2、计算机在底层存储数据的时候，一律存储的是“二进制的补码形式”
    计算机采用补码形式存储数据的原因是：补码形式效率最高。
3、什么是补码呢？
    实际上是这样的，二进制有：原码 反码 补码
4、记住：
    对于一个正数来说：二进制原码、反码、补码是同一个，完全相同。
    int i = 1;
对应的二进制原码：00000000 00000000 00000000 00000001
对应的二进制反码：00000000 00000000 00000000 00000001
对应的二进制补码：00000000 00000000 00000000 00000001
    对于一个负数来说：二进制原码、反码、补码是什么关系呢？
    byte i = -1;
对应的二进制原码：10000001
对应的二进制反码（符号位不变，其它位取反）：11111110
对应的二进制补码（反码+1）：11111111
    或者：求负数的补码：可以先求其绝对值的原码，按位取反，加1
5、分析 byte b = (byte)150;
这个b是多少？
    int类型的4个字节的150的二进制码是什么？
    00000000 00000000 00000000 10010110
    将以上的int类型强制类型转为1个字节的byte，最终在计算机中的二进制码是：
    10010110（最高位是1，表示负数）
    千万要注意：计算机永远存储的都是二进制补码形式。也就是说上面
    10010110 这个是一个二进制补码形式，你可以采用逆推导的方式推算出
    这个二进制补码对应的原码是啥！！！！！！
    10010110 ---> 二进制补码形式
    10010101 ---> 二进制反码形式
    11101010 ---> 二进制原码形式（-106）
```

1.2混合运算

- byte、char、short做混合运算的时候，各自先转换成int再做运算。
- 多种数据类型做混合运算的时候，最终的结果类型是“最大容量”对应的类型。
  char+short+byte 这个除外。
  因为char + short + byte混合运算的时候，会各自先转换成int再做运算。

```
char c1 = 'a';
byte b = 1;
System.out.println(c1 + b); // 98

// 错误: 不兼容的类型: 从int转换到short可能会有损失
short s = c1 + b;   // 编译器不知道这个加法最后的结果是多少。只知道是int类型。
```

2、浮点型

- long类型占用8个字节；float类型占用4个字节。
  注意：任意一个浮点型都比整数型空间大。float容量 > long容量。
- java中规定，任何一个浮点型数据默认被当做double来处理。如果想让这个浮点型字面量被当做float类型来处理，那么
  请在字面量后面添加F/f。
- 1.0 那么1.0默认被当做double类型处理。
  1.0F 这才是float类型。 (1.0f)

3、转换规则

- 第一条：八种基本数据类型中，除 boolean 类型不能转换，剩下七种类型之间都可以进行转换；
- 第二条：如果整数型字面量没有超出 byte,short,char 的取值范围，可以直接将其赋值给byte,short,char 类型的变量；
- 第三条：小容量向大容量转换称为自动类型转换，容量从小到大的排序为：
  byte < short(char) < int < long < float < double，其中 short和 char 都占用两个字节，但是char 可以表示更大的正整数；
- 第四条：大容量转换成小容量，称为强制类型转换，编写时必须添加“强制类型转换符”，但运行时可能出现精度损失，谨慎使用；
- 第五条：byte,short,char 类型混合运算时，先各自转换成 int 类型再做运算；
- 第六条：多种数据类型混合运算，各自先转换成容量最大的那一种再做运算；

三、运算符

- ++i 先加再用 i++ 先用再加 但只要执行了，i的值必加1

```
int c = 90;
System.out.println(c++);  // 90
System.out.println(c); // 91
int d = 80;
System.out.println(++d); //81
System.out.println(d); // 81
```

```
短路&&  短路||  （大部分用这两，效率高）
int m = 10;
int n = 11;
// 使用短路与&&的时候，当左边的表达式为false的时候，右边的表达式不执行，使用短路与||的时候，当左边的表达式为true的时候，右边的表达式不执行，这种现象被称为短路。
System.out.println(m > n && m > n++);
System.out.println(n); // 11
```

- 使用扩展赋值运算符的时候，永远都不会改变运算结果类型。

```
byte x = 100;
x += 1;        //x自诞生以来是byte类型，那么x变量的类型永远都是byte。不会变。不管后面是多大的数字。
```

```
i += 10 和 i = i + 10   //是不一样的
byte x = 100;// 错误: 不兼容的类型: 从int转换到byte可能会有损失
x = x + 1; // 编译器检测到x + 1是int类型，int类型不可以直接赋值给byte类型的变量x
x += 1;  //这是可以的，其实 x += 1 等同于：x = (byte)(x + 1);
System.out.println(x); // 101
x += 199; // x = (byte)(x + 199);
System.out.println(x); // 44 （会自动损失精度了。）
```

- 字符串连接+

  当 + 运算符两边都是数字类型的时候，求和。
  当 + 运算符两边的“任意一边”是字符串类型，那么这个+会进行字符串拼接操作。
  一定要记住：字符串拼接完之后的结果还是一个字符串。

- 拼接口诀：加一个双引号""，然后双引号之间加两个加号："++"，然后两个加号中间加变量名："+name+"

```
int a = 100;int b = 200;
System.out.println(a + b + "110"); // 最后一定是一个字符串："300110"
System.out.println(a + "+" + b + "=" + a + b); //100+200=100200
System.out.println(a + "+" + b + "=" + (a + b));//100+200=300
String name = "李四";
System.out.println("登录成功欢迎"+name+"回来");
```

```java
int k = 10;
k = k++;
System.out.println(k);  //10
// java中运行结果是10  c++中运行结果是11 为什么？因为java和c++的编译器是不同的人开发的。原理不同。
//其实k = k++;对应的是下面三行代码
int temp = k;       k++;        k = temp;
```

四、控制语句

- 选择语句：if语句 switch语句
- 循环语句：for循环 while循环 do..while..循环
- 转向语句：break continue return
    - switch语句本质上是只支持int和String，但是byte,short,char也可以，因为byte short char可以进行自动类型转换。（long类型不可以，需要加强转）如果分支执行了，但是分支最后没有“break;”，此时会发生case穿透现象。（下个分支不再进行匹配，会直接执行，直到遇到break)

> 强调一下：在for循环当中声明的变量只在for循环中有效，属于for循环的局部变量，for循环结束后i的内存就释放了。这个i变量属于for循环域。在main方法中没有办法直接使用。
>
> 循环嵌套，分析需要用几层循环，每层实现什么功能

- break;语句可以用在哪里呢?
  第一个位置：switch语句当中，用来终止switch语句的执行。防止case穿透现象。
  第二个位置：break;语句用在循环语句当中，用来终止循环的执行。*/
- break;语句会让离它最近的循环终止结束掉。
- continue；终止当前"本次"循环，直接进入下一次循环继续执行。

```java
a:for(int k = 0; k < 2; k++){
  b:for(int i = 0; i < 10; i++){
    if(i == 5){
      break a; // 终止指定的循环。
    }
    System.out.println("i ===> " + i);
  }
}
```

五、方法

- 方法是一段可以完成某个特定功能的并且可以被重复利用的代码片段。

- 方法不调用是不会执行的（main方法是JVM调用的）
- 方法中的形参变量都属于局部变量，方法结束之后，局部变量占用的内存会自动释放。因为在同一块内存空间中，变量不能重名，不释放的话，下次调用重新声明变量，会冲突。
- 如果返回值类型“不是void”，那么你在方法体执行结束的时候必须使用"return 值;"这样的语句来完成“值”的返回。
- return只要执行，当前所在的方法（离他最近）结束，记住：不是整个程序结束。
- 如果返回值类型是void，那么在方法体当中不能有“return 值;”这样的语句。但是可以有“return;”语句。这个语句“return;”的作用就是用来终止当前方法的。

> 注：return不能返回多个值，要返回多个值可以用数组
>
> 语句“return;”只能出现在返回值类型是void的方法中

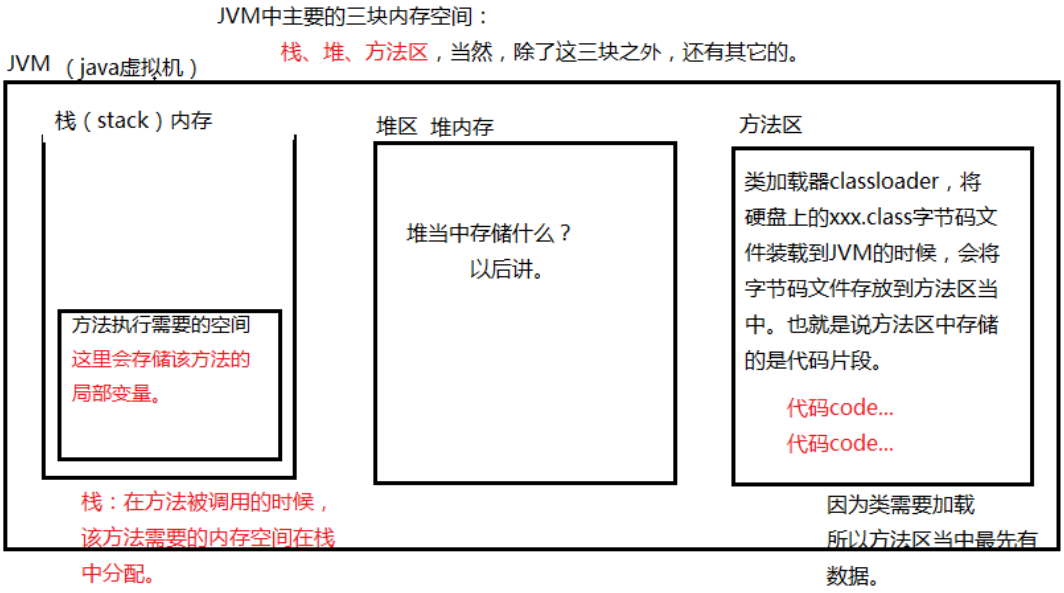
5.1调用

- 类名.方法名(实际参数列表);
- a()方法调用b()方法的时候，a和b方法都在同一个类中，“类名.”可以省略。如果不在同一个类中“类名.”不能省略。
- 在同一个域当中，"return语句"下面不能再编写其它代码。编写之后编译报错。

```
//错误：缺少返回语句
public static int m(){
  boolean flag = true;
  if(flag){
    return 1;
  }
} //编译器觉得这行代码可能会执行，当然也可能不会执行。编译器为了确保程序不出现任何异常，所以编译器说：缺少返回语句
```

5.2栈数据结构

- 5.2.1 JVM内存结构



- 5.2.2数据结构：是计算机存储数据的容器，该容器存在不同的结构。

  方法调用叫做：压栈。分配空间
  方法结束叫做：弹栈。释放空间

  方法执行过程遵循自上而下的顺序依次执行，采用先进先出的栈数据结构保证了代码的执行顺序

5.3方法重载

- 前提：方法功能相似
- 满足三个条件：在同一个类当中；方法名相同；参数列表不同
- 方法重载和方法的“返回值类型”无关。
- 方法重载和方法的“修饰符列表”无关。
- 方法重载只和方法名和形式参数列表有关

> println 就是写好的方法，利用方法重载，定义了一系列的println方法，只是后面的形式参数类型不同

5.4方法递归

- 方法自己调用自己，就是方法递归
- 当递归时程序没有结束条件，一定会发生：栈内存溢出错误：StackOverflowError所以：递归必须要有结束条件。（这是一个非常重要的知识点。原因：一直压栈，没有弹栈，栈内存不够
- 递归假设是有结束条件的，就一定不会发生栈内存溢出错误吗？
  假设这个结束条件是对的，是合法的，递归有的时候也会出现栈内存溢出错误。因为有可能递归的太深，栈内存不够了。因为一直在压栈。
- 在实际的开发中，假设有一天你真正的遇到了：StackOverflowError 你怎么解决这个问题，可以谈一下你的思路吗？
  第一步：先检查递归的结束条件对不对。如果递归结束条件不对，必须对条件进一步修改，直到正确为止。

第二步：假设递归条件没问题，怎么办？这个时候需要手动的调整JVM的栈内存初始化大小。可以将栈内存的空 间调大点。（可以调整大一些。）
第三步：调整了大小，运行还是出现这个错误，没办法只能继续扩大栈的内存（Java -x 可以查看调整堆栈大小的参数）。

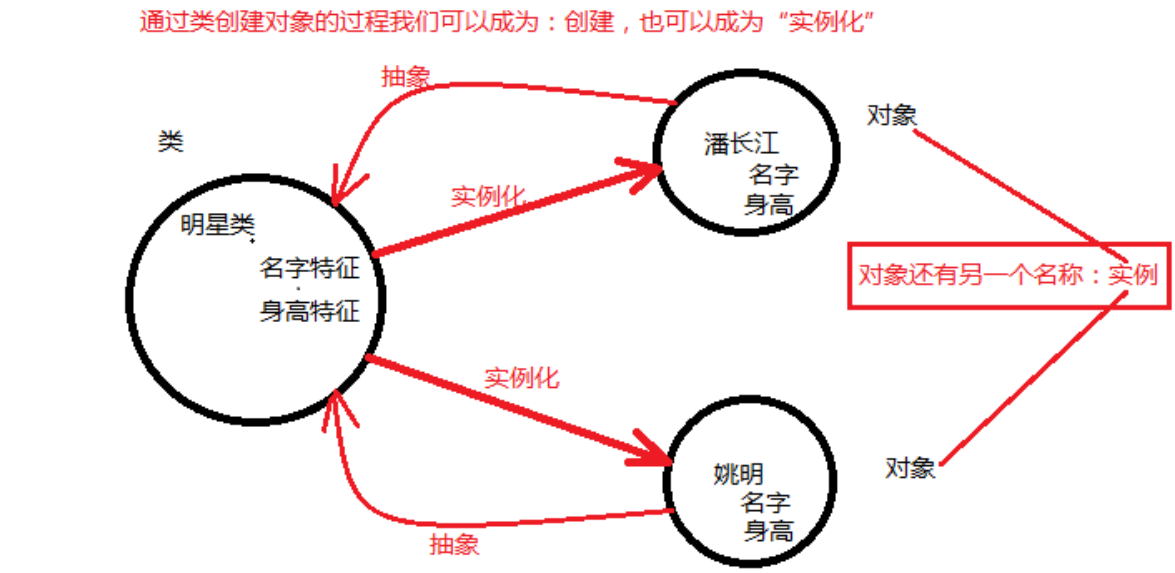
六、面向对象

6.1面向对象和面向过程区别

- 面向过程：注重的是实现这个功能的步骤；另外面向过程也注重实现功能的因果关系。
  缺点：耦合度高，扩展力差
  优点：快速开发，对于小型项目（功能），采用面向过程的方式进行开发，效率较高。
  面向过程最主要是每一步与每一步的因果关系，其中A步骤因果关系到B步骤，A和B联合起来形成一个子模块，子模块和子模块之间又因为因果关系结合在一起，假设其中任何一个因果关系出现问题（错误），此时整个系统的运转都会出现问题。（代码和代码之间的耦合度太高，扩展力太差。）
- 面向对象（OO）：注重把不同的对象，以及组合方式。耦合度低，扩展力强。（对象可以更换）
  三大特征：封装、继承、多态

6.2 类

- 类：本质上是现实世界当中某些事物具有共同特征，将这些共同特征提取出来形成的概念就是一个“类”，“类”就是一个模板。(实际在现实中不存在的)
- 对象：是实际存在的个体。（真实存在的个体）
- 在java语言中，要想得到“对象”，必须先定义“类”，“对象”是通过“类”这个模板创造出来的。一个类可以创建多个对象
- 实例：对象还有另一个名字叫做实例。
- 实例化：通过类这个模板创建对象的过程，叫做：实例化。
- 抽象：多个对象具有共同特征，进行思考总结抽取共同特征的过程。



注意：名字和身高都属于明星对象的共同特征。

- 类 = 属性 + 方法　属性描述状态，方法描述的是行为动作
- JAVA中所有的类都是引用数据类型

```
定义：
    [修饰符列表] class 类名 {
            类体 = 属性 + 方法
            // 属性在代码上以“变量”的形式存在（描述状态）；因为属性以数据形式存在，数据在程序中放在变量中。
            // 方法描述动作/行为
        }
变量根据出现位置进行划分：
    方法体当中声明的变量：局部变量。
    方法体外声明的变量：成员变量。（这里的成员变量就是“属性”）
```

6.3 对象的创建和使用

- 创建：类名 变量名 = new 类名();　--->后面其实是构造方法名

```
XueSheng s1 = new XueSheng();    //s1是变量名（s1不能叫做对象。s1只是一个变量名字。）
                                 // XueSheng是变量s1的数据类型（引用数据类型）  new XueSheng() 这是一个对象。（学生类创建出来的学生对象。）
```

- 使用

对于成员变量：没有手动赋值，系统会默认赋值（值见下图，其中引用数据类型默认赋值为：null）。

对于局部变量：必须先声明，再赋值然后才能访问

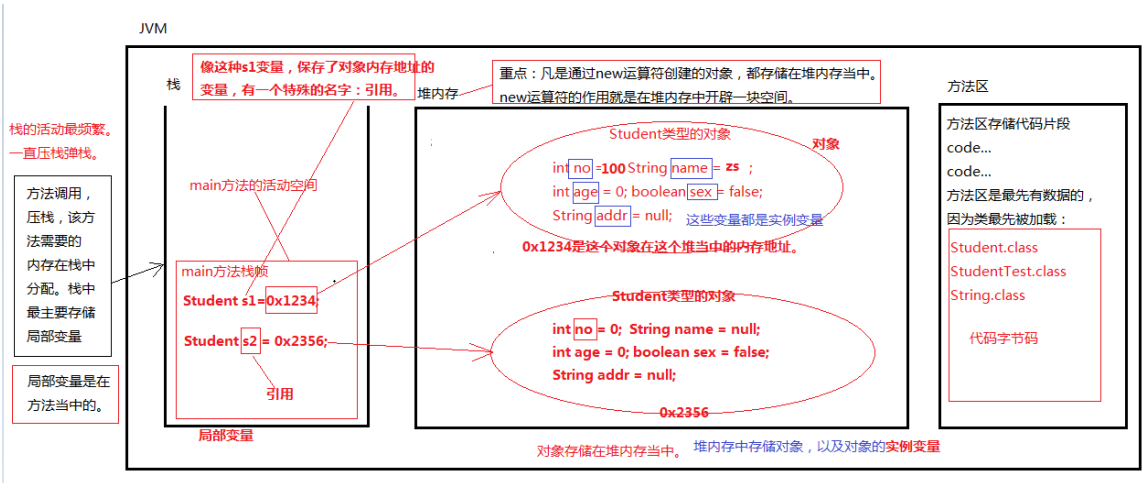
```
什么是实例变量？
    对象又被称为实例。实例变量实际上就是：对象级别的变量。
```

```
    public class 明星类{
        double height;
    }
    身高这个属性所有的明星对象都有，但是每一个对象都有“自己的身高值”。
    假设创建10个明星对象，height变量应该有10份。
    所以这种变量被称为对象级别的变量。属于实例变量。
    //实例变量在访问的时候，是不是必须先创建对象？  是的

怎么访问实例变量？
    //语法：引用.实例变量名
    public class Student{
        int no;
        String name;
        int age;
    }
    Student s1 = new Student();  //创建对象，s1叫引用，s1中存储的是对象在堆区的地址
    System.out.println(s1.no);
    System.out.println(s1.name);
```

一定注意：不能通过类名直接访问实例变量，Student.name(这是错误的，必须先创建对象)。因为实例变量无论是手动赋值还是系统赋值，都是在构造方法执行的时候进行的，不进行创建对象是不会在堆区开辟空间的，实例变量也不会被赋值。

6.4 java虚拟机内存管理

- 堆内存中存储对象以及对象的实例变量
  栈中存放局部变量
- 对象和引用的区别？
  对象是通过new出来的，在堆内存中存储。
  引用：是变量，并且该变量中保存了内存地址指向了堆内存当中的对象的。
- 每new一下，就会在堆内存中开辟一块空间



> 注：只要这个变量中保存的是一个对象的内存地址，那么这个变量就叫做“引用”。

> 引用不一定是局部变量

```
public class Address{
    String city;
    String street;
    String zipcode;
}
public class User{
    // 类=属性+方法
    // 以下3个都是属性，都是实例变量。（对象变量。）
    int id;
    String username;

    // 家庭住址,Address是一种引用数据类型
    // addr是成员变量并且还是一个实例变量
    Address addr;   // addr是一个引用,但在堆区存储，是一个实例变量
}
一定注意，addr中只能存储内存地址
```

6.5 空指针异常--NullPointerException

- 这是运行错误
- 出现空指针异常的前提条件是："空引用"访问实例【对象相关】相关的数据时，都会出现空指针异常。(包括实例变量和实例方法)

```
class Customer{
    int id;
}
public class NullPointerTest{
    public static void main(String[] args){
        Customer c = new Customer();
        System.out.println(c.id); // 0

        c = null;
        System.out.println(c.id);//空指针异常 编译没问题，因为编译器只检查语法，编译器发现c是Customer类型，Customer类型中有id属性，所以可以：c.id。语法过了。  但是运行的时候需要对象的存在，但是对象没了,就只能出现一个异常。
    }
```

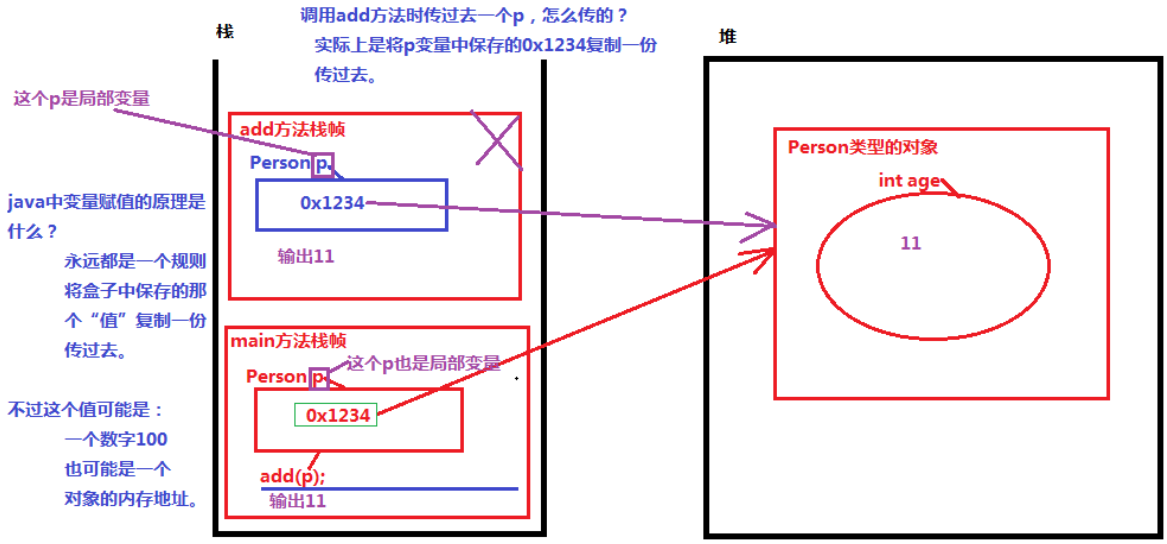
- 在java语言中，垃圾回收器主要针对的是堆内存。当一个java对象没有任何引用指向该对象的时候，GC会考虑将该垃圾对象释放回收掉。

6.6方法调用时参数传递

- 不管你是基本数据类型，还是引用数据类型，实际上在传递的时候都是将变量中保存的那个"值"复制一份，传过去。

- 在传递引用数据类型时，由于传递的这个值是java对象的内存地址，所以会导致两个引用指向同一个堆内存中的java对象。通过其中一个去改变堆内存中对象的值，另一个也会改变。

```java
public class Test2{
    public static void main(String[] args){
        Person p = new Person();
        p.age = 10;
        add(p);
        System.out.println("main--->" + p.age); //11
    }
    // 方法的参数可以是基本数据类型，也可以是引用数据类型，只要是合法的数据类型就行。
    public static void add(Person p){ // p是add方法的局部变量。
        p.age++;
        System.out.println("add--->" + p.age); //11
    }
}
class Person{
    int age;
}
```



6.7 构造方法(构造器、Constructor)

- 构造方法是用来创建对象，并且同时给对象的属性赋值。（注意：实例变量没有手动赋值的时候，系统会赋默认值。）
- 重点：当一个类没有提供任何构造方法，系统会默认提供一个无参数的构造方法。（而这个构造方法被称为缺省构造器。）
  重点：当一个类中手动的提供了构造方法，那么系统将不再默认提供无参数构造方法。建议将无参数构造方法手动的写出来，这样一定不会出问题。
- 调用构造方法怎么调用呢？
  使用new运算符来调用构造方法： new 构造方法名(实际参数列表);
  普通方法调用是：类名.方法名(实际参数列表);

```
语法结构：
[修饰符列表] 构造方法名(形式参数列表){
    构造方法体；
    通常在构造方法体当中给属性赋值，完成属性的初始化。
}
注意：第一：修饰符列表目前统一写：public。千万不要写public static。
    第二：构造方法名和类名必须一致。
    第三：构造方法不需要指定返回值类型，也不能写void，写上void表示普通方法，就不是构造方法了。
    第四：构造方法的放回值类型是当前 类 的类型
    第五：构造方法支持方法重载，在一个类中构造方法可以有多个，并且所有的构造方法名字都是一样的。
```

- 无参数构造方法和有参数的构造方法都可以调用。
  Student x = new Student(); ------>这是调用无参的
  Student y = new Student(123);------>这是有参的

```
public class Student{
    int no;
    String name;
    int age;
    // 无参数的构造方法（不写的话系统会默认提供，但是写了有参数的，系统就不再提供了，最好手动写出来）
    public Student(){
    }
    // 定义一个有参数的构造方法
    public Student(int i){
    }
}
```

- 实例变量默认赋值是在构造方法执行过程中进行赋值的

```
public Student(){
        //这里实际上有三行代码你看不见。
        // 无参数构造方法体当中虽然什么代码都没写，但是实际上是在这个方法体里面进行的实例变量默认值初始化.有参数的也是一样的，不写就会默认赋值。
        /*
        no = 0;
        name = null;
        age = 0;
        */

        // 这就表示不再采用系统默认值，手动赋值了。
        id = 111;
        name = "lisi";
        age = 30;
    }
```

七、封装

- 作用：保证内部结构的安全；屏蔽复杂，暴露简单
- 步骤：第一步：属性私有化（使用private关键字进行修饰。）第二步：对外提供对外提供公开的set方法和get方法作为操作入口,并且都不带static。都是实例方法。。
- 实例方法(对象级别的方法)：没有static。通过“引用.”访问实例方法，普通方法是通过“类名.”访问
- private 表示私有的，被这个关键字修饰之后，该数据只能在本类中访问。

7.1 get和set

```
get方法的要求：
    public 返回值类型 get+属性名首字母大写(无参){
    return xxx;
}
set方法的要求：
    public void set+属性名首字母大写(有1个参数){
    xxx = 参数;
}
```

八、this和static

8.1 static-"静态"

- 所有static关键字修饰的都是类相关的，类级别的。
- 所有static修饰的，都是采用“类名.”的方式访问。
- static修饰的变量：静态变量；static修饰的方法：静态方法
- 变量在方法体当中声明的变量叫做：局部变量。在方法体外声明的变量叫做：成员变量。成员变量又可以分为：实例变量和静态变量。
- 成员变量系统会赋默认值

```
class VarTest{
    // 以下实例的，都是对象相关的，访问时采用“引用.”的方式访问。需要先new对象。
    // 实例相关的，必须先有对象，才能访问，可能会出现空指针异常。

    // 成员变量中的实例变量
    int i;
    // 实例方法
    public void m2(){
        // 局部变量
        int x = 200;
    }

```

```
    // 以下静态的，都是类相关的，访问时采用“类名.”的方式访问。不需要new对象。
    // 不需要对象的参与即可访问。没有空指针异常的发生。

    // 成员变量中的静态变量
    static int k;
    // 静态方法
    public static void m1(){
        // 局部变量
        int m = 100;
    }
}
```

> 注：只要是方法，不管是静态方法、实例方法、构造方法，它们在运行时都需要压栈。

- 重点：静态变量在类加载时初始化，不需要new对象，静态变量的空间就开出来了。静态变量存储在方法区，静态变量是类级别的；实例变量是对象级别，存储在堆区；局部变量存储在栈区。
- 静态变量可以用：“类名.”访问，当也可以用“引用.”访问，但不建议用
- 空指针异常：只有在“空引用”访问“实例”相关的，都会出现空指针异常

```
public class StaticTest03{
    public static void main(String[] args){
        // 通过"类名."的方式访问静态变量
        System.out.println(Chinese.country);
        // 创建对象
        Chinese c1 = new Chinese("1111111", "张三");
        System.out.println(c1.idCard); // 1111111
        System.out.println(c1.name); // 张三
        System.out.println(c1.country); // 中国    不建议这么写

        // c1是空引用
        c1 = null;
        // 分析这里会不会出现空指针异常？
        // 不会出现空指针异常。
        // 因为静态变量不需要对象的存在。
        // 实际上以下的代码在运行的时候，还是：System.out.println(Chinese.country);
        System.out.println(c1.country);

        // 这个会出现空指针异常，因为name是实例变量。
        //System.out.println(c1.name);
    }
}
class Chinese{

    // 实例变量
    String idCard;
```

```
    String name;

    // 静态变量
    static String country = "中国";

    //构造方法
    public Chinese(String x, String y){
        idCard = x;
        name = y;
    }
}
```

- 定义一个方法，方法体中直接访问了实例变量，这个方法就是实例方法。
- 静态代码块---类加载时执行。并且只执行一次。静态代码块在类加载时执行，并且在main方法执行之前执行。静态代码块一般是按照自上而下的顺序执行。
- 注：静态代码块存储在方法区
- 作用：静态代码块这种语法机制实际上是SUN公司给我们java程序员的一个特殊的时刻/时机。这个时机叫做：类加载时机。可以用来记录日志

```
语法：
static {
        java语句;
        java语句;
}
```

> 总结：第一：对于一个方法来说，方法体中的代码是有顺序的，遵循自上而下的顺序执行。
>
> 第二：静态代码块1和静态代码块2是有先后顺序的。
>
> 第三：静态代码块和静态变量是有先后顺序的。

- 实例语句块----不是在类加载时执行。只要是构造方法执行，必然在构造方法执行之前，自动执行“实例语句块”中的代码。实际上这也是SUN公司为java程序员准备一个特殊的时机，叫做对象构建时机。
- 语法：{
  java语句;
  }

```
public class InstanceCode{
    //入口
    public static void main(String[] args){
        System.out.println("main begin");
        new InstanceCode();
        new InstanceCode();

        new InstanceCode("abc");
        new InstanceCode("xyz");
```

```
    }
    //实例语句块
    {
        System.out.println("实例语句块执行！");
    }
    // Constructor
    public InstanceCode(){
        System.out.println("无参数构造方法");
    }
    // Constructor
    public InstanceCode(String name){
        System.out.println("有参数的构造方法");
    }

}
```

8.2 this

- this是一个变量，是一个引用。this保存当前对象的内存地址，指向自身。一个对象一个this。
- 所以，严格意义上来说，this代表的就是“当前对象”;this存储在堆内存当中对象的内部。
- this可以使用在实例方法中也可以使用在构造方法中
- “this.”大部分情况下是可以省略的。
- 为什么this不能使用在静态方法中？this代表当前对象，静态方法中不存在当前对象。

```
public class ThisTest01{
    public static void main(String[] args){
        Customer c1 = new Customer("张三");
        c1.shopping();

        Customer c2 = new Customer("李四");
        c2.shopping();

        Customer.doSome();
    }
}
// 顾客类
class Customer{
    String name;
    public Customer(){

    }
    public Customer(String s){
        name = s;
    }

```

```
    // 顾客购物的方法
    // 实例方法
    public void shopping(){
        // 这里的this是谁？this是当前对象。
        // c1调用shopping(),this是c1
        // c2调用shopping(),this是c2
        //System.out.println(this.name + "正在购物!");

        // this. 省略的话，还是默认访问“当前对象”的name。
        System.out.println(name + "正在购物!");
    }

    // 静态方法
    public static void doSome(){
        // this代表的是当前对象，而静态方法的调用不需要对象。矛盾了。
        // 错误: 无法从静态上下文中引用非静态 变量 this
        System.out.println(this);
    }
}
```

- get和set为什么是实例方法？因为这两个方法中都直接访问了实例变量
- 注：静态方法中不能引用实例变量

```
public class ThisTest02{
    int i = 100; // 实例变量，这个i变量必须先new对象才能访问。
    static int k = 111;//静态变量

    //静态方法
    public static void main(String[] args){
        System.out.println(i);// 错误: 无法从静态上下文中引用非静态 变量 i，要访问的话必须先创建对象通过“引用.”来访问。
        System.out.println(k);//z
    }
}
```

- 注：在实例方法中，或者构造方法中，为了区分局部变量和实例变量，this. 是不能省略的。

```
public class ThisTest03{
    public static void main(String[] args){

        Student s = new Student();
        s.setNo(111);
        System.out.println("学号: " + s.getNo());
    }
}
class Student{
    //学号
```

```
    private int no;
    private String name;
    public Student(){

    }
    public Student(int no, String name){
        this.no = no;
        this.name = name;
    }
    public void setNo(int no){
        //no是局部变量,this.no 是指的实例变量。
        this.no = no; // this. 的作用是：区分局部变量和实例变量。
    }
}
```

- 新语法：this(实际参数列表);-----调用本类中的其他构造方法
- 通过一个构造方法1去调用构造方法2，可以做到代码复用。但需要注意的是：“构造方法1”和“构造方法2” 都是在同一个类当中。
- 对于this()的调用只能出现在构造方法的第一行

```
class Date{
    private int year;
    private int month;

    public Date(){
        //System.out.println(11);//错误：对this的调用必须是构造器中的第一个语句
        this(1970, 1); //相当于this.year = 1970;this.month = 1;
    }
    // 构造方法有参数
    public Date(int year, int month){
        this.year = year;
        this.month = month;
    }
```

8.3总结

- 所有的实例相关的都是先创建对象，通过“引用.”来访问。
- 所有的静态相关的都是直接采用“类名.”来访问。
- 静态变量、静态方法执行时编译器会自动在前面加"类名."；实例变量，实例方法会自动加"this."(前提是创建好对象)
- 只要负责调用的方法a和被调用的方法b在同一个类当中：this. 可以省略;类名. 可以省略
- 局部变量直接访问

九、继承(extends)

- 基本作用：子类继承父类，代码可以得到复用。
- 主要(重要)作用：因为有了继承关系，才有了后期的方法覆盖和多态机制。

9.1 相关特性

- B类继承A类，则称A类为超类(superclass)、父类、基类，B类则称为子类(subclass)、派生类、扩展类。
- java 中的继承只支持单继承，不支持多继承，C++中支持多继承，换句话说，java 中不允许这样写代码：class B extends A,C{ } 这是错误的。
- 虽然 java 中不支持多继承，但有的时候会产生间接继承的效果，例如：class C extends B，class B extends A，也就是说，C 直接继承 B，其实 C 还间接继承 A。
- java 中规定，子类继承父类，除构造方法不能继承之外，剩下都可以继承。但是私有的属性无法在子类中直接访问。(父类中private修饰的不能在子类中直接访问。可以通过间接的手段来访问。)
- java 中的类没有显示的继承任何类，则默认继承 Object类，Object类是java 语言提供的根类（老祖宗类），也就是说，一个对象与生俱来就有Object类型中所有的特征。
- 继承缺点：耦合度非常高

```
class people{
    private String name;
    public people(String name){
        this.name=name;
    }
    //封装
    public void setName(String name){
        this.name=name;
    }
    public String getName(){
        return name;
    }
    public void speak(){
        System.out.println(name+"...");
    }
}
class Chinese extends people{
    public void speak(){
        System.out.println(this.getName()+"中国话");
    }
}
```

9.2 System.out

- 在实际开发中凡是采用“is a”能描述的，都可以继承。

- System.out.println("Hello World!");-----System.out 中，out后面没有小括号，说明out是变量名。另外System是一个类名，直接使用类名System.out，说明out是一个静态变量。System.out 返回一个对象，然后采用“对象.”的方式访问println()方法。
- System.out.println(引用);当直接输出一个“引用”的时候，println()方法会先自动调用“引用.toString()”，然后输出toString()方法的执行结果。----toString()是Object类中的方法。
- toString()方法存在的作用就是：将java对象转换成字符串形式。大多数的java类toString()方法都是需要覆盖的。因为Object类中提供的toString()方法输出的是一个java对象的内存地址。

十、方法覆盖和多态

10.1、方法覆盖

- 子类继承父类之后，当继承过来的方法无法满足当前子类的业务需求时，子类有权利对这个方法进行重新编写，有必要进行“方法的覆盖”。
- 方法覆盖又叫做：方法重写（重新编写），英语单词叫做：Override、Overwrite，
- 当子类对父类继承过来的方法进行“方法覆盖”之后，子类对象调用该方法的时候，一定执行覆盖之后的方法

10.1.1 覆盖条件

- 前提：两个类必须要有继承关系
- 重写之后的方法和之前的方法具有相同的返回值类型、相同的方法名、相同的形式参数列表。
- 访问权限不能更低，可以更高。（这个先记住。）
- 重写之后的方法不能比之前的方法抛出更多的异常，可以更少。（这个先记住）

10.1.2注意事项

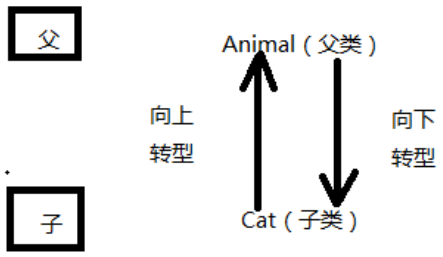
- 注意1：方法覆盖只是针对于方法，和属性无关。
- 注意2：私有方法不能被继承，所以无法覆盖。
- 注意3：构造方法不能被继承，所以 也不能被覆盖
- 注意4：方法覆盖只是针对于“实例方法”，“静态方法覆盖”没有意义（但不是不能覆盖）。

```
1  public class OverrideTest05{
2      public static void main(String[] args){
3          // 静态方法可以使用“引用.”来调用吗？可以，虽然使用“引用.”来调用，但是和对象无关。
4          Animal a = new Cat(); //多态
5          a.doSome();//Animal的doSome方法执行！  静态方法和对象无关。虽然使用“引用.”来调用。但是实际运行的时候还是：Animal.doSome()
```

```
    }
}
class Animal{
    // 父类的静态方法
    public static void doSome(){
        System.out.println("Animal的doSome方法执行！");
    }
}
class Cat extends Animal{
    //对静态方法重写
    public static void doSome(){
        System.out.println("Cat的doSome方法执行！");
    }
}
```

10.2 多态

- 前提是封装形成独立个体，独立体之间存在继承关系，从而产生多态机制。
- 多态是同一个行为具有多个不同表现形式或形态的能力。
- 向上转型(upcasting)：子--->父（自动类型转换）；向下转型 (downcasting)：父--->子（强制类型转换）。其中Animal a1=new Cat();是父类型引用指向了子类型对象，属于向上转型。
- 什么时候必须使用“向下转型”？-----当你需要访问的是子类对象中“特有”的方法，此时必须进行向下转型。例：Cat c = (Cat)a1
- 向上转型a1可以调用Cat()中覆盖后的方法。



无论向上还是向下转型，两种类型之间必须要有继承关系，这是首要条件。没有继承关系，就没有向上和向下转型。

- 父类型引用指向子类型对象。包括编译阶段和运行阶段。编译期：静态绑定。运行期：动态绑定。
- Animal a = new Cat();编译的时候编译器发现a的类型是Animal，所以编译器会去Animal类中找move()方法，找到了，绑定，编译通过。但是运行的时候和底层堆内存当中的实际对象有关，真正执行的时候会自动调用“堆内存中真实对象”的相关方法。

```
// 动物类：父类
```

```java
public class Animal{
    // 移动的方法
    public void move(){
        System.out.println("动物在移动！！！");
    }
}
// 猫类，子类
public class Cat extends Animal{
    // 对move方法进行重写
    public void move(){
        System.out.println("cat走猫步！");
    }
    // 这个行为是子类型对象特有的方法。
    public void catchMouse(){
        System.out.println("猫正在抓老鼠！！！！");
    }
}
```

```java
public class Test01{
    public static void main(String[] args){
        Animal a1 = new Animal();
        a1.move(); //动物在移动！！！

        Animal a5 = new Cat(); // 底层对象是一只猫。
        a5.catchMouse();   //错误：找不到符号。因为编译器只知道a5的类型是Animal，去Animal.class文件中找catchMouse()方法结果没有找到，所以静态绑定失败，编译报错。无法运行。（语法不合法。）

        Cat x = (Cat)a5;   //向下转型
        x.catchMouse(); //猫正在抓老鼠！！！！

        Animal a6 = new Bird(); //表面上a6是一个Animal，运行的时候实际上是一只鸟儿。
        Cat y = (Cat)a6;
        y.catchMouse();//错误,强制类型转换存在风险，应该用instanceof判断
        /*
            分析以上程序，编译报错还是运行报错？？？
                编译器检测到a6这个引用是Animal类型，
                而Animal和Cat之间存在继承关系，所以可以向下转型。
                编译没毛病。

                运行阶段，堆内存实际创建的对象是：Bird对象。
                在实际运行过程中，拿着Bird对象转换成Cat对象
                就不行了。因为Bird和Cat之间没有继承关系。

            运行是出现异常，这个异常和空指针异常一样非常重要，也非常经典：
            java.lang.ClassCastException：类型转换异常。

```

```
            java.lang.NullPointerException：空指针异常。这个也非常重要。
        */
    }
}
```

- 怎么避免ClassCastException异常的发生?

运算符：instanceof （运行阶段动态判断）

- 第一：instanceof可以在运行阶段动态判断引用指向的对象的类型。
- 第二：instanceof的语法：(引用 instanceof 类型)
- 第三：instanceof运算符的运算结果只能是：true/false
- 第四：c是一个引用，c变量保存了内存地址指向了堆中的对象。
  (c instanceof Cat)为true表示：c引用指向的堆内存中的java对象是一个Cat。
  (c instanceof Cat)为false表示：c引用指向的堆内存中的java对象不是一个Cat。

> 规范：任何时候，任何地点，对类型进行向下转型时，一定要使用instanceof 运算符进行判断。

10.3 多态在开发中作用

- OCP（开闭原则）：对扩展开放，对修改关闭（最好很少的修改现有程序）。
- 作用：降低程序的耦合度，提高程序的扩展力。
- 面向抽象编程，不建议面向具体编程。

```
public class Master{
            public void feed(Dog d){}
            public void feed(Cat c){}
        } //Master和Dog以及Cat的关系很紧密（耦合度高）。导致扩展力很差。
public class Master{
            public void feed(Pet pet){
                pet.eat();
            }
        } //Master和Dog以及Cat的关系就脱离了，Master关注的是Pet类。
```

- 在方法覆盖中，关于方法的返回值类型必须一致；学习了多态机制之后：
  对于返回值类型是基本数据类型来说，必须一致。
  对于返回值类型是引用数据类型来说，重写之后返回值类型可以变的更小（但意义不大，实际开发中没人这样写。）。

10.4 动态绑定机制（重点）

- 当调用对象方法时，该方法会和该对象的内存地址/运行类型绑定
- 当调用对象属性时，没有动态绑定机制，哪里声明，哪里使用
- 在分析代码时，一定要区分编译类型和运行类型

```

```

```java
public class Test02 {
    public static void main(String[] args) {
        A a = new B();
        System.out.println(a.sum());  //30   //a的编译类型是A，运行类型是B，在调用父类sum方法时，由于动态绑定是B，所以geti()方法会去调B中的，将B中的i返回
        System.out.println(a.sum1()); //20    //调用父类sum1()方法时，由于对象属性没有动态绑定机制，所以i的值是用父类A中的10
    }
}
class A{
    public int i=10;
    public int sum(){
        return geti()+10;
    }
    public int sum1(){
        return i+10;
    }
    public int geti(){
        return i;
    }
}
class B extends A{
    public int i=20;
    public int geti(){
        return i;
    }
}
```

十一、super

- super代表的是当前对象（this）的父类型特征。

```
this:
            this能出现在实例方法和构造方法中。
            this的语法是：“this.”、“this()”
            this不能使用在静态方法中。
            this. 大部分情况下是可以省略的。
            this.什么时候不能省略呢？ 在区分局部变量和实例变量的时候不能省略。
                public void setName(String name){
                    this.name = name;
                }
            this() 只能出现在构造方法第一行，通过当前的构造方法去调用“本类”中
            其它的构造方法，目的是：代码复用。
```

- super能出现在实例方法和构造方法中。

- super的语法是：“super.”、“super()”
- super不能使用在静态方法中。
- super. 大部分情况下是可以省略的。
- super. 什么时候不能省略？---如果父类和子类有同名属性，并且想要通过子类访问父类属性，则不能省略。
- super() 只能出现在构造方法第一行，通过当前的构造方法去调用“父类”中的构造方法，目的是：创建子类对象的时候，先初始化父类型特征。
- 结论:当一个构造方法第一行,既没有this()又没有super()的话，默认会有一个super();表示通过当前子类的构造方法调用父类的无参数构造方法。所以必须保证父类的无参数构造方法是存在的。
- super(实际参数列表)；----调用父类中有参构造方法

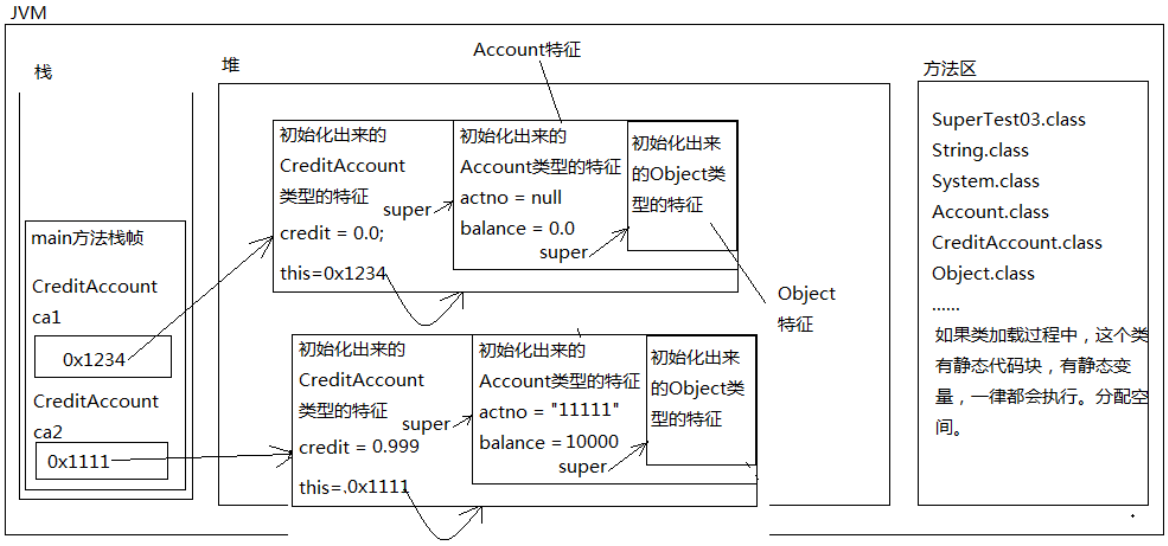
```
class A {
    public A(){
        //super(); // 这里也是默认有这一行代码的，调用的是Object的无参构造方法
        System.out.println("A类的无参数构造方法！");
    }
}
class B extends A{
    public B(){
        //super();   默认有，调用A类的无参构造方法
        System.out.println("B类的无参数构造方法！");
    }
}
public class SuperTest01{
    public static void main(String[] args){
        new B();     //输出结果：A类的无参数构造方法！
                                B类的无参数构造方法！
    }
}
```

- 结论：this()和super() 不能共存，它们都是只能出现在构造方法第一行。this(实际参数列表);---调用本类中的其他构造方法
- 无论是怎样折腾，父类的构造方法是一定会执行的。（百分百的。）
- 在java语言中无论是创建那个java对象，老祖宗Object类中的无参构造方法是必然执行的。（Object类的无参数构造方法是处于“栈顶部”,最后调用，最先结束）。
- 如果无参数构造方法丢失的话，可能会影响到“子类对象的构建”。

```
class Account{
    // 属性
    private String actno;
    private double balance;
    // 构造方法
    public Account(){}
```

```
7
8       public Account(String actno, double balance){
9           // super();
10          this.actno = actno;
11          this.balance = balance;
12      }
13  }
14  // 信用账户
15  class CreditAccount extends Account{
16      private double credit;
17      public class CreditAccount(){
18          //super();
19          //this.credit = 0.0;
20      }
21      public CreditAccount(String actno, double balance, double
    credit){
22          // 以下会报错，私有的属性，只能在本类中访问。
23          /*
24          this.actno = actno;
25          this.balance = balance;
26          */
27
28          // 通过子类的构造方法调用父类的构造方法。
29          super(actno, balance);
30          this.credit = credit;
31      }
32  }
33  public class SuperTest03{
34      public static void main(String[] args){
35
36          CreditAccount ca1 = new CreditAccount();
37          CreditAccount ca2 = new CreditAccount("1111", 10000.0,
    0.999);
38      }
39  }
```

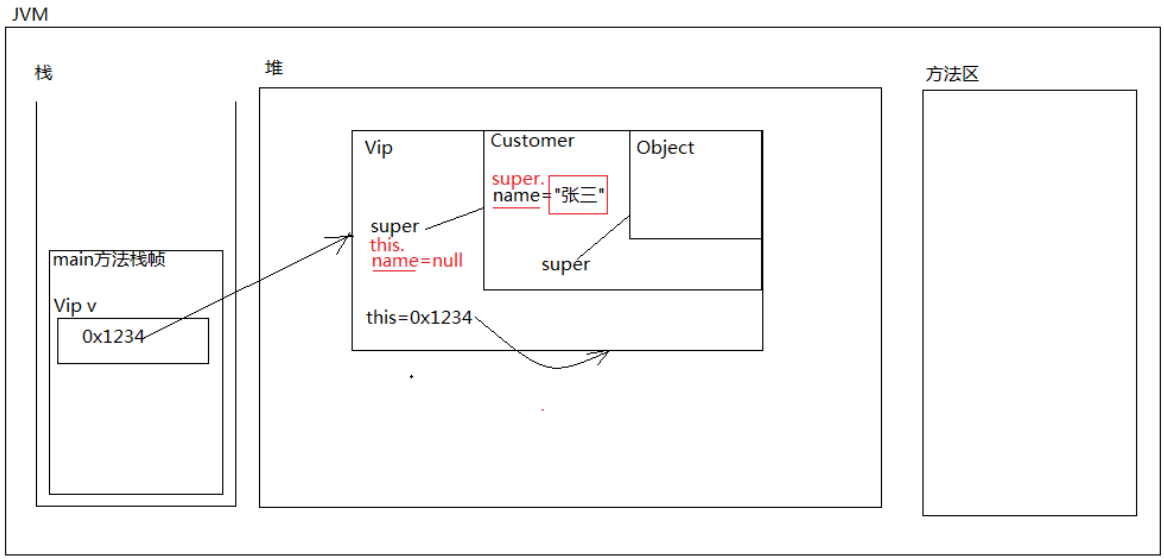
super代表的是“当前对象(this)”的“父类型特征”。　　此图不展示构造方法调用压栈的过程。



- 注意：super(实参)的作用是：初始化当前对象的父类型特征。并不是创建新对象，实际上对象只创建了1个。
  super关键字代表的就是“当前对象”的那部分父类型特征。
- super不是引用，并不指向独立的对象，不保存某个对象的内存地址。super其实是this的一部分。
- 父类中私有化的变量在子类中不能直接被访问，可以用super();来访问父类的有参构造方法，或者用this.get()和super.get()都能访问

```
public class SuperTest05{
    public static void main(String[] args){
        Vip v = new Vip("张三");
        v.shopping();
    }
}
class Customer {
    String name;
    public Customer(){}
    public Customer(String name){
        super();
        this.name = name;
    }
}
class Vip extends Customer{
    // 假设子类也有一个同名属性
    // java中允许在子类中出现和父类一样的同名变量/同名属性。
    String name; // 实例变量
    public Vip(){
    }
    public Vip(String name){
        super(name);
        // this.name = null;
    }
    public void shopping(){
        /*
```

```
27              java是怎么来区分子类和父类的同名属性的？
28                  this.name：当前对象的name属性
29                  super.name：当前对象的父类型特征中的name属性。
30          */
31          System.out.println(this.name + "正在购物!"); // null 正在购
    物
32          System.out.println(super.name + "正在购物!"); // 张三正在购物
33          System.out.println(name + "正在购物!"); //null 正在购物
34      }
35  }
```



- java中允许在子类中出现和父类一样的同名变量/同名属性。
- java是怎么来区分子类和父类的同名属性的？
  this.name：当前对象的name属性
  super.name：当前对象的父类型特征中的name属性。
- 在父和子中有同名的属性，或者说有相同的方法，如果此时想在子类中访问父中的数据，必须使用“super.”加以区分。
  super.属性名 【访问父类的属性】
  super.方法名(实参)【访问父类的方法】（重写前的方法）
  super(实参) 【调用父类的构造方法】

十二、idea

- 快速生成main方法：psvm
- 快速生成System.out.println()：sout
- 删除一行：ctrl + y
- 复制一行：ctrl + d
- 左侧窗口中的列表怎么展开？怎么关闭？----左箭头关闭。右箭头展开。上下箭头移动。
- ea中退出任何窗口，都可以使用esc键盘。（esc就是退出）
- 任何新增/新建/添加的快捷键是：alt + insert
- 窗口变大，变小：ctrl + shift + F12

- 切换java程序：从HelloWorld切换到User-----alt + 右箭头 或者 alt + 左箭头
- 切换窗口：alt + 标号----alt + 1（打开，关闭）
- 提示方法的参数：ctrl + p
- 注释：单行注释：ctrl + /　多行注释：ctrl + shift + /
- idea中怎么定位方法/属性/变量？-----光标停到某个单词的下面，这个单词可能是：方法名、变量名，停到单词下面之后，按ctrl键，出现下划线，点击跳转。
- 纠正错误的快捷键：alt+回车

十三、进阶-面向对象

13.1 final关键字

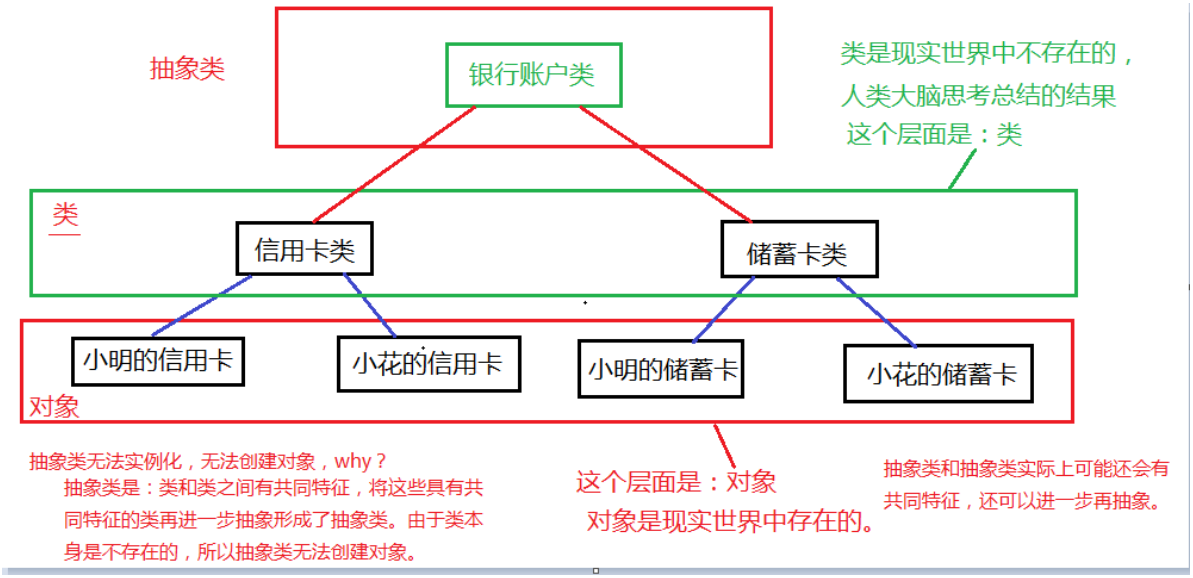
- final表示最终的，不可改变
- final修饰的类无法继承。
- final修饰的方法无法覆盖。但不影响调用
- final修饰的变量只能赋一次值，不能被修改
- final修饰的引用只能指向1个对象，则不能再重新指向其它对象，但该引用指向的对象内部的数据是可以修改的。
  并且在该方法执行过程中，该引用指向对象之后，该对象不会被垃圾回收器回收。直到当前方法结束，才会释放空间。
- final修饰的实例变量必须手动赋值（在构造方法中赋值也行），系统不负责赋值。
- final修饰的实例变量一般和static联合使用，称为常量。实际上常量和静态变量一样，都存储在方法区，都在类加载时初始化。
  public static final double PI = 3.1415926;常量一般是公共的：public

```
final class Test{
    public final void play(){
        final int i=2;
    }
}
```

13.2 抽象类

- 类和类之间具有共同特征，将这些共同特征提取出来，形成的就是抽象类。
  类本身是不存在的，所以抽象类无法创建对象,《无法实例化》。
- 抽象类也属于引用数据类型。
- ```
  语法：
      [修饰符列表] abstract class 类名{
      类体;
  }
  ```
- 抽象类是无法实例化的，无法创建对象的，所以抽象类是用来被子类继承的。
  ```

- 抽象类不能被final修饰，final和abstract不能联合使用，这两个关键字是对立的。
- 抽象类的子类可以是抽象类。也可以是非抽象类。
- 抽象类虽然无法实例化（不能new对象），但是抽象类有构造方法，这个构造方法是供子类使用的。



- 抽象类关联到一个概念：抽象方法。什么是抽象方法呢？抽象方法表示没有实现的方法，没有方法体的方法。例如：public abstract void doSome();
  抽象方法特点是：特点1：没有方法体，以分号结尾。
  特点2：前面修饰符列表中有abstract关键字。
- 抽象类中不一定有抽象方法，抽象方法必须出现在抽象类中。

```
abstract class Account{
    // 非抽象方法
    public void doOther(){

    }
    // 抽象方法
    public abstract void withdraw();
}
```

- 重要结论：一个非抽象的类继承抽象类，必须将抽象类中的抽象方法进行重写。如果子类中不复写该抽象方法，必须将此类再次声明为抽象类。

```
public class AbstractTest02{
    public static void main(String[] args){

        Animal a = new Bird();  // 向上转型。（自动类型转换）
        a.move();    //编译时静态绑定父类方法；运行时动态绑定子类中move方法，
    }
}
// 动物类（抽象类）
```

```java
abstract class Animal{
    // 抽象方法
    public abstract void move();
}
class Bird extends Animal{
    public void move(){
        System.out.println("鸟儿在飞翔！");
    }
}
```

```
面试题（判断题）：java语言中凡是没有方法体的方法都是抽象方法。
    不对，错误的。
    Object类中就有很多方法都没有方法体，都是以“;”结尾的，但他们
        都不是抽象方法，例如：
        public native int hashCode();
    这个方法底层调用了C++写的动态链接库程序。
    前面修饰符列表中没有：abstract。有一个native。表示调用JVM本地程序。
```

13.3 接口

- 接口通常提取的是行为动作。
- 1、接口也是一种“引用数据类型”。编译之后也是一个class字节码文件。
- 2、接口是完全抽象的。（抽象类是半抽象。）或者也可以说接口是特殊的抽象类。
- ```
  语法：[修饰符列表] interface 接口名{}
  ```
- 3.接口支持多继承，一个接口可以继承多个接口。

```java
interface A{

}
interface B{

}
interface C extends A,B{

}
```

- 5、接口中只包含两部分内容，常量+抽象方法。接口中没有其它内容了。只有以上两部分。接口中没有构造方法
- 6、接口中所有的元素都是public修饰的。（都是公开的。）
- 7、接口中的抽象方法定义时：public abstract修饰符可以省略。
- 8、接口中的方法不能有方法体(不允许有大括号)。
- 9、接口中的常量的public static final可以省略。接口中常量必须初始化
```

```java
interface MyMath{
    // 常量， public static final可以省略
    //public static final double PI = 3.1415926;
    double PI = 3.1415926;

    // 接口中随便写一个变量就是常量。常量：值不能发生改变的变量。
    int k = 100;

    // 抽象方法，public abstract可以省略
    //public abstract int sum(int a, int b);
    int sum(int a, int b);

    void doSome(){}  // 错误：接口抽象方法不可以有方法体

    // 相减的抽象方法
    int sub(int a, int b);
}
```

- 类和类之间叫做继承，类和接口之间叫做实现；仍然可以将"实现"看做“继承”。
- 继承使用extends关键字完成。实现使用implements关键字完成。
- 10.重要结论：当一个非抽象的类实现接口的话，必须将接口中所有的抽象方法全部实现（覆盖、重写）。（public不能省略）

```java
interface MyMath{
    double PI = 3.1415926;
    int sum(int a, int b);
    int sub(int a, int b);
}
class MyMathImpl implements MyMath {

    //错误：正在尝试分配更低的访问权限；以前为public,所以public不能省
    /*
    int sum(int a, int b){
        return a + b;
    }
    */

    // 重写/覆盖/实现 接口中的方法（通常叫做实现。）
    public int sum(int a, int b){
        return a + b;
    }
    public int sub(int a, int b){
        return a - b;
    }
}
```

- 接口不能new对象

```java
public class Test02{
    public static void main(String[] args){
        //错误: MyMath是抽象的; 无法实例化
        //new MyMath();

        // 父类型的引用指向子类型的对象
        MyMath mm = new MyMathImpl();
        // 调用接口里面的方法（面向接口编程。）
        int result1 = mm.sum(10, 20);
        System.out.println(result1);

        int result2 = mm.sub(20, 10);
        System.out.println(result2);
    }
}
```

- 11.一个类可以同时实现多个接口
- 这种机制弥补了java中的哪个缺陷?
  java中类和类只支持单继承。实际上单继承是为了简单而出现的，现实世界中存在多继承，java中的接口弥补了单继承带来的缺陷。

```java
interface A{
    void m1();
}

interface B{
    void m2();
}

interface C{
    void m3();
}

// 实现多个接口，其实就类似于多继承。
class D implements A,B,C{
    // 实现A接口的m1()
    public void m1(){
    }
    // 实现B接口中的m2()
    public void m2(){
        System.out.println("m2 ....");
    }
    // 实现接口C中的m3()
    public void m3(){

    }
}
```

- (非重点)接口A和接口B虽然没有继承关系，但是写代码的时候，可以互转。编译器没意见。但是运行时可能出现：ClassCastException
- 无论向上转型还是向下转型，两种类型之间必须要有继承关系，没有继承关系编译器会报错。（这句话不适用在接口方面。）最终实际上和之前还是一样，需要加：instanceof运算符进行判断。

```
public class Test03{
    public static void main(String[] args){
        // 多态该怎么用呢？
        // 都是父类型引用指向子类型对象
        A a = new D();
        //a.m2(); // 编译报错。A接口中没有m2()方法。
        B b = new D();
        C c = new D();

        // 这个编译没问题，运行也没问题。
        // 调用其他接口中的方法，你需要转型（接口转型。）
        B b2 = (B)a;
        b2.m2();

        // 直接向下转型为D可以吗？可以
        D d = (D)a;
        d.m2();

        M m = new E();
        // 经过测试：接口和接口之间在进行强制类型转换的时候，没有继承关系，也可以强转。
        // 但是一定要注意，运行时可能会出现ClassCastException异常。
        // 编译没问题，运行有问题。
        //K k = (K)m;
        if(m instanceof K){
            K k = (K)m;
        }
    }
}
interface K{
}

interface M{
}

class E implements M{
}
```

- 12.继承和实现都存在的话，代码应该怎么写？----extends 关键字在前。implements 关键字在后。
- 13、使用接口，写代码的时候，可以使用多态（父类型引用指向子类型对象）。

```java
class Animal{
}
interface Flyable{
    void fly();
}
// 接口通常提取的是行为动作。// Flyable是一个接口，是一对翅膀的接口，通过接口插到猫身上，让猫变的可以飞翔。
class Cat extends Animal implements Flyable{
    public void fly(){
        System.out.println("飞猫起飞，翱翔太空的一只猫，很神奇，我想做一只猫！！");
    }
}
public class Test04{
    public static void main(String[] args){
        Flyable f = new Cat(); //多态。
        f.fly();
    }
}
```

13.4 接口在开发中的作用--解耦合

- 多态：面向抽象编程，不要面向具体编程。降低程序的耦合度。提高程序的扩展力。
- 面向接口编程，可以降低程序的耦合度，提高程序的扩展力。符合OCP开发原则。接口的使用离不开多态机制。（接口+多态才可以达到降低耦合度。）
- 类型和类型之间的关系：is a（继承）、has a（关联）、like a（实现）
  Cat is a Animal，凡是能够满足is a的表示“继承关系”---A extends B

  凡是能够满足has a关系的表示“关联关系”，通常以属性的方式存在。---A{

  B b;}

  Cooker like a FoodMenu（厨师像一个菜单一样）凡是能够满足like a关系的表示“实现关系”，实现关系通常是：类实现接口。A implements B

```java
// MySelf has a Friend;
class MySelf{
    Friend f;
    public MySelf(Friend f){
        this.f = f;
    }
    public static void main(String[] args){
        // 创建朋友对象
        Friend f = new Friend(); //朋友对象有了
        // 创建对象的同时交朋友。
        MySelf m2 = new MySelf(f);
}
// “朋友”类
```

```
class Friend{

}
```

- 接口可以解耦合，解开的是调用者和实现者的耦合！！---调用者面向接口调用。实现者面向接口编写实现。

13.4.1接口和抽象类的区别

- 抽象类是半抽象的。接口是完全抽象的。
- 抽象类中有构造方法。接口中没有构造方法。
- 接口和接口之间支持多继承。类和类之间只能单继承。
- 一个类可以同时实现多个接口。一个抽象类只能继承一个类（单继承）。
- 接口中只允许出现常量和抽象方法。
- 接口一般都是对“行为”的抽象。

13.5 package和import

13.5.1 package

- 包机制的作用是为了方便程序的管理，不同功能的类分别存放在不同的包下(不同文件夹)
- package语句只允许出现在java源代码的第一行。
- 包名命名规范：公司域名倒序 + 项目名 + 模块名 + 功能名
- 用法：package+包名 例如package com.bjpowernode.javase.chapter17;
- 带着包名描述，表示完整类名。如果没有带包，描述的话，表示简类名。
- 类名是：com.bjpowernode.javase.chapter17.HelloWorld
  编译： javac -d . HelloWorld.java
  javac 负责编译的命令
  -d 带包编译
  . 代表编译之后生成的东西放到当前目录下（点代表当前目录）
  HelloWorld.java 被编译的java文件名
  运行： java com.bjpowernode.javase.chapter17.HelloWorld

```
package com.bjpowernode.javase.chapter17;
public class HelloWorld{
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

13.5.2 import

- import什么时候使用?
  A类中使用B类:A和B类都在同一个包下,不需要import。
  A和B类不在同一个包下，需要使用import。
- java.lang.*;这个包下的类不需要使用import导入。String类就在lang包下
- import语句只能出现在package语句之下，class声明语句之上。
- 使用：import 完整类名;
  import 包名.*;

```
package com;
//import com.bjpowernode.javase.chapter17.HelloWorld;
import com.bjpowernode.javase.chapter17.*;
```

- java.util.Scanner s = new java.util.Scanner(System.in); 中java.util就是Scanner类的包名。

```
import java.util.*;
public class Test03{
    public static void main(String[] args){
        Scanner s = new Scanner(System.in);
        String str = s.next();
        System.out.println("您输入的字符串是--->" + str);
    }
}
```

13.6 访问控制权限

- 有四个：private　私有-----只能在本类中访问
  public 公开-----在任何位置都可以访问
  protected 受保护-----只能在本类、同包、子类中访问
  默认-----只能在本类，以及同包下访问。

```
访问控制修饰符
              本类          同包          子类（不同包）    任意位置
          ------------------------------------------------------
  public     可以          可以          可以          可以
  protected  可以          可以          可以          不行
  默认        可以          可以          不行          不行
  private    可以          不行          不行          不行
```

- 修饰类和接口只能用public和默认
- 修饰方法和属性四个都可以

```java
public class C {
        private int i;
        public int age;
        protected  int weigh;
        String name;
}
```

十四、Object类

- API：应用程序编程接口。（Application Program Interface）
  整个JDK的类库就是一个javase的API。每一个API都会配置一套API帮助文档。
  SUN公司提前写好的这套类库就是API。（一般每一份API都对应一份API帮助文档。）

```
常用方法：
    protected Object clone()   // 负责对象克隆的。
    int hashCode()  // 获取对象哈希值的一个方法。
    boolean equals(Object obj)  // 判断两个对象是否相等
    String toString()  // 将对象转换成字符串形式
    protected void finalize()  // 垃圾回收器负责调用的方法
```

- toString() 方法：----返回：类名@对象的内存地址转换为十六进制的形式
- toString()方法的设计目的是：通过调用这个方法可以将一个“java对象”转换成“字符串表示形式”，建议所有的子类都去重写toString()方法。
- System.out.println(引用); 这里会自动调用“引用”的toString()方法。
- String类已经重写了toString方法，直接可以用

```
源码：
public String toString() {
    return this.getClass().getName() + "@" +
Integer.toHexString(hashCode());
            }
```

```java
public class Test01{
    public static void main(String[] args){
        MyTime t1 = new MyTime(1970, 1, 1);

        //MyTime类重写toString()方法之前
        //System.out.println(t1.toString()); // MyTime@28a418fc

        //MyTime类重写toString()方法之后
        System.out.println(t1.toString()); // 1970/1/1

        // 注意：输出引用的时候，会自动调用该引用的toString()方法。
        System.out.println(t1);  //1970/1/1
    }
```

```
}
class MyTime{
    int year;
    int month;
    int day;
    public MyTime(){

    }
    public MyTime(int year, int month, int day){
        this.year = year;
        this.month = month;
        this.day = day;
    }
    public String toString(){
        return this.year + "/" + this.month + "/" + this.day;
    }
}
```

- equals()方法----判断两个java对象是否相等
- 判断两个java对象是否相等，不能使用“==”，因为“==”比较的是两个对象的内存地址。我们应该判断两个java对象的内容是否相等。所以需要子类重写equals。
- 父类Object中的equals()方法是比较的内存地址。

```
源码：
    public boolean equals(Object obj) {
        return (this == obj);     //这是判断对象的内存地址
    }
```

- 1、String类已经重写了equals方法，比较两个字符串不能使用==，必须使用equals。equals是通用的。
- 编程习惯：当两个字符串比较时，“abc”.equals(s2)，采用这种方式可以避免空指针异常，"abc"算一个对象
- 2、String类已经重写了toString方法。
- 结论：java中基本数据类型比较是否相等，使用==
  java中所有的引用数据类型统一使用equals方法来判断是否相等。

```
public class Test03{
    public static void main(String[] args){

        // 大部分情况下，采用这样的方式创建字符串对象
        String s1 = "hello";
        String s2 = "abc";

        // 实际上String也是一个类。不属于基本数据类型。
        // 既然String是一个类，那么一定存在构造方法。
        String s3 = new String("Test1");
        String s4 = new String("Test1");
```

```
        // new两次，两个对象内存地址，s3保存的内存地址和s4保存的内存地址不同。
        // == 判断的是内存地址。不是内容。
        System.out.println(s3 == s4); // false

        // 比较两个字符串能不能使用双等号？
        // 不能，必须调用equals方法。String类已经重写equals方法了。
        System.out.println(s3.equals(s4)); // true

        // String类有没有重写toString方法呢？
        String x = new String("动力节点");
        // 如果String没有重写toString()方法，输出结果：java.lang.String@十六进制的地址
        // 经过测试：String类已经重写了toString()方法。
        System.out.println(x.toString()); //动力节点
        System.out.println(x); //动力节点
    }
}
```

在编写程序判断是，不要一直嵌套if...else语句，要采用“过关斩将”的方式，先将不满足的情况剔除，最后剩下满足的情况，再编写代码

```
重写equals方法：
public boolean equals(Object obj){
        // 用户名和用户名相同，住址和住址相同的时候，认定是同一个用户。
        if(obj == null || !(obj instanceof User)) return false;
        if(this == obj) return true;    //先将两个不满足的剔除

        User u = (User)obj;
        if(this.name.equals(u.name) && this.addr.equals(u.addr)){
            return true;
        }
        return false;
    }
```

- （非重点）finalize()方法：------垃圾回收器（GC）负责调用的方法
- 这个方法不需要程序员手动调用，JVM的垃圾回收器负责调用这个方法。finalize()只需要重写，重写完将来自动会有程序来调用。
- finalize()方法实际上是SUN公司为java程序员准备的一个时机，垃圾销毁时机。如果希望在对象销毁时机执行一段代码的话，这段代码要写到finalize()方法当中。

```
源码：
    protected void finalize() throws Throwable { }
```

- hashCode方法：public native int hashCode();
  这个方法不是抽象方法，带有native关键字，底层调用C++程序。
- 实际上就是一个java对象的内存地址，经过哈希算法，得出的一个值

十五、内部类

- 内部类：在类的内部又定义了一个新的类。被称为内部类。
- 静态内部类：类似于静态变量

  实例内部类：类似于实例变量
  局部内部类：类似于局部变量
- 匿名内部类是局部内部类的一种。

```
class Test01{

    // “静态内部类,Test01.Inner1是类名”
    static class Inner1{
        public static void m1(){         //静态方法
            System.out.println("123");
        }
        public void m2(){                //实例方法
            System.out.println("456");
        }
    }

    //没有static叫做实例内部类。
    class Inner2{
    }

    // 方法
    public void doSome(){
        // 局部内部类。
        class Inner3{
        }
    }

    //调用静态内部类中的方法
    public static void main(String[] args){
        Test01.Inner1.m1();     //123
        Test01.Inner1 t1= new Test01.Inner1();   //创建静态内部类对象
        t1.m2();    //456

    }
}
```

- 不采用匿名内部类，需要再定义一个类实现接口
- 采用匿名内部类，直接new接口后面跟{}，对接口进行实现

```java
class Test01{
    public static void main(String[] args){
        // 调用MyMath中的mySum方法。
        MyMath mm = new MyMath();
        mm.mySum(new ComputeImpl(), 100, 200);

        // 使用匿名内部类，表示这个ComputeImpl这个类没名字了。
        // 这里表面看上去好像是接口可以直接new了，实际上并不是接口可以new了。
        // 后面的{} 代表了对接口的实现。
        // 不建议使用匿名内部类，为什么？
        // 因为一个类没有名字，没有办法重复使用。另外代码太乱，可读性太差。
        mm.mySum(new Compute(){
            public int sum(int a, int b){
                return a + b;
            }
        }, 200, 300);
    }
}
// 负责计算的接口
interface Compute{
    int sum(int a, int b);
}

class ComputeImpl implements Compute{
    // 对方法的实现
    public int sum(int a, int b){
        return a + b;
    }
}
// 数学类
class MyMath{
    // 数学求和方法
    public void mySum(Compute c, int x, int y){
        int retValue = c.sum(x, y);
        System.out.println(x + "+" + y + "=" + retValue);
    }
}
```

十六、数组

- Java语言中的数组是一种引用数据类型。不属于基本数据类型。数组的父类是Object。
- 数组当中可以存储“基本数据类型”的数据，也可以存储“引用数据类型”的数据。
- 数组因为是引用类型，所以在堆内存当中。
- 数组中不能直接存储java对象，实际上存储的是对象的“引用（内存地址）”。
- 数组长度不可变

- length属性(java自带的)---用来获取数组中元素的个数。
- 数组中元素的类型统一。比如int类型数组只能存储int类型，Person类型数组只能存储Person类型（Person子类也能存放）。
- 数组中的元素内存地址连续，并且以首元素内存地址作为整个数组对象的内存地址。
- 数组下标从0开始，以1递增。最后一个元素的下标是：length - 1
- 优点：查询/查找/检索某个下标上的元素时效率极高。
  为什么检索效率高?
  第一：每一个元素的内存地址在空间存储上是连续的。
  第二：每一个元素类型相同，所以占用空间大小一样。
  第三：知道第一个元素内存地址，知道每一个元素占用空间的大小，又知道下标，所以可以计算出某个下标上元素的内存地址。直接通过内存地址定位元素，所以数组的检索效率是最高的。
- 缺点： 第一：数组上随机删除或者增加元素的时候，效率较低，因为随机增删元素会涉及到后面元素统一向前或者向后位移的操作。
  第二：数组不能存储大数据量。 因为很难在内存空间上找到一块特别大的连续的内存空间。

16.1 一维数组

```
1、定义语法格式：
    int[] array1;
    double[] array2;
    boolean[] array3;
    String[] array4;
    Object[] array5;
2、怎么初始化一个一维数组呢？
    包括两种方式：静态初始化一维数组，动态初始化一维数组。
静态初始化语法格式：
    int[] array = {100, 2100, 300, 55};  //int a[]={1,2}也行
动态初始化语法格式：
    int[] array = new int[5]; // 这里的5表示数组的元素个数。初始化一个5个长度的int类型数组，每个元素默认值0
    String[] names = new String[6]; // 初始化6个长度的String类型数组，每个元素默认值null。
```

- array[array.length-1]-------数组中最后一个元素
- array[arryy.length]-------ArrayIndexOutOfBoundsException（比较著名的异常。）

```
public class ArrayTest04 {
    public static void main(String[] args) {
        // 静态初始化一维数组
        int[] a = {1,2,3};
        printArray(a);

```

```
        // 没有这种语法。
        //printArray({1,2,3});
        // 如果直接传递一个静态数组的话，语法必须这样写。
        printArray(new int[]{1,2,3});

        // 动态初始化一维数组
        int[] a2 = new int[4];
        printArray(a2);
        printArray(new int[3]);//这个可以
    }

    // 为什么要使用静态方法？方便呀，不需要new对象啊。
    public static void printArray(int[] array){
        for (int i = 0; i < array.length; i++) {
            System.out.println(array[i]);
        }
    }
}
```

- main方法中传入的String[] args，默认长度为0，相当于JVM传了一个{}
- main方法上面的String[] args数组主要是用来接收用户输入参数的
- 用户可以在控制台上输入参数，这个参数自动会被转换为“String[] args”
  例如这样运行程序：java ArrayTest05 abc def xyz　JVM会自动将“abc def xyz”通过空格的方式进行分离，分离完成之后，自动放到“String[] args”数组当中.
- IDEA中在run-->edit configurations-->Program arguments，添加参数

16.1.1 数组扩容

- 数组长度一旦确定不可变，java中对数组的扩容是：先新建一个大容量的数组，然后将小容量数组中的数据一个一个拷贝到大数组当中。
- 数组扩容效率较低

```
System.arraycopy(5个参数:拷贝源，起点，拷贝目标，起点，拷贝长度);

int[] src = {1, 11, 22, 3, 4};    // 拷贝源（从这个数组中拷贝）

int[] dest = new int[20];      // 拷贝目标（拷贝到这个目标数组上）
System.arraycopy(src, 1, dest, 3, 2);
```

16.2 二维数组

- 二维数组其实是一个特殊的一维数组，特殊在这个一维数组当中的每一个元素是一个一维数组。
- 三维数组是一个特殊的二维数组，特殊在这个二维数组中每一个元素是一个一维数组。

```
二维数组静态初始化
        int[][] a = {{1,2,1},{2,3,4,5},{0,0,0,0}};
a.length  //3
int[] b = a[0]      //{1,2,1}
a[0][1]    //2
二维数组动态初始化：
        int[][] a = new int[3][4];
```

```
public static void printArray(int[][] array){
    // 遍历二维数组。
    for (int i = 0; i < array.length; i++) {
        for (int j = 0; j < array[i].length; j++) {
            System.out.print(array[i][j] + " ");
        }
        System.out.println();
    }
}

int[][] a = {{1,2,3,4},{4,5,6,76},{1,23,4}};
printArray(a);

// 没有这种语法
//printArray({{1,2,3,4},{4,5,6,76},{1,23,4}});

// 可以这样写。
printArray(new int[][]{{1,2,3,4},{4,5,6,76},{1,23,4}});
```

16.3 Arrays工具类

- 工具类中方法大部分是静态的，为了方便，不需要创建对象,直接用类名调用
- SUN公司已经为我们程序员写好了一个数组工具类。java.util.Arrays;

```
int[] arr = {3,6,4,5,12,1,2,32,5,5};
// 排序
Arrays.sort(arr);
// 二分法查找（建立在排序基础之上。）
int index = Arrays.binarySearch(arr, 5);
```

十七、算法

17.1 冒泡排序

- 每一次循环结束之后，都要找出最大的数据，放到参与比较的这堆数据的最右边。（冒出最大的那个气泡。）
- 拿着左边的数字和右边的数字比对，当左边 > 右边的时候，交换位置。
- 考虑数组下标越界问题

- 外层循环控制一共需要遍历几次，内层循环控制每次需要比较几次

```
public class MaoPaoPaiXu {
    public static void main(String[] args) {
        int[] arr={3,2,1,7,8,0};
        int m;
        /*for (int i = 0; i < arr.length-1; i++) {
            for(int j=0;j<arr.length-1-i;j++){
                if(arr[j] > arr[j+1]){
                    m=arr[j+1];
                    arr[j+1]=arr[j];
                    arr[j]=m;
                }
            }
        }*/
        for (int i = arr.length-1; i >0 ; i--) {
            for(int j=0;j<i;j++){
                if(arr[j] > arr[j+1]){
                    m=arr[j+1];
                    arr[j+1]=arr[j];
                    arr[j]=m;
                }
            }
        }
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i]);
        }
    }
}
```

17.2 选择排序

- 关键点：每一次从这堆“参与比较的数据当中”找出最小值，拿着这个最小值和“参与比较的这堆最前面的元素”交换位置。
- 选择排序比冒泡排序好在：每一次的交换位置都是有意义的。
- 选择排序比冒泡排序效率高，两者比较次数一样，但选择排序交换次数少
-

```
public class XuanZePaiXu {
    public static void main(String[] args) {
        int[] arr = {3,1,8,5,2,0};
        for (int i = 0; i < arr.length-1 ; i++) {
            // i正好是“参加比较的这堆数据中”最左边那个元素的下标，是起点下标。
            int min =i;  //第一个参与比较的,假设最小值下标是i
            for (int j =i+1;j<arr.length;j++){
                if(arr[min] > arr[j]){
                    min=j;
                }
```

```
            }
            if(min != i){
                // 表示存在更小的数据，需要交换位置
                int temp;
                temp = arr[min];
                arr[min] = arr[i];
                arr[i] = temp;
            }
        }
        // 排序之后遍历
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i]);
        }
    }
}
```

17.3 二分查找

- 基于排序的基础上，必须注意排序是从大到小排还是从小到大排
- 以下代码是基于数组从小到大排的

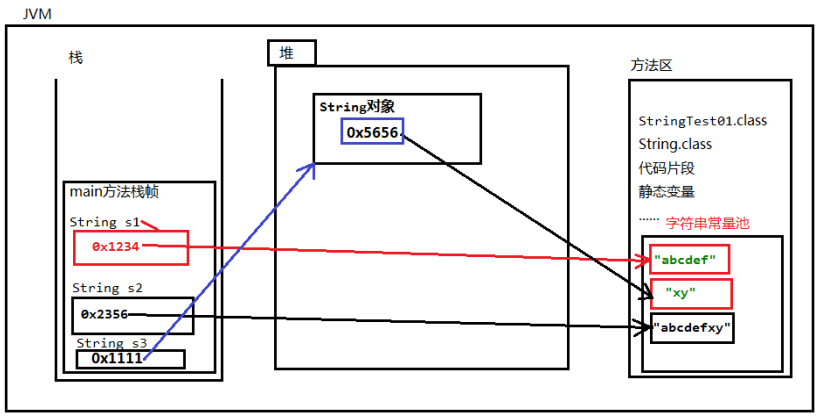
```
public class ErFenChaZhao {
    public static void main(String[] args) {
        int[] arr = {12, 34, 45, 78, 99, 101, 345};
        int index = ArrSearch(arr, 345);
        System.out.println(index == -1 ? "该元素不存在" : "该元素下标是" + index);
    }
    public static int ArrSearch(int[] arr, int ele) {
        // 开始下标
        int begin = 0;
        // 结束下标
        int end = arr.length - 1;
        // 开始元素的下标只要在结束元素下标的左边，就有机会继续循环。
        while (begin <= end) {
            // 中间元素下标
            int mid = (begin + end) / 2;
            if (arr[mid] == ele) {
                return mid;
            } else if (arr[mid] < ele) {
                // 目标在“中间”的右边,开始元素下标需要发生变化（开始元素的下标需要重新赋值）
                begin = mid + 1; // 一直增
            } else {
                // arr[mid] > dest 目标在“中间”的左边,修改结束元素的下标
                end = mid - 1; // 一直减
            }
        }
```

```
        return -1;
    }
}
```

十八、常用类

- String表示字符串类型,java中规定，双引号括起来的字符串，是不可变的
- 只要见到双引号括起来的字符串，例如："abc" "def"都是直接存储在“方法区”的“字符串常量池”当中的。
  因为字符串在实际的开发中使用太频繁。为了执行效率，所以把字符串放到了方法区的字符串常量池当中。
- 常量池中的数据是在编译期赋值的，也就是生成class文件时就把它放在常量池了
- 垃圾回收器不会释放常量的

```
public class StringTest01 {
    public static void main(String[] args) {
        // 这两行代码表示底层创建了3个字符串对象，都在字符串常量池当中。
        String s1 = "abcdef";
        String s2 = "abcdef" + "xy";

        // 分析：这是使用new的方式创建的字符串对象。这个代码中的"xy"是从哪里来的？
        // 凡是双引号括起来的都在字符串常量池中有一份。
        // new对象的时候一定在堆内存当中开辟空间。
        String s3 = new String("xy");

        // i变量中保存的是100这个值。
        int i = 100;
        // s变量中保存的是字符串对象的内存地址。
        // s引用中保存的不是"abc"，是0x1111，而0x1111是"abc"字符串对象在“字符串常量池”当中的内存地址。
        String s = "abc";
    }
}
```



- 上面直接创建字符串对象，s1和s2保存的内存地址直接指向方法区中常量池；new一个字符串对象，s3保存的 内存地址指向堆中的对象，堆中保存的内存地址指向方法区中 的常量池。
- 如果采用双引号引起来的字符串常量，首先会去常量池中查找，如果存在就不再分配，如果不存在就分配。
- 使用String时，不建议使用new关键字，因为使用new会创建两个对象（一个堆区，一个方法区）
- 堆区中是在运行期分配的，常量池是编译期分配的

```
String s1="abc";
String s2="abc";
System.out.println(s1==s2); //true,两个保存的是同一个内存地址
String s3=new String("abc");
System.out.println(s2==s3);  //false,s2保存的是方法区中常量池的内存地址，s3保存的是堆区内存地址
```

- String常用构造方法(6个)

```
 String s1 =  "hello world!";

byte[] bytes = {97, 98, 99}; // 97是a，98是b，99是c
String s2 = new String(bytes);
 System.out.println(s2.toString()); //abc,String类已经重写了toString()方法。
 System.out.println(s2); //abc

// String(字节数组,数组元素下标的起始位置,长度)
// 将byte数组中的一部分转换成字符串。
String s3 = new String(bytes, 1, 2);
System.out.println(s3); // bc

// 将char数组全部转换成字符串
char[] chars = {'我','是','中','国','人'};
String s4 = new String(chars);
System.out.println(s4);
// 将char数组的一部分转换成字符串
String s5 = new String(chars, 2, 3);
System.out.println(s5);

String s6 = new String("helloworld!");
System.out.println(s6); //helloworld!
```

18.1 String常用方法

- 通过一个字符串就可以去调用以下方法，大部分是实例方法。字符串就是一个对象。
- 下面方法，前面表示返回类型

```
1（掌握）.char charAt(int index)--返回指定索引的 char值
    char c = "中国人".charAt(1); // "中国人"是一个字符串String对象。只要是对象就能“点.”
    System.out.println(c); // 国

2（了解）.int compareTo(String anotherString)----比较两个字符串，可以看出谁大谁小。
    int result2 = "abcd".compareTo("abce");
    System.out.println(result2); //-1（小于0） 前小后大
    System.out.println("xyz".compareTo("yxz")); // -1  拿着字符串第一个字母和后面字符串的第一个字母比较。能分胜负就不再比较了。

3（掌握）.boolean contains(CharSequence s)---如果只有当此字符串包含指定的字符序列的字符串值，则返回真值。

 System.out.println("HelloWorld.java".contains(".java")); // true
System.out.println("http://www.baidu.com".contains("https://")); // false

4（掌握）. boolean endsWith(String suffix)---测试如果这个字符串以指定的后缀结束。
    System.out.println("test.txt".endsWith(".java")); // false
    System.out.println("test.txt".endsWith(".txt")); // true

5（掌握）.boolean equals(Object anObject)----比较是否相等，String类重写了equals方法，底层是两个Byte数组在比较
    System.out.println("abc".equals("abc")); // true

6（掌握）.boolean equalsIgnoreCase(String anotherString)----判断两个字符串是否相等，并且同时忽略大小写。
    System.out.println("ABc".equalsIgnoreCase("abC")); // true

 7（掌握）.byte[] getBytes()----将字符串对象转换成字节数组
     byte[] bytes = "abcdef".getBytes();

8（掌握）.int indexOf(String str)----判断某个子字符串在当前字符串中第一次出现处的索引（下标）。

 System.out.println("oraclejavac++.netc#phppythonjavaoraclec++".indexOf("java")); // 6

9（掌握）.boolean isEmpty()----判断某个字符串是否为“空字符串”。底层源代码调用的应该是字符串的length()方法。
    String s = "";
    System.out.println(s.isEmpty());//true
```

```
10（掌握）. int length()---判断字符串长度
    System.out.println("".length()); // 0

11（掌握）.int lastIndexOf(String str)----判断某个子字符串在当前字符串中最后一次出现的索引（下标）

 System.out.println("oraclejavac++javac#phpjavapython".lastIndexOf("java")); //22

12（掌握）. String replace(CharSequence target, CharSequence replacement)----替换，用后面的替换前面的，String的父接口就是：CharSequence
    String newString2 = "name=zhangsan&password=123&age=20".replace("=", ":");
    System.out.println(newString2); //name:zhangsan&password:123&age:20

13（掌握）.String[] split(String regex)----拆分字符串，返回一个String数组
    String[] ymd = "1980-10-11".split("-"); //"1980-10-11"以"-"分隔符进行拆分。

14（掌握）、boolean startsWith(String prefix)---判断某个字符串是否以某个子字符串开始。

 System.out.println("http://www.baidu.com".startsWith("http")); // true

15（掌握）、 String substring(int beginIndex) 参数是起始下标。----截取字符串

 System.out.println("http://www.baidu.com".substring(7)); //www.baidu.com

16（掌握）、String substring(int beginIndex, int endIndex)----截取（左闭右开），beginIndex起始位置（包括）endIndex结束位置（不包括）
    System.out.println("http://www.baidu.com".substring(7, 10)); //www

17(掌握)、char[] toCharArray()---- 将字符串转换成char数组
    char[] chars = "我是中国人".toCharArray();

18（掌握）、String toLowerCase()----转换为小写。
    System.out.println("ABCDefKXyz".toLowerCase());

19（掌握）、String toUpperCase();----转换为大写
    System.out.println("ABCDefKXyz".toUpperCase());
```

```
63
64  20（掌握）. String trim();----去除字符串前后空白
65      System.out.println("           hello      world
        ".trim());
66
67  21（掌握）. valueOf()-----String中只有一个方法是静态的，不需要
    new对象，作用：将“非字符串”转换成“字符串”
68      String s1 = String.valueOf(100);
69
70      String s2 = String.valueOf(new Customer());
71      System.out.println(s2);//valueOf()中参数是一个对象时，底层
    会自动调用该对象的toString()方法，由于Customer类没有重写
    toString()方法，会调用父类Object中方法，输出：类名@内存地址
```

- 注：判断数组长度和判断字符串长度不一样----判断数组长度是length属性，判断字符串长度是length()方法。
- System.out.println(引用);为什么会调用父类Object的toString()方法

  因为println方法底层调用String.valueOf()，而valueOf()中参数是一个对象时会调用对象的toString()方法。

  本质上System.out.println()这个方法在输出任何数据的时候都是先转换成字符串，再输出。

18.2 StringBuffer-字符串缓存区

- 字符串每一次拼接都会产生新字符串,这样会占用大量的方法区内存。造成内存空间的浪费。String s = "abc";s += "hello";---以上两行代码，就导致在方法区字符串常量池当中创建了3个对象：
- StringBuffer底层实际上是一个byte[]数组，在StringBuffer中放字符串实际上是放到byte数组中了。StringBuffer的初始容量是16;
- StringBuffer是可变对象，内存满了自动扩容（重新建一个大的数组）；而String是不可变的。

  因为String类源代码中，byte数组被final修饰了，只能赋一次值;
- 如何优化StringBuffer的性能?
  在创建StringBuffer的时候尽可能给定一个合适的初始化容量，可以提高程序的执行效率。
- 以后拼接字符串统一调用append()方法。

```
1  StringBuffer sb = new StringBuffer(100);
2  sb.append("hello");// 拼接字符串，以后拼接字符串统一调用 append()方法。
3  sb.append("world");
4  System.out.println(sb);
```

18.3 StringBuilder

- StringBuffer中的方法都有：synchronized关键字修饰。表示StringBuffer在多线程环境下运行是安全的，但速度慢。
- StringBuilder中的方法都没有：synchronized关键字修饰，表示StringBuilder在多线程环境下运行是不安全的，但速度快。

```
//使用StringBuilder也是可以完成字符串的拼接。
StringBuilder sb = new StringBuilder();
sb.append(100);
```

18.4 基础类型对应的8个包装类

- 当一个方法的参数是Object类型，基本数据类型是无法传递到方法中的，为了方便开发，定义了包装类。
- java中为8种基本数据类型又对应准备了8种包装类型。8种包装类属于引用数据类型，父类是Object。
- 所有包装类都是final的 ,所以不能创建其子类，包装类都是不可变对象。
- 这几个类中都重写了toString()方法

```
8种基本数据类型对应的包装类型名是什么？
    基本数据类型                包装类型
    -------------------------------------
    byte                    java.lang.Byte（父类Number）
    short                   java.lang.Short（父类Number）
    int                     java.lang.Integer（父类Number）
    long                    java.lang.Long（父类Number）
    float                   java.lang.Float（父类Number）
    double                  java.lang.Double（父类Number）
    boolean                 java.lang.Boolean（父类Object）
    char                    java.lang.Character（父类Object）
```

```
Number是一个抽象类，无法实例化对象。
    Number类中有这样的方法：
    byte byteValue() ---以 byte 形式返回指定的数值。
    abstract  double doubleValue()---以 double 形式返回指定的数值。
    abstract  float floatValue()---以 float 形式返回指定的数值。
    abstract  int intValue()--- int 形式返回指定的数值。
    abstract  long longValue()---以 long 形式返回指定的数值。
    short shortValue()---以 short 形式返回指定的数值。
    这些方法其实所有的数字包装类的子类都有，这些方法是负责拆箱的。
```

```java
// 123这个基本数据类型，进行构造方法的包装达到了：基本数据类型向引用数据类型的转换。
// 基本数据类型 -(转换为)->引用数据类型（装箱）
Integer i = new Integer(123);

// 将引用数据类型--(转换为)-> 基本数据类型(拆箱)
float f = i.floatValue();
System.out.println(f); //123.0
```

- Integer类的构造方法，有两个：
  Integer(int)
  Integer(String)
- 常量：Integer.MAX_VALUE---最大值　Integer.MIN_VALUE---最小值
- Integer构造方法中传的必须是数字，String类型的也算，否则会出异常java.lang.NumberFormatException

```java
// 将数字100转换成Integer包装类型（int --> Integer）
Integer x = new Integer(100);
// 将String类型的数字，转换成Integer包装类型。（String --> Integer）
Integer y = new Integer("123");
System.out.println(y);  //123

// 通过访问包装类的常量，来获取最大值和最小值
System.out.println("int的最大值：" + Integer.MAX_VALUE);
System.out.println("int的最小值：" + Integer.MIN_VALUE);
System.out.println("byte的最大值：" + Byte.MAX_VALUE);
System.out.println("byte的最小值：" + Byte.MIN_VALUE);
```

```java
Integer a = new Integer("中文");  //java.lang.NumberFormatException 数字格式化异常
// 编译的时候没问题，一切符合java语法，运行时会出问题
// 不是一个“数字”可以包装成Integer吗？不能。运行时出现异常。
```

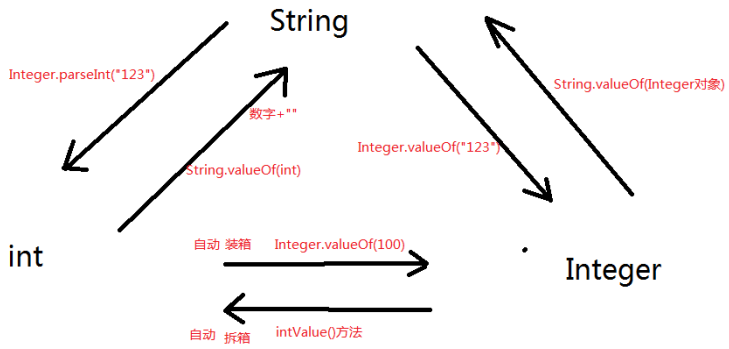
- 自动装箱和自动拆箱：自动将基础类型转换为对象，自动将对象转换为基础类型
- 有了自动拆箱之后，Number类中的方法就用不着了！

```java
// 基本数据类型 --(自动转换)--> 包装类型：自动装箱
Integer x = 900;
System.out.println(x+1);  //不会报错，会进行自动拆箱

// x是包装类型，y是基本数据类型
// 包装类型 --(自动转换)--> 基本数据类型：自动拆箱
int y = x;
System.out.println(y);
```

- Integer类常用方法：intValue();----Integer类型转换为int类型（手动拆箱）
- 重要方法：static int parseInt(String s)------静态方法，传参String，返回int

- String--->int 方法一、 Integer.parseInt() 方法二、 Integer.valueof().intValue()
- int--->String 方法一、String.valueof() 方法二、字符串拼接 方法三、 Integer.toString()

```
//网页上文本框中输入的100实际上是"100"字符串。后台数据库中要求存储100数字，此时java程序需要将"100"转换成100数字。
    int retValue = Integer.parseInt("123"); // String转换-> int

    int retValue = Integer.parseInt("中文"); // 异常 NumberFormatException

// 照葫芦画瓢
    double retValue2 = Double.parseDouble("3.14");
    System.out.println(retValue2 + 1); //4.140000000000001（精度问题）

    float retValue3 = Float.parseFloat("1.0");
    System.out.println(retValue3 + 1); //2.0

        // int --> String
        String s2 = i1 + ""; // "100"字符串
        System.out.println(s2 + 1); // "1001"

        // int --> Integer
        // 自动装箱
        Integer x = 1000;

        // Integer --> int
        // 自动拆箱
        int y = x;

        // String --> Integer
        Integer k = Integer.valueOf("123");

        // Integer --> String
        String e = String.valueOf(k);
```



- 非重点（了解）

```
        // static String toBinaryString(int i)----静态的：将十进制转换成二进制字符串。
        String binaryString = Integer.toBinaryString(3);
        System.out.println(binaryString); //"11" 二进制字符串

        // static String toHexString(int i)----静态的：将十进制转换成十六进制字符串。
        String hexString = Integer.toHexString(16);
        System.out.println(hexString); // "10"

        //static String toOctalString(int i)----静态的：将十进制转换成八进制字符串。
        String octalString = Integer.toOctalString(8);
        System.out.println(octalString); // "10"

        System.out.println(new Object());
//java.lang.Object@6e8cf4c6

        // valueOf方法:static Integer valueOf(int i)----静态的：int->Integer
        Integer i1 = Integer.valueOf(100);
        System.out.println(i1);

        // static Integer valueOf(String s)----静态的：String-->Integer
        Integer i2 = Integer.valueOf("100");
        System.out.println(i2);
```

18.5 日期Date

- java.util.Date类(使用时需要导包)
- java.text.SimpleDateFormat类---专门负责日期格式化的,有format()、parse()方法

  yyyy 年(年是4位)

MM 月（月是2位）
dd 日
HH 时
mm 分
ss 秒
SSS 毫秒（毫秒3位，最高999。1000毫秒代表1秒）

- 注意：在日期格式中，除了y M d H m s S这些字符不能随便写之外，剩下的符号格式自己随意组织。
- Date--->String　　format(Date对象);返回一个String类型
  String--->Date　　parse(String);返回一个Date类型

```
// 获取系统当前时间（精确到毫秒的系统当前时间）
    Date nowTime = new Date();

// java.util.Date类的toString()方法已经被重写了。
// 输出的应该不是一个对象的内存地址，应该是一个日期字符串。
    System.out.println(nowTime); //Thu Mar 05 10:51:06 CST 2020

//Date --转换成具有一定格式的日期字符串-->String
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");
    //SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");

    String nowTimeStr = sdf.format(nowTime);
    System.out.println(nowTimeStr);

// 假设现在有一个日期字符串String，怎么转换成Date类型？String --> Date
    String time = "2008-08-08 08:08:08 888";

// 注意：字符串的日期格式和SimpleDateFormat对象指定的日期格式要一致。不然会出现异常：java.text.ParseException
    SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");
    Date dateTime = sdf2.parse(time);
    System.out.println(dateTime); //Fri Aug 08 08:08:08 CST 2008
```

- System.currentTimeMillis()----获取自1970年1月1日 00:00:00 000到当前系统时间的总毫秒数。
- 可以统计一个方法执行所耗费的时长

```
long nowTimeMillis = System.currentTimeMillis();
System.out.println(nowTimeMillis); //1583377912981，在不断变化

// 统计一个方法耗时，在调用目标方法之前记录一个毫秒数
long begin = System.currentTimeMillis();
print();
// 在执行完目标方法之后记录一个毫秒数
```

```
long end = System.currentTimeMillis();
System.out.println("耗费时长"+(end - begin)+"毫秒");
}
// 需求：统计一个方法执行所耗费的时长
public static void print(){
    for(int i = 0; i < 1000000000; i++){
        System.out.println("i = " + i);}}
```

- Date()----无参构造方法，获取系统当前时间
- Date(毫秒)----有参构造方法，指1970-1-1 00：00：00 000作为起点，经过的毫秒数

```
Date time = new Date(1); // 注意：参数是一个毫秒
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");
String strTime = sdf.format(time);
// 北京是东8区。差8个小时。
System.out.println(strTime); // 1970-01-01 08:00:00 001

// 获取昨天的此时的时间。
Date time2 = new Date(System.currentTimeMillis() - 1000 * 60 * 60 * 24);
String strTime2 = sdf.format(time2);
System.out.println(strTime2);
```

18.6 数字类

- java.text.DecimalFormat专门负责数字格式化的。
- DecimalFormat df = new DecimalFormat("数字格式");
  调用format()方法，返回一个字符串
- 数字格式： # 代表任意数字 ，代表千分位 . 代表小数点 0 代表不够时补0

```
//###,###.##---表示：加入千分位，保留2个小数。
    DecimalFormat df = new DecimalFormat("###,###.##");
    String s = df.format(1234.561232);
    System.out.println(s); // "1,234.56"

    DecimalFormat df2 = new DecimalFormat("###,###.0000"); //保留4个小数位，不够补上0
    String s2 = df2.format(1234.56);
    System.out.println(s2); // "1,234.5600"
```

- java.math.BigDecimal-----BigDecimal 属于大数据，精度极高。不属于基本数据类型，属于java对象（引用数据类型）这是SUN提供的一个类。专门用在财务软件当中。

- ```java
  // 这个100不是普通的100，是精度极高的100
  BigDecimal v1 = new BigDecimal(100);
   BigDecimal v2 = new BigDecimal(200);
  BigDecimal v3 = v1.add(v2); // 调用方法求和。
  BigDecimal v4 = v2.divide(v1);//除
  ```

18.7 随机数

- java.util.Random

```java
// 创建随机数对象
Random random = new Random();

// 随机产生一个int类型取值范围内的数字。
int num1 = random.nextInt();

// 产生[0~100]之间的随机数。不能产生101。
// nextInt翻译为：下一个int类型的数据是101，表示只能取到100.
int num2 = random.nextInt(101); //不包括101
System.out.println(num2);
```

18.8枚举类

- 枚举是一种引用数据类型,编译之后也是生成class文件,枚举中的每一个值可以看做是常量。
- 结果只有两种情况的，建议使用布尔类型。结果超过两种并且还可以一枚一枚列举出来的，建议使用枚举类型。
  例如：颜色、四季、星期等都可以使用枚举类型。

```
语法：
    enum 枚举类型名{
        枚举值1,枚举值2
    }
```

```java
public enum Season {
    SPRING, SUMMER, AUTUMN, WINTER   //春夏秋冬
}
```

18.9 System类

- java.lang 包下
```

```
    System.out 【out是System类的静态变量。】
    System.out.println() 【System.out返回一个PrintStream类，println()方法不是System类的，是PrintStream类的方法。】
    System.gc() 建议启动垃圾回收器
    System.currentTimeMillis() 获取自1970年1月1日到系统当前时间的总毫秒数。
    System.exit(0) 退出JVM。
```

十九、异常

- 程序在运行过程中出现的错误就是异常，作用：增强程序的健壮性
- 异常信息由JVM打印的

```
空指针异常：NullPointerException
类型转换异常：ClassCastException
数组下标越界异常：ArrayIndexOutOfBoundsException
数字格式化异常：NumberFormatException
          java.text.ParseException
迭代器结构改变，但未重新获取：
java.util.ConcurrentModificationException
TreeSet中存放对象未实现Comparable接口：
 java.lang.ClassCastException:class
com.bjpowernode.javase.collection.Person cannot be cast to class
java.lang.Comparable
对象没有实现Serializable接口，无法进行序列化
 java.io.NotSerializableException:
```

19.1 异常的存在

- java.lang.Exception
- 异常在java中以类的形式存在，每一个异常类都可以创建异常对象。
- 在程序执行过程中出现异常JVM都会new一个异常对象出来，然后将异常对象抛出，打印输出信息到控制台。

```
// 通过“异常类”实例化“异常对象”
    NumberFormatException nfe = new NumberFormatException("数字格式化异常！");
    System.out.println(nfe);    ////
java.lang.NumberFormatException: 数字格式化异常！
```

19.2 异常的处理机制

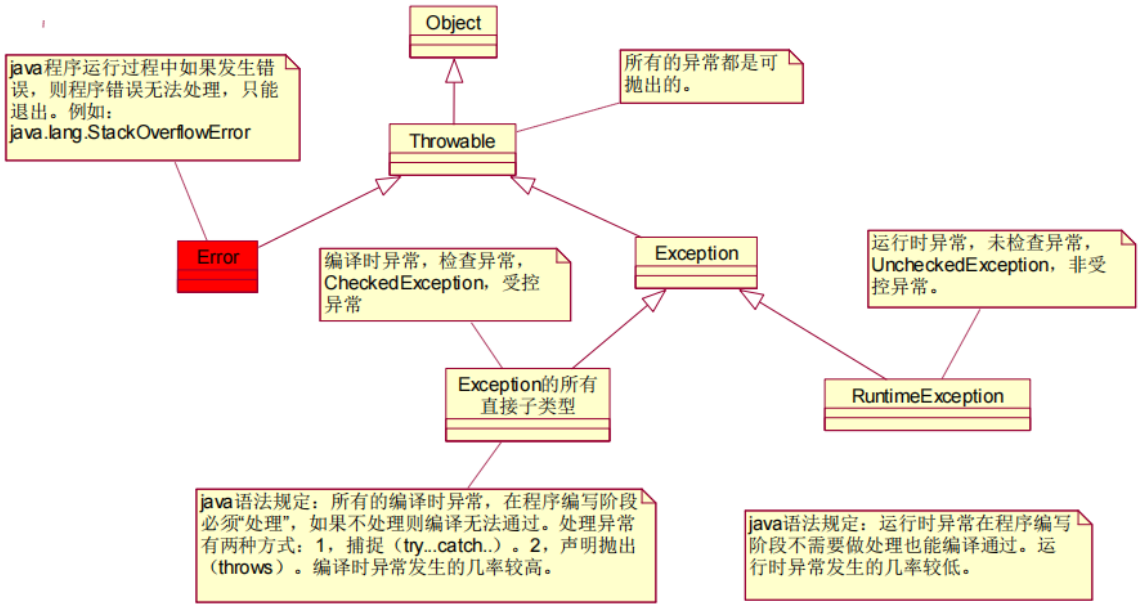
- 异常的继承结构：
  Object
  Object下有Throwable（可抛出的）
  Throwable下有两个分支：Error（不可处理，直接退出JVM）和Exception（可

处理的)

Exception下有两个分支：

Exception的直接子类：编译时异常（要求程序员在编写程序阶段必须预先对这些异常进行处理，如果不处理编译器报错，因此得名编译时异常。）。

RuntimeException：运行时异常。（在编写程序阶段程序员可以预先处理，也可以不管，都行。）



- 所有异常都是在运行阶段发生的。因为只有程序运行阶段才可以new对象。异常的发生就是new异常对象。编译时异常和运行时异常，都是发生在运行阶段，编译阶段，异常是不会发生的。
- 编译时异常因为什么而得名？---因为编译时异常必须在编译(编写)阶段预先处理，如果不处理编译器报错，因此得名。
- 处理方式：第一种方式：在方法声明的位置上，使用throws关键字，抛给上一级。

  第二种方式：使用try..catch语句进行异常的捕捉。
- 如果我选择了上抛，抛给了我的调用者，调用者处理这个异常同样有两种处理方式。
- Java中异常发生之后如果一直上抛，最终抛给了main方法，main方法继续向上抛，抛给了调用者JVM，JVM知道这个异常发生，只有一个结果。终止java程序的执行。

```
try{
    代码;
}catch(异常名 变量e){    //e引用保存的内存地址是new出来异常对象的内存地址，以下代码可以使用引用e
    代码;
}catch(异常名 变量e){

}
            //catch后面的小括号中的类型可以是具体的异常类型，也可以是该异常类型的父类型。
            //catch可以写多个。建议catch的时候，精确的一个一个处理。这样有利于程序的调试。
            //catch写多个的时候，从上到下，异常必须遵守从小到大。
```

```
//main方法中的异常建议使用try..catch进行捕捉。main就不要继续上抛了。
public static void main(String[] args) {
        try {
            System.out.println(100 / 0); // 这个异常是运行时异常，编写程序时可以处理，也可以不处理。
            m1();
            // 以上代码出现异常，直接进入catch语句块中执行。
            System.out.println("hello world!");
        } catch (FileNotFoundException e){
            System.out.println("文件不存在");
            System.out.println(e);
//java.io.FileNotFoundException: D:\course\01-课\学习方法.txt (系统找不到指定的路径。)
        }
        System.out.println("main over"); //// try..catch把异常抓住之后，这里的代码会继续执行。
    }

    private static void m1() throws FileNotFoundException {
        System.out.println("m1 begin");
        m2();
        // 以上代码出异常，这里是无法执行的。
        System.out.println("m1 over");
    }

    private static void m2() throws FileNotFoundException {
        System.out.println("m2 begin");
        new FileInputStream("D:\\course\\01-课\\学习方法.txt");  //出现异常

        // 以上如果出现异常，这里是无法执行的!
        System.out.println("m2 over");
    }
```

- 注意：只要异常没有捕捉，采用上报的方式，出现异常后的代码不会执行。另外，try语句块中的某一行出现异常，该行后面的代码不会执行。 try..catch捕捉异常之后，后续代码可以执行。
- 标准：如果希望调用者来处理，选择throws上报。其它情况使用捕捉的方式。

```
// JDK8的新特性！ catch后异常可以用|符号连接
    try {
        FileInputStream fis = new
FileInputStream("D:\\document\\JavaSE进阶讲义\\JavaSE进阶-01-面向对
象.pdf");
    } catch(FileNotFoundException | ArithmeticException |
NullPointerException e) {
        System.out.println("文件不存在？数学异常？空指针异常？都有可
能！");
    }
```

19.3 异常的两个方法

- 获取异常简单的描述信息：String msg = exception.getMessage();
  打印异常追踪的堆栈信息：exception.printStackTrace();
- 异常信息追踪信息，从上往下一行一行看。但是需要注意的是：SUN写的代码就不用看了(看包名就知道是自己的还是SUN的。)。主要的问题是出现在自己编写的代码上。

```
NullPointerException e = new NullPointerException("空指针异常
fdsafdsafdsafds");
String msg = e.getMessage();
System.out.println(msg);   //空指针异常fdsafdsafdsafds
e.printStackTrace();

public static void main(String[] args) {
        try {
            m1();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
}
```

19.4 finally

```
try{
    代码;
}catch(异常名 变量e){
    代码;
}finally{
    代码;
}                   //在finally子句中的代码是最后执行的，并且是一定会执行的，即使try语句块中的代码出现了异常。
                    //finally子句必须和try一起出现，不能单独编写。
```

- finally语句通常使用在哪些情况下呢?
  通常在finally语句块中完成资源的释放/关闭。因为finally中的代码比较有保障。
- try不能单独使用，try和finally可以连用
- 放在finally语句块中的代码是一定会执行的【再次强调！！！】(除非退出JVM)

```
try {
    System.out.println("try...");
    return;
} finally {
    System.out.println("finally...");
}
/*上面代码的执行顺序：
            先执行try...
            再执行finally...
            最后执行 return （return语句只要执行方法必然结束。）*/
 try {
     System.out.println("try...");
     // 退出JVM
     System.exit(0); // 退出JVM之后，finally语句中的代码就不执行了！
 } finally {
     System.out.println("finally...");
 }
```

19.5 自定义异常

- 第一步：编写一个类继承Exception或者RuntimeException.
- 第二步：提供两个构造方法，一个无参数的，一个带有String参数的，调用父类构造方法super(s)。

```
public class MyException extends Exception{ // 编译时异常
    public MyException(){

    }
    public MyException(String s){
        super(s);
    }
}
```

```
9
10 public class MyException extends RuntimeException{ // 运行时异常
11     public MyException(){
12
13     }
14     public MyException(String s){
15         super(s);
16     }
17 }
```

19.6 异常在开发中的作用--throw

- throws是在方法声明位置上使用，表示上报异常信息给调用者；throw是手动抛出异常
- 自己定义一个异常，然后手动抛出 throw new 异常
- 出现异常上报之后，后续代码不会执行，(代替了return)

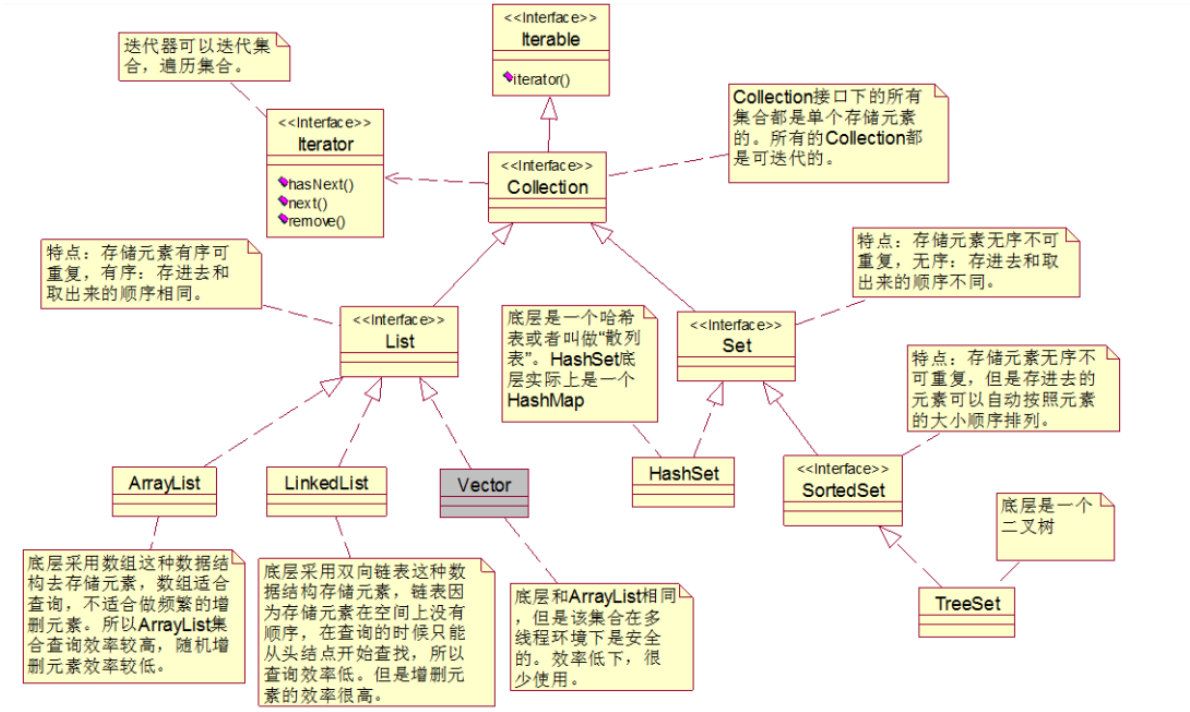
```
1  //自定义异常
2  public class MyStackOperationException extends Exception{   // 编译时异常！
3      public MyStackOperationException(){}
4
5      public MyStackOperationException(String s){
6          super(s);
7      }
8  }
9  //模拟栈数据结构
10 public class MyStack {
11     private Object[] elements;
12     private int index;
13     public MyStack() {
14         this.elements = new Object[3];
15         this.index = -1;
16     }
17     // 压栈
18     public void push(Object obj) throws MyStackOperationException{
19         if(index >= elements.length - 1){
20             // 创建异常对象
21             //MyStackOperationException e = new MyStackOperationException("压栈失败，栈已满！");
22             //throw e; // 手动将异常抛出去！
23             // 合并（手动抛出异常！）
24             throw new MyStackOperationException("压栈失败，栈已满！");
25         }
26         index++;
27         elements[index] = obj;
```

```
28          System.out.println("压栈" + obj + "元素成功，栈帧指向" +
index);
29      }
30  }
31  //测试程序
32  public class ExceptionTest16 {
33      public static void main(String[] args) {
34          // 创建栈对象
35          MyStack stack = new MyStack();
36          // 压栈
37          try {
38              stack.push(new Object());
39              stack.push(new Object());
40              stack.push(new Object());
41              // 这里栈满了
42              stack.push(new Object());
43          } catch (MyStackOperationException e) {
44              // 输出异常的简单信息。
45              System.out.println(e.getMessage());    //压栈失败，栈已
满！
46          }
```

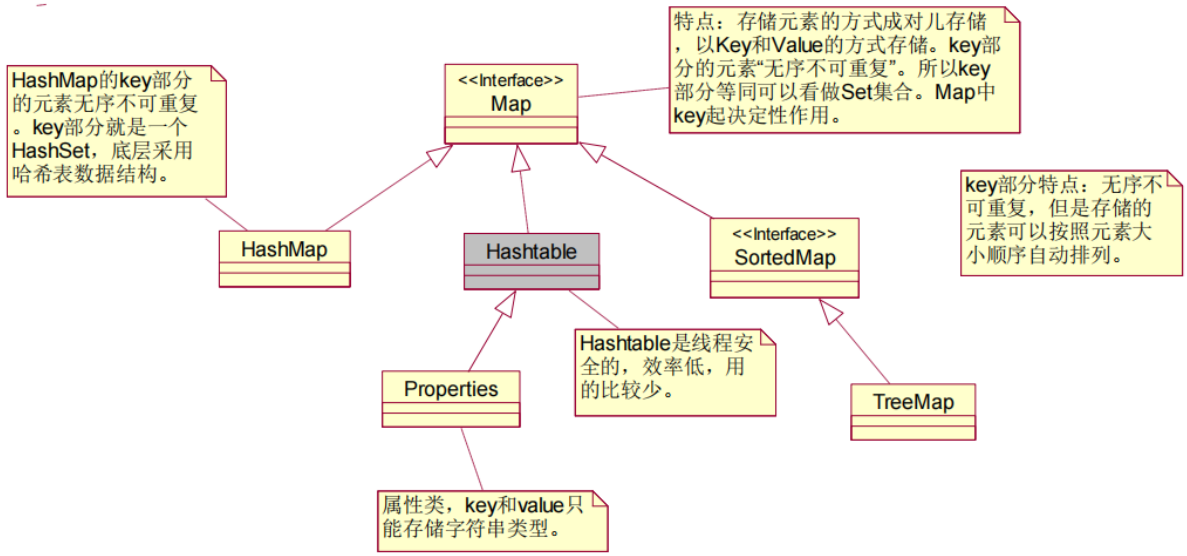
- 重写之后的方法不能比重写之前的方法抛出更多（更宽泛）的异常，可以更少。

二十、集合

- 集合实际上就是一个容器，可以来容纳其它类型的数据。
- 集合不能直接存储基本数据类型，也不能直接存储java对象，集合当中存储的都是java对象的内存地址。（或者说集合中存储的是引用。）
- 集合也是一个对象，也有内存地址
- 每一个不同的集合，底层会对应不同的数据结构。所有的集合类和集合接口都在java.util包下。



- 上图实线空心箭头表示继承，is a；虚线空心箭头表示实现，like a ;单箭头表示关联，has a
- list集合都有下标，从0开始，以1递增；set集合没有下标
- HashSet底层是HashMap，放到HashSet集合中的元素等同于放到了HashMap集合的key部分。
- TreeSet底层实际上是TreeMap,TreeMap底层数据结构是一个二叉树。TreeMap集合中的key可以自动按照大小排序。
- Map集合的key，就是一个set集合。在set集合中放数据，实际上放到了Map集合的key部分。



- Map中key和value都是存储java对象的内存地址

20.1 集合类型

- List:有序集合，可以放重复数据
- Set:无序集合，不允许放重复的数据
- Map：是一个无序集合，集合中以键值对的方式存储，键对象不可重复，值对象可以重复

20.2 Collection常用方法

- Collection中没有使用“泛型”之前，Collection中可以存储Object的所有子类型,使用了“泛型”之后，Collection中只能存储某个具体的类型。注：集合不能直接存储基本数据类型，基本数据类型通过自动装箱来存储对象
- boolean add(Object e) 向集合末尾添加元素(自动装箱)
- int size() 获取集合中元素的个数
- void clear() 清空集合
- boolean contains(Object o) 判断当前集合中是否包含元素o，包含返回true，不包含返回false
- boolean remove(Object o) 删除集合中的某个元素。
- boolean isEmpty() 判断该集合中元素的个数是否为0
- Object[] toArray() 调用这个方法可以把集合转换成数组。【作为了解，使用不多。】----数组中存储的都是引用

```
    Collection c = new ArrayList();  //多态
// 测试Collection接口中的常用方法
    c.add(1200); // 自动装箱(java5的新特性。),实际上是放进去了一个对象的内存地址。Integer x = new Integer(1200);
    c.add(new Object());
    c.add(true); // 自动装箱
    c.add("hello"); // "hello"对象的内存地址放到了集合当中。

    // 获取集合中元素的个数
    System.out.println("集合中元素个数是：" + c.size()); // 4

    // 清空集合
    //c.clear();

    // 判断集合中是否包含"hello"
    boolean flag = c.contains("hello");
    System.out.println(flag); // true

    // 删除集合中某个元素
    c.remove(1200);
    System.out.println("集合中元素个数是：" + c.size()); // 3

    // 判断集合是否为空（集合中是否存在元素）
    System.out.println(c.isEmpty()); // false

    // 转换成数组（了解，使用不多。）
    Object[] objs = c.toArray();
```

```
    for(int i = 0; i < objs.length; i++){
        // 遍历数组
        Object o = objs[i];
        System.out.println(o);    //会调用toString()方法
```

20.2.1 迭代器Iterator

- 以下讲解的遍历方式/迭代方式，是所有Collection通用的一种方式。在Map集合中不能用。在所有的Collection以及子类中使用。
- 首先通过集合对象调用Iterator()方法，返回一个迭代器对象Iterator it = c.iterator();
- boolean m=it.hasNext();----true,表示还有元素可以迭代；false表示没有元素
  Object obj=it.next();-----让迭代器前进一位，并将指向元素返回。不管原先存储的是什么，统一返回Object类型（除非用泛型）
  it.remove();-----删除的一定是迭代器指向的当前元素，迭代器删除时会自动更新迭代器。

```
// 创建集合对象
    Collection c = new ArrayList(); // 后面的集合无所谓，主要是看前面的Collection接口
    // 添加元素
    c.add("abc");
    c.add("def");
    c.add(100);
    c.add(new Object());

    // 对集合Collection进行遍历/迭代
    // 第一步：获取集合对象的迭代器对象Iterator
    Iterator it = c.iterator();
    // 第二步：通过以上获取的迭代器对象开始迭代/遍历集合。
    while(it.hasNext()){
        Object obj = it.next();
        System.out.println(obj);
    }
```

- 重点：当集合的结构发生改变时，迭代器必须重新获取，如果还是用以前老的迭代器，会出现 异常：java.util.ConcurrentModificationException
- 重点：在迭代集合元素的过程中，不能调用集合对象的remove方法，删除元素：c.remove(o); 迭代过程中不能这样。会出异常
- 重点：在迭代元素的过程当中，一定要使用迭代器Iterator的remove方法，删除元素，不要使用集合自带的remove方法删除元素。迭代器去删除时，会自动更新迭代器，并且更新集合（删除集合中的元素）
- 迭代器相当于对当前集合的状态拍了一个快照，迭代时会参照这个快照进行迭代。

```
    Collection c = new ArrayList();
    Iterator it = c.iterator();  //获取迭代器
// 添加元素
    c.add(1); // Integer类型
    c.add(2);
//Iterator it = c.iterator();
    while(it.hasNext()){
        Object obj = it.next();  //会报错，重新添加元素后，必须重新获取迭代器
    //c.remove(1); // 直接通过集合去删除元素，没有通知迭代器。（导致迭代器的快照和原集合状态不同。）
        it.remove(); // 删除的一定是迭代器指向的当前元素，迭代器删除时会自动更新迭代器。
        System.out.println(obj);
}
```

20.2.2 contains方法

- contains方法是用来判断集合中是否包含某个元素的方法，它在底层是调用了对象的equals方法进行比对, equals方法返回true，就表示包含这个元素。
- 存放在集合中的类型一定要重写equals方法，这样调用contains方法就不会比较内存地址了，而是根据内容判断

```
// 创建集合对象
    Collection c = new ArrayList();
// 向集合中存储元素
    String s1 = new String("abc"); // s1 = 0x1111
    c.add(s1); // 放进去了一个"abc"
// 新建的对象String
    String x = new String("abc"); // x = 0x5555
// c集合中是否包含x？结果猜测一下是true还是false？
    System.out.println(c.contains(x)); // true,调用equals方法，而String重写了equals方法，是通过内容来判断的
```

- remove() 方法也调用了equals()方法

```java
// 创建集合对象
    Collection cc = new ArrayList();
    String s1 = new String("hello");。
    cc.add(s1);

// 创建了一个新的字符串对象
    String s2 = new String("hello");
// 删除s2
    cc.remove(s2); // s1.equals(s2) java认为s1和s2是一样的。删除s2就是删除s1。注意s2没有添加进集合中
// 集合中元素个数是？
    System.out.println(cc.size()); // 0
```

20.3 List

- List集合存储元素特点：有序可重复。有序：List集合中的元素有下标， 从0开始，以1递增。；可重复：存储一个1，还可以再存储1.
- 以下只列出List接口特有的常用的方法（父接口方法都有）：
  void add(int index, Object element)---指定位置添加元素

  Object set(int index, Object element) ---修改指定位置的元素

  Object get(int index)----根据下标获取元素
  int indexOf(Object o)-----获取指定对象第一次出现处的索引。
  int lastIndexOf(Object o)---获取指定对象最后一次出现处的索引。
  Object remove(int index)---删除指定下标位置的元素
- list有自己特殊的遍历方法，根据下标取值

```java
// 创建List类型的集合。
        List myList = new ArrayList();

        // 添加元素
        myList.add("A"); // 默认都是向集合末尾添加元素。
        myList.add(1, "KING");

        // 迭代
        Iterator it = myList.iterator();
        while(it.hasNext()){
            Object elt = it.next();
            System.out.println(elt);
        }
        // 根据下标获取元素
        Object firstObj = myList.get(0);
        System.out.println(firstObj);

        // 因为有下标，所以List集合有自己比较特殊的遍历方式
        // 通过下标遍历。【List集合特有的方式，Set没有。】
        for(int i = 0; i < myList.size(); i++){
```

```
            Object obj = myList.get(i);
            System.out.println(obj);
}
```

20.3.1 ArrayList

- 默认初始化容量10（底层先创建了一个长度为0的数组，当添加第一个元素的时候，初始化容量10。）
- 集合底层是一个Object[]数组。
- 构造方法：new ArrayList();----无参
  new ArrayList(20);---有参，指定容量
- ArrayList集合的扩容：增长到原容量的1.5倍。(源代码中增长值为：原长度向右移一位（位运算，右移一位相当于除2））
  优化策略：尽可能少的扩容。因为数组扩容效率比较低，建议在使用ArrayList集合的时候预估计元素的个数，给定一个初始化容量。
- 数组优点：检索效率比较高。
  数组缺点：随机增删元素效率比较低。向数组末尾添加元素，效率很高，不受影响。另外数组无法存储大数据量。（很难找到一块非常巨大的连续的内存空间。）
- ArrayList集合是非线程安全的。（不是线程安全的集合。）

```
// 默认初始化容量是10，数组的长度是10
    List list1 = new ArrayList();
// 集合的size()方法是获取当前集合中元素的个数。不是获取集合的容量。
    System.out.println(list1.size()); // 0
```

20.3.2 LinkedList

- 单链表中的节点。节点是单向链表中基本的单元。
- 每一个节点Node都有两个属性：一个属性：是存储的数据。另一个属性：是下一个节点的内存地址。
- 链表优点：链表在空间上内存地址不连续，故增删元素不涉及大量元素位移，随机增删元素效率较高
  缺点：查询效率较低，每次查找某个元素都需要从头节点开始往下遍历
- LinkedList也是有下标的
- 双向链表，节点包含三部分，上一个节点内存地址；数据；下一个节点内存地址
- LinkedList集合没有初始化容量，first和last都是null

```
//实现单链表添加元素
//定义一个节点
public class Node{
    Object data;//存储的数据
    Node next; //下一个节点的内存地址
    public Node(){}
    public Node(Object data,Node next){
```

```
        this.data=data;
        this.next=next;
    }
}
public class Link{
    Node header;//头节点
    public void add(Object data){
        if(header==null){       //说明还没有节点
            header=new Node(data,null);
        }else{                  //找到末尾节点
            Node endNode=findLast(header);
            endNode.next=new Node(data,null);
        }
    }
    private Node findLast(Node node){   //找末尾节点方法
        if(node.next==null){
            return node;        //说明第一个节点就是末尾节点
        }
        return findLast(node.next);   //递归
    }
}
```

20.3.3 Vector

- 底层也是一个数组。初始化容量：10
- 扩容：扩容之后是原容量的2倍。
- Vector中所有的方法都是线程同步的，都带有synchronized关键字，是线程安全的。效率比较低，使用较少了。
- 怎么将一个线程不安全的ArrayList集合转换成线程安全的呢?
  使用集合工具类：java.util.Collections;

```
// 创建一个Vector集合
List vector = new Vector();
//Vector vector = new Vector();

Iterator it = vector.iterator();
while(it.hasNext()){
    Object obj = it.next();
    System.out.println(obj);
}

// 这个可能以后要使用！！！！
    List myList = new ArrayList(); // 非线程安全的。
// 变成线程安全的
    Collections.synchronizedList(myList); // 这里没有办法看效果，因为多线程没学，你记住先！
// myList集合就是线程安全的了。
    myList.add("111");
```

```
    myList.add("222");
    myList.add("333");
```

20.4 泛型

- 泛型这种语法机制，只在程序编译阶段起作用，只是给编译器参考的。（运行阶段泛型没用！）
- 使用了泛型好处是什么？
  第一：集合中存储的元素类型统一了。
  第二：从集合中取出的元素类型是泛型指定的类型，不需要进行大量的“向下转型”！
- 泛型的缺点是什么？ 导致集合中存储的元素缺乏多样性！

```
// 使用泛型List<Animal>之后，表示List集合中只允许存储Animal类型的数据。
        List<Animal> myList = new ArrayList<Animal>();
        // 指定List集合中只能存储Animal，那么存储String就编译报错了。
        // 这样用了泛型之后，集合中元素的数据类型更加统一了。
        //myList.add("abc");

        Cat c = new Cat();
        Bird b = new Bird();

        myList.add(c);
        myList.add(b);

        // 获取迭代器
        // 这个表示迭代器迭代的是Animal类型。
        Iterator<Animal> it = myList.iterator();
        while(it.hasNext()){
            // 使用泛型之后，每一次迭代返回的数据都是Animal类型。
            Animal a = it.next();
            // 这里不需要进行强制类型转换了。直接调用。
            a.move();

            // 调用子类型特有的方法还是需要向下转换的！
            Animal a = it.next();
            if(a instanceof Cat) {
                Cat x = (Cat)a;
                x.catchMouse();
            }
            if(a instanceof Bird) {
                Bird y = (Bird)a;
                y.fly();
            }
        }
    }
class Animal {
    // 父类自带方法
```

```
    public void move(){
        System.out.println("动物在移动！");
    }
}
class Cat extends Animal {
    // 特有方法
    public void catchMouse(){
        System.out.println("猫抓老鼠！");
    }
}
class Bird extends Animal {
    // 特有方法
    public void fly(){
        System.out.println("鸟儿在飞翔！");
    }
}
```

- 自动类型推断机制。（又称为钻石表达式）

```
// ArrayList<这里的类型会自动推断>()，前提是JDK8之后才允许。
        List<Animal> myList = new ArrayList<>();

        myList.add(new Animal());
        myList.add(new Cat());
        myList.add(new Bird());
        // 遍历
        Iterator<Animal> it = myList.iterator();
        while(it.hasNext()){
            Animal a = it.next();  //直接返回一个Animal类型
            a.move();
        }

```

- 创建对象时，前面的类型用泛型指定固定类型
- 自定义泛型时<>中就是一个标识符，随便写（一般是E或T），起决定作用的还是创建时前面的指定类型

```
class MyIterator<T> {
    public T get(T age){
        return age;
    }

        MyIterator<String> mi = new MyIterator<>();
        String s1 = mi.get("123");
        MyIterator<Animal> mi2 = new MyIterator<>();
        Animal a = mi2.get(new Animal());
```

20.5 foreach

```
语法：
    for(元素类型 变量名 : 数组或集合){
            System.out.println(变量名);
        }
```

```
int[] arr = {432,4,65,46,54,76,54};
for(int data : arr) {
            // data就是数组中的元素（数组中的每一个元素。）
            System.out.println(data);
        }

    List<String> strList = new ArrayList<>();
    // 添加元素
    strList.add("hello");
    strList.add("world!");
    for(String s : strList){ // 因为泛型使用的是String类型，所以是：String s
        System.out.println(s);
    }
```

- foreach有一个缺点：没有下标。在需要使用下标的循环中，不建议使用增强for循环。

20. 6 Map

- Map和Collection没有继承关系。
- Map集合以key和value的方式存储数据：键值对。key和value都是引用数据类型。 key和value都是存储对象的内存地址。

```
Map接口中常用方法：
    V put(K key, V value) -----向Map集合中添加键值对
    V get(Object key) -----通过key获取value
    void clear()    ----清空Map集合
    boolean containsKey(Object key) -----判断Map中是否包含某个key
    boolean containsValue(Object value) ---判断Map中是否包含某个value
    boolean isEmpty()  --- 判断Map集合中元素个数是否为0
    V remove(Object key) ---通过key删除键值对
    int size() ----获取Map集合中键值对的个数。
    Collection<V> values() ----获取Map集合中所有的value，返回一个Collection
    Set<K> keySet()---- 获取Map集合所有的key（所有的键是一个set集合）
    Set<Map.Entry<K,V>> entrySet()-----将Map集合转换成Set集合

    假设现在有一个Map集合，如下所示：
    map1集合对象
    key             value
    ----------------------------
    1               zhangsan
```

```
    2               lisi
    3               wangwu
    4               zhaoliu

    Set set = map1.entrySet();
    set集合对象
    1=zhangsan 【注意：Map集合通过entrySet()方法转换成的这个Set集合，Set集合中元素的类型是 Map.Entry<K,V>】
    2=lisi     【Map.Entry和String一样，都是一种类型的名字，只不过：Map.Entry是静态内部类，是Map中的静态内部类】
    3=wangwu
    4=zhaoliu ---> 这个东西是个什么？Map.Entry
```

```
    // 创建Map集合对象
    Map<Integer, String> map = new HashMap<>();  //泛型
    // 向Map集合中添加键值对
    map.put(1, "zhangsan"); // 1在这里进行了自动装箱。
    map.put(2, "lisi");
    // 通过key获取value
    String value = map.get(2);
    System.out.println(value);
    // 获取所有的value
    Collection<String> values = map.values();
```

20.6.1Map集合遍历

- 方法一：先取出所有的的key（keySet()方法），然后通过遍历key去取出所有的值(get(Object key)方法)

```
    Map<Integer, String> map = new HashMap<>();
    map.put(1, "zhangsan");
    map.put(2, "lisi");
    map.put(3, "wangwu");
    // 获取所有的key，所有的key是一个Set集合
    Set<Integer> keys = map.keySet();
    // 遍历key，通过key获取value

    // 迭代器可以
    /*Iterator<Integer> it = keys.iterator();
            while(it.hasNext()){
                // 取出其中一个key
                Integer key = it.next();
                // 通过key获取value
                String value = map.get(key);
                System.out.println(key + "=" + value);
            }*/
    // foreach也可以
    for(Integer key : keys){
```

```
        System.out.println(key + "=" + map.get(key));
    }
```

- 方法二：通过Set<Map.Entry<K,V>> entrySet()把Map集合直接全部转换成Set集合。Set集合中元素的类型是：Map.Entry，其中用两个方法getKey(),getValue()来取出键和值

```
    Set<Map.Entry<Integer,String>> set = map.entrySet();
    // 遍历Set集合，每一次取出一个Node
    // 迭代器
    /*Iterator<Map.Entry<Integer,String>> it2 = set.iterator();
            while(it2.hasNext()){
                Map.Entry<Integer,String> node = it2.next();
                Integer key = node.getKey();
                String value = node.getValue();
                System.out.println(key + "=" + value);
            }*/

    // foreach
    // 这种方式效率比较高，因为获取key和value都是直接从node对象中获取的属性值。
    // 这种方式比较适合于大数据量。
    for(Map.Entry<Integer,String> node : set){
        System.out.println(node.getKey() + "--->" + node.getValue());
    }
```

20.6.2HashMap

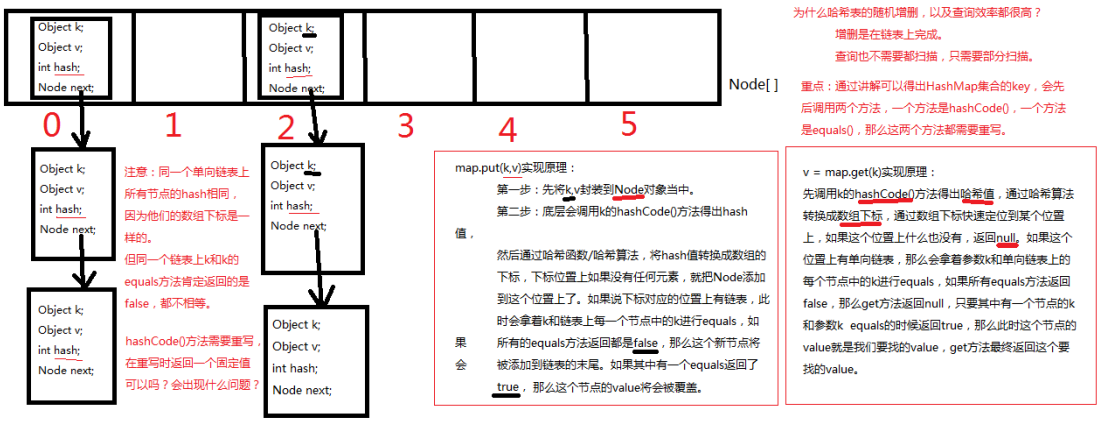
- HashMap集合底层是哈希表/散列表的数据结构。
- 哈希表是一个数组和单向链表的结合体,查询效率和增删效率都比较高

```
HashMap集合底层的源代码：
    public class HashMap{
        Node<K,V>[] table;// HashMap底层实际上就是一个数组。（一维数组）

        // 静态的内部类HashMap.Node
        static class Node<K,V> {
            final int hash; // 哈希值（哈希值是key的hashCode()方法的执行结果。hash值通过哈希函数/算法，可以转换存储成数组的下标。）
            final K key; // 存储到Map集合中的那个key
            V value; // 存储到Map集合中的那个value
            Node<K,V> next; // 下一个节点的内存地址。
        }
    }
哈希表/散列表：一维数组，这个数组中每一个元素是一个单向链表。（数组和链表的结合体。）
```

最主要掌握的是：

```
map.put(k,v)
v = map.get(k)
```

以上这两个方法的实现原理，是必须掌握的。



- 如果哈希表单向链表中元素超过8个，单向链表这种数据结构会变成红黑树数据结构，当红黑树上节点小于6时，会重新把红黑树变成单向链表数据结构。
- HashMap集合的key部分特点：无序，不可重复。
  无序：因为不一定挂到哪个单向链表上。
  不可重复： equals方法来保证HashMap集合的key不可重复。如果key重复了，value会覆盖。
- 重点：放在HashMap集合key部分的，以及放在HashSet集合中的元素，需要同时重写hashCode方法和equals方法。
- HashMap集合的默认初始化容量是16，默认加载因子是0.75。这个默认加载因子是当HashMap集合底层数组的容量达到75%的时候，数组开始扩容，扩容之后是原容量的2倍。(HashSet也一样)
- 重点：HashMap集合初始化容量必须是2的倍数，这也是官方推荐的，这是因为达到散列均匀，为了提高HashMap集合的存取效率，所必须的。

哈希表HashMap使用不当时无法发挥性能！

假设将所有的hashCode()方法返回值固定为某个值，那么会导致底层哈希表变成了纯单向链表。这种情况我们成为：散列分布不均匀。

什么是散列分布均匀？

假设有100个元素，10个单向链表，那么每个单向链表上有10个节点，这是最好的，

是散列分布均匀的。

假设将所有的hashCode()方法返回值都设定为不一样的值，可以吗，有什么问题？

不行，因为这样的话导致底层哈希表就成为一维数组了，没有链表的概念了。

也是散列分布不均匀。

散列分布均匀需要你重写hashCode()方法时有一定的技巧。

- 向Map集合中存，以及从Map集合中取，都是先调用key的hashCode方法，然后再调用equals方法！

equals方法有可能调用，也有可能不调用，数组下标位置上如果是null，equals不需要执行。

- 注意：如果一个类的equals方法重写了，那么hashCode()方法必须重写，并且equals方法返回如果是true，hashCode()方法返回的值必须一样。
  equals方法返回true表示两个对象相同，在同一个单向链表上比较，那么对于同一个单向链表上的节点来说，他们的哈希值都是相同的。所以hashCode()方法的返回值也应该相同。
  比如存放十个内容相同的对象，如果不重写，哈希值都不同，在没有哈希碰撞的前提下就会都存进去，而这违背了Map集合不可重复的特点。
- 对于哈希表数据结构来说：如果o1和o2的hash值相同，一定是放到同一个单向链表上。
  当然如果o1和o2的hash值不同，但由于哈希算法执行结束之后转换的数组下标可能相同，此时会发生“哈希碰撞”。

```
    Map<Integer,String> map = new HashMap<>();
    map.put(1111, "zhangsan");
    map.put(2222, "zhaoliu");
    map.put(2222, "king"); //key重复的时候value会自动覆盖。

    System.out.println(map.size()); // 2

    // 遍历Map集合
    Set<Map.Entry<Integer,String>> set = map.entrySet();
    for(Map.Entry<Integer,String> entry : set){
        // 验证结果：HashMap集合key部分元素：无序不可重复。
        System.out.println(entry.getKey() + "=" +
entry.getValue());
```

- HashMap集合key允许为null，但是要注意：HashMap集合的key null值只能有一个。value也可以为null的。

```
    Map map = new HashMap();

    // HashMap集合允许key为null
    map.put(null, null);
    System.out.println(map.size()); // 1

    // key重复的话value是覆盖！
    map.put(null, 100);
    System.out.println(map.size()); //1

    // 通过key获取value
    System.out.println(map.get(null)); // 100

```

20.6.3 Hashtable

- Hashtable的key和value都是不能为null的。
  HashMap集合的key和value都是可以为null的。
- Hashtable方法都带有synchronized：线程安全的。线程安全有其它的方案，这个Hashtable对线程的处理
  导致效率较低，使用较少了。
- Hashtable和HashMap一样，底层都是哈希表数据结构。Hashtable的初始化容量是11，默认加载因子是：0.75f。Hashtable的扩容是：原容量 * 2 + 1

```
    Map map = new Hashtable();
    map.put(null, "123");//NullPointerException
    map.put(100, null);//NullPointerException
```

- Properties属性类,Properties是一个Map集合，继承Hashtable，Properties的key和value都是String类型。
- Properties被称为属性类对象。Properties是线程安全的

```
    // 创建一个Properties对象
    Properties pro = new Properties();

    // 需要掌握Properties的两个方法，一个存，一个取。
    pro.setProperty("url",
"jdbc:mysql://localhost:3306/bjpowernode");
    pro.setProperty("driver","com.mysql.jdbc.Driver");

    // 通过key获取value
    String url = pro.getProperty("url");
    String driver = pro.getProperty("driver");
```

20.6.4 TreeSet

- TreeSet集合底层实际上是一个TreeMap
- TreeMap集合底层是一个二叉树,放到TreeSet集合中的元素，等同于放到TreeMap集合key部分了。
- TreeSet集合中的元素：无序不可重复，但是可以按照元素的大小顺序自动排序,称为：可排序集合。一边存一边排序
- TreeSet类中有无参构造方法和有参构造方法，调用两种方法创建对象都可以排序。有参构造方法传递的是一个比较器，需要自己定义一个。

```
    // 创建一个TreeSet集合
    TreeSet<String> ts = new TreeSet<>();
    // 添加String
    ts.add("zhangsan");
    ts.add("lisi");
    ts.add("wangwu");
```

```
        ts.add("zhangsi");
        ts.add("wangliu");
        // 遍历
        for(String s : ts){
            // 按照字典顺序，升序！
            System.out.println(s);
        }
    /*
    数据库中有很多数据：
        userid  name     birth
        ------------------------------------
        1       zs          1980-11-11
        2       ls          1980-10-11
        3       ww          1981-11-11
        4       zl          1979-11-11
        编写程序从数据库当中取出数据，在页面展示用户信息的时候按照生日升序或者降序。
        这个时候可以使用TreeSet集合，因为TreeSet集合放进去，拿出来就是有顺序的。
```

- TreeSet（无参构造）排序必须存放的对象实现Comparable接口，并且重写compareTo方法，在方法中指定比较规则，然后才可以排序
- 对于String和Interage类来说，底层已经重写了compareTo方法，所以直接存放可以排序；对于自定义的类型来说，就必须实现Comparable接口，并且手动重写compareTo方法。
- compareTo方法的返回值：
  返回0表示相同，value会覆盖。
  返回>0，会继续在右子树上找，将传进来的数存放到右子树。【10 - 9 = 1，1 > 0的说明左边这个数字比较大。所以在右子树上找。】
  返回<0，会继续在左子树上找，将传进来的数存放在左子树。
- 程序运行的时候出现了这个异常：java.lang.ClassCastException:class com.bjpowernode.javase.collection.Person cannot be cast to class java.lang.Comparable
  出现这个异常的原因是：Person类没有实现java.lang.Comparable接口。

```
public class TreeSetTest04 {
    public static void main(String[] args) {
        Customer c1 = new Customer(32);
        Customer c2 = new Customer(20);
        Customer c3 = new Customer(30);
        Customer c4 = new Customer(25);

        // 创建TreeSet集合
        TreeSet<Customer> customers = new TreeSet<>();
        // 添加元素
        customers.add(c1);
        customers.add(c2);
```

```
        customers.add(c3);
        customers.add(c4);

        // 遍历
        for (Customer c : customers){
            System.out.println(c);
        }
    }
}

// 放在TreeSet集合中的元素需要实现java.lang.Comparable接口。
// 并且实现compareTo方法。equals可以不写。
class Customer implements Comparable<Customer>{
    int age;
    public Customer(int age){
        this.age = age;
    }
    // 需要在这个方法中编写比较的逻辑，或者说比较的规则，按照什么进行比较！
    // k.compareTo(t.key) 拿着参数k和集合中的每一个key进行比较，返回值可能是>0 <0 =0。
    @Override
    public int compareTo(Customer c) {
        // this是本次添加的对象，c是原来集合中有的
        return this.age-c.age;          //升序，最后中序遍历输出是升序
    }

    public String toString(){
        return "Customer[age="+age+"]";
    }
}

```

- 重点：放到TreeSet或者TreeMap集合key部分的元素要想做到排序,包括两种方式：
  第一种：放在集合中的元素实现java.lang.Comparable接口，重写并且重写compareTo方法。
  第二种：在构造TreeSet或者TreeMap集合的时候，给它传一个比较器对象实现Comparator接口。
- Comparable和Comparator怎么选择呢?
  当比较规则不会发生改变的时候，或者说当比较规则只有1个的时候，建议实现Comparable接口。
  如果比较规则有多个，并且需要多个比较规则之间频繁切换，建议使用Comparator接口。
- Comparator接口的设计符合OCP原则。

```
public class TreeSetTest06 {
    public static void main(String[] args) {

```

```
        // 创建TreeSet集合的时候，需要使用这个比较器。
        TreeSet<WuGui> wuGuis = new TreeSet<>(new WuGuiComparator());

        /*大家可以使用匿名内部类的方式（这个类没有名字。直接new接口。）
        TreeSet<WuGui> wuGuis = new TreeSet<>(new Comparator<WuGui>() {
            @Override
            public int compare(WuGui o1, WuGui o2) {
                return o1.age - o2.age;
            }
        });*/
        wuGuis.add(new WuGui(1000));
        wuGuis.add(new WuGui(800));
        wuGuis.add(new WuGui(810));

        for(WuGui wuGui : wuGuis){
            System.out.println(wuGui);
        }
    }
}
// 乌龟
class WuGui{
    int age;
    public WuGui(int age){
        this.age = age;
    }
    @Override
    public String toString() {
        return "小乌龟[" +
                "age=" + age +
                ']';
    }
}

// 单独在这里编写一个比较器
// 比较器实现java.util.Comparator接口。（Comparable是java.lang包下的。Comparator是java.util包下的。）
class WuGuiComparator implements Comparator<WuGui> {
    @Override
    public int compare(WuGui o1, WuGui o2) {
        // 指定比较规则，按照年龄排序，从小到大，o1是本次新添加的对象，o2是原来树中有的
        return o1.age - o2.age;
    }
```

20.6.5 自平衡二叉树

- TreeSet\TreeMap是自平衡二叉树，遵循左小右大原则存放。
- 遍历二叉树：前序遍历：根左右
  中序遍历：左根右
  后序遍历：左右根
- TreeSet\TreeMap集合采用中序遍历方式，Iterator迭代器采用的是中序遍历
- 中序遍历出来就是从小到大排序的

20.7 集合工具类

- java.util.Collection 集合接口
- java.util.Collections 集合工具类，方便集合的操作。

```
    List<String> list = new ArrayList<>();
// 变成线程安全的
    Collections.synchronizedList(list);

    list.add("abf");
    list.add("abx");
    list.add("abc");
    list.add("abe");
// 排序
    Collections.sort(list);
// 注意：对List集合中元素排序，需要保证List集合中的元素实现了：Comparable接口。
// 对Set集合怎么排序呢？
        Set<String> set = new HashSet<>();
        set.add("king");
        set.add("kingsoft");
        set.add("king2");
        set.add("king1");
        // 将Set集合转换成List集合
        List<String> myList = new ArrayList<>(set);
        Collections.sort(myList);
        for(String s : myList) {
            System.out.println(s);
        }

        // 这种方式也可以排序。
        //Collections.sort(list集合/ 比较器对象);
```

20.8集合间转换

- List和Set通过调用有参构造方法就可以转换

```
    ArrayList<String> list=new ArrayList<>();
    list.add("123");
    list.add("456");
    list.add("123");
    System.out.println(list.size());    //3
    HashSet<String> hashSet=new HashSet<>(list);
    System.out.println(hashSet.size());   //2
```

- Map转换为Set,通过entrySet()方法
- LIst有三种遍历方式，Set有两种，Map有两种
- TreeSet默认排序是从小到大，传入比较器，可以改变排序顺序

```
// 编写比较器可以改变规则。
    TreeSet<Integer> ts = new TreeSet<>(new Comparator<Integer>() {  //匿名内部类
        @Override
        public int compare(Integer o1, Integer o2) {
            return o2 - o1; // 自动拆箱,从大到小排序，边存边排，o1是本次添加的数，o2是里面原先有的
        }
    });
    // 添加元素
    ts.add(1);
    ts.add(100);
    ts.add(10);
    ts.add(0);
    // 遍历（foreach）
    for(Integer x : ts){
        System.out.println(x);   //100 10 1 0
    }

```

二十一、IO流

- I : Input O : Output 通过IO可以完成硬盘文件的读和写。
- 分类：输入流、输出流 字节流、字符流
- 一种方式是按照流的方向进行分类：以内存作为参照物
  往内存中去，叫做输入(Input)。或者叫做读(Read)。
  从内存中出来，叫做输出(Output)。或者叫做写(Write)。
- 另一种是按照读取数据方式不同进行分类：
  有的流是按照字节的方式读取数据，一次读取1个字节byte，等同于一次读取8个二进制位。这种流是万能的，什么类型的文件都可以读取。包括：文本文件，图片，声音文件，视频文件等....
  有的流是按照字符的方式读取数据的，一次读取一个字符，这种流是为了方便读取普通文本文件而存在的，这种流不能读取：图片、声音、视频等文件。只能读取纯文本文件，连word文件都无法读取。

- windows中'a'占一个字节，'中'占两个字节；java中char都占两个字节
- java中所有的流都是在：java.io.*;下。
- 四大家族(都是抽象类)：

  java.io.InputStream 字节输入流
  java.io.OutputStream 字节输出流
  java.io.Reader 字符输入流
  java.io.Writer 字符输出流

  注意：在java中只要“类名”以Stream结尾的都是字节流。以“Reader/Writer”结尾的都是字符流。
- 所有的流都实现了：java.io.Closeable接口，都是可关闭的，都有close()方法。流毕竟是一个管道，这个是内存和硬盘之间的通道，用完之后一定要关闭，不然会耗费(占用)很多资源。养成好习惯，用完流一定要关闭。
- 所有的输出流都实现了：java.io.Flushable接口，都是可刷新的，都有flush()方法。养成一个好习惯，输出流在最终输出之后，一定要记得flush()刷新一下。这个刷新表示将通道/管道当中剩余未输出的数据强行输出完（清空管道！）刷新的作用就是清空管道。注意：如果没有flush()可能会导致丢失数据。

```
java.io包下需要掌握的流有16个：
    文件专属：
        java.io.FileInputStream（掌握）
        java.io.FileOutputStream（掌握）
        java.io.FileReader
        java.io.FileWriter

    转换流：（将字节流转换成字符流）
        java.io.InputStreamReader
        java.io.OutputStreamWriter

    缓冲流专属：
        java.io.BufferedReader
        java.io.BufferedWriter
        java.io.BufferedInputStream
        java.io.BufferedOutputStream

    数据流专属：
        java.io.DataInputStream
        java.io.DataOutputStream

    标准输出流：
        java.io.PrintWriter
        java.io.PrintStream（掌握）

    对象专属流：
        java.io.ObjectInputStream（掌握）
        java.io.ObjectOutputStream（掌握）
```

21.1 java.io.FileInputStream

- 文件字节输入流，万能的，任何类型的文件都可以采用这个流来读。（硬盘--->内存）
- 构造方法：FileInputStream(String name)-----name指文件路径
  FileInputStream(File file)-----传递一个File文件对象
- 在finally语句块当中确保流一定关闭。
- int read()---读取文件内容，一次读取一个字节,返回int类型，读字符串也是返回int,需要调方法转成字符串
- 已经读到文件的末尾了，再读的时候读取不到任何数据，返回-1

```
public class FileInputStreamTest01 {
    public static void main(String[] args) {
        FileInputStream fis = null;         //写try里面在finally中无法关闭
        try {
            // 创建文件字节输入流对象。
            // fis = new FileInputStream("D:\\course\\JavaProjects\\02-JavaSE\\temp");
            // 写成这个/也是可以的,绝对路径。
            fis = new FileInputStream("D:/course/JavaProjects/02-JavaSE/temp");//abc

            // 开始读
            int readData = 0;
            while((readData = fis.read()) != -1){
                System.out.println(readData);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            // 在finally语句块当中确保流一定关闭。
            if (fis != null) { // 避免空指针异常!
                // 关闭流的前提是：流不是空。流是null的时候没必要关闭。
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

- int read(byte[] b)-----一次最多读取 b.length 个字节，返回值是读取到的字节数量(不是字节本身)，读不到返回-1
- 减少硬盘和内存的交互，提高程序的执行效率。
- 往byte[]数组当中读，第二次读的话是从下标0的位置开始覆盖，读几位覆盖几位。
- IDEA默认的当前路径是：工程Project的根
- 字节数组--->String：调用String构造方法，new String(byte[]，起始下标，长度)

```
        FileInputStream fis = null;
        try {
            // 相对路径一定是从当前所在的位置作为起点开始找!
            fis = new FileInputStream("chapter23/tempfile2");  // 文件内容：abcdef
            byte[] bytes = new byte[4]; // 准备一个4个长度的byte数组，一次最多读取4个字节。

            int readCount = fis.read(bytes);// 这个方法返回值是：读取到的字节数量（不是字节本身）
            System.out.println(readCount); // 第一次读到了4个字节。
            // 将字节数组全部转换成字符串
            //System.out.println(new String(bytes)); // abcd
            // 不应该全部都转换，应该是读取了多少个字节，转换多少个。
            System.out.println(new String(bytes,0, readCount)); //abcd

            readCount = fis.read(bytes); // 第二次只能读取到2个字节，会覆盖前两个。
            System.out.println(readCount); // 2
            // 将字节数组全部转换成字符串
            //System.out.println(new String(bytes)); // efcd
            System.out.println(new String(bytes,0, readCount)); //ef

            readCount = fis.read(bytes); // 1个字节都没有读取到返回-1
            System.out.println(readCount); // -1

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
```

```
35          }
```

- 最终版(掌握)

```
    byte[] bytes=new byte[4];
/*while(true){
    int readcount=fis.read(bytes);
    if(readcount==-1){
        break;
    }
    System.out.print(new String(bytes,0,readcount));
}*/
    int readCount = 0;
    while((readCount = fis.read(bytes)) != -1) {
        System.out.print(new String(bytes, 0, readCount));
    }
```

- FileInputStream类的其它常用方法：
  int available()：返回流当中剩余的没有读到的字节数量
  long skip(long n)：跳过几个字节不读。

```
    FileInputStream fis = null;
    try {
        fis = new FileInputStream("tempfile");  //abcdef
        System.out.println("总字节数量：" + fis.available());  //6
        byte[] bytes = new byte[fis.available()]; // 这种方式不太适合太大的文件，因为byte[]数组不能太大。
        // 不需要循环了。直接读一次就行了。
        int readCount = fis.read(bytes); // 6
        System.out.println(new String(bytes)); // abcdef

        // skip跳过几个字节不读取，这个方法也可能以后会用！
        fis.skip(3);
        System.out.println(fis.read()); //100(c)

```

21.2 FileOutputStream

- 文件字节输出流，负责写。从内存到硬盘。
- 构造方法：FileOutputStream(String name)-----用指定名称写入文件，文件不存在的时候会自动新建，存在的话会把原文件清空，然后重新写入。
  FileOutputStream(String name，true)-----以追加的方式在文件末尾写入。不会清空原文件内容
- void write(byte[] b)----将byte数组全部写出
  void write(byte[] b，int off,int len)----byte数组从off开始，长度为len的写出
- String---->byte数组：s.getBytes()
- 注意：写完之后要记得刷新，fos.flush();

```java
public class FileOutputStreamTest01 {
    public static void main(String[] args) {
        FileOutputStream fos = null;
        try {
            // myfile文件不存在的时候会自动新建!
            // 这种方式谨慎使用，这种方式会先将原文件清空，然后重新写入。
            //fos = new FileOutputStream("myfile");

            // 以追加的方式在文件末尾写入。不会清空原文件内容。
            fos = new FileOutputStream("chapter23/src/tempfile3", true);
            // 开始写。
            byte[] bytes = {97, 98, 99, 100};
            // 将byte数组全部写出!
            fos.write(bytes); // abcd
            // 将byte数组的一部分写出!
            fos.write(bytes, 0, 2); // 再写出ab

            // 字符串
            String s = "我是一个中国人，我骄傲！！！";
            // 将字符串转换成byte数组。
            byte[] bs = s.getBytes();
            // 写
            fos.write(bs);

            // 写完之后，最后一定要刷新
            fos.flush();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

21.3 拷贝

- 使用FileInputStream + FileOutputStream完成文件的拷贝。
- 拷贝的过程应该是一边读，一边写。
  使用以上的字节流拷贝文件的时候，文件类型随意，万能的。什么样的文件都能

拷贝。

- 注意，最后关闭的时候应该分开抓异常，一起try的时候，其中一个出现异常，可能会影响到另一个流的关闭。
  别忘记输出流最后要刷新

```
        FileInputStream fis = null;
        FileOutputStream fos = null;
        try {
            // 创建一个输入流对象
            fis = new FileInputStream("D:\\course\\02-JavaSE动力节点-JavaSE-杜聚宾.avi");
            // 创建一个输出流对象
            fos = new FileOutputStream("C:\\动力节点-JavaSE-杜聚宾.avi");

            // 最核心的：一边读，一边写
            byte[] bytes = new byte[1024 * 1024]; // 1MB（一次最多拷贝1MB。）
            int readCount = 0;
            while((readCount = fis.read(bytes)) != -1) {
                fos.write(bytes, 0, readCount);
            }
            // 刷新，输出流最后要刷新
            fos.flush();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            // 分开try，不要一起try。
            // 一起try的时候，其中一个出现异常，可能会影响到另一个流的关闭。
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```
41
```

21.4 FileReader

- 文件字符输入流，只能读取普通文本。按照字符的方式读取
- int read(char[] c)
- char数组转为String:new String(chars,起始,长度)

```
    FileReader reader = null;
    try {
        // 创建文件字符输入流
        reader = new FileReader("tempfile");
        char[] chars = new char[4]; // 一次读取4个字符
        int readCount = 0;
        while((readCount = reader.read(chars)) != -1) {
            System.out.print(new String(chars,0,readCount));
        }

        /*//准备一个char数组
        char[] chars = new char[4];
        // 往char数组中读
        reader.read(chars); // 按照字符的方式读取：第一次e，第二次f，第三次 风....
        for(char c : chars) {
            System.out.println(c);
        }*/
```

21.5FileWriter

- 文件字符输出流。写。只能输出普通文本。
- void write(char[] c)
  void write(char[] c,int off,int len)
  void write(String s)-----可以直接写入字符串
- 记得最后要刷新

```
        FileWriter out = null;
        try {
            // 创建文件字符输出流对象
            //out = new FileWriter("file");
            out = new FileWriter("file", true);  //不会清空原文件，在后面追加

            // 开始写。
            char[] chars = {'我','是','中','国','人'};
            out.write(chars);
            out.write(chars, 2, 3);

```

```
            out.write("我是一名java软件工程师！ ");
            // 写出一个换行符。
            out.write("\n");
            out.write("hello world!");
            // 刷新
            out.flush();
```

21.6 缓冲流

21.6.1 BufferedReader

- 带有缓冲区的字符输入流。
- 使用这个流的时候不需要自定义char数组，或者说不需要自定义byte数组。自带缓冲。
- 构造方法：BufferedReader(Reader in) -----需要传进来一个字符流，不能传字节流，需要转换，
- 这个被传进来的流叫做：节点流。外部负责包装的这个流，叫做：包装流，还有一个名字叫做：处理流。
- String readLine()----读取一行，返回一个字符串，读不到数据返回null
- 包装流关闭的时候只需要关闭最外层的流就行，因为底层会关闭里面的节点流

```
public class BufferedReaderTest01 {
    public static void main(String[] args) throws Exception{

        FileReader reader = new FileReader("Copy02.java");
        // 当一个流的构造方法中需要一个流的时候，这个被传进来的流叫做：节点流。
        // 外部负责包装的这个流，叫做：包装流，还有一个名字叫做：处理流。
        // 像当前这个程序来说：FileReader就是一个节点流。BufferedReader就是包装流/处理流。
        BufferedReader br = new BufferedReader(reader);

        // br.readLine()方法读取一个文本行，但不带换行符。
        String s = null;
        while((s = br.readLine()) != null){
            System.out.print(s);
        }
        // 关闭流
        // 对于包装流来说，只需要关闭最外层流就行，里面的节点流会自动关闭。（可以看源代码。）
        br.close();
```

21.6.2 BufferedWriter

- 带有缓冲的字符输出流
- 构造方法中只能传一个字符流

```
public class BufferedWriterTest {
    public static void main(String[] args) throws Exception{
        // 带有缓冲区的字符输出流
        //BufferedWriter out = new BufferedWriter(new FileWriter("copy"));

        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(new FileOutputStream("copy", true)));
        // 开始写。
        out.write("hello world!");
        out.write("\n");
        out.write("hello kitty!");
        // 刷新
        out.flush();
        // 关闭最外层
        out.close();
```

21.7 转换流

- InputStreamReader:将字节流转换成字符流
- OutputStreamWriter：将字节流转换成字符流

```
         /*// 字节流
        FileInputStream in = new FileInputStream("CopyO2.java");

        // 通过转换流转换（InputStreamReader将字节流转换成字符流。）
        // in是节点流。reader是包装流。
        InputStreamReader reader = new InputStreamReader(in);

        // 这个构造方法只能传一个字符流。不能传字节流。
        // reader是节点流。br是包装流。
        BufferedReader br = new BufferedReader(reader);*/

        // 合并
        BufferedReader br = new BufferedReader(new InputStreamReader(new FileInputStream("CopyO2.java")));

        String line = null;
        while((line = br.readLine()) != null){
            System.out.println(line);
        }
        // 关闭最外层
        br.close();
```

21.8 数据流

- java.io.DataOutputStream：数据专属的流。这个流可以将数据连同数据的类型一并写入文件。
- 注意：这个文件不是普通文本文档。（这个文件使用记事本打不开。）
- 构造方法：DataOutputStream(OutputStream out)------传入一个字节流

```
    // 创建数据专属的字节输出流
    DataOutputStream dos = new DataOutputStream(new FileOutputStream("data"));
    // 写数据
    byte b = 100;
    int i = 300;
    boolean sex = false;
    char c = 'a';
    // 写
    dos.writeByte(b); // 把数据以及数据的类型一并写入到文件当中。
    dos.writeInt(i);
    dos.writeBoolean(sex);
    dos.writeChar(c);
    // 刷新
    dos.flush();
    // 关闭最外层
    dos.close();
```

- DataInputStream:数据字节输入流。
- DataOutputStream写的文件，只能使用DataInputStream去读。并且读的时候你需要提前知道写入的顺序。
  读的顺序需要和写的顺序一致。才可以正常取出数据。

```
    DataInputStream dis = new DataInputStream(new FileInputStream("data"));
    // 开始读
    byte b = dis.readByte();
    int i = dis.readInt();
    boolean sex = dis.readBoolean();
    char c = dis.readChar();

    System.out.println(i + 1000);
    dis.close();
```

21.9 标准输出流(重点)

- java.io.PrintStream：标准的字节输出流。默认输出到控制台。
- 标准输出流不需要手动close()关闭。
- System.out返回一个PrintStream对象
- System.setOut(PrintStream out) -----可以更改输出方向，输出到指定文件

```
         // 联合起来写
        System.out.println("hello world!");
        // 分开写
        PrintStream ps = System.out;
        ps.println("hello zhangsan");

        // 可以改变标准输出流的输出方向吗？ 可以
        // 标准输出流不再指向控制台，指向“log”文件。
        PrintStream out = new PrintStream(new
FileOutputStream("log"));
        // 修改输出方向，将输出方向修改到"log"文件。
        System.setOut(out);
        // 再输出
        System.out.println("hello world");  //就会输出到log文件中
```

```
//日志工具
public class Logger {
    public static void log(String msg) {
        try {
            // 指向一个日志文件
            PrintStream out = new PrintStream(new
FileOutputStream("log.txt", true));
            // 改变输出方向
            System.setOut(out);
            // 日期当前时间
            Date nowTime = new Date();
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss SSS");
            String strTime = sdf.format(nowTime);

            System.out.println(strTime + ": " + msg);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

21.10 对象专属流

- 序列化（Serialize）:将Java对象转换为二进制，写进磁盘，将java对象的状态保存下来的过程叫序列化（切割java对象）-------ObjectOutputStream，调用writeObject()方法
- 反序列化（DeSerialize）：从磁盘读出完整的Java对象-------ObjectInputStream，调用readObject()方法
- 参与序列化和反序列化的对象，必须实现Serializable接口，否则会出异常：java.io.NotSerializableException:
- 实现Serializable接口后java虚拟机会为该类自动生成一个序列化版本号

- java语言中是采用什么机制来区分类的?
  第一：首先通过类名进行比对，如果类名不一样，肯定不是同一个类。
  第二：如果类名一样，再怎么进行类的区别？靠序列化版本号进行区分。
- 这种自动生成的序列化版本号缺点是：一旦代码确定之后，不能进行后续的修改，因为只要修改，必然会重新编译，此时会生成全新的序列化版本号这个时候java虚拟机会认为这是一个全新的类。
- 结论：凡是一个类实现了Serializable接口，建议给该类提供一个固定不变的序列化版本号。以后这个类即使代码修改了，但是版本号不变，java虚拟机会认为是同一个类。如果序列号改了之后就无法进行反序列化了

```
注意：通过源代码发现，Serializable接口只是一个标志接口：
    public interface Serializable {
    }
        这个接口当中什么代码都没有。那么它起到一个什么作用呢？
        起到标识的作用，标志的作用，java虚拟机看到这个类实现了这个接口，可能会对这个类进行特殊待遇。
        Serializable这个标志接口是给java虚拟机参考的，java虚拟机看到这个接口之后，会为该类自动生成一个序列化版本号。
```

```
public class Student implements Serializable {
    // IDEA工具自动生成序列化版本号。
    //private static final long serialVersionUID = -7998917368642754840L;
    // 建议将序列化版本号手动的写出来。不建议自动生成
    private static final long serialVersionUID = 1L; // 手动固定一个序列化版本号

    private int no;
    private transient String name;  //transient 关键字表示游离的，不参与序列化操作
}
```

```
public class ObjectOutputStreamTest01 {
    public static void main(String[] args) throws Exception{
        // 创建java对象，Student要实现Serializable接口
        Student s = new Student(1111, "zhangsan");
        // 序列化
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("students"));
        // 序列化对象
        oos.writeObject(s);
        // 刷新
        oos.flush();
        // 关闭
        oos.close();

```

```
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("students"));
        // 开始反序列化，读
        Object obj = ois.readObject();
        // 反序列化回来是一个学生对象，所以会调用学生对象的toString方法。
        System.out.println(obj);
        ois.close();
    }
```

- 一次序列化多个对象：可以将对象放到集合当中，序列化集合
- 参与序列化的ArrayList集合以及集合中的元素User都需要实现java.io.Serializable接口。

```
    List<User> userList = new ArrayList<>();
    userList.add(new User(1,"zhangsan"));
    userList.add(new User(2, "lisi"));
    ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("users"));

    // 序列化一个集合，这个集合对象中放了很多其他对象。
    oos.writeObject(userList);
    oos.flush();
    oos.close();
    //反序列化
    ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("users"));
    //Object obj = ois.readObject();
    List<User> userList = (List<User>)ois.readObject();
    for(User user : userList){
        System.out.println(user);
    }
```

- transient 关键字表示游离的，不参与序列化操作
- 比如不想让某个实例变量序列化，可以用transient修饰；

21.11 File类

- File类不能完成文件的读和写。File类和四大家族没有关系
- File对象代表：文件和目录路径名的抽象表示形式。一个File对象有可能对应的是目录，也可能是文件。

```
    // 创建一个File对象
    File f1 = new File("D:\\file");

常用方法：
    // 判断是否存在！
    System.out.println(f1.exists());

```

```
    // 如果D:\file不存在，则以文件的形式创建出来
    if(!f1.exists()) {
        // 以文件形式新建
        f1.createNewFile();
    }

    如果D:\file不存在，则以目录的形式创建出来
    if(!f1.exists()) {
        // 以目录的形式新建。
        f1.mkdir();
    }

    // 可以创建多重目录吗？
    File f2 = new File("D:/a/b/c/d/e/f");
    if(!f2.exists()) {
        // 多重目录的形式新建。
        f2.mkdirs();
    }

    File f3 = new File("D:\\course\\01-开课\\学习方法.txt");
    // 获取文件的父路径
    String parentPath = f3.getParent();
    System.out.println(parentPath);    //D:\course\01-开课
    //获取父文件
    File parentFile = f3.getParentFile();
    //获取绝对路径
    System.out.println("获取绝对路径：" +
parentFile.getAbsolutePath());
```

```
    File f1 = new File("D:\\course\\01-开课\\开学典礼.ppt");
    // 获取文件名
    System.out.println("文件名：" + f1.getName());

    // 判断是否是一个目录
    System.out.println(f1.isDirectory()); // false

    // 判断是否是一个文件
    System.out.println(f1.isFile()); // true

    // 获取文件最后一次修改时间
    long haoMiao = f1.lastModified(); // 这个毫秒是从1970年到现在的总
毫秒数。
    // 将总毫秒数转换成日期?????
    Date time = new Date(haoMiao);
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss SSS");
    String strTime = sdf.format(time);
    System.out.println(strTime);

```

```
    // 获取文件大小
    System.out.println(f1.length()); //216064字节。
```

- File[] listFiles()---- 获取当前目录下所有的子文件。

```
    File f = new File("D:\\course\\01-开课");
    File[] files = f.listFiles();
    // foreach
    for(File file : files){
        //System.out.println(file.getAbsolutePath());  //绝对路径
        System.out.println(file.getName());
    }
```

- 带有目录的拷贝，需要先用File创建好拷贝源和拷贝目标，再调用方法。关键是把目录在拷贝目标中创建好，再进行文件的读和写。-----代码见Learn-String-io-Homework

21.12 IO和Properties的联合应用

- 调用Properties对象的load方法将文件中的数据加载到Map集合中

```
/*
IO+Properties的联合应用。
非常好的一个设计理念：
    以后经常改变的数据，可以单独写到一个文件中，使用程序动态读取。
    将来只需要修改这个文件的内容，java代码不需要改动，不需要重新
    编译，服务器也不需要重启。就可以拿到动态的信息。

    类似于以上机制的这种文件被称为配置文件。
    并且当配置文件中的内容格式是：
        key1=value
        key2=value   (最后不要有空格，用冒号也行)
    的时候，我们把这种配置文件叫做属性配置文件(#是注释)，属性配置文件中key重复的话，value会自动覆盖。

    java规范中有要求：属性配置文件建议以.properties结尾，但这不是必须的。
    这种以.properties结尾的文件在java中被称为：属性配置文件。
    其中Properties是专门存放属性配置文件内容的一个类。
 */
public class IoPropertiesTest01 {
    public static void main(String[] args) throws Exception{
        /*
        Properties是一个Map集合，key和value都是String类型。
        想将userinfo文件中的数据加载到Properties对象当中。
         */
        // 新建一个输入流对象
        FileReader reader = new FileReader("chapter23/userinfo.properties");//username=ad
```

```
26
27          // 新建一个Map集合
28          Properties pro = new Properties();
29
30          // 调用Properties对象的load方法将文件中的数据加载到Map集合中。
31          pro.load(reader); // 文件中的数据顺着管道加载到Map集合中，其中等
    号=左边做key，右边做value
32
33          // 通过key来获取value呢？
34          String username = pro.getProperty("username");
35          System.out.println(username);
36      }
37  }
```

二十二、线程

- 进程是一个应用程序（1个进程是一个软件）。线程是一个进程中的执行场景/执行单元，一个进程可以启动多个线程。
- 进程A和进程B的内存独立不共享。
- 线程A和线程B，堆内存和方法区内存共享。但是栈内存独立，一个线程一个栈，互不干扰。多线程并发可以提高效率。
- main方法结束，只代表主线程结束了，其他线程可能还在执行。
- 对于单核的CPU来说，在某一个时间点上实际上只能处理一件事情，但是由于CPU的处理速度极快，多个线程之间频繁切换执行，跟人来的感觉是：多个事情同时在做！

22.1 线程的创建和启动

- 第一种方式：编写一个类，直接继承java.lang.Thread，重写run方法。
  怎么创建线程对象？ new就行了。
  怎么启动线程呢？ 调用线程对象的start()方法。
- start()方法的作用是：启动一个分支线程，在JVM中开辟一个新的栈空间
  启动成功的线程会自动调用run方法，并且run方法在分支栈的栈底部（压栈）；
  run方法在分支栈的栈底部，main方法在主栈的栈底部。run和main是平级的。
- 注意：创建线程对象，直接调用run()方法是不会启动线程的，必须先调用start()方法启动线程。

```
1   public class ThreadTest02 {
2       public static void main(String[] args) {  // main方法在主栈中运
    行。
3           // 新建一个分支线程对象
4           MyThread t = new MyThread();
5           //t.run(); // 不会启动线程，不会分配新的分支栈。（这种方式就是单
    线程。）
6           // start()方法的作用是：启动一个分支线程，在JVM中开辟一个新的栈空
    间，这段代码任务完成之后，瞬间就结束了。
```

```
        // 这段代码的任务只是为了开启一个新的栈空间，只要新的栈空间开出来，start()方法就结束了。线程就启动成功了。
        t.start();

        // 这里的代码还是运行在主线程中。
        for(int i = 0; i < 1000; i++){
            System.out.println("主线程--->" + i);
        }
    }
}
//定义线程类
class MyThread extends Thread {
    @Override
    public void run() {
        // 编写程序，这段程序运行在分支线程中（分支栈）。
        for(int i = 0; i < 1000; i++){
            System.out.println("分支线程--->" + i);
        }
    }
}
//注：上面程序主线程和分支线程同时执行，输出结果谁在前谁在后不确定
```

- 实现线程的第二种方式，编写一个类实现java.lang.Runnable接口,重写run()方法。(常用，因为实现了接口还可以继承别的类)
- 调用有参构造方法Thread(Runnable r),将可运行的对象封装成一个线程对象

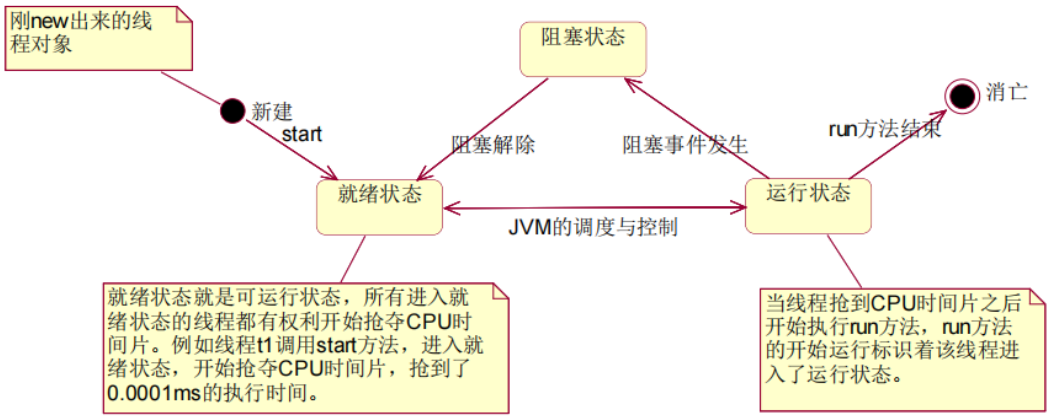
```
public class ThreadTest03 {
    public static void main(String[] args) {
        // 创建一个可运行的对象
        //MyRunnable r = new MyRunnable();
        // 将可运行的对象封装成一个线程对象
        //Thread t = new Thread(r);

        Thread t = new Thread(new MyRunnable()); // 合并代码
        // 启动线程
        t.start();

        for(int i = 0; i < 100; i++){
            System.out.println("主线程--->" + i);
        }
    }
}
// 这并不是一个线程类，是一个可运行的类。它还不是一个线程。
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for(int i = 0; i < 100; i++){
            System.out.println("分支线程--->" + i);
        }
```

```
    }
}
```

```
//采用匿名内部类
public class ThreadTest03 {
    public static void main(String[] args) {
        Thread t = new Thread(new Runnable(){
                @Override
                public void run() {
                    for(int i = 0; i < 100; i++){
                        System.out.println("t线程---> " + i);
                    }
                }
            });

        // 启动线程
        t.start();
    }
```

22.2 线程的生命周期

- 五个状态：新建、就绪、运行、阻塞、死亡



- 新建：采用new创建线程对象
- 执行start后，就进入了就绪状态，开始抢夺CPU时间片
- run()方法的执行标志着线程进入了运行状态。时间片用完后，会再进入就绪状态进行抢夺。
- 当一个线程遇到阻塞事件，例如接受用户输入，或者sleep方法/join()方法等，此时线程会进入阻塞状态，阻塞状态的线程会放弃之前占有的CPU时间片。
- 阻塞状态解除需要再次回到就绪状态抢夺CPU时间片。
- run()方法结束，表示线程死亡

22.3 获取线程名字

- 获取当前线程对象：Thread t = Thread.currentThread();返回值t就是当前线程。（静态方法）
- 获取线程对象的名字：String name = 线程对象.getName();
- 修改线程对象的名字： 线程对象.setName("线程名字");
- 当线程没有设置名字的时候，默认的名字为 Thread-0、Thread-1、Thread-2

```
public class ThreadTest05 {
    public static void main(String[] args) {

        //t就是当前线程对象
        // 这个代码出现在main方法当中，所以当前线程就是主线程。
        Thread t = Thread.currentThread();
        System.out.println(t.getName());  //main

        // 创建线程对象
        MyThread2 t = new MyThread2();
        // 设置线程的名字
        t.setName("t1");
        // 获取线程的名字
        String tName = t.getName();
        System.out.println(tName);

        MyThread2 t2 = new MyThread2();
        t2.setName("t2");
        System.out.println(t2.getName());
        t2.start();

        // 启动线程
        t.start();
    }
}

class MyThread2 extends Thread {
    public void run(){
        for(int i = 0; i < 100; i++){
            // currentThread就是当前线程对象。当前线程是谁呢？
            // 当t1线程执行run方法，那么这个当前线程就是t1
            // 当t2线程执行run方法，那么这个当前线程就是t2
            Thread currentThread = Thread.currentThread();
            System.out.println(currentThread.getName() + "-->" +
i);
        }
    }
}
```

22.4 sleep(静态方法)

- sleep()方法：static void sleep(long millis)----静态方法，参数是毫秒，作用：让当前线程进入休眠，进入“阻塞状态”，放弃占有CPU时间片，让给其它线程使用
- 使用sleep方法要捕捉异常
- Thread.sleep()方法可以间隔特定的时间，去执行一段特定的代码，每隔多久执行一次。
- 注意，sleep()是让当前线程进入休眠，和谁调用无关，静态方法，只看出现的位置。

```
public class ThreadTest06 {
    public static void main(String[] args) {
        // 让当前线程进入休眠，睡眠5秒，当前线程是主线程！！！
        try {
            Thread.sleep(1000 * 5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 5秒之后执行这里的代码
        System.out.println("hello world!");
    }
}
```

- 重点：run()当中的异常不能throws，只能try catch。因为run()方法在父类中没有抛出任何异常，子类不能比父类抛出更多的异常。
- 中断线程的睡眠-----t.interrupt();这种终断睡眠的方式依靠了java的异常处理机制，run中的sleep会报异常，捕捉之后就继续往下执行代码了。

```
public class ThreadTest08 {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable2());
        t.setName("t");
        t.start();
        // 希望5秒之后，t线程醒来（5秒之后主线程手里的活儿干完了。）
        try {
            Thread.sleep(1000 * 5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 终断t线程的睡眠（这种终断睡眠的方式依靠了java的异常处理机制。）
        t.interrupt(); // 干扰，一盆冷水过去！
    }
}

class MyRunnable2 implements Runnable {
    // 重点：run()当中的异常不能throws，只能try catch
    // 因为run()方法在父类中没有抛出任何异常，子类不能比父类抛出更多的异常。
    @Override
```

```
    public void run() {
        System.out.println(Thread.currentThread().getName() + "---> begin");
        try {
            // 睡眠1年
            Thread.sleep(1000 * 60 * 60 * 24 * 365);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //1年之后才会执行这里
        System.out.println(Thread.currentThread().getName() + "---> end");
    }
```

22.4.1 终止线程

```
在java中怎么强行终止一个线程的执行。----t.stop(); // 已过时（不建议使用。）
    这种方式存在很大的缺点：容易丢失数据。因为这种方式是直接将线程杀死了，线程没有保存的数据将会丢失。不建议使用。
```

- 合理终止线程----打布尔标记

```
public class ThreadTest10 {
    public static void main(String[] args) {
        MyRunable4 r = new MyRunable4();
        Thread t = new Thread(r);
        t.setName("t");
        t.start();
        // 模拟5秒
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 终止线程,你想要什么时候终止t的执行，那么你把标记修改为false，就结束了。
        r.run = false;
    }
}
class MyRunable4 implements Runnable {

    // 打一个布尔标记
    boolean run = true;

    @Override
    public void run() {
        for (int i = 0; i < 10; i++){
```

```
            if(run){

 System.out.println(Thread.currentThread().getName() + "--->" +
i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }else{
                //终止当前线程
                return;
            }
        }
    }
}

```

22.4 线程调度

- 常见的线程调度模型：
  抢占式调度模型：那个线程的优先级比较高，抢到的CPU时间片的概率就高一些/多一些。java采用的就是抢占式调度模型。
  均分式调度模型：平均分配CPU时间片。每个线程占有的CPU时间片时间长度一样。
- 实例方法：
  void setPriority(int newPriority) 设置线程的优先级
  int getPriority() 获取线程优先级-----最低优先级1;默认优先级是5;最高优先级10
- 静态方法：
  static void yield() ------让位方法，yield()方法不是阻塞方法。让当前线程让位，让给其它线程使用。yield()方法的执行会让当前线程从“运行状态”回到“就绪状态”。注意：在回到就绪之后，有可能还会再次抢到。Thread.yield()
- 实例方法：void join() -------合并线程（会进入阻塞），栈并不会消失

```
class MyThread1 extends Thread {
    public void doSome(){
        MyThread2 t = new MyThread2();
        t.join(); // 当前线程进入阻塞，t线程执行，直到t线程结束。当前线程才可以继续。
    }
}
class MyThread2 extends Thread{
}
```

22.5 线程安全(重点)

- 什么时候数据在多线程并发的环境下会存在安全问题呢？
  条件1：多线程并发。
  条件2：有共享数据。
  条件3：共享数据有修改的行为。满足以上3个条件之后，就会存在线程安全问题。
- 怎么解决线程安全问题呢？
  使用“线程同步机制”，实际上就是线程不能并发了，线程必须排队执行。
- 异步编程模型：线程t1和线程t2，各自执行各自的，t1不管t2，t2不管t1，谁也不需要等谁，这种编程模型叫做：异步编程模型。其实就是：多线程并发（效率较高。）
  同步编程模型：线程t1和线程t2，在线程t1执行的时候，必须等待t2线程执行结束，或者说在t2线程执行的时候，必须等待t1线程执行结束，两个线程之间发生了等待关系，这就是同步编程模型。（效率较低）
- 异步就是并发。同步就是排队。
- 重点：synchronized() 括号里面传的：必是需要排队执行的线程所共享的。

```
线程同步机制的语法是：
            synchronized(){
                // 线程同步代码块。
            }
    synchronized后面小括号中传的这个“数据”是相当关键的。这个数据必须是多线程共享的数据。才能达到多线程排队。
        ()中写什么？
        那要看你想让哪些线程同步。假设t1、t2、t3、t4、t5，有5个线程，你只希望t1 t2 t3排队，t4 t5不需要排队。怎么办？
        你一定要在()中写一个t1 t2 t3共享的对象。而这个对象对于t4 t5来说不是共享的。

            在java语言中，任何一个对象都有“一把锁”，其实这把锁就是标记。（只是把它叫做锁。）
            100个对象，100把锁。1个对象1把锁。

            以下代码的执行原理？
                1、假设t1和t2线程并发，开始执行以下代码的时候，肯定有一个先一个后。
                2、假设t1先执行了，遇到了synchronized，这个时候自动找“后面共享对象”的对象锁，
                找到之后，并占有这把锁，然后执行同步代码块中的程序，在程序执行过程中一直都是
                占有这把锁的。直到同步代码块代码结束，这把锁才会释放。
                3、假设t1已经占有这把锁，此时t2也遇到synchronized关键字，也会去占有后面
                共享对象的这把锁，结果这把锁被t1占有，t2只能在同步代码块外面等待t1的结束，
                直到t1把同步代码块执行结束了，t1会归还这把锁，此时t2终于等到这把锁，然后
                t2占有这把锁之后，进入同步代码块执行程序。
```

```
22
23                  这样就达到了线程排队执行。
24                  这里需要注意的是：这个共享对象一定要选好了。这个共享对象一
定是你需要排队
25                  执行的这些线程对象所共享的。
26  public class AccountThread extends Thread {
27
28      // 两个线程必须共享同一个账户对象。
29      private Account act;
30      // 通过构造方法传递过来账户对象
31      public AccountThread(Account act) {
32          this.act = act;
33      }
34      public void run(){
35          double money = 5000;
36          act.withdraw(money);
37      }
38  }
39  public class Account {
40          // 账号
41           private String actno;
42           // 余额
43          private double balance; //实例变量。
44           //对象
45          Object obj = new Object();
46
47     //取款的方法
48      public void withdraw(double money){
49          synchronized (this){          --t1和t2共享账户对象
50          //synchronized (obj) {        ----这么写也可以
51          //synchronized ("abc") {      ----这么写也可以 ，"abc"在字符串
常量池当中。但这么写所有线程都会同步
52              double before = this.getBalance();
53              double after = before - money;
54              try {
55                  Thread.sleep(1000);
56               } catch (InterruptedException e) {
57                  e.printStackTrace();
58              }
59              this.setBalance(after);
60          }
61  }
62  public class Test {
63      public static void main(String[] args) {
64          // 创建账户对象（只创建1个）
65          Account act = new Account("act-001", 10000);
66          // 创建两个线程
67          Thread t1 = new AccountThread(act);
68          Thread t2 = new AccountThread(act);
```

```
69
70          // 设置name
71          t1.setName("t1");
72          t2.setName("t2");
73          // 启动线程取款
74          t1.start();
75          t2.start();
76      }
77  }
```

- 线程在运行状态下遇到synchronized的关键字，会进入锁池，在这里找共享对象的对象锁，并且会释放之前占有的CPU时间片。有可能找到，有可能没找到，没找到就在锁池中等待，如果找到了就会进入就绪状态继续抢夺CPU时间片
- 在实例方法上可以使用synchronized，锁的是this

```
1           synchronized出现在实例方法上，一定锁的是this。
2           没得挑。只能是this。不能是其他的对象了。
3           所以这种方式不灵活。
4
5           另外还有一个缺点：synchronized出现在实例方法上，
6           表示整个方法体都需要同步，可能会无故扩大同步的范围，导致程序的执行效率降低。所以这种方式不常用。
7           synchronized使用在实例方法上有什么优点？：代码写的少了。节俭了。
8
9           如果共享的对象就是this，并且需要同步的代码块是整个方法体，建议使用这种方式。
10      public synchronized void withdraw(double money){
11          double before = this.getBalance(); // 10000
12          double after = before - money;
13          try {
14              Thread.sleep(1000);
15          } catch (InterruptedException e) {
16              e.printStackTrace();
17          }
18          this.setBalance(after);
```

- 实例变量：在堆中;静态变量：在方法区;局部变量：在栈中。以上三大变量中：局部变量永远都不会存在线程安全问题。因为局部变量不共享。（一个线程一个栈。）局部变量在栈中。所以局部变量永远都不会共享。

  实例变量在堆中，堆只有1个。静态变量在方法区中，方法区只有1个。堆和方法区都是多线程共享的，所以可能存在线程安全问题。
- 局部变量+常量：不会有线程安全问题。成员变量：可能会有线程安全问题。
- 在实例方法上使用synchronized，表示锁的对象是this,并且同步代码块是整个方法体。一个对象一把锁
- 在静态方法上使用synchronized，表示找类锁。类锁永远只有1把。
- 对象锁：1个对象1把锁，100个对象100把锁。类锁：100个对象，也可能只是1把类锁。

```
4、如果使用局部变量的话：建议使用：StringBuilder。
    因为局部变量不存在线程安全问题。选择StringBuilder。
    StringBuffer是线程安全的，但效率比较低。

    ArrayList是非线程安全的。
    Vector是线程安全的。
    HashMap HashSet是非线程安全的。
    Hashtable是线程安全的。

5、总结：
    synchronized有三种写法：

        第一种：同步代码块
            灵活
            synchronized(线程共享对象){
                同步代码块;
            }

        第二种：在实例方法上使用synchronized
            表示共享对象一定是this,并且同步代码块是整个方法体。

        第三种：在静态方法上使用synchronized
            表示找类锁。类锁永远只有1把。
            就算创建了100个对象，那类锁也只有一把。类锁为了保证静态变量的安全

        对象锁：1个对象1把锁，100个对象100把锁。类锁：100个对象，也可能只是1把类锁。

6、聊一聊，我们以后开发中应该怎么解决线程安全问题？

    是一上来就选择线程同步吗？synchronized
        不是，synchronized会让程序的执行效率降低，用户体验不好。系统的用户吞吐量降低。用户体验差。在不得已的情况下再选择线程同步机制。

    第一种方案：尽量使用局部变量代替“实例变量和静态变量”。

    第二种方案：如果必须是实例变量，那么可以考虑创建多个对象，这样实例变量的内存就不共享了。（一个线程对应1个对象，100个线程对应100个对象，对象不共享，就没有数据安全问题了。）

    第三种方案：如果不能使用局部变量，对象也不能创建多个，这个时候就只能选择synchronized了。线程同步机制。
```

22.6 死锁

- synchronized在开发中最好不要嵌套使用，容易造成死锁

```
```

```java
public class DeadLock {
    public static void main(String[] args) {
        Object o1 = new Object();
        Object o2 = new Object();

        // t1和t2两个线程共享o1,o2
        Thread t1 = new MyThread1(o1,o2);
        Thread t2 = new MyThread2(o1,o2);

        t1.start();
        t2.start();
    }
}

class MyThread1 extends Thread{
    Object o1;
    Object o2;
    public MyThread1(Object o1,Object o2){
        this.o1 = o1;
        this.o2 = o2;
    }
    public void run(){
        synchronized (o1){
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (o2){

            }
        }
    }
}

class MyThread2 extends Thread {
    Object o1;
    Object o2;
    public MyThread2(Object o1,Object o2){
        this.o1 = o1;
        this.o2 = o2;
    }
    public void run(){
        synchronized (o2){
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
            synchronized (o1){

            }
        }
    }
}
```

22.7 守护线程

- java语言中线程分为两大类：一类是：用户线程；一类是：守护线程（后台线程）。其中具有代表性的就是：垃圾回收线程（守护线程）。
- 守护线程的特点：一般守护线程是一个死循环，所有的用户线程只要结束，守护线程自动结束。
  注意：主线程main方法是一个用户线程。
- 将一个线程设置为守护线程：t.setDaemon(true);

```
public class ThreadTest14 {
    public static void main(String[] args) {
        Thread t = new BakDataThread();
        t.setName("备份数据的线程");

        // 启动线程之前，将线程设置为守护线程
        t.setDaemon(true);

        t.start();

        // 主线程：主线程是用户线程
        for(int i = 0; i < 10; i++){
            System.out.println(Thread.currentThread().getName() + "--->" + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class BakDataThread extends Thread {
    public void run(){
        int i = 0;
        // 即使是死循环，但由于该线程是守护者，当用户线程结束，守护线程自动终止。
        while(true){
            System.out.println(Thread.currentThread().getName() + "--->" + (++i));
            try {
```

```
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

22.8 定时器

- 定时器的作用：间隔特定的时间，执行特定的程序。
- 在java的类库中已经写好了一个定时器：java.util.Timer
- 无参构造：Timer() 创建一个新的计时器。
- void schedule(TimerTask task, Date firstTime, long period) -----定时任务, 第一次执行时间, 间隔多久执行一次
- 子类去继承TimerTask 这个类，需要重写run方法，在run中写入要执行的任务

```
public class TimerTest {
    public static void main(String[] args) throws Exception {

        // 创建定时器对象
        Timer timer = new Timer();
        //Timer timer = new Timer(true); //守护线程的方式

        // 指定定时任务
        //timer.schedule(定时任务, 第一次执行时间, 间隔多久执行一次);
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        Date firstTime = sdf.parse("2020-03-14 09:34:30");
        timer.schedule(new LogTimerTask() , firstTime, 1000 * 10);

        //匿名内部类方式
        timer.schedule(new TimerTask(){
            @Override
            public void run() {
                // code....
            }
        } , firstTime, 1000 * 10);

    }
}
// 编写一个定时任务类
// 假设这是一个记录日志的定时任务
class LogTimerTask extends TimerTask {

    @Override
    public void run() {
        // 编写你需要执行的任务就行了。
```

```
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String strTime = sdf.format(new Date());
        System.out.println(strTime + ":成功完成了一次数据备份！");
    }
}
```

22.9 线程创建的第三种方式

- 实现线程的第三种方式：实现Callable接口。这种方式实现的线程可以获取线程的返回值。之前讲解的那两种方式是无法获取线程返回值的，因为run方法返回void。
- 首先创建一个“未来任务类”对象，FutureTask，然后创建线程采用有参构造方法
- 采用get()方法获取其他线程的返回值。
- 优点：可以获取到线程的执行结果。
  缺点：效率比较低，在获取t线程执行结果的时候，当前线程受阻塞，效率较低。

```
public class ThreadTest15 {
    public static void main(String[] args) throws Exception {

        // 第一步：创建一个“未来任务类”对象。
        // 参数非常重要，需要给一个Callable接口实现类对象,采用匿名内部类
        FutureTask task = new FutureTask(new Callable() {
            @Override
            public Object call() throws Exception {  // call()方法就相当于run方法。只不过这个有返回值
                // 线程执行一个任务，执行之后可能会有一个执行结果
                // 模拟执行
                System.out.println("call method begin");
                Thread.sleep(1000 * 10);
                System.out.println("call method end!");
                int a = 100;
                int b = 200;
                return a + b; //自动装箱(300结果变成Integer)
            }
        });

        // 创建线程对象
        Thread t = new Thread(task);
        // 启动线程
        t.start();

        // 这里是main方法，这是在主线程中。
        // get()方法的执行会导致“当前线程阻塞”
        Object obj = task.get();
        System.out.println("线程执行结果:" + obj);

        // main方法这里的程序要想执行必须等待get()方法的结束
```

```
        // 而get()方法可能需要很久。因为get()方法是为了拿另一个线程的执行结果， 另一个线程执行是需要时间的。
        System.out.println("hello world!");
    }
}
```

22.10 wait和notify方法

- 使用wait方法和notify方法实现“生产者和消费者模式”-----生产线程负责生产，消费线程负责消费。生产线程和消费线程要达到均衡。这是一种特殊的业务需求，在这种特殊的情况下需要使用wait方法和notify方法。
- wait和notify方法不是线程对象的方法，是java中任何一个java对象都有的方法，因为这两个方式是Object类中自带的。wait方法和notify方法不是通过线程对象调用，
- wait和notify方法建立在synchronized线程同步的基础上；因为多线程要同时操作一个仓库。有线程安全问题。

```
wait()方法作用？
    Object o = new Object();
    o.wait();
表示：
    o.wait()让正在o对象上活动的线程t进入等待状态，并且释放掉t线程之前占有的o对象的锁。
    o.wait();方法的调用，会让“当前线程（正在o对象上活动的线程）”进入等待状态。

notify()方法作用？
    Object o = new Object();
    o.notify();
表示：o.notify()让正在o对象上等待的线程唤醒，只是通知，不会释放o对象上之前占有的锁。

还有一个notifyAll()方法：这个方法是唤醒o对象上处于等待的所有线程。
```

```
/**
 * 两个线程交替输出，t1输出奇数，t2输出偶数
 */
public class WaitNotify {
    public static void main(String[] args) {
        //创建一个数字对象
        Num num=new Num(1);
        //创建两个线程对象，共享数字
        Thread1 t1=new Thread1(num);
        Thread2 t2=new Thread2(num);

        t1.setName("t1");
        t2.setName("t2");
```

```
14
15          t1.start();
16          t2.start();
17      }
18  }
19  //定义数字类
20  class Num{
21      int i;
22
23      public Num(int i) {
24          this.i = i;
25      }
26  }
27  //创两个线程对象
28  class Thread1 extends Thread{
29      Num num;
30
31      public Thread1(Num num) {
32          this.num = num;
33      }
34      @Override
35      public void run() {
36          //输出奇数
37          while(true){
38              synchronized (num){
39                  if(num.i%2==0){
40                      try {
41                          num.wait();    //当数是偶数时，当前t1线程等待，
    释放锁，t2线程执行
42                      } catch (InterruptedException e) {
43                          e.printStackTrace();
44                      }
45                  }
46                  try {
47                      Thread.sleep(1000);
48                  } catch (InterruptedException e) {
49                      e.printStackTrace();
50                  }
51
     System.out.println(Thread.currentThread().getName()+"-->"+num.i);
52                  num.i++;
53                  num.notify();       //t1执行结束，唤醒t2
54              }
55          }
56      }
57  }
58  class Thread2 extends Thread{
59      Num num;
60
```

```
    public Thread2(Num num) {
        this.num = num;
    }

    @Override
    public void run() {
        //输出偶数
        while(true){
            synchronized (num){
                if(num.i%2==1){
                    try {
                        num.wait(); //当数是奇数时，当前t2线程等待，释放锁，t1线程执行
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

 System.out.println(Thread.currentThread().getName()+"-->"+num.i);
                num.i++;
                num.notify();
            }
        }
    }
}
```

二十三、反射机制

- 反射机制的相关类在哪个包下？java.lang.reflect.*;
- 作用：通过java语言中的反射机制可以操作字节码文件，让程序更加灵活
- 反射机制相关的重要的类有哪些？
  java.lang.Class：代表整个字节码，代表一个类型，代表整个类。
  java.lang.reflect.Method：代表字节码中的方法字节码。代表类中的方法。
  java.lang.reflect.Constructor：代表字节码中的构造方法字节码。代表类中的构造方法
  java.lang.reflect.Field：代表字节码中的属性字节码。代表类中的成员变量（静态变量+实例变量）。
- 要操作一个类的字节码，需要首先获取到这个类的字节码，怎么获取java.lang.Class实例？
  三种方式:第一种：Class c = Class.forName("完整类名带包名");--静态方法，方法的参数是一个字符串

第二种：Class c = 对象.getClass();

第三种：Class c = 任何类型.class;

```
public class ReflectTest01 {
    public static void main(String[] args) {
        1、/*
        Class.forName()
            1、静态方法
            2、方法的参数是一个字符串。
            3、字符串需要的是一个完整类名。
            4、完整类名必须带有包名。java.lang包也不能省略。
         */
        Class c1 = null;
        Class c2 = null;
        try {
            c1 = Class.forName("java.lang.String"); // c1代表String.class文件，或者说c1代表String类型。
            c2 = Class.forName("java.util.Date"); // c2代表Date类型
            Class c3 = Class.forName("java.lang.Integer"); // c3代表Integer类型
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        2、// java中任何一个对象都有一个方法：getClass()
        String s = "abc";
        Class x = s.getClass(); // x代表String.class字节码文件，x代表String类型。
        System.out.println(c1 == x); // true（==判断的是对象的内存地址。）

        Date time = new Date();
        Class y = time.getClass();
        System.out.println(c2 == y); // true (c2和y两个变量中保存的内存地址都是一样的，都指向方法区中的字节码文件。)

        3、// 第三种方式，java语言中任何一种类型，包括基本数据类型，它都有.class属性。
        Class z = String.class; // z代表String类型
        Class k = Date.class; // k代表Date类型
        Class f = int.class; // f代表int类型
        Class e = double.class; // e代表double类型

        System.out.println(x == z); // true

    }
}
```

23.1 实例化对象

- 获取到Class，能干什么？通过Class的newInstance()方法来实例化对象。
- 注意：newInstance()方法内部实际上调用了无参数构造方法，必须保证无参构造存在才可以。
- 可以将类名写入properties配置文件，通过IO流来读取类名，通过Class来创建对象，更加的灵活

```
public class ReflectTest02 {
    public static void main(String[] args) {

        // 这是不使用反射机制，创建对象
        User user = new User();
        System.out.println(user);

        // 下面这段代码是以反射机制的方式创建对象。
        try {
            // 通过反射机制，获取Class，通过Class来实例化对象
            Class c =
Class.forName("com.bjpowernode.java.bean.User"); // c代表User类型。
            Object obj = c.newInstance();

            System.out.println(obj);    //
com.bjpowernode.java.bean.User@10f87f48
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        }
    }
}
```

- Class.forName(类名)--这个方法执行，会进行类加载，此时类中的静态代码块会执行。
- 结论：如果希望类的静态代码块执行，而其他代码不执行的话，可以采用Class.forName的方式。

```
public class ReflectTest04 {
    public static void main(String[] args) {
        try {
            // Class.forName()这个方法的执行会导致：类加载。
            Class.forName("com.bjpowernode.java.reflect.MyClass");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
```

```
    }
}
public class MyClass {
    // 静态代码块在类加载时执行，并且只执行一次。
    static {
        System.out.println("MyClass类的静态代码块执行了！");
    }
}
```

23.2 文件路径

- 以下讲解的这种方式是通用的。但前提是：文件需要在类路径下才能用这种方式。src是类的根路径。
- String path = Thread.currentThread().getContextClassLoader().getResource("文件名").getPath();
- 采用以上的代码可以拿到一个文件的绝对路径，但注意文件名必须写从src下的路径（相对src的路径）

```
/*解释：
   Thread.currentThread() 当前线程对象
   getContextClassLoader() 是线程对象的方法，可以获取到当前线程的类加载器对象。
   getResource() 【获取资源】这是类加载器对象的方法，当前线程的类加载器默认从类的根路径下加载资源。*/

        // 获取db.properties文件的绝对路径（从类的根路径下作为起点开始）
        String path2 = Thread.currentThread().getContextClassLoader()

.getResource("com/bjpowernode/java/bean/db.properties").getPath();
// 这种方式获取文件绝对路径是通用的。
        System.out.println(path2);

        // 直接以流的形式返回。
        InputStream reader = Thread.currentThread().getContextClassLoader()
                .getResourceAsStream("classinfo2.properties");
        Properties pro = new Properties();
        pro.load(reader);
        reader.close();

```

23.3 资源绑定器

- java.util包下提供了一个资源绑定器，便于获取属性配置文件中的内容。

- 资源绑定器，只能绑定xxx.properties文件。并且这个文件必须在类路径下(以src为起点)。文件扩展名也必须是properties，在写路径的时候，路径后面的扩展名不能写。
- 使用以下这种方式的时候，属性配置文件xxx.properties必须放到类路径下。
- 代替了IO-Properties 的方法

```
public class ResourceBundleTest {
    public static void main(String[] args) {

        // 资源绑定器，只能绑定xxx.properties文件。并且这个文件必须在类路径下。文件扩展名也必须是properties
        // 并且在写路径的时候，路径后面的扩展名不能写。
        //ResourceBundle bundle = ResourceBundle.getBundle("classinfo2");

        ResourceBundle bundle = ResourceBundle.getBundle("com/bjpowernode/java/bean/db");

        String className = bundle.getString("className");  //获取配置文件中的类名
        System.out.println(className);
    }
}
```

23.4 类加载器

```
3、关于JDK中自带的类加载器：（聊一聊，不需要掌握，知道当然最好！）
    3.1、什么是类加载器？-----专门负责加载类的命令/工具。
        ClassLoader

    3.2、JDK中自带了3个类加载器
        启动类加载器:rt.jar
        扩展类加载器:ext/*.jar
        应用类加载器:classpath

    3.3、假设有这样一段代码：
        String s = "abc";

        代码在开始执行之前，会将所需要类全部加载到JVM当中。通过类加载器加载，看到以上代码类加载器会找String.class文件，找到就加载，那么是怎么进行加载的呢？

        首先通过“启动类加载器”加载。
            注意：启动类加载器专门加载：C:\Program Files\Java\jdk1.8.0_101\jre\lib\rt.jar
                rt.jar中都是JDK最核心的类库。

        如果通过“启动类加载器”加载不到的时候，会通过"扩展类加载器"加载。
```

```
            注意：扩展类加载器专门加载：C:\Program Files\Java\jdk1.8.0_101\jre\lib\ext\*.jar

        如果“扩展类加载器”没有加载到，那么会通过“应用类加载器”加载。
            注意：应用类加载器专门加载：classpath中的类。

    3.4、java中为了保证类加载的安全，使用了双亲委派机制。
        优先从启动类加载器中加载，这个称为“父”。“父”无法加载到，再从扩展类加载器中加载，这个称为“母”。双亲委派。如果都加载不到，才会考虑从应用类加载器中加载。直到加载到为止。
```

23.5 类Field

- Field翻译为字段，其实就是属性/成员。通过字节码文件，去获取类中的成员变量
- Class类中的方法：
- getName()-----获取完整类名，返回一个字符串
  getSimpleName()----获取简类名，返回一个字符串
  getFields()-----获取类中所有的public修饰的成员变量，返回一个Field[] 数组
  getDeclaredFields();-------获取类中所有的成员变量，返回一个Field[] 数组
- Field类中方法：
  field.getModifiers()-----获取属性的修饰符列表，返回的修饰符是一个数字，每个数字是修饰符的代号
  Modifier.toString(i)----静态方法，将这个“代号”数字转换成“字符串”
  field.getType();-----获取属性的类型，返回一个Class，再调用Class中getSimpleName()获取类型名字
  field.getName()----- 获取属性的名字

```java
public class Student {
    // 5个Field，分别采用了不同的访问控制权限修饰符
    private String name; // Field对象
    protected int age; // Field对象
    boolean sex;
    public int no;
    public static final double MATH_PI = 3.1415926;
}
public class ReflectTest05 {
    public static void main(String[] args) throws Exception{

        // 获取整个类
        Class studentClass = Class.forName("com.bjpowernode.java.bean.Student");

        //com.bjpowernode.java.bean.Student
        String className = studentClass.getName();
        System.out.println("完整类名：" + className);

```

```
        String simpleName = studentClass.getSimpleName();
        System.out.println("简类名: " + simpleName);

        // 获取类中所有的public修饰的Field
        Field[] fields = studentClass.getFields();
        System.out.println(fields.length); // 测试数组中只有1个元素
        // 取出这个Field
        Field f = fields[0];
        // 取出这个Field它的名字
        String fieldName = f.getName();
        System.out.println(fieldName);

        // 获取所有的Field
        Field[] fs = studentClass.getDeclaredFields();
        System.out.println(fs.length); // 4

        System.out.println("==================================");
        // 遍历
        for(Field field : fs){
            // 获取属性的修饰符列表
            int i = field.getModifiers(); // 返回的修饰符是一个数字,每个数字是修饰符的代号!!!
            System.out.println(i);
            // 可以将这个“代号”数字转换成“字符串”吗?
            String modifierString = Modifier.toString(i);
            System.out.println(modifierString);
            // 获取属性的类型
            Class fieldType = field.getType();
            //String fName = fieldType.getName();
            String fName = fieldType.getSimpleName();
            System.out.println(fName);
            // 获取属性的名字
            System.out.println(field.getName());
        }
    }
}
```

23.6 反编译(了解)

- 通过字节码文件去获得java源码中成员变量

```
public class ReflectTest06 {
    public static void main(String[] args) throws Exception{

        // 创建这个是为了拼接字符串。
        StringBuilder s = new StringBuilder();

        //Class studentClass = Class.forName("com.bjpowernode.java.bean.Student");
```

```
        Class studentClass = Class.forName("java.lang.Thread");

        s.append(Modifier.toString(studentClass.getModifiers()) +
" class " + studentClass.getSimpleName() + " {\n");

        Field[] fields = studentClass.getDeclaredFields();
        for(Field field : fields){
            s.append("\t");
            s.append(Modifier.toString(field.getModifiers()));
            s.append(" ");
            s.append(field.getType().getSimpleName());
            s.append(" ");
            s.append(field.getName());
            s.append(";\n");
        }

        s.append("}");
        System.out.println(s);
    }
}
```

23.7 访问对象属性

- 必须掌握：怎么通过反射机制访问一个java对象的属性？
  给属性赋值set
  获取属性的值get
- getDeclaredField(String name)-----获取类中一个成员变量,返回一个field对象
  注：上面方法是Class类中的方法

```
public class ReflectTest07 {
    public static void main(String[] args) throws Exception{

        // 我们不使用反射机制，怎么去访问一个对象的属性呢？
        Student s = new Student();

        // 给属性赋值
        s.no = 1111; //三要素：给s对象的no属性赋值1111
                     //要素1：对象s
                     //要素2：no属性
                     //要素3：1111

        // 读属性值
        // 两个要素：获取s对象的no属性的值。
        System.out.println(s.no);

        // 使用反射机制，怎么去访问一个对象的属性。（set get）
        Class studentClass =
Class.forName("com.bjpowernode.java.bean.Student");
```

```
        Object obj = studentClass.newInstance(); // obj就是Student对象。（底层调用无参数构造方法）

        // 获取no属性（根据属性的名称来获取Field）
        Field noFiled = studentClass.getDeclaredField("no");

        // 给obj对象(Student对象)的no属性赋值
        /*
        虽然使用了反射机制，但是三要素还是缺一不可：
            要素1：obj对象
            要素2：no属性
            要素3：2222值
        注意：反射机制让代码复杂了，但是为了一个“灵活”，这也是值得的。
         */
        noFiled.set(obj, 22222); // 给obj对象的no属性赋值2222

        // 读取属性的值
        // 两个要素：获取obj对象的no属性的值。
        System.out.println(noFiled.get(obj));

        // 可以访问私有的属性吗？
        Field nameField = studentClass.getDeclaredField("name");

        // 打破封装（反射机制的缺点：打破封装，可能会给不法分子留下机会！！！）
        // 这样设置完之后，在外部也是可以访问private的。
        nameField.setAccessible(true);

        // 给name属性赋值
        nameField.set(obj, "jackson");
        // 获取name属性的值
        System.out.println(nameField.get(obj));
    }
}

```

23.8 Method

- 可变长度参数,语法是：类型... args（注意：一定是3个点。）
- 可变长度参数要求的参数个数是：0~N个。
- 可变长度参数在参数列表中必须在最后一个位置上，而且可变长度参数只能有1个。
- 可变长度参数可以当做一个数组来看待

```
public class ArgsTest {
    public static void main(String[] args) {
        m2(200, "abc");
        m2(200, "abc", "def");

```

```
        m3("ab", "de", "kk", "ff");
        // 也可以传1个数组
        String[] strs = {"a","b","c"};
        m3(strs);

        // 直接传1个数组
        m3(new String[]{"我","是","中","国", "人"}); //没必要
        m3("我","是","中","国", "人");
    }

    //public static void m2(int... args2, String... args1){}    //错误

    // 必须在最后，只能有1个。
    public static void m2(int a, String... args1){

    }

    public static void m3(String... args){
        //args有length属性，说明args是一个数组!
        // 可以将可变长度参数当做一个数组来看。
        for(int i = 0; i < args.length; i++){
            System.out.println(args[i]);
        }
    }

}
```

- 反射Method---作为了解内容（不需要掌握）：

```
public class ReflectTest08 {
    public static void main(String[] args) throws Exception{

        // 获取类了
        Class userServiceClass = Class.forName("com.bjpowernode.java.service.UserService");

        // 获取所有的Method（包括私有的！）
        Method[] methods = userServiceClass.getDeclaredMethods();
        //System.out.println(methods.length); // 2

        // 遍历Method
        for(Method method : methods){
            // 获取修饰符列表
            System.out.println(Modifier.toString(method.getModifiers()));
            // 获取方法的返回值类型
```

```
 System.out.println(method.getReturnType().getSimpleName());
            // 获取方法名
            System.out.println(method.getName());
            // 方法的修饰符列表（一个方法的参数可能会有多个。）
            Class[] parameterTypes = method.getParameterTypes();
            for(Class parameterType : parameterTypes){
                System.out.println(parameterType.getSimpleName());
            }
        }
    }
}
```

- 通过反射机制调用对象的方法----必须掌握
- 先获取Class文件，再通过Class去创建对象，然后获取method方法
- Class类中方法：
  getDeclaredMethod(String name, Class<?>... parameterTypes)------获取对象中的一个方法，参数是方法名，参数列表中数据类型对应的Class，返回值是一个Method
- 可能会有数据重载，故需要通过形参列表去区分,注意上面括号中传递的是.class
- Method类中方法：
  invoke(Object obj, Object... args) ----------调用方法，参数是对象名，传递的参数，返回一个Object

```
public class ReflectTest10 {
    public static void main(String[] args) throws Exception{

        // 使用反射机制来调用一个对象的方法
        Class userServiceClass = Class.forName("com.bjpowernode.java.service.UserService");
        // 创建对象
        Object obj = userServiceClass.newInstance();
        // 获取Method方法
        Method loginMethod = userServiceClass.getDeclaredMethod("login", String.class, String.class);
        //Method loginMethod = userServiceClass.getDeclaredMethod("login", int.class);

        // 调用方法
        // 调用方法有几个要素？ 也需要4要素。
        /*
        四要素：
        loginMethod方法
        obj对象
        "admin","123" 实参
        retValue 返回值
         */
```

```
        Object retValue = loginMethod.invoke(obj,
"admin","123123");
        System.out.println(retValue);
    }
}
// 用户业务类
public class UserService {

    public boolean login(String name,String password){
        if("admin".equals(name) && "123".equals(password)){
            return true;
        }
        return false;
    }
    public void login(int i){

    }
    public void logout(){
        System.out.println("系统已经安全退出！");
    }
}
```

23.9 Constructor

- 通过反射机制调用构造方法实例化java对象。（这个不是重点）

```
public class ReflectTest12 {
    public static void main(String[] args) throws Exception{

        // 使用反射机制怎么创建对象呢？
        Class c = Class.forName("com.bjpowernode.java.bean.Vip");
        // 调用无参数构造方法
        Object obj = c.newInstance();
        System.out.println(obj);

        // 调用有参数的构造方法怎么办？
        // 第一步：先获取到这个有参数的构造方法
        Constructor con = c.getDeclaredConstructor(int.class,
String.class, String.class,boolean.class);
        // 第二步：调用构造方法new对象
        Object newObj = con.newInstance(110, "jackson", "1990-10-
11", true);
        System.out.println(newObj);

        // 获取无参数构造方法
        Constructor con2 = c.getDeclaredConstructor();
        Object newObj2 = con2.newInstance();
        System.out.println(newObj2);
    }
```

```
}
public class Vip {
    int no;
    String name;
    String birth;
    boolean sex;

    public Vip() {
    }
    public Vip(int no, String name, String birth, boolean sex) {
        this.no = no;
        this.name = name;
        this.birth = birth;
        this.sex = sex;
    }
}
```

- 重点：给你一个类，怎么获取这个类的父类，已经实现了哪些接口？

```
public class ReflectTest13 {
    public static void main(String[] args) throws Exception{

        // String举例
        Class stringClass = Class.forName("java.lang.String");

        // 获取String的父类
        Class superClass = stringClass.getSuperclass();
        System.out.println(superClass.getName());

        // 获取String类实现的所有接口（一个类可以实现多个接口。）
        Class[] interfaces = stringClass.getInterfaces();
        for(Class in : interfaces){
            System.out.println(in.getName());
        }
    }
}
```

二十四、注解

- 注解，或者叫做注释类型，英文单词是：Annotation
- 注解Annotation是一种引用数据类型。编译之后也是生成xxx.class文件。
- 自定义注解的语法格式：

  [修饰符列表] @interface 注解类型名{

  }
- 注解使用：@注解类型名
- 注解可以出现在类上、属性上、方法上、变量上等....注解还可以出现在注解类型上。

```
JDK内置了哪些注解呢？
        java.lang包下的注释类型：

            掌握：
            Deprecated 用 @Deprecated 注释的程序元素，表示标识的元素已过时
            不鼓励程序员使用这样的元素，通常是因为它很危险或存在更好的选择。

            掌握： Override 表示一个方法声明打算重写超类中的另一个方法声明。

            不用掌握：
            SuppressWarnings 指示应该在注释元素（以及包含在该注释元素中的
            所有程序元素）中取消显示指定的编译器警告。

3.6、元注解
        什么是元注解？----用来标注“注解类型”的“注解”，称为元注解。

        常见的元注解有哪些？
            Target
            Retention

    Target注解：这个Target注解用来标注“被标注的注解”可以出现在哪些位置上。

            @Target(ElementType.METHOD)：表示“被标注的注解”只能出现在方法上。
            @Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, MODULE, PARAMETER, TYPE})
                表示该注解可以出现在：构造方法上、字段上、局部变量上、方法上、类上...

    Retention注解：Retention注解用来标注“被标注的注解”最终保存在哪里。

    @Retention(RetentionPolicy.SOURCE)：表示该注解只被保留在java源文件中。
    @Retention(RetentionPolicy.CLASS)：表示该注解被保存在class文件中。
    @Retention(RetentionPolicy.RUNTIME)：表示该注解被保存在class文件中，并且可以被反射机制所读取。

    Retention的源代码
            public @interface Retention {
                //属性
                RetentionPolicy value();
            }     //RetentionPolicy是枚举类
```

24.1Override

- @Override这个注解只能注解方法。该注解只被保留在java源文件中
- 标识性注解，给编译器做参考的。
- 编译器看到方法上有这个注解的时候，编译器会自动检查该方法是否重写了父类的方法。如果没有重写，报错。
- 这个注解只是在编译阶段起作用，和运行期无关!

```
public class AnnotationTest02 {
    @Override
    public String toString() {
        return "toString";
    }
}
Override 源码:
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

24.2 Deprecated

- Deprecated这个注解标注的元素已过时。
- 该注解被保存在class文件中，并且可以被反射机制所读取。

```
public class AnnotationTest03 {

    @Deprecated
    private String s;

    // 这个注解主要是向其它程序员传达一个信息，告知已过时，有更好的解决方案存在。
    @Deprecated
    public static void doOther(){
        System.out.println("do other...");
    }
}
```

24.3 自定义注解

- 自定义注解可以定义属性
- 如果一个注解当中有属性，那么必须给属性赋值。（除非该属性使用default指定了默认值。）
- 如果一个注解的属性的名字是value，并且只有一个属性的话，在使用的时候，该属性名可以省略。其他参数名不可以省。
- 属性的类型可以是：基本数据类型、String、Class、枚举类型、数组等。

```
public @interface MyAnnotation {
```

```
    String name();
    //颜色属性
    String color();
    //年龄属性
    int age() default 25; //属性指定默认值
}

public class MyAnnotationTest {

    //@MyAnnotation(属性名=属性值,属性名=属性值,属性名=属性值)
    //age指定了默认值，可以不写
    @MyAnnotation(name = "zhangsan", color = "红色")
    public void doSome(){
    }
}

public @interface MyAnnotation11 {
    String value();
}
public class MyAnnotationTest {
    @MyAnnotation11("haha")
    public void doOther(){
    }
}
```

```
public @interface OtherAnnotation {
    int age();
    String[] email();
    /**
     * 季节数组，Season是枚举类型
     * @return
     */
    Season[] seasonArray();
}
public class OtherAnnotationTest {
    // 数组是大括号
    @OtherAnnotation(age = 25, email = {"zhangsan@123.com",
"zhangsan@sohu.com"}, seasonArray = Season.WINTER)
    public void doSome(){
    }

    // 如果数组中只有1个元素：大括号可以省略。
    @OtherAnnotation(age = 25, email = "zhangsan@123.com",
seasonArray = {Season.SPRING, Season.SUMMER})
    public void doOther(){
    }
}
```

24.4 反射机制获取注解

- 前提：元注解是 @Retention(RetentionPolicy.RUNTIME)，可以被反射机制读取
- 获取类-->判断类上有没有注解-->如果有获取注解对象--->获取注解对象的属性

```
public class ReflectAnnotationTest {
    public static void main(String[] args) throws Exception{
        // 获取这个类
        Class c =
Class.forName("com.bjpowernode.java.annotation5.MyAnnotationTest")
;
        // 判断类上面是否有@MyAnnotation

 //System.out.println(c.isAnnotationPresent(MyAnnotation.class));
// true
        if(c.isAnnotationPresent(MyAnnotation.class)){
            // 获取该注解对象
            MyAnnotation myAnnotation =
(MyAnnotation)c.getAnnotation(MyAnnotation.class);
            //System.out.println("类上面的注解对象" + myAnnotation);
// @com.bjpowernode.java.annotation5.MyAnnotation()

            // 获取注解对象的属性怎么办？和调接口没区别。
            String value = myAnnotation.value();
            System.out.println(value);
        }

        // 判断String类上面是否存在这个注解
        Class stringClass = Class.forName("java.lang.String");

 System.out.println(stringClass.isAnnotationPresent(MyAnnotation.c
lass)); // false
    }
}
```

```
注解在开发中有什么用呢？

        需求：
            假设有这样一个注解，叫做：@Id
            这个注解只能出现在类上面，当这个类上有这个注解的时候，
            要求这个类中必须有一个int类型的id属性。如果没有这个属性
            就报异常。如果有这个属性则正常执行！

    在测试程序中获取类，然后判断类中有没有ID属性，没有的话抛异常

//通过反射机制去读取注解
    public class Test {
    public static void main(String[] args) throws Exception{
```

```
        // 获取类
        Class userClass = Class.forName("com.bjpowernode.java.annotation7.User");
        // 判断类上是否存在Id注解

        if(userClass.isAnnotationPresent(MustHasIdPropertyAnnotation.class)){
            // 当一个类上面有@MustHasIdPropertyAnnotation注解的时候，要求类中必须存在int类型的id属性
            // 如果没有int类型的id属性则报异常。
            // 获取类的属性
            Field[] fields = userClass.getDeclaredFields();
            boolean isOk = false; // 给一个默认的标记
            for(Field field : fields){
                if("id".equals(field.getName()) && "int".equals(field.getType().getSimpleName())){
                    // 表示这个类是合法的类。有@Id注解，则这个类中必须有int类型的id
                    isOk = true; // 表示合法
                    break;
                }
            }

            // 判断是否合法
            if(!isOk){
                throw new HasNotIdPropertyException("被@MustHasIdPropertyAnnotation注解标注的类中必须要有一个int类型的id属性！");
            }

        }
    }
}

@Target(ElementType.TYPE)
// 该注解可以被反射机制读取到
@Retention(RetentionPolicy.RUNTIME)
public @interface MustHasIdPropertyAnnotation {

}

@MustHasIdPropertyAnnotation
public class User {
    int id;
    String name;
    String password;
}
```

编码经验

- 在编写程序判断是，不要一直嵌套if...else语句，要采用“过关斩将”的方式，先将不满足的情况剔除，最后剩下满足的情况，再编写代码

```
1、引用等于null的这个判断最好放到所有条件的最前面。
if(username == null || username.length() < 6 || username.length() > 14){
再分享一个经验：username == null 不如写成 null == username

2、 "abc".equals(username) 比 username.equals("abc") 好。---防止空指针异常

3、类在强制类型转换过程中，如果是类转换成接口类型。那么类和接口之间不需要存在继承关系，也可以转换，
见D:\Program\IdeaProject\Learn\homework
```

面试题

```
1、面试题：String为什么是不可变的？
    我看过源代码，String类中有一个byte[]数组，这个byte[]数组采用了final修饰，因为数组一旦创建长度不可变。并且被final修饰的引用一旦指向某个对象之后，不可再指向其它对象，所以String是不可变的！
        "abc" 无法变成 "abcd"

2、StringBuilder/StringBuffer为什么是可变的呢？
```

```
    我看过源代码，StringBuffer/StringBuilder内部实际上是一个byte[]数组，这个byte[]数组没有被final修饰，StringBuffer/StringBuilder的初始化容量我记得应该是16，当存满之后会进行扩容，底层调用了数组拷贝的方法System.arraycopy()...是这样扩容的。所以StringBuilder/StringBuffer适合于使用字符串的频繁拼接操作。

    String s = "abc"; //"abc"放到了字符串常量池当中。"abc"不可变。
    // s变量是可以指向其它对象的。
    // 字符串不可变不是说以上变量s不可变。说的是"abc"这个对象不可变。
    s = "xyz";//"xyz"放到了字符串常量池当中。"xyz"不可变。

3、StringBuffer和StringBuilder区别？
    StringBuffer中的方法都有：synchronized关键字修饰。表示StringBuffer在多线程环境下运行是安全的，但速度慢。StringBuilder中的方法都没有：synchronized关键字修饰，表示StringBuilder在多线程环境下运行是不安全的，但速度快。
```

```
4、Integer非常重要的面试题：Integer在类加载时会初始化整数型常量池（256个对象）
    Integer x = 127;
    Integer y = 127;
// == 永远判断的都是两个对象的内存地址是否相同。
    System.out.println(x == y); //true
    原因：java中为了提高程序的执行效率，将[-128到127]之间所有的包装对象提前创建好，放到了一个方法区的“整数型常量池”当中了，目的是只要用这个区间的数据不需要再new了，直接从整数型常量池当中取出来。
    原理：x变量中保存的对象的内存地址和y变量中保存的对象的内存地址是一样的。
        Integer a = 128;
        Integer b = 128;
        System.out.println(a == b); //false
```

```
5、finally面试题
    public static void main(String[] args) {
        int result = m();
        System.out.println(result); //100，但finally中i++肯定执行了
    }
    public static int m(){
    int i = 100;
try {
    // 这行代码出现在int i = 100;的下面，所以最终结果必须是返回100
    // return语句还必须保证是最后执行的。一旦执行，整个方法结束。
    return i;
} finally {
    i++;
}
}
/*执行原理:
public static int m(){
```

```
    int i = 100;
    int j = i;
    i++;
    return j;
}
*/
```

```
6、final finally finalize有什么区别？
    final 关键字
        final修饰的类无法继承
        final修饰的方法无法覆盖
        final修饰的变量不能重新赋值。

    finally 关键字
        和try一起联合使用。
        finally语句块中的代码是必须执行的。

    finalize 标识符
        是一个Object类中的方法名。
        这个方法是由垃圾回收器GC负责调用的。
```

```
7、这么多的集合中，你用哪个集合最多？
    ArrayList集合。因为往数组末尾添加元素，效率不受影响。另外，我们检索/查找某个元素的操作比较多。每个元素占用空间大小相同，内存地址是连续的，知道首元素内存地址，然后知道下标，通过数学表达式计算出元素的内存地址，所以检索效率最高。
```

```
8、关于Thread.sleep()方法的一个面试题：
public class ThreadTest07 {
    public static void main(String[] args) {
        // 创建线程对象
        Thread t = new MyThread3();
        t.setName("t");
        t.start();

        // 调用sleep方法
        try {
            // 问题：这行代码会让线程t进入休眠状态吗？---不会
            t.sleep(1000 * 5); // 在执行的时候还是会转换成：Thread.sleep(1000 * 5);
                                // 这行代码的作用是：让当前线程进入休眠，也就是说main线程进入休眠。
                                // 这样代码出现在main方法中，main线程睡眠。
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 5秒之后这里才会执行。
```

```
        System.out.println("hello World!");
    }
}
class MyThread3 extends Thread {
    public void run(){
        for(int i = 0; i < 10000; i++){
            System.out.println(Thread.currentThread().getName() +
"--->" + i);
        }
    }
}
```

```
9、面试题：doOther方法执行的时候需要等待doSome方法的结束吗？
    //需要，因为synchronized出现在实例方法上是锁this,t1和t2共用一个对象，
一个对象一把锁
public class Exam01 {
    public static void main(String[] args) throws
InterruptedException {
        MyClass mc = new MyClass();

        Thread t1 = new MyThread(mc);
        Thread t2 = new MyThread(mc);

        t1.setName("t1");
        t2.setName("t2");

        t1.start();
        Thread.sleep(1000); //这个睡眠的作用是：为了保证t1线程先执行。
        t2.start();
    }
}

class MyThread extends Thread {
    private MyClass mc;
    public MyThread(MyClass mc){
        this.mc = mc;
    }
    public void run(){
        if(Thread.currentThread().getName().equals("t1")){
            mc.doSome();
        }
        if(Thread.currentThread().getName().equals("t2")){
            mc.doOther();
        }
    }
}

class MyClass {
    public synchronized void doSome(){
```

```
        System.out.println("doSome begin");
        try {
            Thread.sleep(1000 * 10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("doSome over");
    }
    public synchronized void doOther(){
        System.out.println("doOther begin");
        System.out.println("doOther over");
    }
}
```

```
10、面试题：doOther方法执行的时候需要等待doSome方法的结束吗？
    //需要，因为静态方法是类锁，不管创建了几个对象，类锁只有1把。
public class Exam01 {
    public static void main(String[] args) throws
InterruptedException {
        MyClass mc1 = new MyClass();
        MyClass mc2 = new MyClass();

        Thread t1 = new MyThread(mc1);
        Thread t2 = new MyThread(mc2);

        t1.setName("t1");
        t2.setName("t2");

        t1.start();
        Thread.sleep(1000); //这个睡眠的作用是：为了保证t1线程先执行。
        t2.start();
    }
}

class MyThread extends Thread {
    private MyClass mc;
    public MyThread(MyClass mc){
        this.mc = mc;
    }
    public void run(){
        if(Thread.currentThread().getName().equals("t1")){
            mc.doSome();
        }
        if(Thread.currentThread().getName().equals("t2")){
            mc.doOther();
        }
    }
}

```

```
class MyClass {
    // synchronized出现在静态方法上是找类锁。
    public synchronized static void doSome(){
        System.out.println("doSome begin");
        try {
            Thread.sleep(1000 * 10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("doSome over");
    }
    public synchronized static void doOther(){
        System.out.println("doOther begin");
        System.out.println("doOther over");
    }
}
```