JavaSE-itheima

1-java 入门

1-1_Java 语言简介

1.Java 创始人-James Gosling，Sun 公司，现在属于 Oracle 公司
2. 计算机基础
3. 二进制和十进制的转换: 8421
4. 字节：计算机存储的最小单位，1bit=8B
5. 常用 DOS 命令
    切换盘符: D:
    进入盘符文件：cd d:\develop
    清屏：cls
    自动补全：tab
    退到上一目录：cd ..
    **退到根目录：cd **
    查看目录下的所有文件：dir
    查看历史命令：上键和下键

1-2_Java 环境配置

1.JDK、JRE 和 JVM
    JVM: Java 虚拟机，Java 跨平台的实现原理
    JRE: Java 运行环境
    JDK：Java 开发包
    三者的关系：JDK>JRE>JVM
2.JDK9 下载，Oracle 官网下载
3.Java 环境变量配置
    JAVA_HOME: jdk 所在目录
    把 JAVA_HOME 添加进 path 变量: **%JAVA_HOME%\bin;**

1-3_HelloWorld 入门程序

```
public class HelloWorld{// 定义一个类 类名为 HelloWorld    类名和文件名要一致
    // main 方法是程序的入口,程序运行的时候,JVM 就会去找 main 方法执行
    public static void main(String[] args){// main 方法的格式是固定的
        /*
            这些东西都是被注释了的
            xxxx
            yyyy
        */
        System.out.println("Hello World!!!");// 输出语句  ()括号里面的数据会原样输出来
    }
}
```

java 编译成.class 字节码文件，解释器解释，jvm 运行，javac 编译，java 解释
类名和文件名保持一致，类名大写
关键字:在 java 语言中有特定含义的单词，在编辑器中有特殊的颜色标记

标识符

命令规则：由字母、数字、_ 、$组成，不能以数字开头，不能是 java 关键字，要做到见名知意

注释: 解释代码的, 不参与程序的运行

单行注释：//

多行注释：/**/

1-4_变量和常量

1.常量:程序运行期间不可改变的数据(固定不变的数据)

整型常量: 66

浮点常量：3.14

布尔常量：只有两个数据, true 或者 false

字符串常量：必须要用双引号引起来,在 java 里面凡是看到双引号引起来的就是字符串

双引号、可以为空值

字符常量：必须要用单引号引起来,并且单引号里面只能有一个字符

单引号，只能有一个字符

单个字符：'A', '9'

转义字符：'\n', '\"', '\'"

Unicode 编码：'\u66f0'

空常量: null, 没有任何数据

2.变量:可以变化的数据

作用: 变量其实就是一个容器,可以用来存储数据

定义：数据类型 变量名=值;

基本数据类型和引用数据类型

基本数据类型：4 类 8 种

整型: 默认是 int

byte(1 个字节): -128-127

short(2 个字节): -32768-32767

int(4 个字节)：范围大约 21 亿左右

long(8 个字节)：定义 long 类型的变量时，**值后面加 L 或者 l**

浮点型

float: 值后面加 F 或者 f

double: 浮点型默认是 double

字符型：char, 2 个字节

布尔型：boolean, true 或者 false

引用数据类型: 类, String 类, 数组, 接口

1-5_注意事项

1.变量名称：在同一个大括号范围内，变量的名字不可以相同。

2.变量赋值：定义的变量，不赋值不能使用。

3.给 float 和 long 类型的变量赋值,记得加 F 和 L

4.超过了变量的作用域,就不能再使用变量了　　作用域: 从定义变量的位置开始,一直到直接所属的大括号结束为止

5.同一条语句可以同时定义多个同类型的变量

6.如果给 byte 和 short 类型的变量赋值,记得不要超过该类型所表示的范围

提示: Java 里面是严格区分大小写的

2-类型转换、运算符和+

2-1_类型转换

Java 程序中要求参与的计算的数据，必须要保证数据类型的一致性，如果数据类型不一致将发生类型的转换

1.自动类型转换:又叫隐式类型转换

条件：表述范围小的向表述范围大的转换，且转换是安全的

byte, short, char-->int, float-->double, long-->float

char, short, byte 类型的变量在参与运算时会自动提升为 int 类型，然后参与运算

经典用例：System.out.println('中'+0); 可以确定中的 ASCII 码

2.强制类型转换

条件：表述范围大的向表述范围小的转换，转换不安全

转换规则：目标数据类型 变量名 = (目标数据类型) 原数据类型值;

byte b=100; //ok, Java 里面常量优化机制,在编译阶段就可以确定 100 是在 byte 所能表示的数据范围

3.类型转换注意事项

类型转换的注意事项:

强制类型转换，可能造成数据损失精度。

byte \ short \ char 这三种数据类型可以发生数学运算,但是会首先提升到 int 类型

boolean 类型的不能发生强制类型转换

2-2_Java 运算符

单元运算符: 运算的时候只需要一个数参与运算 num++ !true

二元运算符: 运算的时候只需要 2 个数参与运算 1 + 2 1 * 2 1 > 2

三元运算符: 运算的时候需要 3 个参数参与运算

赋值运算符：

=， +=， -=， *=，/=, %=

算术运算符：

+, -, *, /, %, ++, --

/：取的结果是商

%：取得结果是余数

a++和++a 的区别

++运算符: 自增运算符 使变量自增 1

使用:

独立使用:单独使用 num++ ++num,不管是前++,还是后++,都是使变量自身的值+1

混合使用:与其他运算符混合一起 变量的值 表达式的值

前++(前自增): ++运算符在变量的前面 例如 ++num

运算法则: 先将变量自身的值+1,然后再将变量的值作为前自增表达式值

后++(后自增): ++运算符在变量的后面 例如 num++

运算法则: 先取变量自身的值作为后自增表达式的值,然后变量自身的值+1

比较运算符:比较表达式的结果一定是 boolean 类型的数据

>, >=, <, <=, ==, !=

逻辑运算符:逻辑运算符 2 边的表达式的值一定要是 boolean 类型

&&(短路与), &, **||(短路或)**, |, !, ^

三目运算符

三元运算符格式：

数据类型 变量名 = 布尔类型表达式？结果 1 ： 结果 2;// 数据类型 变量名 = 三元表达式的值;

三元运算符计算方式：

尔类型表达式结果是 true，三元运算符整体结果为结果 1，赋值给变量。

布尔类型表达式结果是 false，三元运算符整体结果为结果 2，赋值给变量。

++(--)和扩展运算符(+=, -=等等)系统会自动强制类型转换

```
byte b = 2;
b++; // ok
System.out.println(b); //3
```

2-3_方法

方法：就是将一个功能抽取出来，把代码单独定义在一个大括号内，形成一个单独的功能

定义方法:

格式:

修饰符 返回值类型 方法名(参数列表){

方法体...

return 语句;

}

修饰符: public static

返回值类型: void

方法名: 自己取的名字,符合标识符命名规则和规范

参数: 空着　小括号里面空着,啥也不写

方法体: 功能代码

return 语句: 由于返回值类型是 void,所以不需要 return 语句

调用方法:

格式: 方法名();

注意事项:

1.方法定义在类里面,方法的外面

2.方法里面不能定义方法

3.方法和方法之间是独立的关系

4.方法不调用不会执行

2-4_+的用法

1.对于数值来说,那就是加法

2.对于字符 char 类型来说,在计算之前,char 会被提升为 int 类型,然后再进行计算

char 类型 和 int 类型之间的对照关系表: ASCII Unicode

3.对于字符串 String 来说,加号代表字符串的拼接

4.字符串和任意类型的数据相加,都会拼接成一个新的字符串

2-5_JShell 工具

cmd 命令行， 输入 jshell， 进入 jshell 命令行，可以输入 java 片段代码

推出 jshell 命令行：/exit

JDK9 新特性

2-6_ACII 码

每一个字符都有一个对应的 int 类型的数

ASCII 码表: 美国标准信息交换代码

Unicode 码表: 万国表　前面 0-127 和 ASCII 码表是一样的,但是从 128 开始包含更多的字符

3-流程控制语句

3-1_流程控制语句

3-1-1_顺序结构

程序的执行顺序: 从上往下执行

3-1-2_条件选择语句

```
1.if 语句
if (关系表达式) {
      语句;
}
关系表达式的值一定要是布尔类型的值
适合: 如果...就...

if...else 语句:
 if (关系表达式) {
      语句 1;
 } else {
      语句 2;
 }
适合: 如果...就...否则就...
if...else if...else 语句:
 if (关系表达式 1) {
      语句 1;
  } else if (关系表达式 2) {
      语句 2;
  } else if (关系表达式 3) {
      语句 3;
  }
      ...
 else {
      语句 n;
 }
 适合：大于等于 3 种情况以上的
          2.switch 语句
switch(表达式) {
    case  常量值 1:
        语句体 1;
        break;
    case  常量值 2:
        语句体 2;
        break;
    ...
    default:
     语句体 n;
     break;
}

执行流程: 首先拿到表达式的值与 case 后面的值匹配, 如果匹配成功, 就执行对应 case 的语句体, 如果没有碰到 break 就一直
```

往下执行直到配到 break 或者"}"结束

注意事项:

- 表达式的值必须是：byte, short, char, int, String, enum 枚举
- 前后顺序可以颠倒(case 1 可以放在 case 2 后面, default 可以放在最前面), break 语句可以省略，但是如果不写 break 会有穿透现象，合理利用 case 穿透
- **case 穿透:switch 语句如果 case 后面不写 break，将会出现穿透现象，也就是不会去判断下一个 case 的值，直接向后执行，直接遇到 break 才结束或者整体 switch 语句结束**
- default 语句后面的 break 可以省略, default 语句也可以省略
- 多个 case 后面的值不能相同
- 能用 switch 一定能用 if，但是能用 if 的不一定能用 switch(需要转换，有时候不一定能转换出来)
- 等值判断时用 switch

3-1-3_循环语句

1.for 循环

格式:

```
for (初始化表达式 1; 布尔表达式 2; 步进表达式 4) {
    循环体 3;
}
```

执行顺序: 1234->234->234...2 直到 2 不满足为止，跳出循环

初始化表达式 1：循环变量初始化，只执行一次

布尔表达式 2: 判断是否满足循环条件，不满足就跳出循环，满足就执行循环体 3

循环体 3: 如果布尔表达式 2 满足为 true 时，那么就会执行

步进表达式 4: 循环体 3 执行完之后，循环变量的变化情况

嵌套的 for 循环:

```
for(初始化表达式①; 循环条件②; 步进表达式⑦) {
    for(初始化表达式③; 循环条件④; 步进表达式⑥) {
        执行语句⑤;
    }
}
```

嵌套循环执行流程：

执行顺序：①②③④⑤⑥>④⑤⑥>⑦②③④⑤⑥>④⑤⑥

外循环一次，内循环多次。

比如跳绳：一共跳 5 组，每组跳 10 个。5 组就是外循环，10 个就是内循环。

2.while 循环

```
初始化表达式 1;
while (布尔表达式 2) {
    循环体 3;
    步进表达式 4;
}
```

执行顺序: 1234->234->234...2 直到 2 不满足为止，跳出循环

3.do...while 循环

```
初始化表达式 1;
do {
    循环体 3;
    步进表达式 4;
} while (布尔表达式 2);
```

执行顺序: 1342->342->342...2 直到 2 不满足为止，跳出循环

注意 while(布尔表达式);后面有一个;

4.for, do...while, while 大的区别

1.三者的书写格式不一样

2.for 循环和 while 循环可以相互转换，如果次数确定就是用 for 循环，

如果次数不确定就使用 while(登录时用户名,密码的校验，因为不知道用户要输入的次数)

3.while 和 do...while 循环的循环变量可以作用于循环体外面，而 for 循环不行

4.while 和 for 循环不满足条件一次都不执行，如果 do...while 循环至少执行一次

5.死循环

```
while (true) {
    ...
}
for (; ture; ) {
    ...
}
for (; ; ) {
    ...
}
```

6.break 和 continue 关键字

break: 终止 switch 或者循环

使用场景:

1.在 switch 语句中

2.在循环语句中

3.离开了使用场景没意义

continue: 结束本次循环，继续下一次循环

4-IDEA 使用和方法

4-1_IDEA 的使用

1. Idea 的常用设置

1.字体设置: File->settings->(直接使用快捷键 alt+shift+s)Editor->Font

2.主题风格设置: settings->Appearances && Behavior->Apperance

3.快捷键修改: settings->Keymap->Main menu->Code->Completion->Basic->选择右键先移除原先的快捷，在 Add 新的快捷键->直接输入快捷键，比如 alt+/，键盘直接输入：先输入 art，在输入/

4.修改小写也能提示(忽略大小写): settings->Editor->General->Code Completion->在 Case sensitive completion->选择 None

5.显示工具按钮: View->勾选 Tool Buttons

6.取消显示形参名: settings->Editor->Appearance->取消"Show parameter name hints"前面的勾

2.Idea 常用快捷键

1.编辑快捷键

1. main 方法的快捷键: **psvm+tab 键(enter)**, println 的快捷键: **sout+enter(tab 键)**

2. Ctrl+Y: 删除光标所在行

3. Ctrl+D: 复制光标所在行代码到光标下一行

4. Ctrl+Alt+L: 格式化代码

5. Ctrl+/ : 单行注释, Ctrl+Shift+/: 多行注释

6. Alt+Shift+上下箭头: 移动当前代码上下移动

7. Alt+Enter: 导入包，自动修正代码

8. Alt+Insert: 可以自动生成构造器, getter/setter 等常用方法

9. Ctrl+Alt+V: 可以自动引入变量定义，例如: new String(); 自动导入变量定义后: String s = new String();

10. Ctrl+Shift+空格: 代码自动补全

11.Ctrl+Shift+U: 大小写转化

12.Ctrl+Enter: 当前光标所在行的上插一行空行, Shift+Enter: 当前光标所在行的下插一行空行

2. 搜索快捷键

1. 双击 Shift: 项目所有目录查找
2. Ctrl+F: 当前文件查找特定内容
3. Ctrl+R: 当前文件替换选定的内容, Ctrl+Shift+R: 整个项目替换选定的内容
4. Ctrl+Shift+F: 当前项目查找包含特定内容的文件
5. Ctrl+N: 查找类, Ctrl+Shift+N: 查找文件
6. Ctrl+E: 打开最近的文件
7. Alt+F7: 找到你变量, 方法, 类所引用到的地方
8. F2 或 Shift+F2,高亮错误或警告快速定位
9.Ctrl+Shift+F7,高亮显示所有该文本,按 Esc 高亮消失
10. Alt+Home,跳转到导航栏

3.视图快捷键

1. Shift+Esc: 隐藏当前视图窗口
2. Alt+1: 打开或者隐藏项目工程目录, Alt+7: 查看当前类结构, Alt+6:TODO
3. Ctrl+Alt+S: 快速打开 settings 界面
4. Ctrl+Alt+Shift+S: 快速打开 Project Structure 界面
5.Ctrl+F12: 查看当前类的结构图
6. Ctrl+H,显示类结构图(类的继承层次)
7. Ctrl+F4: 关闭当前文件, 关闭当前文件之外的所有文件: 按住 Alt+左键单击当前文件右上角 X,
8. Ctrl+Tab,转到下一个拆分器, Ctrl+Shift+Tab: 转到上一个拆分器

4.Debug 调试

1. F7: 单步调试, 一条语句逐步执行
2. F8: 跳过方法执行
3. Ctrl+F5: 重新 run debug
4. F9：跳过当前断点执行下一个断点(如果没有下一个断点就终止)
5. Ctrl+F8: 当前光标行打上/取消断点(鼠标左键点击)

5.重构

1.Ctrl+O,重写方法
2.Shift+F6: 重构: 重命名类，变量，方法等
3.Ctrl+Alt+M: 重构方法

4-2_方法

1.方法的定义
1.方法定义的格式
修饰符 返回值类型 方法名(参数列表){
方法体
}
2.方法定义注意事项
1.方法名定义: 要符合标识符命名规范, 首个单词小写, 其余单词首字母大写
2.return 的作用
1.结束方法
2.方法产生的数据返回给调用者
3.返回值类型要和 return 返回的值得类型保持一致
4.方法返回值是否为空为 void: 可以不用写 return, 如果要写就这样写: return;不为 void: 一定要写 return 返回值;
5.方法定义在类的内部, 其他方法的外面

2.方法的调用
1.方法的使用
1.明确方法的返回值类型
2.明确方法名
3.明确方法的参数
4.明确方法的方法体
2.方法的调用(3 种方式)
1.直接调用: 方法名(实际参数);

2.赋值调用: 返回值类型 变量名 = 方法名(实参);
3.打印调用: System.out.println(方法名(实参));

3.方法的重载

指在同一个类中，允许存在一个以上的同名方法，只要它们的参数列表不同即可，与修饰符和返回值类型无关。

1. 构成重载的条件

1.同一个类中
2.方法名相同
3.参数不同

2.重载的类型

1.参数的类型可以不同
2.参数的个数可以不同
3.参数的顺序可以不同

3.方法重载与以下元素无关

1.参数名无关
2.返回值类型无关
3.修饰无关

5-数组

5-1_数组的定义以及初始化

数组的定义:

动态方式:动态方式创建数组,必须指定数组的长度,不确定数组中的元素

格式:

数据类型[] 数组名 = new 数据类型[长度];

数据类型: 数组中元素的数据类型(数组中存储的数据的数据类型)

[]: 代表是数组

数组名: 自己取的名字,符合标识符命名规则和规范

new : 代表创建数组的关键字

数据类型: 和左边的数据类型一致

[长度] : 数组的长度

静态方式:静态方式创建数组,不指定数组的长度,但是确定数组中的元素,系统根据元素的个数 计算数组的长度

格式:

数据类型[] 数组名 = new 数据类型[]{元素 1,元素 2,...};

省略格式:

数据类型[] 数组名 = {元素 1,元素 2,元素 3,...};

使用:

如果确定数组中的元素,那么就使用静态方式创建数组

如果只能确定数组的长度,具体的元素不清楚,就使用动态方式创建数组

数组的特点:

1.长度固定

2.数组中元素的数据类型一致

3.数组中的每一个元素都有一个下标\角标\索引 并且**索引是从 0 开始的**

int 类型的数组中的元素默认值:0

byte 类型的数组中的元素默认值:0

short 类型的数组中的元素默认值:0

long 类型的数组中的元素默认值:0

double 类型的数组中的元素默认值:0.0

float 类型的数组中的元素默认值:0.0

char 类型的数组中的元素默认值:'\u0000'

boolean 类型的数组中的元素默认值: false
String 类型的数组中的元素默认值: null
"" 空字符串　空字符串数据
null　空　没有任何数据

int[] : int 数组类型　　说明数组中只能存储 int 类型的数据(也就是说数组中的元素是 int 类型)
double[] : double 数组类型　　说明数组中只能存储 double 类型的数据(也就是说数组中的元素是 double 类型)
char[] : char 数组类型　　说明数组中只能存储 char 类型的数据(也就是说数组中的元素是 char 类型)
String[] : String 数组类型　　说明数组中只能存储 String 类型的数据(也就是说数组中的元素是 String 类型)
十六进制数:　0 1 2 3　4 5 6 7 8 9 A B C D E F 10
十进制数:　0 1 2 3 4 5 6 7 8 9 10 11
八进制数:　0 1 2 3 4 5 6 7 10 11

抽取方法的快捷键: ctrl+alt+M

1.定义数组
2.取出数组中的元素值
3.给数组中的元素赋值
4.求数组的长度
经典错误:
ArrayIndexOutOfBoundsException：数组索引越界异常　也就是访问了不存在的索引空间
NullPointerException：空指针异常

5-2_数组的使用

访问数组中的元素:　**数组名[索引];**
给数组中的元素赋值; **数组名[索引] = 值;**
数组在内存中的存储情况:
数组名(引用)存在栈区，　数组中的元素存堆区
内存:
寄存器: 跟 cpu 有关系　不需要 java 程序员考虑
本地方法区: 跟系统有关系, 不需要 java 程序员考虑
栈区: 存储局部变量
堆区: 存储对象　凡是 new 出来的都是在堆区
方法区: 存储字节码文件的　　静态区,常量池,方法区,代码区

6-OO 和继承

6-1_面向对象(OO)

1.面向对象:
面向对象:以对象的思维去思考, 万事万物皆对象, 用对象去完成我们需要完成的功能。
面向过程:一步一步地去完成功能。
区别:面向对象强调对象, 面向过程强调过程
面向对象 3 大特征: 封装、继承和多态
2.类和对象
类: 一群具有共同特征和行为的事物的特征, 模板
对象: 就是以类为模板创建出的对象, 具体的实例
类和对象的关系:
1.类里面有什么, 对象就有什么
2.对象是通过类来创建的, 一个类可以创建多个对象

3.对象与对象之间是相互独立的
4.类是抽象的(不可以直接使用)，对象是具体存在的(可以直接使用)

3.类的定义

定义类的格式:

```
public class 类名 {
    // 成员变量
    // 构造方法
    // 成员方法
}
```

4.对象的定义以及使用

对象的创建格式: 类名 对象名 = new 类名();

对象的使用:

属性: 对象名.属性名

方法: 对象名.方法名(参数);

成员变量的默认值:

byte、short、int、long: 0

double、float: 0.0

char: '\u0000' - 空字符，什么都没有

boolean: false

数组、类、接口: null

5.对象内存图

内存分布结构：

局部变量: 栈区

new 出来的东西: 堆区

方法区:

代码区: 存放.class 文件

方法区: 存放方法的代码

常量池: 字符串常量

静态区: 静态常量

创建对象内存图:

类名 对象名 = new 类名();

对象名.属性 = 值;

对象名.成员方法();

对象名存栈区，new 类名();这一行代码表示: 在堆区创建一个对象，成员变量赋默认值，成员方法存放在方法区，堆内存存的是成员方法的地址

对象名.属性 = 值; 在堆区给对象的属性赋值

对象名.成员方法(); 调用成员方法时，成员方法入栈，然后进行后面的操作，方法执行完之后，弹栈

6.对象作为方法的参数和返回值

对象作为方法的参数，传递的对象的引用; 对象作为方法的返回值，返回的是对象的引用

7.成员变量和局部变量的区别

1.定义的位置不同

成员变量定义在类的里面，方法的外面

局部变量定义在类的里面

2.作用域不同

成员变量的作用域是整个类，局部变量的作用域是方法里面

3.初始值不同

成员变量有一个默认值，局部变量必须要赋值后才能使用

4.在内存中存放的位置不同

成员变量存在在堆区，局部变量存放在栈区

5.生命周期不同

成员变量随着对象的创建而存在，随着对象的消失而消失

局部变量随着方法的调用而存在，随着方法的调用完毕而消失

6-2_封装(encapsulation)

1.封装的概述

封装：就是防止该类的代码或者数据被其他类随意访问，要访问该类的代码必须通过指定的方式(方法的调用)

封装的好处：代码更易理解和维护，也加强了代码的安全性

原则: 将属性(成员变量)隐藏起来，若要访问某个属性，提供公共方法对其访问

2.封装的操作

private 关键字：

1.private 是一个修饰符，代表最小的访问权限

2.可以修饰成员方法和成员变量

3.被 private 修饰的成员方法和成员变量只能在本类中访问

private 修饰成员属性: private 数据类型 变量名;

private 修饰成员方法: private 返回值类型 方法名(参数类型 参数名, ...) {}

3.封装的优化-this 关键字和构造方法

this：

this 关键字: 代表当前对象的引用(地址值)

格式: this.成员变量;

成员方法被哪个调用那么 this 就代表那个对象

this 的作用: 为了避免局部变量和成员变量重名

成员方法中只有一个成员变量时，那么 this 可以省略

构造方法：

构造方法定义格式: public 类名(){}或者 public 类名(数据类型 参数名, ...) {}

构造方法作用: 初始化对象(1.创建对象，2.给创建出来的对象的属性初始化)

注意事项:

1.构造方法没有返回值, void 也不能写

2.构造方法通过 new 来调用

3.如果类中不顶用构造方法，那么系统会默认给一个构造方法

4.如果类中定了构造方法，那么系统就不会给一个默认的构造方法，需要自己手动添加

5.构造方法是可以重载的，既可以参数，也可以不定义参数

4.JavaBean-标准代码

符合 **JavaBean** 规范的类:

1.public 修饰类

2.有无参的构造方法

3.需提供的 **setXxx()**和 **getXxx()**方法

4.成员变量用 **private** 修饰

7-Scanner、Random 和 ArrayList

7-1_API

API(Application Programming Interface)，应用程序编程接口

java.lang 包下面的类不需要 import，除此之外都要 import

查看 api 文档:

1.显示->索引->搜索

2.看类属于哪个包

3.看类的说明

4.看类的构造方法

5.看类的方法

7-2_Scanner 类

1.Scanner 类
    接收用户的键盘输入
    在 java.util.Scanner 包里面
2.Scanner 使用步骤
    导包: import java.util.Scanner;
    创建: Scanner sc = new Scanner(System.in);
    调用方法:
        int number = sc.nextInt(); // 接收 int 类型的数字
        int number2 = sc.nextInt(); // 第二次也可以用 sc 对象的 nextInt()方法
        String str = sc.nextLint(); // 接收输入的行
3.匿名对象
    定义：创建对象时并没有把对象引用赋值给变量，没有变量名的对象
    格式：new 类名(参数列表);
        比如, new Scanner(System.in).nextInt();
    应用场景:
        1.创建匿名对象直接调用方法，没有变量名
        2.不需要调用两次即以上的方法(一个匿名对象只能使用一次)
        3.匿名对象可以作为方法的参数和返回值
    匿名对象作为方法的参数: 传递的是一个引用
        method(new Scanner(System.in));
    匿名对象作为方法的返回值: 返回的是一个引用
        public Scanner method() {
            return new Scanner(System.in);
        }

7-3_Random 类

1.Random 类在 java.util 包下，这个类可以随机产生一个数
2.使用 Random 类
    Random r = new Random();
    int i3 = r.nextInt(n-m+1) + m; // 随机产生一个[m, n)之间的数字
    int i = r.nextInt(); // 随机产生一个 int 类型的数字
    int i2 = r.nextInt(n); // 随机产生一个[0,n)之间的数字
    int i3 = r.nextInt(20) + 30; // 随机产生一个[30, 49]之间的数

7-4_ArrayList 类

1.基本数据类型和包装类
    基本数据类型 包装成 基本数据类型包装类
        int Integer
        byte Byte
        short Short
        long Long
        float Float
        double Double
        char Character
        boolean Boolean
    自动装箱: 基本数据类型自动转换成包装类
        int i = 2;
        Integer i2 = i;

自动拆箱: 包装类自动转换成基本数据类型

Integer i = 2;

int i2 = i;

2.ArrayList 类是一个集合类

1.定义一个 ArrayList 类, 限定集合里面的元素只能是 String

ArrayList<String> list = new ArrayList<>(); // JDK1.7 之后的写法, 推荐

ArrayList<String> list = new ArrayList<String>(); // 完整写法

ArrayList<String> list = new ArrayList(); //不推荐使用

2.ArrayList 类的常用方法(增删改查)

- public boolean add(E e): 添加指定元素到集合的尾部
- public E remove(int index): 删除指定位置的元素, 返回删除的元素
- **public E get(int index): 获取指定位置的元素, 返回获取的元素**
- public int size(): 获取集合中的元素个数(集合的大小或者长度)
- **public boolean contains(Object o)**: 判断指定元素是否在集合中
- public E set(int index, E element): 用指定元素代替集合中指定位置的元素, 返回被替代的元素

3.ArrayList 其他方法

- void add(int index, E element) 将指定的元素插入此列表中的指定位置。
- int indexOf(Object o) 返回此列表中首次出现的指定元素的索引，或如果此列表不包含元素，则返回 -1。
- boolean isEmpty() 如果此列表中没有元素，则返回 true
- int lastIndexOf(Object o) 返回此列表中最后一次出现的指定元素的索引，或如果此列表不包含索引，则返回 -1。
- **Object[] toArray()** 按适当顺序（从第一个到最后一个元素）返回包含此列表中所有元素的数组。

8-API(常用类)上和 static

8-1_String 类

1.String 类代表字符串, Java 程序中用双引号引起来的数据, 都是 String 类的对象。

2.String:

- 代表字符串
- **字符串是不可变的**
- **字符串是有索引的, 索引从 0 开始**
- String 字符串底层是字符数组实现的

String str = "abc"

等价于: char[] data = {'a', 'b', 'c'};

String newStr = new String(data);

3.构造方法: 创建对象 给对象的属性初始化

- String(): 初始化一个新建的 String 对象, 使其表示一个空字符序列
- String(char[] value): 分配一个新的 String, 使其表示字符数组参数中当前包含的字符序列
- String(byte[] bytes): 通过使用平台的默认字符集解码指定的 byte 数组, 构造一个新的 String

```
byte[] bys = {99, 98, 97};
String str = new String(bys);
System.out.println(str); //cba
```

- String(String original): 初始化一个新创建的 String 对象，使其表示一个与参数相同的字符序列；换句话说，新创建的字符串是该参数字符串的副本。
- 用的最多的是: String str = "jack";

4.成员方法:

比较功能的方法:

- **boolean equals(Object anObject):** 将此字符串与指定的对象比较。
- boolean equalsIgnoreCase(String anotherString):将此 String 与另一个 String 比较，不考虑大小写。

获取功能的方法:

- **public int length()**: 获取字符串的长度，也就是获取有多少个字符
- public String concat(String str): 将指定的字符串连接到改字符串的末尾
- **public char charAt(int index)**: 返回指定索引处的 char 值
- public int indexOf(String str): 返回指定子字符串第一次出现在该字符串内的索引
- public int lastIndexOf(String str): 返回子字符串最后一次出现的索引
- public int indexOf(char ch): 返回字符在字符串中第一次出现的索引
- public int lastIndexOf(char ch): 返回字符在字符串中最后一次出现的索引
- public String substring(int beginIndex): 返回一个字符串，从 beginIndex 开始截取字符串到字符串结尾
- **public String substring(int beginIndex, int endIndex)**: 返回一个子字符串，从 beginIndex 到 endIndex 截取字符串, [beginIndex, endIndex)

转换功能的方法:

- **public char[] toCharArray()**: 将字符串转换为字符数组
- **public byte[] getBytes()**: 使用平台默认的字符集将 String 编码转换为新的字节数组。
- public String replace(CharSequence target, CharSequence replacement): 将与 target 匹配的字符串使用 replacement 替换，返回的是一个新的字符串，原先的值并没有改变
- public String replace(char oldChar, char newChar) 返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。

拆分功能的方法:

- **public String[] split(String regex)**: 将此字符串按照给定的 regex(规则)拆分为字符串数组

其他方法:

- **boolean contains(CharSequence s)** 当且仅当此字符串包含指定的 char 值序列时，返回 true。
- boolean endsWith(String suffix) 测试此字符串是否以指定的后缀结束。
- boolean isEmpty() 当且仅当 length() 为 0 时返回 true。
- **boolean startsWith(String prefix)** 测试此字符串是否以指定的前缀开始。
- **String trim()** 返回字符串的副本，忽略前导空白和尾部空白。
- String concat(String str) 将指定字符串连接到此字符串的结尾。

基本数据类型转换为字符串(valueOf()):

- static String valueOf(boolean b) 返回 boolean 参数的字符串表示形式。
- static String valueOf(char c) 返回 char 参数的字符串表示形式。
- **static String valueOf(char[] data)** 返回 char 数组参数的字符串表示形式。
- static String valueOf(char[] data, int offset, int count) 返回 char 数组参数的特定子数组的字符串表示形式。
- static String valueOf(double d) 返回 double 参数的字符串表示形式。
- static String valueOf(float f) 返回 float 参数的字符串表示形式。
- **static String valueOf(int i)** 返回 int 参数的字符串表示形式。
- static String valueOf(long l) 返回 long 参数的字符串表示形式。
- static String valueOf(Object obj) 返回 Object 参数的字符串表示形式。

5.求长度

1.数组的长度: 数组名.length;

2.ArrayList 集合的长度: 集合名.size();

3.String 字符串的长度: 字符串对象.length();

6.==和 equals()的比较

==: 比较的是字符串对象的地址值

如果是基本数据类型比较的是数值

如果是引用数据类型比较的是地址值

字符串常量放在常量区且共享一份相同的内容

```
String str = "hello";
String str2 = "hello"; //str==str2, true
```

字符串对象放在堆区，不同的字符串对象在堆区的地址不一样

equals(): 比较的是字符串的内容

字符串比较推荐使用 equals()

7.String 字符串特点探究:

1.字符串常量在常量池中永远只有一份拷贝

```
        String str = "abc";
        String str2 = "abc"; // "abc"已经在常量池存在，所以 str2 和 str1 都是指向的相同地址
        String str3 = "a" + "bc";// 编译期间已经确定是字符串常量了，存在了常量池，由于在常量池中存在"abc",
        // 所以 str3 也是指向相同的地址
        System.out.println("str==str2:" + (str == str2)); // true
        System.out.println("str==str3:" + (str == str3)); // true
```

2.字符串常量连接在编译时就确定了连接之后的数据是一个字符串常量，存储在了常量池中

3.字符串的不可变性

```
        String str = "hello";
        str += "world"; // 这里的 str 已经不等同于上面的 str，因为已经是不同的对象了，存的地址不一样了。
        // 等价于
        /*
           StringBuffer temp = new StringBuffer(str);
           str = temp.append("world").toString();
        */
```

4.new 产生的字符串会先去常量池中查找，如果没有就在常量池中创建一个此字符串的对象，然后再在堆中创建一个常量池此"china"对象的拷贝对象。

面试题:

```
String s = new String("xyz");
/*
    答案:1 个或 2 个
    如果常量池没有"xyz"，先在常量池中创建一个对象，然后再在堆区创建一个在常量池创建好的对象的拷贝对象
*/
```

5.字符串是一个特殊的包装类，其引用存放在栈里的，而对象内容根据创建方式的不同来确定定(常量池和堆),在编译期就创建好的(字符串常量或者字符串常量的拼接)就存放在常量池中，而在运行时才被创建的存放在堆区(new 出来的字符串对象)。

8-1-1_String、StringBuilder 和 StringBuffer 的区别

主要区别在于两方面：运行速度和线程安全

1、执行效率：StringBuilder > StringBuffer > String

因为 **String** 是字符串常量，一旦 **String** 对象创建好了就不能更改，但是 **StringBuffer** 和 **StringBuilder** 是字符串变量，是可以更改的。

2、线程安全

StringBuilder 是线程不安全的，而 **StringBuffer** 是线程安全的

3、总结

String：适用于少量的字符串操作的情况

StringBuilder：适用于单线程下在字符缓冲区进行大量操作的情况

StringBuffer：适用多线程下在字符缓冲区进行大量操作的情况

8-2_static 关键字

1.static: 静态的意思

修饰成员变量:

格式: static 数据类型 变量名;

特点:

1.被 static 修饰的成员变量会被所有对象共享

2.使用的时候: 类名.静态成员变量名;

修饰成员方法:

格式: 权限修饰符 static 返回值类型 方法名(参数列表) {...}

特点:

1.使用的时候: 类名.静态成员方法名();

2.静态方法里面不能有 this

3.静态方法里面只能访问静态成员，不能直接访问非静态成员，反之非静态成员里面可以访问静态成员。

2.static 修饰的成员和对象没有关系，它是属于类的

3.特点:

静态成员会随着类的加载而加载，只会加载一次

4.静态代码块

1.静态代码块: 定义在类中，方法外

2.定义:

```
static {
    // 静态代码块
}
```

3.随着类的加载而执行且只执行一次，优先于 main 方法和构造方法的执行

5.静态原理图解:

1.类的静态成员变量和静态成员方法存在静态区(方法区的静态域)，堆区对象里面存的是静态成员变量和静态成员方法的引用(地址)

2.当类加载的时候, .class 文件会在方法区里面，然后类里面的静态成员(静态成员变量和静态成员方法)会在静态域里面创建一份对应的静态成员，然后方法区的静态成员指向静态域中的这份静态成员，当对类里面的静态成员赋值时就找到静态域中的静态成员赋值

3.static 修饰的内容优于对象存在，所以可以被所有对象共享

8-3_工具类

工具类的类名: 功能名+Utils

工具类特点:

只有静态方法

调用方法时: 工具类名.静态方法名();

8-4_Arrays 类

Arrays 类是用来操作数组的工具类

常用的方法:

- static String toString(int[] a) 返回指定数组内容的字符串表示形式。
- static void sort(int[] a) 对指定的 int 型数组按数字升序进行排序。
- static <T> List<T> asList(T... a) 返回一个受指定数组支持的固定大小的列表。

数组转集合:

static <T> List<T> asList(T... a) 返回一个受指定数组支持的固定大小的列表

1.如果是基本数据类型的数组,那么就把整个数组看成是一个元素

```
int[] arr = {1,2,3,4,5,6};
List<int[]> list = Arrays.asList(arr);// list 集合中存储的是 int[] 的地址
System.out.println(list.size());//1
System.out.println(list);//[[I@5f150435]
```

2.如果是引用数据类型的数组,那么就是把引用数据类型的数组里面的每一个元素 添加到 List 集合中

```
String[] arr3 = {"jack","rose","lily","lucy"};
List<String> list2 = Arrays.asList(arr3);
System.out.println(list2.size());//4
System.out.println(list2);// ["jack","rose","lily","lucy"]
```

集合转数组：

Object[] toArray() 按适当顺序（从第一个到最后一个元素）返回包含此列表中所有元素的数组。

```
ArrayList<String> list3 = new ArrayList<>();
list3.add("张柏芝");
list3.add("谢霆锋");
```

```
        list3.add("王菲");
        list3.add("陈冠希");
        Object[] arr5 = list3.toArray();
        for(int i = 0;i<arr5.length;i++){
           System.out.println(arr5[i]);
        }
```

8-5_Math 类

• static int abs(int a)　返回 int 值的绝对值。
• static double ceil(double a) 返回最小的（最接近负无穷大）double 值，该值大于等于参数，并等于某个整数，翻译: 获取 大于等于参数 的 最小整数　例如 : 3.4 -----> 4.0　　-3.4 ----> -3.0
• static double floor(double a) 返回最大的（最接近正无穷大）double 值，该值小于等于参数，并等于某个整数。
翻译: 获取　小于等于参数 的 最大整数　　例如: 3.4---> 3.0　　-3.4----> -4.0
• static int min(int a, int b)　返回两个 int 值中较小的一个。
• static int max(int a, int b)　返回两个 int 值中较大的一个。
• static int round(float a)　返回最接近参数的 int。
• static long round(double a)　返回最接近参数的 long。

9-继承和抽象类

9-1_继承

1.定义

子类继承父类的的属性和方法，使得子类对象具有与父类相同的属性、相同的行为。子类可以直接访问父类中的非私有的属性和方法

好处：

1.提高代码的复用性

2.多态的前提

2.格式:

通过 extends 关键字，可以声明一个类继承另一个类

```
public class 父类 {
    ...
}
public class 子类 extends 父类 {
    ...
}
```

3.继承的特点

a.成员变量:

1.子类和父类没有出现重名的成员变量(非 private 的成员变量)，那么子类访问父类的成员变量是没有问题的

2.子类和父类出现了重名的成员变量(非 private 的成员变量)，那么子类访问父类的成员变量时会有问题，这时候访问成员变量的值实际上是子类的成员变量的值(就近原则:先在方法里面找局部变量,然后在本类中的成员变量找,然后在父类中的成员变量找,都没有在 Object 类中找，如果都没有报错)，这个时候如果我们想访问父类的成员变量(非 private)，就需要使用 super 关键字；通常项目中类的成员变量都设置成私有化(private)，这个时候只能通过父类的公共的 getXxx()和 setXxx()方法进行访问和操作。

格式: super.父类成员变量;

b.成员方法:

1.子类和父类没有出现重名的成员方法(非 private 的成员方法)，那么子类访问父类的成员方法是没有问题的

2.子类和父类出现了重名的成员方法(非 private 的成员方法)，这个时候就叫方法重写(overwrite/override)，方法重写是子类和父类的方法相同(返回值类型，方法名和参数列表相同)，为了不出错直接 copy 父类方法到子类

c.方法重写:子类中出现了和父类同名(返回值类型，方法名和参数列表相同)的方法

重写标志: **@override**

1.子类重写父类的方法，一般是对父类的方法实现不满意，需要对方法进行扩展

子类中独有的方法

子类在父类的基础上添加新的功能

2.子类重写父类方法，子类方法的修饰权限要大于等于父类的修饰权限

private(同一个类)<default(同一个包)<protected(本类和子类，可以位于不同包下)<public(当前项目)

3.调用父类的成员方法(public): super.父类成员方法();

方法重载(overload)：在本类中，方法名一样，方法参数不一样，与返回值无关。

d.构造方法:

1.子类不能继承父类的构造方法，但是可以使用

2.子类的构造方法默认会调用父类的空参构造方法(因为所有类的根类是 Object 类，而 Object 类只有一个空参构造方法)

3.调用父类的构造方法

```
super(); //  调用父类空参构造方法
super(参数); //  调用父类的有参构造方法
```

4.如果父类中没有空参构造方法,只有有参构造方法，子类只能定义有参构造方法，不能定义空参构造方法

```
public class Animal {
    private String name;
    public Animal(String name) {
        this.name = name;
    }
}
public class Dog extends Animal {
    public Dog() {} //error，因为 Animal 定义了有参构造方法，所以就没有默认的构造方法，子类的构造方法会默认调用父类的空参构造方法，而父类没有就会报错
    //  对了
    public Dog(String name) {
        super(name);
    }
}
```

4.super 和 this

- 父类空间优于子类对象产生
- 子类对象包含了父类空间
- super 代表父类引用, this 代表当前对象(谁调用就代表谁)
- this 和 super 的用法

  this(...)　本类构造方法　　super(...) 父类构造方法

  this.成员变量 本类　　super.成员变量　父类

  this.成员方法() 本类　　super.成员方法() 父类
- 子类的每个构造方法都有默认的 super()，手动调用父类构造方法会覆盖默认的 super(), super()和this()都必须在构造方法的第一行，所以不能同时出现。

5.继承特点:

Java 只支持单继承，支持多层继承，所有类的祖先类是 Object 类

9-2_抽象类

1.定义

抽象方法：方法只有声明没有方法体

抽象类: 类里面的方法只有声明没有方法的主体，为了子类重写父类方法，这样的方法叫抽象方法，包含了抽象方法的类叫抽象类

2.抽象方法和抽象类的定义格式

- **抽象方法:**
  修饰符 abstract 返回值类型 方法名(参数列表);
- **抽象类:**
  修饰符 abstract class 类名 {...}

3.抽象的使用

继承抽象类的子类必须要重写父类所有的抽象方法，否则改子类也必须声明为抽象类

```
public abstract class Animal {
  public void say();
}
// 子类不是抽象类
public class Dog extends Animal {
   // 必须重写父类中的 say()方法不然会报错
}
// 子类是抽象类
public abstract class Cat extends Animal {
    // 不需要重写父类中的 say()方法
}
```

子类继承抽象类，对父类抽象方法的重写也叫作实现父类的方法

4.抽象类的注意事项

- **抽象类不能创建对象，但是可以通过抽象类的子类的构造方法初始化抽象类**
- **抽象类可以有构造方法，作用：子类创建对象时初始化父类成员使用的**
- **抽象类不一定包含抽象方法，但是包含了抽象方法的类一定是抽象类**

```
public abstract class PlayGame {
    // 抽象类中可以有成员变量
    private int age;
    // 抽象类中可以有构造方法，通过子类调用父类的构造方法区初始化抽象类
    public PlayGame() {
    }
    public PlayGame(int age) {
    this.age = age;
    }

    // 动物玩耍
    public abstract void play(); // 抽象方法
    // 抽象类中可以有成员方法
    public void show() {
        System.out.println("show 方法");
    }
}
```

抽象类的子类必须重新父类的所有方法，如果不重写，那么子类也要声明为抽象类

```
public class Cat extends PlayGame {
    // 继承了抽象类的子类必须重新抽象类的方法
    @Override
    public void play() {
        System.out.println("猫玩花球");
    }
}
abstract class Dog extends PlayGame {
    // 如果子类继承了抽象类，不想重写父类的方法，那么就声明为抽象类
}
```

10-接口与多态

10-1_接口

1.概述

接口:

• 方法(抽象方法，默认方法，静态方法，私有方法(私有方法和私有静态方法))的封装，用 interface 来定义接口

• 多个类的公共标准规范

2.接口里面的"元素"

总结：接口里面通常只定义抽象方法和常量

a.jdk1.7 及其以前:

常量

接口中只能写常量:

1.定义常量的时候，一定要赋值，并且只能赋值一次

2.接口中的常量有: public static final 修饰的(可以省略，但是不建议)

3.常量的规范：常量名大写

4.接口中的常量供接口直接使用或者供实现类直接使用

格式: public static final 变量名(大写) = 值;

被 final 修饰的变量，会变成常量，只能赋值一次，并且不能改变

抽象方法

格式: **public abstract 返回值类型 方法名()**;

1.接口里面可以视情况可以省略 public 和 abstract

2.实现了接口的类必须重写接口中的所有抽象方法

b.jdk1.8 增加:

默认方法

使用 default 修饰，不可省略，供子类(实现类)调用或者子类重写

1.如果是实现类重写接口中的默认方法，不需要加 default，子接口重写默认方法, default 关键字可以保留

2.默认方法的使用: 1.可以继承, 2.可以重写，但是只能通过实现类的对象来调用

静态方法

静态方法：使用 static 修饰，供接口直接调用(接口名.静态方法名())，不能使用实现类名调用和实现类对象调用

格式: public static 返回值类型 方法名(形式参数列表) {...}

c.jdk1.9 增加:

私有方法

private 返回值类型 方法名(形式参数列表) {...}

1.使用 private 修饰，供接口中的默认方法使用的

私有的静态方法

private static 返回值类型 方法名(形式参数列表) {...}

1.使用 private static 修饰，供默认方法和静态方法以及私有方法(私有方法和私有静态方法)使用

```
public interface 接口名 {
    // 抽象方法
    public abstract void test(); // 接口中的 abstract 可以省略，但是抽象类中的抽象方法 abstract 不能省略
    // 默认方法
    public default void test2() { // default 不能省略
        ...
    }
    // 静态方法
    public static void test3() { // 供接口直接调用
```

```
        ...
    }
    // 私有方法: 私有方法和私有静态方法,  供接口的默认方法和静态方法调用
    private void test4() {
        ...
    }
    private static void test5() {
        ...
    }
}
```

3.接口与接口，接口与类之间的关系

class 类名 [extends 父类名] implements 接口名 1,接口名 2,接口名 3... {
// 重写接口中抽象方法【必须】
// 重写接口中默认方法【不重名时可选】
}

接口和类之间的关系: 实现关系

单实现: 一个类实现一个接口

多实现: 一个类实现多个接口

多实现中抽象方法的特点:

1.多个接口中有不同名的抽象方法，必须全部实现

2.多个接口中有同名的抽象方法，只需要实现一个

多实现中默认方法的特点

1.多个接口中有不同名的默认方法，正常使用，不需要重写

2.多个接口中有同名的默认方法，必须重写

多实现中静态方法的特点

只能供接口使用，没有影响

多实现中私有方法的特点

只能在接口中使用，没有影响

格式:

public class 实现类名 implements 接口 A，接口 B, ... {...}

接口和接口直接的关系

单继承: 一个接口继承一个接口

多继承: 一个接口继承多个接口

多层继承: 一层一层继承

格式:

public interface 接口 A extends 接口 B，接口 C, ...{...}

优先级

类有继承也有实现接口，就要先继承后实现接口

格式:

public class 实现类 extends 父类 implements 接口名 1,接口名 2,...{
}

4.注意事项

- **接口也是一种引用数据类型**
- **接口不能创建对象，可以通过实现类来创建接口的对象**
- 接口文件编译之后也会生成.class 文件
- 接口没有构造方法
- 接口的方法默认省略了 public abstract
- 当一个类，既继承一个父类，又实现若干个接口时，

父类中的成员方法和接口中的默认方法重名时，子类就近选择执行父类的成员方法

- **接口中，没有静态代码块**

10-2_多态

1.概述

不同的对象以自己的方式响应相同名称方法的能力称为多态

实现多态的前提:

1.要有继承或者实现

2.要有方法的重写

3.父类引用指向子类对象

2.多态的体现

父类类型 变量名 = new 子类对象; // 父类类型: 子类继承的父类类型或者实现了的接口类型

变量名.方法名();

3.多态中的成员访问特点

多态中的成员访问特点

成员变量: 编译看左边,运行看左边

成员方法:

非静态方法:编译看左边,运行看右边

静态方法: 编译看左边,运行看左边

编译都看左边, 运行除了非静态方法(看右边)都是看左边!!

注意: 当使用多态调用方法时, 首先检查父类中是否有该方法, 如果没有, 则编译报错; 如果有, 则执行的是子类重写后的方法。

4.多态的优点

扩展性增强: 父类类型作为方法的形参, 传递子类对象给方法, 进行方法的调用

实际开发的过程中，父类类型作为方法形式参数，传递子类对象给方法，进行方法的调用，更能体现出多态的扩展性与便利

```
// 父类
public abstract class Animal {
    public abstract void eat();
}

// 子类
class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
    }
}

// 测试类
public class Test {
    public static void main(String[] args) {
        // 多态形式，创建对象
        Cat c = new Cat();
        Dog d = new Dog();

        // 调用 showCatEat
        showCatEat(c);
        // 调用 showDogEat
```

```
        showDogEat(d);

        /*
        以上两个方法，均可以被 showAnimalEat(Animal a)方法所替代
        而执行效果一致
        */
        showAnimalEat(c); //  父类类型接收子类对象
        showAnimalEat(d); //  父类类型接收子类对象
    }

    public static void showCatEat (Cat c){
        c.eat();
    }

    public static void showDogEat (Dog d){
        d.eat();
    }

        }

        public static void showAnimalEat (Animal a){
            a.eat(); //  执行子类重写的方法
        }
}
```

5.引用类型转型

向上转型:子类对象 赋值给 父类的引用　　向上转型自动完成-安全的　　有点类似基本数据类型的自动类型转换

格式:　父类名　父类引用 =　new 子类名(实际参数);　-->多态体现

向下转型:父类的引用 赋值给 子类的引用 向下转型需要程序员收到完成-不安全的　　有点类似基本数据类型的强制类型转换

格式:　子类名 子类引用 = (子类名)父类引用;

```
//  转型异常
public class Test {
    public static void main(String[] args) {
        //  向上转型
        Animal a = new Cat();
        a.eat();                    //  调用的是 Cat 的 eat

        //  向下转型
        Dog d = (Dog)a;
        d.watchHouse();             //  调用的是 Dog 的 watchHouse 【运行报错: ClassCastException 类型转换异常】
    }
}
```

为了避免类型转换异常的问题，使用 instanceof 关键字，给引用变量做类型的校验

变量名 instanceof 数据类型

如果变量属于该数据类型，返回 true。

如果变量不属于该数据类型，返回 false。

```
public class Test {
    public static void main(String[] args) {
        //  向上转型
        Animal a = new Cat();
```

```
            a.eat();                    // 调用的是 Cat 的 eat

            // 向下转型
            if (a instanceof Cat){
                Cat c = (Cat)a;
                c.catchMouse();          // 调用的是 Cat 的 catchMouse
            } else if (a instanceof Dog){
                Dog d = (Dog)a;
                d.watchHouse();          // 调用的是 Dog 的 watchHouse
            }
        }
}
```

11-final、内部类、权限修饰符

11-1_final 关键字

final 关键字: 最终, 终态

1.修饰类

格式:

```
public final class 类名 {
    //...
}
```

特点: 被 final 修饰的类不能被继承

2.修饰方法

格式:

```
修饰符 final 返回值类型 方法名(参数) {
    //方法体
}
```

特点:被 final 修饰的方法不能被子类重写

3.修饰成员变量

格式:

```
final 数据类型 变量名 = 值;
```

final 数据类型 变量名; 变量名 = 值;

修饰局部变量: 被 final 修饰的局部变量会变成常量, 只能赋值一次

final 修饰基本数据类型的局部变量, 该局部变量的值是不能更改的

final 修饰引用数据类型的局部变量, 该局部变量里面的地址值是不能更改的, 但是指向的对象里面的属性值是可以更改的

```
final Person p = new Person("jack", 20);
p = new Person("rose", 19); // error
p.setName("rose"); // ok
```

修饰成员变量: 被 final 修饰的成员变量会变成一个常量, 只能赋值一次

显示赋值: **final 数据类型 变量名 = 值;**

构造方法赋值: 必须保证所有的构造方法都能被 final 修饰的成员变量赋值;

```
public class Person {
    final int AGE;
    public Person() {
        this.AGE = age;
    }
    public Person(age) {
        this.AGE = age;
    }
```

```
        }
```
特点：常量名的规范是所有字母大写

11-2_权限修饰符

/	public	protected	default	private
同一个类中	√	√	√	√
同一个包中	√	√	√	×
不同包的子类中	√	√	×	×
不同包的非子类中	√	×	×	×

11-3_内部类

概述：类 A 里面包含类 B，类 A 就是外部类，类 B 就是内部类

成员内部类

格式：

```
public class 外部类 {
    class 内部类 {
        ...
    }
}
```

使用特点：

1.内部类可以直接访问外部类的成员，私有的都可以访问

2.外部类不可以直接访问内部类的成员，需要创建内部类对象的访问

成员内部类创建对象的格式：

```
外部类.内部类 对象名 = new 外部类().new 内部类();
```

注意事项：

外部类的权限修饰符: public/default

内部类的权限修饰符: public/protected/default/private

内部类访问外部的成员变量：外部类名.this.外部成员变量名;

匿名内部类

匿名对象：没有名字的对象

```
new Dog().eat();
```

匿名内部类

格式：

```
new 类名/接口名() {
    // 实现抽象方法
};
```

本质：继承了类的子类(实现了接口的实现类) 的匿名对象

```
new Animal() {
    public void eat() {
        System.out.println("匿名内部类的 eat 方法");
    }
}.eat(); // 调用了 Animal 中的 eat()方法
```

等价于: new Dog().eat();

11-4_引用数据类型总结

```
局部变量
public class Hero {
    public void attack() {
        Weapon wp = new Weapon();
        wp.sword();
```

```
    }
}

public class Weapon {
    public void sword() {
        System.out.println("大宝剑");
    }
}
    成员变量
public class Hero {
    private Weapon wp;
    public void attack() {
        wp = new Weapon();
        wp.sword();
    }
}

public class Weapon {
    public void sword() {
        System.out.println("大宝剑");
    }
}
    方法的参数
public class Hero {

    public void attack(Weapon wp) {
        wp = new Weapon();
        wp.sword();
    }
}

public class Weapon {
    public void sword() {
        System.out.println("大宝剑");
    }
}
    方法的返回值
public class Hero {
    private Weapon wp;
    public Weapon attack() {
        wp = new Weapon();
        wp.sword();
        return wp;
    }
}

public class Weapon {
    public void sword() {
        System.out.println("大宝剑");
    }
}
```

12-API(常用类)下

12-1_Object 类

1.Object 类是所有类的根类，位于 **java.lang.Object** 包下.

public boolean equals(Object o) 判断两个对象是否相等

默认比较的是对象的地址值

重写比较对象的属性值是否相同, **alt+insert**

public String toString() 返回对象的字符串表示形式。

tips:

1.打印对象: 包名.类名@哈希地址值(十六进制);

2.如果想要打印一个对象，并且希望打印的是该对象的属性值，那么就需要重写从 Object 类继承过来的 toString()方法，快捷键:alt+insert;

2.Objects 类(jdk7 添加的工具类 java.util.Objects，用于操作对象)，**优化了 Object 类中 equals()方法没有优化的空指针异常问题**

推荐使用: Objects.equals();

public static boolean equals(Object a, Object b) 判断两个对象是否相等

源代码:

```
public static boolean equals(Object a, Object b) {
    return (a == b) || (a != null && a.equals(b));
}
```

12-2_日期时间类

1.Date 类: **java.util.Date** 包下

构造方法:

- **public Date()** 获取当前时间对象
- **public Date(long date)** 获取基于 1970 年 1 月 1 日 00:00:00 的时间戳(单位是毫秒) 1 秒=1000 毫秒，tips: 中国处于东 8 区，我们的基准时间是 1970 年 1 月 1 日 08:00:00

常用方法:

- **public long getTime()** 获取当前时间对象的时间戳(距离 1970.1.1 的毫秒值)

2.DateFormate 类: java.text.DateFormate 包下

抽象类不能创建对象，只能通过其实现类 SimpleDateFormate(java.text.SimpleDateFormate)来创建.

- **格式化: 按照指定的格式，从 Date 对象转换为 String 对象.**
- **解析: 按照指定的格式，从 String 对象转换为 Date 对象**

SimpleDateFormate 类:

构造方法:

public SimpleDateFormate(String pattern) 以指定的格式构造 SimpleDateFormate

常用的格式规则:

y-年, M-月, d-日, H-时, m-分, s-秒

常用方法:

public String formate(Date date) 将 Date 对象格式化为字符串

public Date parse(String source) 将字符串解析为 Date 对象

3.Calendar 类: java.util.Calendar 包下，抽象类，通过 getInstance()方法获取对象

获取日历对象:

public static Calendar getInstance()

常用方法:

public int get(int field)：返回给定日历字段的值。

public void set(int field, int value)：将给定的日历字段设置为给定值。

public abstract void add(int field, int amount)：根据日历的规则，为给定的日历字段添加或减去指定的时间量。

public Date getTime()：返回一个表示此 Calendar 时间值（从历元到现在的毫秒偏移量）的 Date 对象。把日历对象转换为日期对象

字段值	含义
YEAR	年
MONTH	月
DAY_OF_MONTH	月中的天(几号)
HOUR	时(12 小时)
HOUR_OF_DAY	时(24 小时)
MINUTE	分
SECOND	秒
DAY_OF_WEEK	周中的天(周日为 1)

12-3_System 类

System 类: java.lang.System 包

需求: 获取一个循环的执行时间

分析:

1.获取开始时间

2.获取结束时间

3.结束时间-开始时间

常用方法:

- **public static long currentTimeMillis()**：返回以毫秒为单位的当前时间。
- **public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)**：将数组中指定的数据拷贝到另一个数组中。

src-源数组, srcPos-源数组索引起始位置, dest-目标数组, destPos-目标数组索引起始位置, length-复制元素个数

12-4_StringBuilder 类

StringBuilder 类: 可变字符序列, 默认 16 字符空间, 超过会自动扩容

- **String 类的对象时不可变的-->String 类表示的是不可变的字符串对象**
- **StringBuilder 类的对象时可变的-->StringBuilder 类表示的是可变的字符串对象**

构造方法:

public StringBuilder()：构造一个空的 StringBuilder 容器。

public StringBuilder(String str)：构造一个 StringBuilder 容器，并将字符串添加进去。

常用方法:

public StringBuilder append(...)：添加任意类型数据的字符串形式，并返回当前对象自身。

public String toString()：将当前 StringBuilder 对象转换为 String 对象。

public StringBulider reverse() 字符串反转

StringBuilder 和 StringBuffer 的区别:

- **StringBuilder 线程不安全, 但是如果不考虑线程问题, 它的执行效率更高, 推荐使用**
- **StringBuffer 线程安全**

12-5_包装类

基本数据类型对应的类类型-包装类

基本类型	对应的包装类(java.lang 包中)
byte	Byte
short	Short
int	Integer
long	Long

float	Float
double	Double
char	Character
boolean	Boolean

装箱与拆箱

```
装箱：从基本数据类型转换为对应的包装类对象
    Integer i = new Integer(4); // 被弃用了
    Integer i2 = Integer.valueOf(4);
拆箱：从包装类对象转换为对应的基本类型
    int i3 = i.intValue();
自动装箱:
    int a = 1;
    Integer a2 = a;
自动拆箱:
    Integer b = Integer.valueOf(3);
    int b2 = b;
```

基本数据类型与字符串之间的转换

```
基本数据类型转字符串:
    int a = 1;
    String str = a + ""; // 第一种
    String strNew = String.valueOf(a); // 第二种
    StringBuilder sb = new StringBuilder();
    String strNew2 = sb.append(num).toString(); // 第三种
字符串转基本数据类型:
    String str = "20";
    int num = Integer.parseInt(str);
String 转换成对应的基本类型
  除了 Character 类之外，其他所有包装类都具有 parseXxx 静态方法可以将字符串参数转换为对应的基本类型：
    public static byte parseByte(String s)：将字符串参数转换为对应的 byte 基本类型。
    public static short parseShort(String s)：将字符串参数转换为对应的 short 基本类型。
    public static int parseInt(String s)：将字符串参数转换为对应的 int 基本类型。
    public static long parseLong(String s)：将字符串参数转换为对应的 long 基本类型。
    public static float parseFloat(String s)：将字符串参数转换为对应的 float 基本类型。
    public static double parseDouble(String s)：将字符串参数转换为对应的 double 基本类型。
    public static boolean parseBoolean(String s)：将字符串参数转换为对应的 boolean 基本类型。
```

13-Collection、Iterator 和泛型

13-1_Collection 集合

```
1.集合概念
        数组：是一个容器，是一个存储固定个数单一数据类型的容器
            特点：数组长度是固定的， 数组中的元素的数据类型是一致的
        集合：是一个容器，是一个可以存储多个数据的容器
            特点：集合的长度是不固定的，集合中的元素的数据类型可以一致，也可以不一致。
2.集合分类
        单列集合：
            Collection 接口：
                List 接口: 元素有序，可重复
```

Vector：过时，底层是数组结构，增删慢，查询快，线程安全的，效率低。
ArrayList: 数组结构，增删慢，查询快，线程不安全的，效率高，常用
LinkedList: 链表结构，增删快，查询慢，线程安全的，效率高
Set 接口：元素无序，不能重复
HashSet：
LinkedHashSet:
TreeSet:
双列集合：
Map:
HashMap
TreeMap

3.Collection 接口常用方法
public boolean add(E e): 把给定的对象添加到当前集合中 。
public void clear(): 清空集合中所有的元素。
public boolean remove(E e): 把给定的对象在当前集合中删除。
public boolean contains(E e): 判断当前集合中是否包含给定的对象。
public boolean isEmpty(): 判断当前集合是否为空。
public int size(): 返回集合中元素的个数。
public Object[] toArray(): 把集合中的元素，存储到数组中。

13-2_Iterator 迭代器

1.作用：用于遍历集合中的元素
2.常用方法：
获取集合中的迭代器：
public Iterator iterator(): 获取集合对应的迭代器，用来遍历集合中的元素的。
迭代器常用方法：
public E next():返回迭代的下一个元素。
public boolean hasNext():如果仍有元素可以迭代，则返回 true。
3.增强 for 循环
for (元素的数据类型 变量 : Collection 集合 Or 数组) {
...
}

13-3_泛型

1.概念
未知的数据类型，在使用的时候才确定数据类型
2.注意：
泛型里面只能是引用数据类型!!!
3.使用泛型的好处
- 将运行时期的 ClassCastException，转移到了编译时期变成了编译失败
- 避免了类型强转的麻烦

4.泛型的定义和使用
- **泛型类的定义和使用**

概述: 泛型定义在类上
格式：
修饰符 class 类名<泛型> {
...
}
使用：
在创建该类的对象的时候确定泛型的类型
- **泛型方法的定义和使用**

概述：泛型定义在方法上

格式：

修饰符 <泛型> 返回值类型 方法名(参数) {

...

}

使用：

在调用泛型方法的时候确定泛型的类型

- 泛型接口的定义和使用

概述：泛型定义在接口上

格式：

修饰符 interface 接口名<泛型> {

...

}

使用：

1.定义实现类的时候确定泛型

格式：

修饰符 class 类名 implements 接口名<具体的类型> {

...

}

2.一直都不确定泛型的类型，直到创建实现类的对象的时候，确定泛型

格式：

修饰符 class<泛型> 类名 implements 接口名<泛型> {

...

}

- 泛型的通配符

不知道使用什么类型来接收的时候，此时可以使用?, ?表示未知通配符

Collection<?> col - 集合里面可以接收任意类型的数据

- 泛型的高级通配符

泛型的上限：

- 格式： 类型名称 <? extends 类 > 对象名称
- 意义： 只能接收该类型及其子类

泛型的下限：

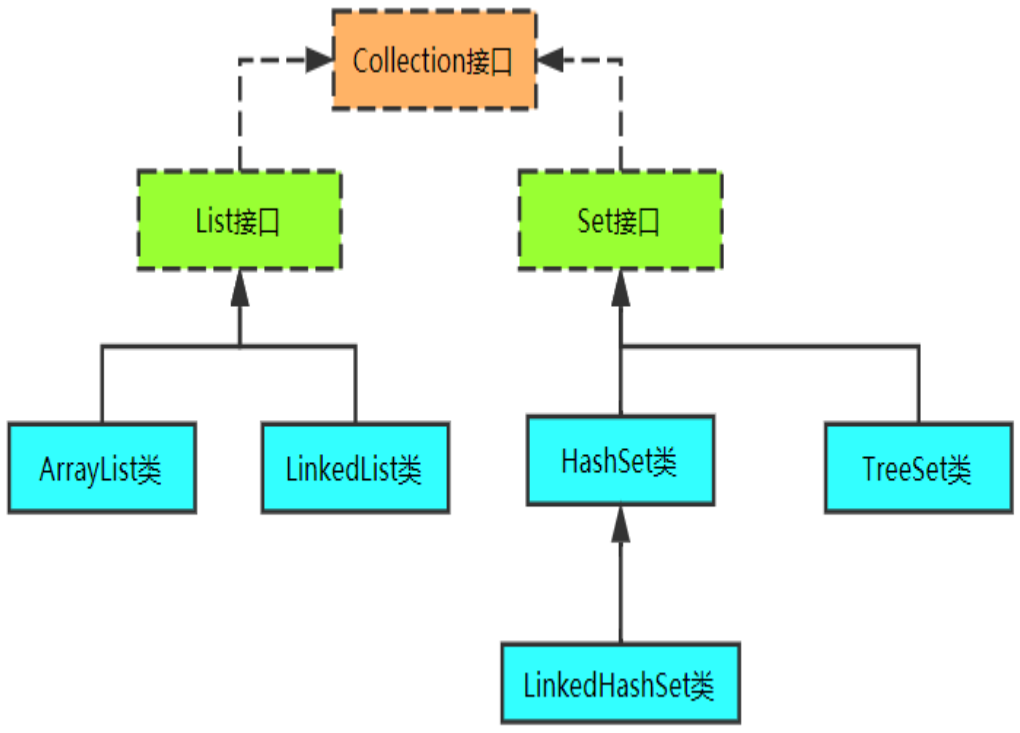
- 格式： 类型名称 <? super 类 > 对象名称
- 意义： 只能接收该类型及其父类型

注意：

1.如果泛型一直都没有指定具体的数据类型，那么默认就是 Object 类型

2.泛型不存在继承关系 Collection<Object> list = new ArrayList<String>();这种是错误的。

14-List、Set、Collections 和可变参数



Collection是顶层接口
list和set都属于单列集合
list：有序且元素可重复，有索引
set：无序且元素不可重复
应用场景：
ArrayList: 查询快，线程不安全，常用方法:增、删、改、查
LinkedList: 增删快，线程安全，常用方法：增、删、改、查
HashSet: 去重(哈希表), 重写equals和hashCode方法，常用方法：增、删
LinkedHashSet: 去重(哈希表)+存储顺序，重写equals和hashCode方法，常用方法：增、删
TreeSet: 去重(重写equals和hashCode方法)+升/降序(compareTo方法保证)，常用方法：增、删
特点：
都有迭代器
两个比较器：Comparable接口的compareTo方法(TreeSet默认实现)、Comparator接口的compare方法(ArrayList和LinkedList可用来排序)

14-1_List 接口

1.List 接口

List 接口特点：

1.存取有序

2.有索引

3.元素可以重复

List 接口的常用方法:

增 - public void add(int index, E element): 将指定的元素，添加到该集合中的指定位置上。

注意：添加到指定索引处时原先的值并不会被覆盖，而是向后移动

删 - public E remove(int index): 移除列表中指定位置的元素, 返回的是被移除的元素。

查 - public E get(int index):返回集合中指定位置的元素。

改 - public E set(int index, E element):用指定元素替换集合中指定位置的元素,返回值是更新前的元素。

List 接口的子类(实现类):

Vector 过时

ArrayList

LinkedList

2.ArrayList 集合(List 接口的实现类)

特点：

- **底层是数组实现的，所以查询快，然后增删慢，而且线程不安全**
- **效率高且最常用**

常用方法：

增：

public boolean add(E e): 添加指定元素到集合的尾部

public void add(int index, E element) 将指定的元素插入此列表中的指定位置。

删:

public E remove(int index): 删除指定位置的元素，返回删除的元素

改:

public E set(int index, E element): 用指定元素代替集合中指定位置的元素，返回被替代的元素

查:

根据索引查值：

public E get(int index): 获取指定位置的元素，返回获取的元素

根据值查索引：

public int indexOf(Object o) 返回此列表中首次出现的指定元素的索引，或如果此列表不包含元素，则返回 -1。

public int lastIndexOf(Object o) 返回此列表中最后一次出现的指定元素的索引，或如果此列表不包含索引，则返回 -1。

其他方法：

获取长度：public int size(): 获取集合中的元素个数(集合的大小或者长度)

判断集合是否含有某个元素：public boolean contains(Object o): 判断指定元素是否在集合中

集合是否为空：public boolean isEmpty() 如果此列表中没有元素，则返回 true

转数组：public Object[] toArray() 按适当顺序（从第一个到最后一个元素）返回包含此列表中所有元素的数组。

3.LinkedList 集合(List 接口的实现类)

LinkedList 集合:

查询慢,增删快

LinkedList 集合特有的方法

增：

- public void addFirst(E e):将指定元素插入此列表的开头。
- public void addLast(E e):将指定元素添加到此列表的结尾。
- public void push(E e):将元素推入此列表所表示的堆栈.换句话就是在此集合前面插入一个元素

删：

- public E removeFirst():移除并返回此列表的第一个元素。
- public E removeLast():移除并返回此列表的最后一个元素。
- public E pop():从此列表所表示的堆栈处弹出一个元素。换句话就是删除集合中第一个元素

查：

- public E getFirst():返回此列表的第一个元素。
- public E getLast():返回此列表的最后一个元素。

其他方法：

- public boolean isEmpty()：如果列表不包含元素，则返回 true。

14-2_Set 接口

1.Set 接口

Set 集合:

Set 集合的特点: 存取无序,无索引,元素不能重复

Set 集合的子类(实现类):

HashSet: 由哈希表来保证集合中的元素唯一

LinkedHashSet: 由哈希表来保证集合中的元素唯一,由链表保证集合中的元素存取有序

TreeSet: 由树状结构保证集合中的元素唯一

常用方法：

增：

boolean add(E e) 向集合中添加元素

删：

boolean remove(Object o) 删除元素

无查和改的具体方法，**查询可用 Iterator 迭代器**

2.HashSet 集合(Set 接口的实现类)

HashSet 集合存储元素的规则

HashSet 集合：根据对象的哈希值来确定元素在集合中的存储位置，因此具有良好的存取和查找性能，**HashSet 集合里面不能存储重复的元素，那么这是怎么确保的呢？**

• hashCode()和 equals()方法

如果假设 HashSet 集合要存储自定义类型的对象,实现元素的唯一性,就必须重写 hashCode()和 equals()方法

HashSet 底层原理(非常重要!!!)-熟稔于心

HashSet 的底层是哈希表结构，那么哈希表是啥？

哈希表：

在 **jdk1.8** 之前：数组**+**链表组成

在 **jdk1.8** 之后，数组存哈希值，然后一个哈希值对应一个链表或者红黑树；

如果一个哈希值对应的元素小于 **8** 个，那么哈希表存储方式就采用：数组**+**链表

如果一个哈希值对应的元素大于 **8** 个，那么哈希表存储方式就采用：数组**+**链表**+**红黑树

HashSet 存储元素的规则：

1.先调用元素的 **hashCode()**方法计算出该元素的哈希值；

2.然后判断该哈希值位置上对应额链表**(**红黑树**)**上是否有相同的元素

a.如果该位置上没有相同哈希值的元素，就存储

b.如果该位置上有相同哈希值的元素，就产生哈希冲突

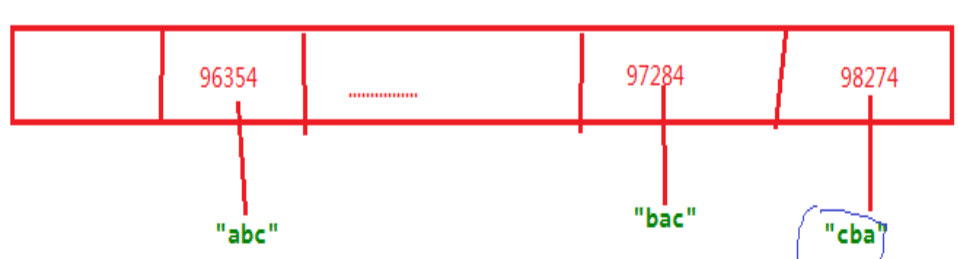
产生了哈希冲突，就会再调用元素的 **equals()**方法，和改位置上的所有元素进行比较：

如果有一个返回 **true**，则不存

所有的返回 **false**，则存储

HashSet是根据对象的哈希值来确定元素在集合中的存储位置，因此具有良好的存取和查找性能。保证元素唯一性的方式依赖于：hashCode与equals方法。

```
HashSet<String> set = new HashSet<>();
set.add("cba");
set.add("bac");
set.add("abc");
set.add("cba");
System.out.println(set); // [cba, abc, bac]
```



HashSet集合的底层是 : 哈希表结构

哈希表:

jdk1.8之前: 数组+链表组成

jdk1.8之后,同一位置上的元素是否超过8

如果同一位置上的元素个数小于8, 就是 数组+ 链表

如果同一位置上的元素个数大于8,就是 数组+链表+红黑树

如果假设HashSet集合要存储自定义类型的对象,实现元素的唯一性,就必须重写hashCode()和equals()方法

HashSet存储元素的规则:

1.先调用元素的hashCode()方法计算出哈希值

2.然后判断该哈希值位置上是否有相同哈希值的元素,

3.如果该位置上没有相同哈希值的元素,就存储

4.如果该位置上有相同哈希值的元素,就产生了哈希冲突

5.如果产生了哈希冲突,就会再调用元素的equals()方法,和该位置上的所有元素进行比较,只要有一个返回true,就不存,所有的都是返回false,就存储

Object类中的equals方法比较的是2个对象的地址值

Object类中的hashCode()方法比较的是2个对象的哈希值

一般我们认为,2个对象是否相等,是看2个对象的属性值是不是全部相同

TreeSet 集合(Set 接口的实现类)

TreeSet 集合的特点:

1.按照自然顺序排序

2.元素唯一

默认规则排序: 写死的规则

下面的知识需要-熟稔于心

TreeSet 存储元素依据的是 compareTo()方法,所以 TreeSet 集合中的元素要实现排序,该元素所属的类必须实现 Coparable 接口中的 compareTo()方法

比较原则: 最后的值排序采用"中序遍历"

compareTo() 返回的值是 0,就表示只存一个

compareTo() 返回的是负数,就倒序存储

compareTo() 返回的是正数,就怎么存就怎么取

this.age - o.age:

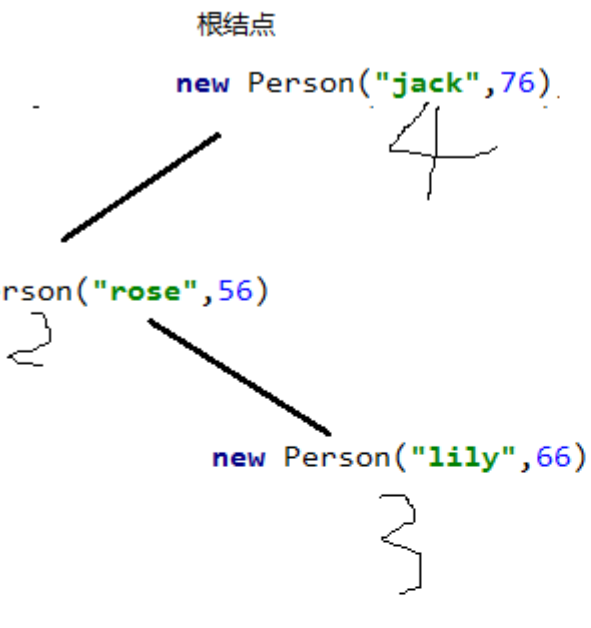
负数存存左面，整数存右面，0 只存一个，最后取值从左子树开始取

前序遍历：根-左子树-右子树

中序遍历：左子树-根-右子树
后序遍历：左子树-右子树-根
到底是升序还是降序？
参数前减后是降序，参数后减前是升序
Comparable 接口：内部比较器　　按照默认规则排序
compareTo()方法
Comparator 接口：外部比较器　　灵活自定义规则排序
compare()方法

TreeSet 集合存储元素的原理

```
// 需求：用TreeSet集合存储Person类型的对象
Person p1 = new Person("jack",76);
Person p2 = new Person("rose",56);
Person p3 = new Person("lily",66);
Person p4 = new Person("lucy",16);
TreeSet<Person> set = new TreeSet<>();
set.add(p1);
set.add(p2);
set.add(p3);
set.add(p4);
System.out.println(set);
```



二叉树:只有2个叉

小的存储在左边,相等就不存,大的就存储在右边

TreeSet集合存储元素依赖于Comparable接口中的compareTo()方法

```
compareTo() 返回的值是0,就表示只存一个
compareTo() 返回的是负数,就倒叙存储
compareTo() 返回的是正数,就怎么存就怎么取
```

```
//按照年龄从小到大的顺序
// 如果this.age - o.age 返回的负数,就存在左边
//  如果this.age - o.age 返回的正数,就存在右边
// 如果this.age - o.age 返回的相等,就不存
return this.age - o.age;
```

14-3_Collections 集合工具类

- java.utils.Collections 是集合工具类，用来对集合进行操作。部分方法如下：
- public static <T> boolean addAll(Collection<T> c, T... elements):往集合中添加一些元素。
- public static void shuffle(List<?> list) 打乱顺序:打乱集合顺序。
- public static <T> void sort(List<T> list):将集合中元素按照默认规则排序。 内部比较器,集合中元素所属的类定义的比较规则
- public static <T> void sort(List<T> list，Comparator<? super T>):将集合中元素按照指定规则排序。外部比较器,临时指定规则
　　排序规则
　　前减后：升序
　　后减前：降序

14-4_可变参数

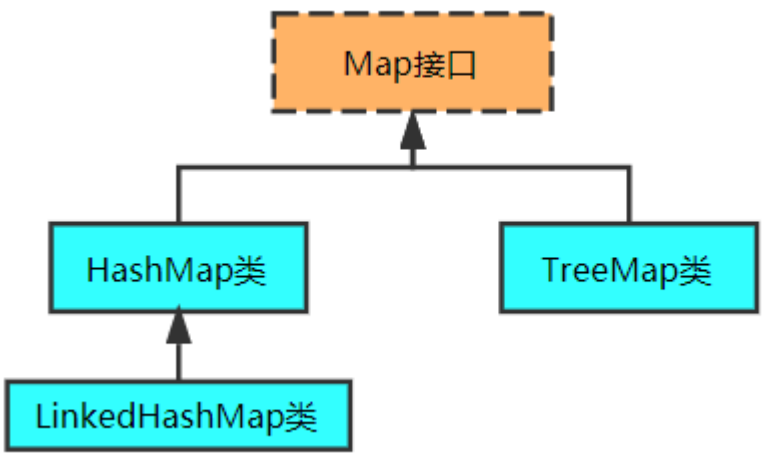
可变参数:如果我们定义一个方法需要接受多个参数，并且多个参数类型一致,就可以定义可变参数
　　修饰符 返回值类型 方法名(参数类型... 形参名){ }
　　注意事项:

如果在方法书写时，这个方法拥有多参数，参数中包含可变参数，可变参数一定要写在参数列表的末尾位置。

14-5_应用场景

如果集合查询多：就用 **ArrayList**
如果集合增删多：就用 **LinkedList**
如果集合要去重：就用 **HashSet-hashCode** 和 **equals** 方法保证唯一性
如果集合要去重，又要按照存取顺序保存：就用 **LinkedHashSet-hashCode** 和 **equals** 方法保证唯一性，链表保证一定顺序
如果集合要去重，要排序：就用 **TreeSet-**比较时默认调用 **Comparable** 接口的 **compareTo()**方法。

15-Map 集合



双列集合:
Map接口:
1.以key-value方式存储
2.key是唯一的，value可以重复，如果键重复，那么后面添加的就会覆盖前面的
3.通过key找value
HashMap<K,V>:
存储数据采用的哈希表结构，元素的存取顺序不能保证一致。由于要保证键的唯一、不重复，需要重写键的hashCode()方法、equals()方法。
LinkedHashMap<K, V>:
HashMap下有个子类LinkedHashMap，存储数据采用的哈希表结构+链表结构。通过链表结构可以保证元素的存取顺序一致； 通过哈希表结构可以保证的键的唯一、不重复，需要重写键的hashCode()方法、equals()方法。
TreeMap<K,V>:
由树状结构保证键的唯一

<K,V> 代表键和值的泛型,键和值的数据类型可以一致,也可以不一致

15-1_Map 接口

双列集合:
Map 接口:
1.以键值对的形式存储元素
2.键是唯一的,值可以重复,如果键重复了,就会覆盖
3.通过键可以找到对应的值
Map 接口中定义的常用方法(重点掌握)
- public V put(K key, V value): 把指定的键与指定的值添加到 Map 集合中。
- public V get(Object key) 根据指定的键，在 Map 集合中获取对应的值。
- public V remove(Object key): 把指定的键 所对应的键值对元素 在 Map 集合中删除，返回被删除元素的值。
- public Set<K> keySet(): 获取 Map 集合中所有的键，存储到 Set 集合中。
- public Set<Map.Entry<K,V>> entrySet(): 获取到 Map 集合中所有的 键值对对象 的集合(Set 集合)。

获取 Map 集合中元素的两种方式(重点掌握)

1.键找值方式：即通过元素中的键，获取键所对应的值

分析步骤：

1. 获取 Map 中所有的键，由于键是唯一的，所以返回一个 Set 集合存储所有的键。方法提示:keyset()
2. 遍历键的 Set 集合，得到每一个键。
3. 根据键，获取键所对应的值。方法提示:get(K key)

2.键值对方式：即通过集合中每个键值对(Entry)对象，获取键值对(Entry)对象中的键与值。

操作步骤与图解：

1. 获取 Map 集合中，所有的键值对(Entry)对象，以 Set 集合形式返回。方法提示:entrySet()。
2. 遍历包含键值对(Entry)对象的 Set 集合，得到每一个键值对(Entry)对象。
3. 通过键值对(Entry)对象，获取 Entry 对象中的键与值。 方法提示:getkey() getValue()

15-2_Map 接口的实现类

HashMap<K,V>：

• 存储数据采用的哈希表结构，元素的存取顺序不能保证一致。

由于要保证键的唯一、不重复，需要重写键的 hashCode()方法、equals()方法。

• 一般我们认为属性都相同的 2 个对象,是同一个对象

• 当给 HashMap 中存放自定义对象时，如果自定义对象作为 key 存在，这时要保证对象唯一，必须复写对象的 hashCode 和 equals 方法(如果忘记，请回顾 HashSet 存放自定义对象)。

LinkedHashMap<K,V>：

• HashMap 下有个子类 LinkedHashMap，存储数据采用的哈希表结构+链表结构。

• 通过链表结构可以保证元素的存取顺序一致；

• 通过哈希表结构可以保证的键的唯一、不重复，需要重写键的 hashCode()方法、equals()方法。

TreeMap<K,V> :

• 由树状结构保证键的唯一

• <K,V> 代表键和值的泛型,键和值的数据类型可以一致,也可以不一致

15-3_JDK9 集合优化和 IDEA 使用 Debug

1、**of()方法只是 Map，List，Set 这三个接口的静态方法**，其父类接口和子类实现并没有这类方法，比如 HashSet，ArrayList 等待；

2、返回的集合是不可变的；

```
// List
List<String> list1 = List.of("jack","rose","lily");
// list1.add("lucy");错误的,因为 list1 是不可变的
System.out.println(list1);
// Set
Set<Integer> set = Set.of(34,2,4,56,6);
System.out.println(set);
// Map
Map<String,String> map = Map.of("k1","v1","k2","v2");
System.out.println(map);
//  如果只是想执行当前这一步,跳转到下一步就用 step over--F8
//  如果想看这一步里面具体怎么执行的,就用 step into--F7
```

16-异常与线程上篇

16-1_复习

集合：java.util 包下

概述：集合就是一个容器，可以存储多个数据
单列集合：
Collection：定义了单列集合的共性方法
add(E e), remove(Object o), size(), isEmpty(), contains(Object o), clear()
迭代器 Iterator　　遍历方式：迭代器遍历，增强 for 循环遍历
list: 存取有序，有索引，元素可重复
遍历方式：迭代器，增强 for 循环，for 循环
增　add(int index, E e)
删　remove(int index), remove(Obejct o)，带有索引的 remove 方法优先
改　set(int index, E e)
查　get(int index)
Vector　过时
数组结构，查询快，增删慢，线程安全，效率慢
ArrayList
数组结构，查询快，增删慢，线程不安全，效率高
LinkedList
链表结构，查询慢，增删快，线程不安全，效率高
set: 存取无序，无索引，元素不可重复(唯一)
遍历方式：迭代器遍历，增强 for 循环
HashSet；由哈希表保证元素的唯一, HashSet 集合中的元素的类型必须重写 hashCode()和equals()方法
LinkedHashSet: 存取有序，没有索引，元素唯一的，有链表保证元素存取有序，有哈希表保证元素唯一
TreeSet:　树状结构保证元素的唯一　依赖于 compareTo()方法
TreeSet() 使用的默认比较器　Comparable 接口-->compareTo()方法
TreeSet(Comparator cpt) 依赖于外部比较器　Comparator-->compare()方法
双列集合：
Map: 以键值对的形式存取元素，键是唯一的，值是可以重复的，根据键就可以找到对应值
put(K k, V v), size(), remove(K k), KeySet(), entrySet(), get(K k), containsKey(K k), containsValue(V v)
HashMap: 由 Hash 表保证键的唯一
LinkedHashMap：由链表保证键值对存取有序，由哈希表保证键唯一
TreeMap：由树状结构保证键唯一

16-2_Exception 异常

学知识：
是什么？
什么时候用？
怎么去用？
1.异常：
异常：指的是程序在执行过程中，出现的非正常的情况，最终会导致 jvm 的非正常停止。
程序运行期间出现的不正常情况，导致情况终止。
异常本身就是一个类, 产生异常创建异常对象并抛出了一个异常对象，Java 处理异常的方式是中断处理。
异常对象：异常的类型，异常的信息，异常的位置等信息。
异常的体系结构：
异常的根类：java.lang.Throwable 类
两个子类：java.lang.Error 和 java.lang.Exception
Throwable 类:
Error:　严重错误 Error, 无法通过处理的错误，只能事先避免
服务器宕机, 数据库奔溃
Exception: 异常, 可以通过代码去纠正，是必须要处理的
异常的分类：

编译异常：编译期间出现的异常，不处理，编译不通过

非 RuntimeException 及其子类都是表示运行异常

运行异常：运行期间出现的异常，不处理，编译可以通过

RuntimeException 及其子类都是表示运行异常

常见的异常：

运行异常:

ArrayIndexOutOfBoundsException

StringIndexOutOfBoundsException

NullPointerException

ArithmeticException

ClassCastExcption

编译异常:

ParseException

IOException

FileNotFoundException

异常分析：

throw 用在方法内，用来抛出一个异常对象，将这个异常对象传递给调用者，并结束当前方法的执行。

格式：throw 异常类型(参数);

注意：throw 就是用来抛出异常对象的

Objects 非空判断：

public static <T> requireNonNull(T obj): 查看指定引用对象不是 null。

2.异常处理

1.声明异常处理：throws 关键字，将问题标识出来，报告给调用者

如果方法内通过 throw 抛出了编译时异常，而没有捕获处理，那么必须通过 throws 进行声明，让调用者去处理，throws 运用于方法声明之上，用于表示当前方法不处理异常，而是提醒该方法的调用者来处理异常(抛出异常)

throws 用于进行异常类的声明，若该方法可能有多种异常，那么在 throws 后面可以写多个异常类，用逗号隔开，还可以抛出一个该异常的父类

格式：

```
修饰符 返回值类型 方法名(形参类型 形参名...) throws 异常类型 1, 异常类型 2, ... {
    …
}
```

2.捕获异常处理

格式：

```
try {
    异常代码;
} catch(异常类型 e) {
    处理异常代码;// 记录日志，打印异常，继续抛出异常
} finally {
    一般正常情况下都会执行的，一般用来释放资源
    除非 catch 里面的代码是: System.exit(0);
}
```

Throwable 类的方法：

public String getMessage():获取异常的描述信息,原因(提示给用户的时候,就提示错误原因。

public String toString():获取异常的类型和异常描述信息(不用)。

public void printStackTrace():打印异常的跟踪栈信息并输出到控制台。

3.声明异常和捕获异常的区别：

声明异常，如果出现了异常，就会终止程序，抛出异常

捕获异常，如果出现了异常，就会处理异常，并且继续处理下去。

扩展：

捕获异常的处理形式：三种

多次捕获，多次处理

一次捕获，多次处理，如果多个异常出现了子父类关系，父类异常需要放在子类的后面

一次捕获，一次处理

16-3_线程

1.线程和进程概念
进程:可执行文件(.exe)
线程:进程中的可执行单元路径
2.并发和并行概念
并发:同一时间段,交替执行
并行:同一时间,同时执行
3.多线程
多线程并发:多个线程同时请求 cpu 执行,cpu 一次只能执行一个线程,所以于是让这个多个线程交替执行
多线程并行:多个线程同时执行
多线程的特点: 随机性　　Java 程序中至少有 2 个线程:main 线程　垃圾回收机制线程
4.多线程创建
多线程的创建方式一:
1. 定义 Thread 类的子类，并重写该类的 run()方法，
该 run()方法的方法体就代表了线程需要完成的任务,因此把 run()方法称为线程执行体。
2. 创建 Thread 子类的实例，即创建了线程对象
3. 调用线程对象的 start()方法来启动该线程

一个程序必须让所有线程执行完毕,才会结束

17-线程中篇

17-1_线程的两种创建方式(重要!!!)

1.继承 Thread 类
创建一个类继承 Tread 类
重写 run()方法, 把要执行的代码放进 run 方法里面
创建继承了 Thread 类的类对象
然后通过对象调用 start()方法(默认会调用 run()方法)
2.实现 Runnable 接口
a.通过实现 Runnable 接口
创建一个类实现 Runnable 接口
重写 run()方法
创建实现了 Runnable 接口的类对象
创建一个 Thread 类对象，把实现了 Runnable 接口的类对象放进构造方法里面
然后通过 Thread 类对象调用 start()方法
b.通过匿名内部类的方式
创建一个 Thread 类对象
在 Thread 类的构造方法里
new Runnable(){
@override
public void run(){
...
}
}
通过 Thread 类对象调用 start()方法
3.注意：
1.线程调度是随机的

2.线程是不能重复开启的

4.Thread 和 Runnable 的区别
实现 Runnable 接口比继承 Thread 类所具有的优势：
1. 适合多个相同的程序代码的线程去共享同一个资源。
2. 可以避免 java 中的单继承的局限性。
3. 增加程序的健壮性，实现解耦操作，代码可以被多个线程共享，代码和线程独立。
4. 线程池只能放入实现 Runable 或 Callable 类线程，不能直接放入继承 Thread 的类。

总而言之: 实现 Runnable 接口这种方式创建线程可扩展性比较强,而继承 Thread 类的方式创建线程可扩展性比较弱

实现 Runnable 接口这种方式创建线程代码比较繁琐,而继承 Thread 类的方式创建线程代码比较简单(直接继承 Thread 类,可以直接实现 Thread 类中的方法)

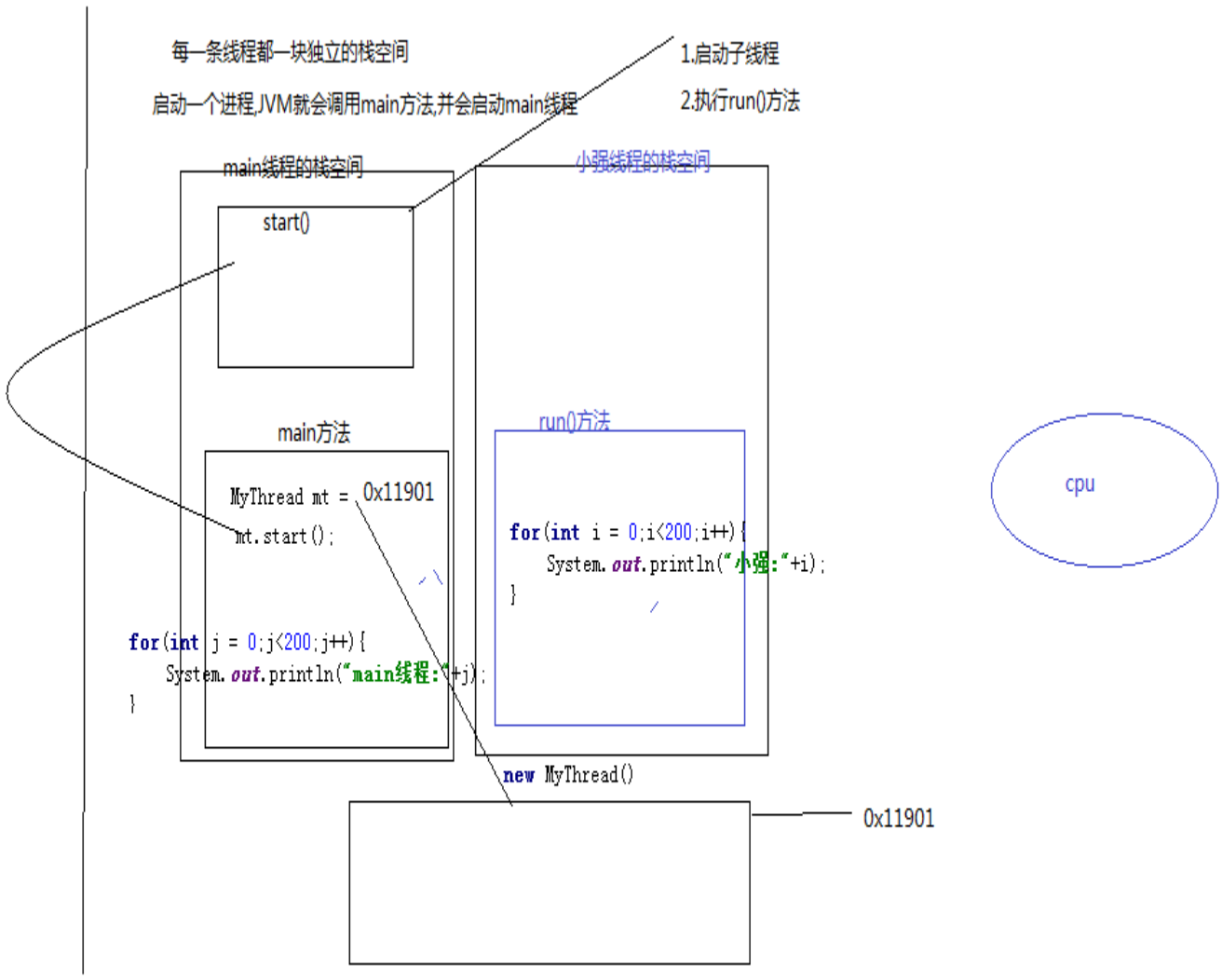
线程池只能放入实现 Runable 或 Callable 类线程，不能直接放入继承 Thread 的类。

多线程原理图：

```
public void run() {
    for(int i = 0;i<200;i++){
        System.out.println("小强:"+i);
    }
}
```

```
public static void main(String[] args) {
    //创建线程
    MyThread mt = new MyThread();
    //开启线程
    mt.start();

    for(int j = 0;j<200;j++){
        System.out.println("main线程:"+j);
    }
}
```



17-2_线程常用方法

构造方法：
- public Thread():分配一个新的线程对象。
- public Thread(String name):分配一个指定名字的新的线程对象。
- public Thread(Runnable target):分配一个带有指定目标新的线程对象。
- public Thread(Runnable target,String name):分配一个带有指定目标新的线程对象 并指定名字

常用方法：
- public void setName(String name) 将此线程的名称更改为等于参数 name 。
- public String getName():获取当前线程名称。
- public void start():导致此线程开始执行; Java 虚拟机调用此线程的 run 方法。
- public void run():此线程要执行的任务在此处定义代码。

- public static void sleep(long millis):使当前正在执行的线程以指定的毫秒数暂停（暂时停止执行）。
- public static Thread currentThread():返回对当前正在执行的线程对象的引用。　获取当前线程对象

17-3_同步机制(加锁)

多个线程访问同一资源的时候，且多个线程中对资源有写的操作，就容易出现线程安全问题
大白话: 多条线程执行一段相同的代码的时候,就有可能出现数据安全问题
解决方法: 使用同步机制(加锁)

1.同步代码块 : 如果有一段代码需要加锁,就使用同步代码块

格式:

```
synchronized(锁对象){
    //需要加锁的代码
}
```

锁对象:

1.可以是任意类型的对象
2.多个线程的锁对象一定要一致

2.同步方法: 如果整个方法的代码都需要加锁,就使用同步方法

格式:

```
修饰符 synchronized 返回值类型 方法名(参数){

}
```

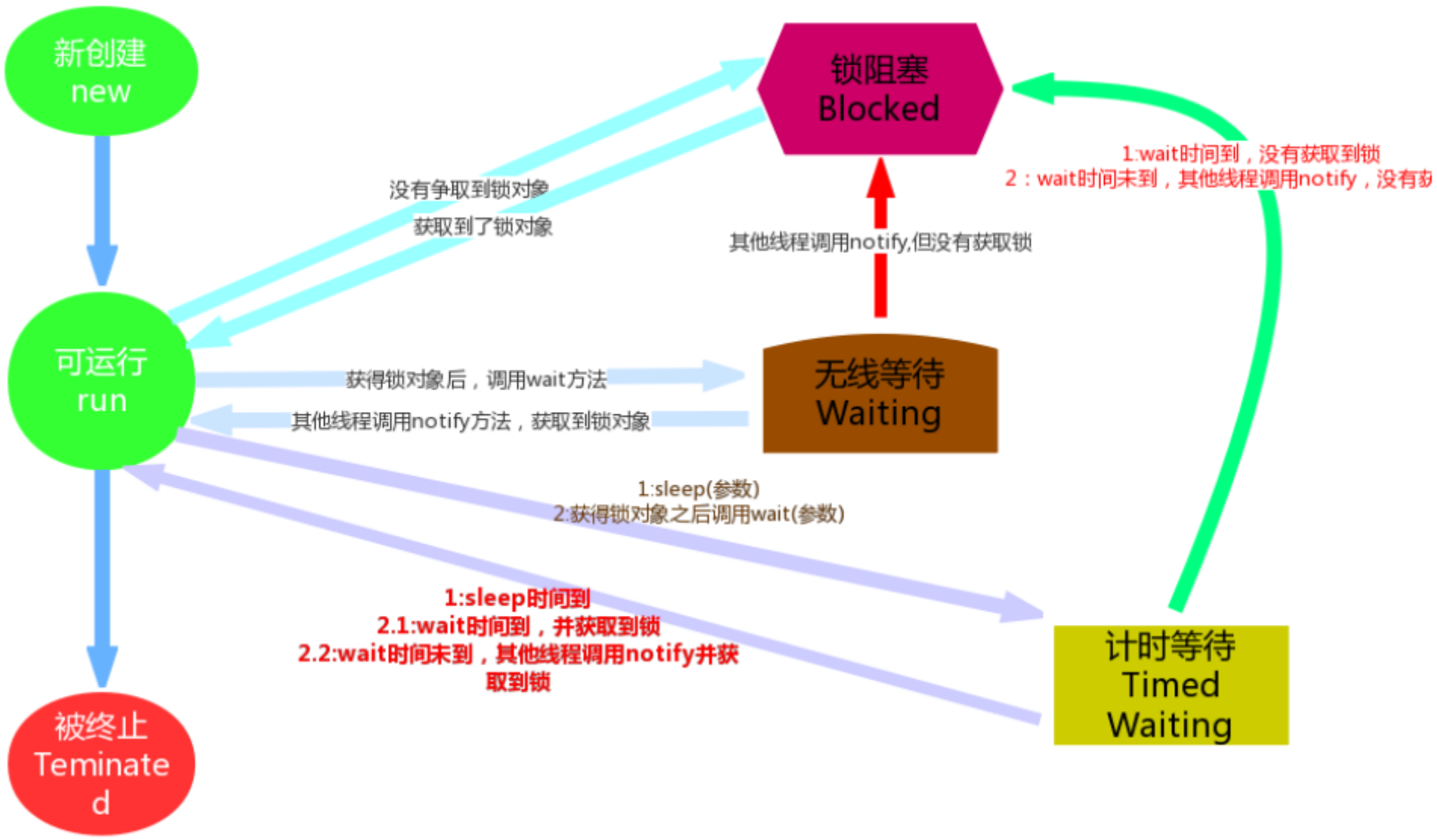
非静态同步方法的锁对象: this
静态同步方法的锁对象:类.class

如果 A 线程和 B 线程要能锁住,A 线程和 B 线程的锁对象要一致
假设 A 线程用的是非静态同步方法 ----> this
假设 B 线程用的是同步代码块-----> 必须也是 this 代表的那个对象

3.Lock 锁 ---> 创建其子类对象 ReentrantLock

- public void lock():加同步锁。
- public void unlock():释放同步锁。

17-4_线程状态(6 种状态)



waiting 状态：一个线程得到了锁，调用 wait 方法，进入无限等待状态，当另一个线程得到锁对象，调用 notify 方法时，前面等待的线程被唤醒 blocked 状态：线程 A 拿到了锁，那么线程 B 就进入 blocked 状态，直到线程 A 执行完毕，线程 B 一直处于 blocked 状态 time-waiting 状态：当线程进入休眠状态时，等待休眠时间结束前的这段时间(等待着另一个线程执行完之前的状态

18-线程下篇和 Lambda

18-1_复习

多线程: 1.创建线程方式 1.继承 Thread 类 1.创建一个子类继承 Thread 类 2.重写 run()方法,把需要子线程执行的任务放入 run()方法中 3.创建子类对象 4.调用 start()启动线程 2.实现 Runnable 接口 1.创建一个实现类去实现 Runnable 接口 2.重写 run()方法,把需要子线程执行的任务放入 run()方法中 3.创建实现类对象,把这个对象以参数的形式传入 Thread 类的构造方法中来创建 Thread 对象 4.调用 start()方法,启动线程 3.匿名内部类方式 1.创建一个 Runnable 接口的匿名内部类,该匿名内部类以参数的形式传入 Thread 类的构造方法中来创建 Thread 对象 2.调用 start()方法启动线程

2.同步机制
1.同步代码块
格式:
synchronized(锁对象){
}
注意:
1.锁对象可以是任意类的对象
2.多个线程必须锁对象一致
2.同步方法
非静态同步方法的锁对象:this
静态同步方法的锁对象:类名.class (该方法所在的类)
3.Lock 锁(互斥锁)
lock() 加锁
unlock() 释放锁
3.线程状态
新建
可运行
锁阻塞
计时等待
sleep(long mills)
wait(long mills);
无限等待 **wait()**
死亡
wait() notify() notifyAll() 调用的 2 个前提
必须在同步中
必须用同步锁对象调用

18-2_等待唤醒机制

研究静态同步方法和非静态同步方法的锁对象
非静态同步方法的锁对象是: this
静态同步方法的锁对象是:类名.class

思路: 线程一执行完了,线程一进入无限等待,通知线程 2 执行
线程二执行完了,线程二进入无限等待,通知线程 1 执行
等待唤醒机制: 有效的利用资源(让这多条线程有规律的执行)
wait() notify() notifyAll()方法
如果想要一个线程进入等待,就用 wait()方法
如果想要唤醒另一个线程,就用 notify()方法
如果想要唤醒所有等待线程,就用 notifyAll()方法
注意:
wait() notify() notifyAll()方法 想要调用有 2 个前提:
1.在同步中调用
2.必须使用同步锁对象调用
两个线程：旗帜变量和 if 判断
三个线程：旗帜变量和 while 循环

经典案例：生产者和消费者

等待唤醒机制实现有效通信

需求:生产一个,就吃一个

没有包子,吃货等待,唤醒包子铺生产包子,生产完了,包子就有了

有包子,包子铺就要等待,唤醒吃货吃包子,吃完了包子,包子就没有了

没有包子,吃货等待,唤醒包子铺生产包子,生产完了,包子就有了

有包子,包子铺就要等待,唤醒吃货吃包子,吃完了包子,包子就没有了

.....

0.资源 ---> 包子

包子类:

String pier;

String xianer;

boolean flag; // 包子状态

1.包子铺线程

如果有包子,包子铺线程就要等待

如果没有包子,就生产包子

2.吃货线程

如果没有包子,吃货就要等待

如果有包子,吃货就吃包子

18-3_线程池

Executors 类中有个创建线程池的方法如下：

- public static ExecutorService newFixedThreadPool(int nThreads)：返回线程池对象。(创建的是有界线程池,也就是池中的线程个数可以指定最大数量)

获取到了一个线程池 ExecutorService 对象，那么怎么使用呢，在这里定义了一个使用线程池对象的方法如下：

- public Future<?> submit(Runnable task):获取线程池中的某一个线程对象，并执行

使用线程池中线程对象的步骤：

1. 创建线程池对象。
2. 创建 Runnable 接口子类对象。(task)
3. 提交 Runnable 接口子类对象。(take task)
4. 关闭线程池(一般不做)。

四.Lambda 表达式

面向对象编程思想:强调是对象以什么形式来做

函数编程思想:强调做什么，而不是以什么形式做。

举例:

买电脑:

面向对象: 找一个电脑高手,电脑高手就应该具备买电脑的功能

函数编程: 获取电脑,怎么获取的电脑,不管

Lambda 的格式

(参数类型 参数名)->{ 代码语句 }

省略规则

在 Lambda 标准格式的基础上，使用省略写法的规则为：

1. 小括号内 参数的类型 可以省略；
2. 如果小括号内有且仅有一个参，则小括号可以省略；
3. 如果大括号内有且仅有一个语句，则无论是否有返回值，都可以省略大括号、return 关键字及语句分号。

Lambda 的语法非常简洁，完全没有面向对象复杂的束缚。但是使用时有几个问题需要特别注意：
1. 使用 Lambda 必须具有接口，且要求接口中有且仅有一个抽象方法。
　　无论是 JDK 内置的 Runnable、Comparator 接口还是自定义的接口，只有当接口中的抽象方法存在且唯一时，才可以使用 Lambda。
2. 使用 Lambda 必须具有上下文推断。
　　也就是方法的参数或局部变量类型必须为 Lambda 对应的接口类型，才能使用 Lambda 作为该接口的实例。
备注：有且仅有一个抽象方法的接口，称为“函数式接口”。

19-File 类与递归

19-1_复习

1.线程通信:多个线程操作同一资源,但是这多个线程的执行任务是不一样的
2.等待唤醒机制就可以有效利用资源
wait() notify() notifyAll()
使用有 2 个前提:
　　1.必须在同步中
　　2.必须用同步锁对象调用
注意:
　　wait() 让线程进入无限等待状态 ,当其他线程调用了 notify()或者 notifyAll()方法,就会唤醒等待线程,
3.等待唤醒机制使得线程有效通信的案例(2 条线程的有效通信,3 条线程的有效通信,吃包子的案例)
4.线程池---> 游泳池
　　1.如何创建一个线程池对象
　　2.如何提交任务并执行
5.Lambda 表达式
　　1.Lambda 表达式的标准格式
　　(参数类型 1 参数名 1,参数类型 2 参数名 2,...)->{ 代码语句;};
　　2.Lambda 表达式的省略规则
　　1).小括号里面的参数类型可以省略
　　2).如果小括号里面只有一个参数,小括号可以省略
　　3).如果大括号里面只有一条语句,那么无论是否有返回值,都可以省略大括号,return 以及分号
　3.Lambda 表达式的使用前提
　　1.用 Lambda 表达式表示的接口里面必须只有一个抽象方法
　4.Lambda 表达式使用案例
　　1.使用 Lambda 表达式的标准格式使用系统中的 Runnable 接口和 Comparator 接口
　　2.使用 Lambda 表达式的省略格式使用系统中的 Runnable 接口和 Comparator 接口
　　3.使用 Lambdas 表达式的标准格式使用自定义的接口
　　4.使用 Lambdas 表达式的省略格式使用自定义的接口

19-2_File 操作

java.io.File 类是文件和目录路径名的抽象表示，主要用于文件和目录的创建、查找和删除等操作。
绝对路径和相对路径
- 绝对路径：从盘符开始的路径，这是一个完整的路径。
- 相对路径：相对于项目目录的路径，这是一个便捷的路径，开发中经常使用。
构造方法:

- public File(String pathname) ： 通过将给定的路径名字符串转换为抽象路径名来创建新的 File 实例。常用

- public File(String parent, String child) ： 从父路径名字符串和子路径名字符串创建新的 File 实例。

- public File(File parent, String child) ： 从父抽象路径名和子路径名字符串创建新的 File 实例。

获取功能的方法

- public String getAbsolutePath() ： 返回此 File 的绝对路径名字符串。

- public String getPath() ： 将此 File 转换为路径名字符串。

- public String getName() ： 返回由此 File 表示的文件或目录的名称。

- public long length() ： 返回由此 File 表示的文件的长度。获取文件的大小(单位是字节)，获取不了文件夹的大小,只能文件的大小

判断功能的方法

- public boolean exists() ： 此 File 表示的文件或目录是否实际存在。

- public boolean isDirectory() ： 此 File 表示的是否为目录。

- public boolean isFile() ： 此 File 表示的是否为文件。

创建删除功能的方法

- public boolean createNewFile() ： 当且仅当具有该名称的文件尚不存在时，创建一个新的空文件。

- public boolean delete() ： 删除由此 File 表示的文件或目录。 注意不走回收站

- public boolean mkdir() ： 创建由此 File 表示的目录。 只能创建单级文件夹

- public boolean mkdirs() ： 创建由此 File 表示的目录，包括任何必需但不存在的父目录。 可以创建多级文件夹

目录的遍历

- public String[] list() ： 返回一个 String 数组，表示该 File 目录中的所有子文件或目录。

- public File[] listFiles() ： 返回一个 File 数组，表示该 File 目录中的所有的子文件或目录。

注意:如果文件夹写了后缀,并不代表文件的类型

19-3_递归调用

现实生活中的递归:

放羊---> 挣钱 ---> 娶媳妇 ---> 生娃 ---> 放羊 --->挣钱 ---> 娶媳妇 ---> 生娃 ---> 放羊....

从前山里有座庙,庙里有一个老和尚和一个小和尚,有一天小和尚哭了,老和尚说我给你讲个故事,

从前山里有座庙,庙里有一个老和尚和一个小和尚,有一天小和尚哭了,老和尚说我给你讲个故事

从前山里有座庙,庙里有一个老和尚和一个小和尚,有一天小和尚哭了,老和尚说我给你讲个故事

从前山里有座庙,庙里有一个老和尚和一个小和尚,有一天小和尚哭了,老和尚说我给你讲个故事

代码中的递归:

递归：方法自己调用自己

注意:

1.递归的次数不能太多,否则会出现 StackOverflowError 栈内存溢出

2.合理使用递归,也就是说出口不能太晚

学习递归: 出口 规律

递归的使用

1.定义一个方法:计算 1-n 的累加和

分析:

1-n 的累加和 = 1+2+3+4+5...+(n-3)+(n-2)+(n-1)+n n+(1-(n-1)的累加和)

1-(n-1)的累加和 = 1+2+3+4+5+...+(n-2)+(n-1) (n-1)+(1-(n-2)的累加和)

.....

1-5 的累加和 = 1+2+3+4+5 5+(1-4 累加和)

1-4 的累加和 = 1+2+3+4 4+(1-3 的累加和)

1-3 的累加和 = 1+2+3 3+(1-2 的累加和)

1-2 的累加和 = 1+2 2+(1 的累加和)

1 的累加和 = 1

规律: 计算一个数的累加和其实就是 等于 这个数 + (这个数-1 的累加和)

假设一个数是 n,计算这个 n 的累加和 === n+(n-1 的累加和)

出口: 当计算到 1 的累加和的时候结束

案例 2:

求一个 5 的阶乘

5! = 5*4*3*2*1 = 5*4!

4! = 4*3*2*1 = 4*3!

3! = 3*2*1 = 3*2!

2! = 2*1 = 2*1!

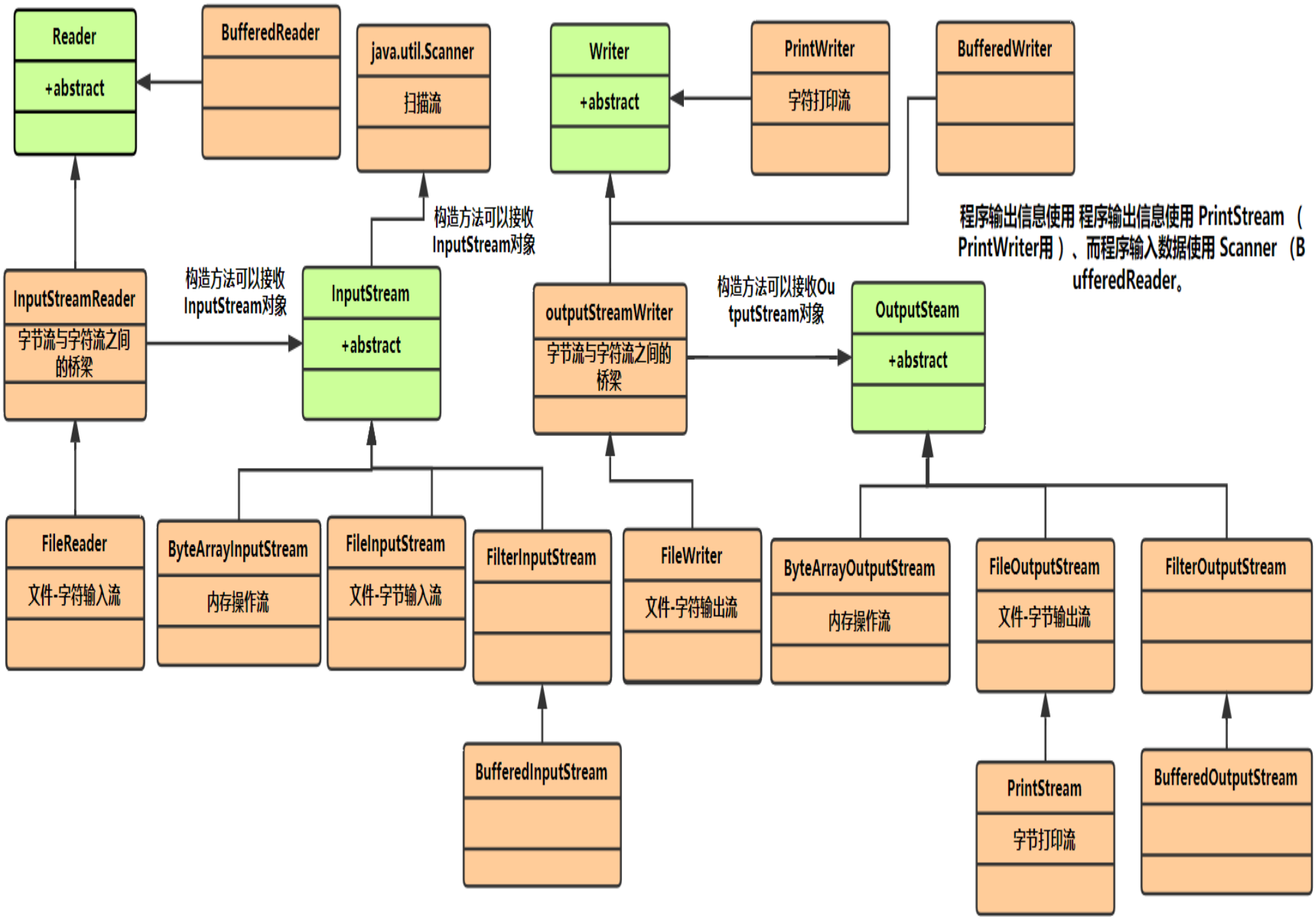
1! = 1

求一个数的阶乘 = 这个数 * (这个数-1)的阶乘

出口: 1

规律: n! = n * (n-1)!

20-IO 流上篇



20-1_IO 流

IO 流:

概述: 用来传输数据的

I:Input - 输入(读取) 把数据从 其他设备 读取到 内存中

O:Output - 输出(写出) 把数据从 内存中 写出到 其他设备

分类:

按类型分类:

1.字节流:以单个字节为基本单位来操作数据

字节输入流:基类 **InputStream** 是表示输入字节流的所有类的超类 抽象类 常用子类:

FileInputStream

字节输出流:基类 **OutputStream** 是表示字节输出流的所有类的超类 抽象类 常用子类
FileOutputStream
2.字符流:以单个字符为基本单位来操作数据
字符输入流:基类 **Reader** 用于读取字符流的抽象类 常用子类:FileReader
字符输出流:基类 **Writer** 用于写入字符流的抽象类 常用子类:FileWriter
按流向分类:
1.输入流: 把数据从 其他设备 读取到 内存中的流
字节输入流:基类 InputStream 是表示输入字节流的所有类的超类 抽象类 常用子类
FileInputStream
字符输入流:基类 Reader 用于读取字符流的抽象类 常用子类:FileReader
2.输出流: 把数据从 内存中 写出到 其他设备的流
字节输出流:基类 OUtputStream 是表示字节输出流的所有类的超类 抽象类 常用子类
FileOutputStream
字符输出流:基类 Writer 用于写入字符流的抽象类 常用子类:FileWriter
注意:
1.**计算机最小的基本存储单位是字节**
2.Windows 系统中的中文默认编码是 gbk 编码,而一个中文在 gbk 编码下占 2 个字节
idea 工具中的中文默认编码是 utf-8 编码,而一个中文在 utf-8 编码下占 3 个字节
今天,遇到的问题:
1.如果是从你电脑上拷贝一个带有中文的 gbk 编码的文件到 idea 中,就会出现乱码
2.如果你之前学过了 eclipse,今天在 idea 中的代码拷贝到 eclipse 里面去运行第一种情况,不会乱码,因为 eclipse 默认编码是 gbk
gbk: 你好 ---> 4 个字节
utf-8: 你好--->4 个字节 --->解码(显示) 以 utf-8 编码显示 --> 乱码

20-2_字节输出流

字节输出流: OutputStream
- public void close() ： 关闭此输出流并释放与此流相关联的任何系统资源。
- public void flush() ： 刷新此输出流并强制任何缓冲的输出字节被写出。
- public void write(byte[] b)：将 b.length 字节从指定的字节数组写入此输出流。
- public void write(byte[] b, int off, int len) ： 从指定的字节数组写入 len 字节， 从偏移量 off 开始输出到此输出流。
- public abstract void write(int b) ： 将指定的字节输出流。

FileOutputStream:
1.构造方法
- public FileOutputStream(File file)： 创建文件输出流以写入由指定的 File 对象表示的文件。
创建字节输出流对象,以 File 对象的形式关联目的地文件
- public FileOutputStream(String name)： 创建文件输出流以指定的名称写入文件。 推荐
创建字节输出流对象,以 String 类型的形式关联目的地文件
- **public FileOutputStream(File file, boolean append)**：创建文件输出流以写入由指定的 File 对象表示的文件。boolean append 参数如果为 true,就代表拼接,如果为 false 就覆盖
- **public FileOutputStream(String name, boolean append)**：推荐，创建文件输出流以指定的名称写入文件

注意:
1.如果指定的路径没有这个文件，会创建该文件
2.如果该文件中有数据,再往这个文件中写数据,就会覆盖

20-3_字节输入流

字节输入流: InputStream
- public void close() ： 关闭此输入流并释放与此流相关联的任何系统资源。
关闭流,释放资源

- public abstract int read()：从输入流读取数据的下一个字节。一次读取一个字节,把读取到的字节数据返回回来,用一个 int 类型的变量接收
- public int read(byte[] b): 从输入流中读取一些字节数，并将它们存储到字节数组 b 中 ，一次读取一个字节数组长度的字节数据,把读取出来的字节数据放入到字节数组中,返回读取到的有效字节数

字节输出流：FileInputStream

构造方法:

- FileInputStream(File file)：通过打开与实际文件的连接来创建一个 FileInputStream ，该文件由文件系统中的 File 对象 file 命名，创建一个字节输入流对象,封装数据源文件
- FileInputStream(String name): 通过打开与实际文件的连接来创建一个 FileInputStream ，该文件由文件系统中的路径名 name 命名。创建一个字节输入流对象,封装数据源文件

注意:

如果指定的数据源文件不存在,就会报文件找不到异常(FileNotFoundException)

20-4_io 流经典案例-copy 文件

IO 流的核心代码

1.创建一个输入流对象,封装数据源文件
2.创建一个输出流对象,封装目的地文件
3.定义一个变量来存储读取到的字节数据
4.循环读取数据,只要满足条件就一直读取,不满足条件就不读
5.使用输出流对象把读取到的数据写入到目的地文件
6.关闭流,释放资源

使用字节输入流和字节输出流一次读写一个字节来拷贝文件

拷贝文件的思路:

1.创建一个输入流对象,封装数据源文件
2.创建一个输出流对象,封装目的地文件
3.定义一个变量来存储读取到的字节数据
4.循环读取数据,只要满足条件就一直读取,不满足条件就不读
5.使用输出流对象把读取到的数据写入到目的地文件
6.关闭流,释放资源

20-5_字符输入流-Reader

当使用字节流读取文本文件时，可能会有一个小问题。

就是遇到中文字符时，可能不会显示完整的字符，那是因为一个中文字符可能占用多个字节存储

Reader 流定义的共性方法:

- public void close() ：关闭此流并释放与此流相关联的任何系统资源。
- public int read()： 从输入流读取一个字符。一次读取一个字符,把读取到的字符用 int 类型的变量来存储
- public int read(char[] cbuf)： 从输入流中读取一些字符，并将它们存储到字符数组 cbuf 中 。一次读取一个字符数组长度的字符数据,把读取到的字符数据存储到字符数组中,返回的是读取带的有效字符数

FileReader 类: 字符输入流

构造方法:

- FileReader(File file)： 创建一个新的 FileReader ，给定要读取的 File 对象。创建一个字符输入流对象,封装数据源文件
- FileReader(String fileName)： 创建一个新的 FileReader ，给定要读取的文件的名称。创建一个字符输入流对象,封装数据源文件

20-6_字符输出流-Writer

字符输出流: Writer

- public abstract void close() ： 关闭此输出流并释放与此流相关联的任何系统资源。
- public abstract void flush() ： 刷新此输出流并强制任何缓冲的输出字符被写出。
  - **flush ： 刷新缓冲区，流对象可以继续使用。**
  - close ： 关闭流，释放系统资源。关闭前会刷新缓冲区。(因为 FileWriter 写数据一开始是写入到缓冲区,如果没有刷新就不会保存到目的地文件中)
- public void write(int c) ： 写出一个字符。一次写入一个字符
- **public void write(String str)** ： 写出一个字符串。一次写入一个字符串
- public void write(char[] cbuf)：将 b.length 字符从指定的字符数组写出此输出流。一次写入一个字符数组
- public abstract void write(char[] b, int off, int len) ： 从指定的字符数组写出 len 字符，从偏移量 off 开始输出到此输出流。一次写入指定长度的字符数组
- public void write(String str, int off,int len) 写一个字符串的一部分

FileWriter 类:

构造方法:

- FileWriter(File file)： 创建一个新的 FileWriter，给定要读取的 File 对象。
- FileWriter(String fileName)： 创建一个新的 FileWriter，给定要读取的文件的名称。
- FileWriter(String fileName, boolean append)：构造一个 FileWriter 对象，给出一个带有布尔值的文件名，表示是否附加写入的数据。
- FileWriter(File file, boolean append) 给一个 File 对象构造一个 FileWriter 对象。

注意:

- **如果指定的目的地文件不存在,会新建目的地文件**
- **如果使用前 2 个构造方法,也就是没有 boolean 类型参数的构造方法,在写入数据的时候会覆盖之前的数据**

20-7_异常处理-IO 流

JDK7 之前处理 IO 异常：

```
public class HandleException1 {
    public static void main(String[] args) {
        // 声明变量
        FileWriter fw = null;
        try {
            //创建流对象
            fw = new FileWriter("fw.txt");
            // 写出数据
            fw.write("jack"); //jack
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (fw != null) {
                    fw.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

JDK7 之后处理 IO 异常

还可以使用 JDK7 优化后的 try-with-resource 语句，该语句确保了每个资源在语句结束时关闭。所谓的资源（resource）是指在程序完成后，必须关闭的对象。

格式：

```
try (创建流对象语句，如果多个,使用';'隔开) {
        // 读写数据
  } catch (IOException e) {
        e.printStackTrace();
   }
```

JDK9 优化的处理 IO 异常

JDK9 中 try-with-resource 的改进，对于引入对象的方式，支持的更加简洁。被引入的对象，同样可以自动关闭，**无需手动 close**，我们来了解一下格式。

格式：

```
// 被 final 修饰的对象
final Resource resource1 = new Resource("resource1");
// 普通对象
Resource resource2 = new Resource("resource2");
// 引入方式：直接引入
try (resource1; resource2) {
      // 使用对象
}
```

20-8_Properties 属性集合

Properties 本质是一个**双列集合**,表示一个永久性的属性集

Properties 集合的键和值默认是 String 类型

特性: 加载配置文件

与流相关的方法

- **public void load(InputStream inStream)：** 从字节输入流中读取键值对。
- **public void store(OutputStream out, String comments)：**保存

  将此属性列表（键和元素对）写入此 Properties 表中，

  以适合于使用 load(InputStream)方法加载到 Properties 表中的格式输出流。

注意:文本中的数据，必须是键值对形式，可以使用空格、等号、冒号等符号分隔

常用方法:

- public Object setProperty(String key, String value) ： 保存一对属性。
- public String getProperty(String key) ：使用此属性列表中指定的键搜索属性值。
- public Set<String> stringPropertyNames() ：所有键的名称的集合。

构造方法

- public Properties() :创建一个空的属性列表

21-IO 流下篇

21-1_缓冲流

1.概述

缓冲流又叫高效流

字节缓冲流：BufferedInputStream、BufferedOutputStream

字符缓冲流：BufferedReader、BufferedWriter

缓冲流的基本原理：是在创建流对象时，会创建一个内置的默认大小的缓冲区数组，通过缓冲区读写，减少系统 IO 次数，从而提高读写的效率。

理解：内存的执行效率比硬盘的效率高，所以我们只需要降低和硬盘的交互次数，就可以提高效率。

2.字节缓冲流：BufferedInputStream、BufferedOutputStream

构造方法

- **public BufferedInputStream(InputStream in)** ：创建一个 新的缓冲输入流。
- **public BufferedOutputStream(OutputStream out)**： 创建一个新的缓冲输出流。

案例：拷贝文件

· 创建输入流对象，封装数据源文件
· 创建输出流对象，封装目的地文件
· 创建变量，用来存储读取到的数据
· 循环读取数据，只要满足条件就一直读取，不满足就结束读取
· 写出数据
· 关闭流，释放资源

总结：

使用普通字节流一次读写一个字节拷贝 curry.jpg 文件 使用了 47130 毫秒
使用普通字节流一次读写一个字节数组拷贝 curry.jpg 使用了 109 毫秒
使用字节缓冲流一次读写一个字节拷贝 curry.jpg 使用了 328 毫秒
使用字节缓冲流一次读写一个字节数组拷贝 curry.jpg 使用了 15 毫秒

3.字符缓冲流：BufferedReader、BufferedWriter

构造方法

- **public BufferedReader(Reader in)** ：创建一个 新的缓冲输入流。
- **public BufferedWriter(Writer out)**： 创建一个新的缓冲输出流。

案例：拷贝文件

使用缓冲流一次读写一个字符拷贝文件
使用缓冲流一次读写一个字符数组拷贝文件

字符缓冲流的基本方法与普通字符流调用方式一致，不再阐述，我们来看它们具备的特有方法。

- **BufferedReader：public String readLine(): 读一行文字。**
- **BufferedWriter：public void newLine(): 写一行行分隔符,由系统属性定义符号。**

案例：

文本排序：对 d.txt 文件内容排序
1.读取 d.txt 中的每一行
2.每获取到一行数据，就以"."分割该字符串行数据
3.把分割出来的 2 个字符串，第一个数字字符串作为键，第二个内容字符串作为值，以键值对的形式存入 HashMap 集合中
4.按照 1-9 的顺序直接取出内容字符串
5.使用输出流写入到 d.txt 中

4.缓冲流原理：

假设文件大小为 8192 字节
普通字节、字符流读取硬盘文件时，要和硬盘交互 8192 次，而缓冲流只需要只需要交互一次(一次读取 8192 字节)，写入同理。

21-2_转换流

1.字符编码：

编码：按照某种规则把字符转换为二进制形式 - 看得懂的 变成 看不懂的
解码：按照某种规则把二进制形式的数据 解析为 字符 - 看不懂的 变成 看得懂的

字符集：是一个系统支持的所有字符的集合，包括各国家文字、标点符号、图形符号、数字等。计算机要准确的存储和识别各种字符集符号，需要进行字符编码，一套字符集必然至少有一套字符编码

ASCII 字符集、ISO-8859-1 字符集、GBXxx 字符集、Unicode 字符集
Windows 中的中文默认是 gbk 编码，一个中文在 gbk 编码下占 2 个字节
idea 中的中文默认是 utf-8 编码，一个中文在 utf-8 编码下占 3 个字节

2.转换流：

InputStreamReader：

转换流 java.io.InputStreamReader，是 Reader 的子类，是从字节流到字符流的桥梁。它读取字节，并使用指定的字符集将其解码为字符。它的字符集可以由名称指定，也可以接受平台的默认字符集。

构造方法：

- **InputStreamReader(InputStream in):** 创建一个使用默认字符集的字符流。

- InputStreamReader(InputStream in, String charsetName): 创建一个指定字符集的字符流。

OutputStreamWriter:

转换流 java.io.OutputStreamWriter 是 Writer 的子类，是从字符流到字节流的桥梁。使用指定的字符集讲字符编码为字节。它的字符集可以由名称指定，也可以接受平台的默认字符集

构造方法：

- OutputStreamWriter(OutputStream in): 创建一个使用默认字符集的字符流。

- OutputStreamWriter(OutputStream in, String charsetName): 创建一个指定字符集的字符流。

案例：将 GBK 编码的文本文件，转换为 UTF-8 编码的文本文件。

· 指定 GBK 编码的转换流，读取文本文件。

· 使用 UTF-8 编码的转换流，写出文本文件。

21-3_序列化

序列化：将对象 存储到 文件中

反序列化：从文件中 读取 一个对象

1.ObjectOutputStream: 可以将对象写入到文件中 - 序列化

构造方法：

public ObjectOutputStream(OutputStream out): 创建一个指定 OutputStream 的 ObjectOutputStream。

序列化操作方法：

public final void writeObject(Object obj): 将指定的对象写出

一个对象要想序列化，必须满足两个条件:

- 该类必须实现 java.io.Serializable 接口，Serializable 是一个标记接口，不实现此接口的类将不会使任何状态序列化或反序列化，会抛出 NotSerializableException 。

- 该类的所有属性必须是可序列化的。如果有一个属性不需要可序列化的，则该属性必须注明是瞬态的，使用 transient 关键字修饰。

2.ObjectInputStream:

构造方法：

public ObjectInputStream(InputStream in)：创建一个指定 InputStream 的 ObjectInputStream。

反序列化操作方法：

public final Object readObject ()：读取一个对象。

反序列注意：

· 如果能找到一个对象的 class 文件，我们可以进行反序列化操作

· 如果找不到该类的 class 文件，则抛出一个 ClassNotFoundException 异常

· 能找到 class 文件，但是 class 文件在序列化对象之后发生了修改，那么反序列化操作也会失败，抛出一个 InvalidClassException 异常，该类的序列版本号与从流中读取的类描述符的版本号不匹配

22-4_打印流

PrintStream 流

平时我们在控制台打印输出，是调用 print 方法和 println 方法完成的，这两个方法都来自于 java.io.PrintStream 类，该类能够方便地打印各种数据类型的值，是一种便捷的输出方式。

构造方法：

public PrintStream(String fileName)： 使用指定的文件名创建一个新的打印流。

常用方法：

- **print()**
- **println()**
- **System.setOut(OutputStream out) - 改变系统打印的目的地**

22-网络编程

22-1_复习

缓冲流:
字节缓冲流:BufferedInputStream,BufferedOutputStream
字符缓冲流:BufferedReader,BufferedWriter
特有的方法:
readLine()---> BufferedReader 读取行
newLine()----> BufferedWriter 换行 writer("\r\n")
转换流:
InputStreamReader,OutputStreamWriter
构造方法有 2 个:
public InputStreamReader(InputStream in);// 默认编码 (idea utf-8)
public InputStreamReader(InputStream in,String charSet); 指定编码
public OutputStreamWriter(OutputStream out);// 默认编码
public OutputStreamWriter(OutputStream out,String charSet);// 指定编码
序列化流:
ObjectInputStream 把二进制数据 转换为 对象 ----> 反序列化
readObject() 读取对象
ObjectOutputStream 把对象 转换为 二进制数据 ----> 序列化
writerObject() 写入对象
打印流
PrintStream
public PrintStream(String path);
print() println()
System.out.print();默认打印到控制台
System.setOut(PrintStream ps);

22-2_网络编程入门

网络编程，就是在一定的协议下，实现两台计算机的通信的程序。
网络通信协议：为计算机网络中进行数据交换而建立的规则、标准或约定的集合.
协议分类:
TCP：传输控制协议 (Transmission Control Protocol): 面向连接(三次握手),传输速度慢,数据安全的
UDP：用户数据报协议(User Datagram Protocol):面向无连接,传输速度快,数据不安全
IP:
概述: 指互联网协议地址（Internet Protocol Address），俗称 IP。
IP 地址用来给一个网络中的计算机设备做唯一的编号
IPV4:是一个 32 位的二进制数，通常被分为 4 个字节，表示成 **a.b.c.d** 的形式 ,大概有 **42.9** 亿个
IPV6:每 16 个字节一组，分成 8 组十六进制数，表示成 ABCD:EF01:2345:6789:ABCD:EF01:2345:6789
端口号:
概述:端口号就可以唯一标识设备中的进程（应用程序）
用两个字节表示的整数，它的取值范围是 **0~65535,使用 1024 以上的端口号**
例子:
李晨 要对 冰冰 表白
1.李晨 找到 冰冰 --------> **IP 地址**
2.李晨 对冰冰 的耳朵说 ---> **端口号**
3.说 撒拉嘿呦 ---> **协议**
中国话 我爱你

22-3_TCP 通信编程

两端通信时步骤：
1. 服务端程序，需要事先启动，等待客户端的连接,并且一般都不会关闭。
2. 客户端主动连接服务器端，连接成功才能通信。服务端不可以主动连接客户端。

在 Java 中，提供了两个类用于实现 TCP 通信程序：
1. 客户端：**java.net.Socket** 类表示。创建 Socket 对象，向服务端发出连接请求，服务端响应请求，两者建立连接开始通信。
2. 服务端：**java.net.ServerSocket** 类表示。创建 ServerSocket 对象，相当于开启一个服务，并等待客户端的连接。

Socket 概述:　客户端
　　该类实现客户端套接字，套接字指的是两台设备之间通讯的端点
Socket 类的方法
　　构造方法:
　　　　public Socket(String host, int port) :创建套接字对象并将其连接到指定主机上的指定端口号
　　常用方法:
　　　　public InputStream getInputStream() ：　返回此套接字的输入流
　　　　public OutputStream getOutputStream() ：　返回此套接字的输出流
　　　　public void close() ：关闭此套接字。
　　　　public void shutdownOutput() ：　禁用此套接字的输出流。
SeverSocket：服务器端
　　public ServerSocket(int port) ：使用该构造方法在创建 ServerSocket 对象时，就可以将其绑定到一个指定的端口号上，参数 port 就是端口号。
　　public Socket accept() ：侦听并接受连接，返回一个新的 Socket 对象，用于和客户端实现通信。该方法会一直阻塞直到建立连接，(如果没有客户端连接服务器,服务器就会阻塞\等待,如果有客户端连接,并且连接成功,就返回连接的 Socket)。

22-4_案例-模拟客户端发生数据到服务器端

思路：
　　客户端：
　　　　1.创建客户端对象 Socket
　　　　2.通过客户端对象获取输出流 OutputStream，向服务端发送数据
　　　　3.写入发送的数据：write()
　　　　4.关闭流
　　服务器端：
　　　　1.创建服务器对象 ServerSocket
　　　　2.通过服务器对象，调用 accept()方法，返回一个 socket 对象，等待客户端请求连接，如果没有请求就阻塞。
　　　　3.通过 socket 对象获取输入流 InputStream，接收客户端数据
　　　　4.开始读取客户端数据：read()
　　　　5.关闭流
　　　　6.关闭 socket
　　　　7.关闭服务器对象(一般不关闭)

【Client.java】

```
package com.jack.socket;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
 /**
 * 案例：模拟客户端发生数据到服务器端
 * 客户端
*/
```

```
public class Client {
	public static void main(String[] args) throws IOException {
		// 创建一个客户端
		Socket socket = new Socket("127.0.0.1", 8888);
		// 通过 socket 对象获取输出流，向服务器端输出数据
		OutputStream output = socket.getOutputStream();
		// 写入数据
		output.write("I love learn java and java is my best favorite programming language.".getBytes());
		// 关闭流
		output.close();
	}
}
```

【Server.java】

```
package com.jack.socket;
import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;
/**
* 服务器端
*/
public class Server {
	public static void main(String[] args) throws IOException {
		// 创建服务器对象
		ServerSocket server = new ServerSocket(8888);
		// 调用 accept()方法接收请求，如果没有请求就阻塞，如果有请求且请求成功就建立连接，返回连接的 socket 对象
		Socket socket = server.accept();
		// 服务器使用 socket 对象获取输入流，来接收客户端发送的数据
		InputStream input = socket.getInputStream();
		// 读取客户端发送的数据
		byte[] data = new byte[1024];
		int len = input.read(data);
		System.out.println(new String(data, 0, len));
		// 关闭流
		input.close();
	}
}
```

22-5_案例:-客户端和服务器端实现双向数据传输

思路：

客户端：

1.创建客户端对象 Socket

2.通过客户端对象获取输出流 OutputStream，向服务端发送数据

3.写入发送的数据：write()

4.通过客户端对象获取输入流 InputStream，接收服务器端发送的数据

5.开始读取接收到的数据：read()

4.关闭流

服务器端：

1.创建服务器对象 ServerSocket

2.通过服务器对象，调用 accept()方法，返回一个 socket 对象，等待客户端请求连接，如果没有请求就阻塞。

3.通过 socket 对象获取输入流 InputStream，接收客户端数据

4.开始读取客户端数据：read()
5.通过 socket 对象获取输出流 OutputStream，向客户端发送数据
6.开始写入发送的数据：write()
5.关闭流
6.关闭 socket
7.关闭服务器对象(一般不关闭)

【Client.java】

```
package com.jack.socket2;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
public class Client {
	public static void main(String[] args) throws IOException {
		// 创建一个客户端对象
		Socket client = new Socket("localhost", 9999);
		// 创建一个输入流对象
		OutputStream output = client.getOutputStream();
		// 开始写入数据，发送数据到服务器端
		output.write("I love learn java and java is my best favorite programming language.".getBytes());
		// 创建一个输出对象
		InputStream input = client.getInputStream();
		// 接收服务器端的数据
		byte[] data = new byte[1024];
		int len = input.read(data);
		System.out.println(new String(data, 0, len));
		// 关闭流
		input.close();
		output.close();
	}
}
```

【Server.java】

```
package com.jack.socket2;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
/**
* 服务器端
*/
public class Server {
	public static void main(String[] args) throws IOException {
		// 创建一个服务器对象
		ServerSocket server = new ServerSocket(9999);
		// 调用 accept()方法，等待客户客户端请求
		Socket socket = server.accept();
		// 请求成功，获取客户端发送的数据
		InputStream input = socket.getInputStream();
		// 读取数据
		byte[] data = new byte[1024];
		int len = input.read(data);
		System.out.println(new String(data, 0, len));
		// 读取完数据，向客户端发送数据
```

```
            OutputStream output = socket.getOutputStream();
            output.write("server accepted the client message and return the success message 'ok'.".getBytes());
            // 关闭流
            output.close();
            input.close();
            socket.close();
      }
}
```

22-6_案例-文件上传

思路：文字使用字符流，其他的建议使用字节流

客户端：

1.创建一个客户端对象 Socket

2.创建一个输入流对象 InputStream

3.通过 Socket 对象获取输出流对象 OutputSteam，保存上传的数据

4.从 InputStream 对象每读取一个字节就向 OutputStream 写入一个字节

5.关闭流

服务器端：

1.创建一个服务器端对象 ServerSocket

2.调用 accept()方法，等待客户端连接

3.获取输入流 InputStream 和创建一个输出流对象 OutputStream

4.每读取一个字节就向硬盘写入一个一个字节

5.关闭流

6.关闭 Socket

7.关闭 Server(一般不关闭)

文件上传-上传图片

【FileUploadClient.java】

```
package com.jack.fileupload;
import java.io.*;
import java.net.Socket;
/**
*  文件上传客户端
*/
public class FileUploadClient {
      public static void main(String[] args) throws IOException {
            Socket socket = new Socket("localhost", 9999);
            OutputStream output = socket.getOutputStream();
            InputStream input = new FileInputStream("day22\\src\\curry.jpg");
            byte[] data = new byte[8192];
            int len = 0;
            while ((len = input.read(data)) != -1) {
                  output.write(data, 0, len);
            }
            input.close();
            output.close();
            socket.close();
            }
}
```

【FileUploadServer.java】

```
package com.jack.fileupload;
import java.io.*;
import java.net.ServerSocket;
```

```
import java.net.Socket;
import java.util.Scanner;
/**
* 文件上传服务器端
*/
public class FileUploadServer {
	public static void main(String[] args) throws IOException {
		ServerSocket server = new ServerSocket(9999);
		Socket socket = server.accept();
		InputStream input = socket.getInputStream();
		OutputStream output = new FileOutputStream("day22\\src\\curry2.jpg");
		byte[] data = new byte[8192];
		int len = 0;
		while ((len = input.read(data)) != -1) {
			output.write(data, 0, len);
		}
		output.close();
		input.close();
		socket.close();
		}
}
```

文件上传-上传文本文件

【FileUploadClient.java】

```
package com.jack.fileupload2;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintStream;
import java.net.Socket;
import java.util.Scanner;
/**
* 文件上传客户端
*/
public class FileUploadClient {
	public static void main(String[] args) throws IOException {
		Socket client = new Socket("localhost", 8888);
		PrintStream out = new PrintStream(client.getOutputStream());
		Scanner in = new Scanner(new FileInputStream("day22\\src\\a.txt"));
		in.useDelimiter("\n");
		while (in.hasNext()) {
			out.write(in.next().getBytes());
		}
		in.close();
		out.close();
		client.close();
	}
}
```

【FileUploadServer.java】

```
package com.jack.fileupload2;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;
```

```
import java.util.Scanner;
/**
 * 文件上传服务器端
 */
public class FileUploadServer {
	public static void main(String[] args) throws IOException {
		ServerSocket server = new ServerSocket(8888);
		Socket socket = server.accept();
		Scanner in = new Scanner(socket.getInputStream());
		PrintStream out = new PrintStream(new FileOutputStream("day22\\src\\aCopy.txt"));
		in.useDelimiter("\n");
		while (in.hasNext()) {
			out.write(in.next().getBytes());
		}
		out.close();
		in.close();
		socket.close();
	}
}
```

22-7_案例-文件上传优化

之前的文件上传存在三个问题：

- 文件名称写死
- 服务器端保存一个文件就关闭了，客户端不能再次上传文件
- 如果上传大文件，可能此时会花费不少的时间，导致其他用户不能上传

解决方法：

- 使用 System.currentTimeMillis()获取当前时间戳，保证唯一性
- while(true){}这样服务器就不会关闭
- 服务器端接收客户端请求 Socket socket = server.accept()放进线程里面

【FileUploadClient】

```
package com.jack.fileupload3;

import java.io.*;
import java.net.Socket;
import java.util.Scanner;

/**
 * 文件上传客户端
 */
public class FileUploadClient {
	public static void main(String[] args) throws IOException {
		Socket client = new Socket("localhost", 8888);
		PrintStream out = new PrintStream(client.getOutputStream());
		Scanner in = new Scanner(new FileInputStream("day22\\src\\a.txt"));
		in.useDelimiter("\n");
		while (in.hasNext()) {
			out.write(in.next().getBytes());
		}
		client.shutdownOutput(); //先前写入的数据发送完之后，终止输出流
		System.out.println("==================================");
		Scanner in2 = new Scanner(client.getInputStream()); //  接收服务器端发送的消息
		PrintStream out2 = new PrintStream(new FileOutputStream("day22\\src\\b.txt"));
		while (in2.hasNext()) {
```

```
                out2.print(in2.nextLine());
            }
            in2.close();
            out2.close();
            in.close();
            out.close();
            client.close();
        }
}
```

【FileUploadServer】

```
package com.jack.fileupload3;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;

/**
 * 文件上传服务器端
 */
public class FileUploadServer {
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(8888);
        while (true) {
            Socket socket = server.accept();
            new Thread(()->{

                try {
                    Scanner in = new Scanner(socket.getInputStream());
                    PrintStream out = new PrintStream(new FileOutputStream("day22\\src\\"+System.currentTimeMillis()+".txt"));
                    in.useDelimiter("\n");
                    while (in.hasNext()) {
                        out.write(in.next().getBytes());
                    }
                    System.out.println("==================");
                    PrintStream out2 = new PrintStream(socket.getOutputStream());
                    out2.write("server accepted the message.".getBytes()); // 向客户端发送数据，实现回写
                    out2.close();
                    out.close();
                    in.close();
                    socket.close();
                } catch (Exception e) {

                }
            }).start();
        }

    }
}
```

22-8_案例-模拟服务器

需求:模拟网站服务器，使用浏览器访问自己编写的服务端程序，查看网页效果。

搭建服务器，指定服务器的端口是 8000, 在浏览器输入网址:http://localhost:8888/day22/web/index.html 可以访问

模拟搭建网站服务器

1.创建 ServerSocket 对象

2.调用 accept()方法,建立连接,获取 Socket 对象

3.服务器要接收请求信息 ---通过 socket 对象后去输入流对象

4.通过 socket 对象获取的输入流对象,来读取第一行数据

5.得到第一行数据之后,通过字符串的 split()方法去分割 str 字符串,得到有效网址\路径 day11/web/index.html

6.创建输入流对象,去有效路径下读取指定的网页信息

7.通过 socket 对象获取输出流对象,把读取到的网页信息回写给浏览器

8.关闭流,释放资源

```
package com.jack.browserserver;
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.HashMap;
import java.util.Scanner;

/**
 * 模拟服务器
 *       模拟网站服务器，使用浏览器访问自己编写的服务端程序，查看网页效果。
 */
public class BrowserServer2 {
    public static void main(String[] args) throws IOException {
        System.out.println("开启服务器...");
        // 创建服务器对象
        ServerSocket server = new ServerSocket(8888);
        // 使用循环保证服务器不关闭
        while (true) {
            // 接收浏览器的请求
            Socket socket = server.accept();
            // 使用线程开启多任务
            new Thread(()->{
                try {
                    // 获取输入流对象
                    Scanner input = new Scanner(socket.getInputStream());
                    // 读取请求的第一行
                    String line = input.nextLine();
                    String[] split = line.split(" ");
                    String path = split[1].substring(1); // 获取请求网页路径
                    System.out.println(path);

                    // 创建一个输入流对象
                    InputStream in = new FileInputStream(path);
                    // 创建一个输出流对象
                    OutputStream output = socket.getOutputStream();
                    output.write("HTTP/1.1 200 ok\r\n".getBytes());
                    output.write("Content-Type:text/html\r\n".getBytes());
                    output.write("\r\n".getBytes()); // 空行
                    byte[] data = new byte[1024];
```

```
                            int len = 0;
                            while ((len = in.read(data)) != -1) {
                                output.write(data, 0, len);
                            }
                            in.close();
                            output.close();
                            input.close();
                            socket.close();
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }).start();
            }
        }
}
```

23-函数式接口和方法引用

23-1_函数式接口

23-1-1_Lambda 表达式

1.是什么

概述:jdk8 专门为有且仅有一个抽象方法的接口 提供的一个实例

2.怎么用

格式:

(参数类型 参数名,...)->{代码语句}

3.什么用

当你需要一个有且仅有一个抽象方法的接口的实例的时候,那么可以传一个 lambda 表达式

23-1-2_函数式接口

概述：**有且仅有一个抽象方法的接口**

函数式接口，即适用于函数式编程场景的接口。而

Java 中的函数式编程体现就是 Lambda，所以**函数式接口就是可以适用于 Lambda 使用的接口**

格式:

```
修饰符 interface 接口名{
    // 该接口中只能有一个抽象方法
    public abstract 返回值类型 方法名(参数列表);
    // 其他的非抽象方法
}
```

自定义一个函数式接口:

```
@FunctionalInterface
    public interface MyFunctionalInterface {
        public abstract void method();// 无参数无返回值的抽象方法
    }
```

使用**@FunctionalInterface** 注解 标识该接口是一个函数式接口

如果接口中有多个抽象方法,那么用@FunctionalInterface 标识会编译报错

目前为止已经接触到的函数式接口：Comparator、Runnable、FileFilter

23-1-3_函数式接口的使用

1、先定义一个函数式接口

```
package com.jack.functioninterface_1;

/**
 * 定义一个函数式接口
 */
@FunctionalInterface
public interface MyFunctionInterface {
	public abstract void method();
}
```

2、实现函数式接口里面的方法

```
public class MyFunctionInterfaceTest {
	public static void main(String[] args) {
		// 第一种方式：使用匿名内部类 实现函数式接口里面的方法
		show(new MyFunctionInterface() {
			@Override
			public void method() {
				System.out.println("这是函数式接口的方法");
			}
		});

		// 第二种方式：使用 lambda 表达式(表示该函数式接口的一个实例) 实现函数式接口里面的方法
		show(()->System.out.println("这是函数式接口的方法"));
	}

	/**
	 * 函数式接口作为方法的参数
	 * @param mfi 函数式接口的对象
	 */
	public static void show(MyFunctionInterface mfi) { // 参数是一个函数式接口
		mfi.method(); // 使用函数式接口的对象 调用函数式接口的 method()方法
	}
}
```

23-2_函数式编程

Lambda 表达式和方法引用是函数式编程的基础

1.Lambda 表达式有一个延迟效果(具体见案例)

2.使用 Lambda 作为参数和返回值

23-2-1_延迟效果

1、定义以函数式接口

```
package com.jack.functioninterface_lambdaapplication_3;

/**
 * 定义一个函数式接口 GetString
 */
public interface GetString {
	public abstract String getString();
}
```

2、测试

```
package com.jack.functioninterface_lambdaapplication_3;

/**
 * Lambda 表达式有一个延迟效果
 */
public class GetStringTest {
    public static void main(String[] args) {
        String str1 = "我是";
        String str2 = "最帅的";
        String str3 = "杰克";
//          printString(1, str1 + str2 + str3);
        /*
            如果等级为 2,str1+str2+str3 该表达式还是会执行,把该表达式的结果传递给 str 形参,然后再去判断等级,是否输出
            翻译: 无论等级是否为 1,str1+str2+str3 该表达式都会执行
         */
//          printString(2, str1 + str2 + str3);
        /*
            需求: 希望等级为 1 的时候,就输出 str,也就是说等级为 1 的时候,
            才去计算 str1+str2+str3 表达式的值,这才是性能最优的
            使用 lambda 解决,因为 lambda 具有延迟效果
            只有当等级为 1 的时候,才会去计算 str1+str2+str3 表达式的值
         */
        printString(2, ()->{
            System.out.println("执行了吗");
            return str1 + str2 + str3;
        });
    }

    /**
     * 定义一个打印字符串的方法
     * @param level
     * @param str
     */
    public static void printString(int level, String str) {
        if (level == 1) {
            System.out.println("等级为 1，打印字符串：" + str);
        }
    }

    /**
     * 定义打印字符串的方法
     *      传入一个函数式接口参数，利用 Lambda 表达式实现延迟效果
     * @param level
     * @param gs
     */
    public static void printString(int level, GetString gs) {
        if (level == 1) {
            String str = gs.getString();
            System.out.println("等级为 1，打印字符串：" + str);
        }
    }
}
```

23-2-2_使用 Lambda 表达式作为函数的参数和返回值

如果方法的参数是一个函数式接口类型，那么就可以使用 Lambda 表达式进行替代。使用 Lambda 表达式作为方法参数，其实就是使用函数式接口作为方法参数。

例如 java.lang.Runnable 接口就是一个函数式接口，假设有一个 startThread 方法使用该接口作为参数，那么就可以使用 Lambda 进行传参。这种情况其实和 Thread 类的构造方法参数为 Runnable 没有本质区别。

```
public class TestDemo {
	private static void startThread(Runnable task) {
		new Thread(task).start();
	}

	public static void main(String[] args) {
		startThread(() -> System.out.println("线程任务执行！"));
	}
}
```

如果一个方法的返回值类型是一个函数式接口，那么就可以直接返回一个 Lambda 表达式。

```
import java.util.Arrays;
import java.util.Comparator;
public class TestDemo {
	private static Comparator<String> newComparator() {
		return (a, b) -> b.length() - a.length();
	}
	public static void main(String[] args) {
		String[] array = { "abc", "ab", "abcd" };
		System.out.println(Arrays.toString(array));
		Arrays.sort(array, newComparator());
		System.out.println(Arrays.toString(array));
	}
}
```

23-3_方法引用

方法引用的出现就是为了更加精简 Lambda 表达式而出现的，Lambda 表达式就是提供一个实现方案，而如果这个实现方案其他地方已经实现了，那么这个时候就用方法引用代替 Lambda 表达式，使得代码更加简捷

方法引用的格式：　　:: 是一个运算符

方法引用的分类

- **通过对象名引用成员方法：　对象名::方法名**
- **通过类名称引用静态方法：类名::方法名**
- **通过 super 引用成员方法：super::父类方法名**
- **通过 this 引用成员方法：this::本类方法名**
- **类的构造器引用：类名::new**
- **数组的构造器引用：数组类型[]::new**

23-3-1_对象名引用成员方法

```
/**
 * 函数式接口
 */
@FunctionalInterface
public interface Printable {
```

```
    void print(String str);
}
```

```
public class StringOperate {
    // 字符转大写
    public void printInfo(String str) {
        System.out.println(str.toUpperCase());
    }
    // 字符串反转后在大写
    public void reverseStr(String str) {
        System.out.println(new StringBuilder(str).reverse().toString().toUpperCase());
    }
}
```

```
/**
 * 方法引用：
 *      对象名::方法名
 */
public class PrintTest {
    public static void main(String[] args) {
        // Lambda 表达式
        printString((str)->{
            StringOperate su = new StringOperate();
            su.printInfo(str);
        });
        // 对象名::方法名
        StringOperate su = new StringOperate();
        printString(su::printInfo);
        printString(su::reverseStr);
    }
    private static void printString(Printable p) {
        p.print("Hello, World.");
    }
}
```

23-3-2_类名称引用静态方法

```
/**
 * 函数式接口
 */
@FunctionalInterface
public interface Calcable {
    int calc(int num);
}
```

```
/**
 * 方法引用：
 *      类名::方法名
 */
public class CalcTest {
    public static void main(String[] args) {
        /*
            希望调用 method 方法,可以输出正数 5
            lambda 的解决方案,其实就是执行 Math 类中的 abs 方法
            该 lambda 的解决方案,其实在 Math 的 abs 方法中已经实现了,这个时候 jdk8 认为可以
            使用方法引用,把 Math 的 abs 方法引用过来,替换 lambda 表达式实现更优
```

```
            */
            method((num)->{return Math.abs(num);});
            method(Math::abs);
        }
        public static void method(Calcable c) {
            int abs = c.calc(-5);
            System.out.println(abs);
        }
}
```

23-3-3_super 引用成员方法

```
@FunctionalInterface
public interface Greetable {
        void greet();
}
```

```
public class Person {
        public void sayHello() {
            System.out.println("大家好,我是渣渣辉,欢迎来玩贪玩蓝月!");
        }
}
```

```
public class Student extends Person {
        public void sayHello(){
            System.out.println("大家好,我是古天乐,欢迎大家来玩贪玩蓝月!");
        }

        //  定义一个方法,使用 Greetable 接口对象调用 greet 来打招呼
        public void method(Greetable greetable){
            greetable.greet();
        }

        // show1()方法,就是调用 method()方法
        public void show1(){
            method(()->{
                // lambda 的打招呼的解决方案是调用 Person 类中的 sayHello()方法来打招呼
                new Person().sayHello();
            });
        }

        public void show2(){
            method(()->{
                // lambda 的打招呼的解决方案是调用 Person 类中的 sayHello()方法来打招呼
                super.sayHello();
            });
        }
        //  我们发现,lambda 中的解决方案,其实就是父类 Person 中的 sayHello()方法
        // jdk8 认为,就可以把 Person 类中的 sayHello()方法直接引用过来替换 lambda 表达式

        //  引用父类的方法      格式: super::方法名
        public void show3(){
            method(super::sayHello);
        }
}
```

```
/**
 * 方法引用
 *          super::父类方法名
 */
public class GreetableTest {
	public static void main(String[] args) {
		Student stu = new Student();
		stu.show1();
		stu.show2();
		stu.show3();
	}
}
```

24-3-4_this 引用成员方法

```
@FunctionalInterface
public interface Greetable {
	void greet();
}
```

```
public class Person {
	public void sayHello() {
		System.out.println("大家好,我是渣渣辉,欢迎来玩贪玩蓝月!");
	}
}
```

```
public class Student extends Person {
	public void sayHello(){
		System.out.println("大家好,我是古天乐,欢迎大家来玩贪玩蓝月!");
	}

	// 定义一个方法,使用 Greetable 接口对象调用 greet 来打招呼
	public void method(Greetable greetable){
		greetable.greet();
	}

	// show1()方法,就是调用 method()方法
	public void show1(){
		method(()->{
			// lambda 的打招呼的解决方案是调用本类中的 sayHello()方法来打招呼
			sayHello();
		});
	}

	public void show2(){
		method(()->{
			// lambda 的打招呼的解决方案是调用本类中的 sayHello()方法来打招呼
			this.sayHello();
		});
	}
	// 我们发现,lambda 中的解决方案,其实就是本类中的 sayHello()方法
	// jdk8 认为,就可以把本类中的 sayHello()方法直接引用过来替换 lambda 表达式

	// 引用本类的成员方法     格式: this::方法名
	public void show3(){
```

```
            method(this::sayHello);
        }
}
```

```
/**
 * 方法引用
 *          this::本类方法名
 */
public class GreetableTest {
        public static void main(String[] args) {
                Student stu = new Student();
                stu.show1();
                stu.show2();
                stu.show3();
        }
}
```

24-3-5_类的构造器引用

```
@FunctionalInterface
public interface BuildPerson {
        Person builderPerosn(String name);
}
```

```
public class Person {
        private String name;

        public Person(String name) {
                this.name = name;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }
}
```

```
public class BuildPersonTest {
        // 需求:定义一个方法,用来创建一个 Person 对象,打印一下 person 对象的姓名
        public static void printPersonName(BuildPerson bp,String name){
                Person p = bp.builderPerosn(name);
                System.out.println(p.getName());
        }

        public static void main(String[] args) {
                // lambda 的解决方案,其实就是调用 Person 类中的构造方法
                // 而 jdk8 中,认为如果 lambda 的解决方法,已经在另一个方法的实现中,那么这个时候我们应该把该方法引用过来,替换 lambda

                printPersonName((String name)->{return new Person(name);}, "jack");

                // 引用 Person 类中的构造方法: Person::new
```

```
            printPersonName(Person::new,"tom");
        }
}
```

24-3-6_数组的构造器引用

```
/**
 * 创建一个函数式接口-实例化数组
 */
@FunctionalInterface
public interface ArrayInstance {
    public abstract int[] InitArray(int len);
}
```

```
/**
 * 方法引用
 *          数组的构造器引用
 *              数组类型[]::new
 */
public class ArrayInstanceTest {
    public static void main(String[] args) {
        // lambda 的解决方案,就是创建一个指定长度的数组
        show(len ->new int[len], 10);
        //  引用数组的创建方式: int[]::new
        show(int[]::new, 10);
    }
    public static void show(ArrayInstance ai, int len) {
        int[] arr = ai.InitArray(len);
        System.out.println("创建出来的数组长度是：" + arr.length);
    }
}
```

23-4_常用函数式接口

生产型-函数式接口：

java.util.function.Supplier<T>接口

仅包含一个无参的方法：**T get()**，用来以获取一个泛型参数指定类型的对象数据。

get()是只出不进

消费型-函数式接口

Consumer 接口

java.util.function.Consumer<T>接口则正好相反，它不是生产一个数据，而是消费一个数据，其数据类型由泛型参数决定。只进不出

抽象方法：accept

Consumer 接口中包含抽象方法 **void accept(T t)**，意为消费一个指定泛型的数据。

default Consumer<T> andThen(Consumer<? super T> after)

作用:消费一个数据的时候，首先做一个操作，然后再做一个操作

翻译: 一个数据可以被消费 2 次,并且可以指定顺序消费

23-4-1_Supplier(生产接口)

```
/**
 * java.util.function.Supplier<T>接口
```

```
 *        仅包含一个无参的方法：T get()，用来以获取一个泛型参数指定类型的对象数据。
 *        get()是只出不进
 */
/*
    练习：求数组元素最大值
    题目
    使用 Supplier 接口作为方法参数类型，
    通过 Lambda 表达式求出 int 数组中的最大值。提示：接口的泛型请使用 java.lang.Integer 类。
 */
public class SupplierTest {
    public static void main(String[] args) {
        printString(()->{return new String("jack");});
        int[] arr = {1, 2, 3, 4, 5, 8};
        int maxNum = getMax(() -> {
            int max = arr[0];
            for (int i = 0; i < arr.length; i++) {
                if (arr[i] > max) {
                    max = arr[i];
                }
            }
            return max;
        });
        System.out.println(maxNum);
}
/**
 * 定义一个方法,生产出来一个字符串数据,并把打印出来
 */
public static void printString(Supplier<String> supplier) {
        String str = supplier.get();
        System.out.println(str);
}
/**
 * 定义一个方法,让 Supplier 接口作为方法参数类型,过 Lambda 表达式求出 int 数组中的最大值
 * @param supplier
 * @return
 */
public static int getMax(Supplier<Integer> supplier) {
        Integer max = supplier.get();
        return max;
    }
}
```

23-4-2_Consumer(消费接口)

```
/**
 * Consumer 接口
 * java.util.function.Consumer<T>接口则正好相反，它不是生产一个数据，
 * 而是消费一个数据，其数据类型由泛型参数决定。只进不出
 *
 * 抽象方法：accept
 * Consumer 接口中包含抽象方法 void accept(T t)，意为消费一个指定泛型的数据。基本使用
 */
public class ConsumerTest1 {
```

```
    public static void main(String[] args) {
        method((str)->{
            // 字符串反转
            str = new StringBuilder(str).reverse().toString();
            System.out.println(str);
        }, "jack come on.");
    }

    /**
     * 定义一个 method 方法,消费传进来的 str 字符串
     * @param consumer
     * @param str
     */
    public static void method(Consumer<String> consumer, String str) {
        consumer.accept(str);
    }
}
```

```
/*
    消费的方法: Consumer 中的 accept(T t)方法;

    default Consumer<T> andThen(Consumer<? super T> after)
    作用:消费一个数据的时候，首先做一个操作，然后再做一个操作
        翻译: 一个数据可以被消费 2 次,并且可以指定顺序消费

    案例:
        小明    去体育馆      小明打篮球
                              小明大乒乓球
 */
public class ConsumerTest2 {
    public static void main(String[] args) {
        method("jack",
                (name)->{
                    System.out.println(name + "打篮球");
                },
                (name)->{
                    System.out.println(name + "踢足球");
                });
    }
    public static void method(String name, Consumer<String> one, Consumer<String> two) {
        two.andThen(one).accept(name);
    }
}
```

```
/*
    练习：格式化打印信息
    题目
    下面的字符串数组当中存有多条信息
    请按照格式“姓名：XX。性别：XX。”的格式将信息打印出来。

    要求将打印姓名的动作作为第一个 Consumer 接口的 Lambda 实例，
    将打印性别的动作作为第二个 Consumer 接口的 Lambda 实例，
    将两个 Consumer 接口按照顺序“拼接”到一起。
 */
public class ConsumerTest3 {
    public static void main(String[] args) {
```

```
            String[] arr = {"jack, man", "rose, woman", "jane, man"};
            method(
                        (str)->{System.out.print("姓名: " + str.split(",")[0]);},
                        (str)->{System.out.println("  性别: " + str.split(",")[1]);},
                        arr
            );
      }
      public static void method(Consumer<String> one, Consumer<String> two, String[] arr) {
            for (String s : arr) {
//                      one.andThen(two).accept(s);
                  one.accept(s);
                  two.accept(s);
            }
      }
}
```

24-函数式接口和 Stream 流

24-1_复习

@FunctionalInterface 注解 用来标识函数式接口

函数式接口是 Lambda 的前提，方法引用和 Lambda 是兄弟关系

如果 Lambda 表达式的解决方案，已经在另一个方法的实现中，那么这个时候就直接把该方法引用过来，替换 Lambda

:: 运算符　由::运算符连接的式子就是方法引用

函数式接口：有且仅有一个抽象方法的接口

1.Lambda

概述：jdk8 专门为函数式接口 提供的一个实例(接口的一个对象)

格式：

(参数)->{代码语句}

省略规则：

- 小括号中的参数类型可以省略
- 如果小括号中有且仅有一个参数的时候，那么小括号也可以省略
- 如果大括号中只有一条语句，那么无论是否有返回值，大括号、return、分号都可以省略

使用：

一般是在方法的参数、方法的返回值

2.方法引用

如果 Lambda 表达式的解决方案，已经在另一个方法的实现中，那么这个时候就直接把该方法引用过来，替换 Lambda

什么时候使用方法引用：

如果 lambda 表达式的解决方案，就是仅仅只是执行另一个方法，那么就直接把那个方法引用过来，替代 lambda

如何引用：

:: 运算符

对象引用成员方法：对象名::成员方法名

类名引用静态方法：类名::静态方法名

super 引用父类方法：super::父类方法名

this 引用本类方法：this::本类方法名

构造器的引用：类名::new

数组的引用: 数据类型[]::new　　int[] :: new

3.常用的函数式接口 2 个
Supplier <T>　　T get()　---->生产接口
Consumer <T>　　void accept(T t) ---->消费接口

24-2_常用函数式接口

24-2-1_Predicate(判断接口)

java.util.function.Predicate<T>接口

1-test 方法

abstract boolean test(T t) - 用于条件判断的场景
用来判断一个数据，是否符合某个规则

Demo:

```
public class TestDemo {
   public static void main(String[] args) {
        method((str)->{return str.length()>3;}, "jack");
        method(new Predicate<String>() {
            @Override
            public boolean test(String s) {
                boolean b = s.length()>5;
                return b;
            }
        }, "rose");
    }
    public static void method(Predicate<String> p, String str) {
        boolean flag = p.test(str);
        System.out.println(flag);
    }
}
```

2-and 方法

default Predicate<T> and(Predicate<? super T> other)
只要是判断条件，就会出现：与、或、非这三种情况
两个 Predicate 条件使用“与”逻辑连接起来实现“并且”的效果时，可以使用 default 方法 and
·predicate.and(anotherPredicate).test(str)

Demo:

```
public class TestDemo {
    public static void main(String[] args) {
          method("jackloverose", (str)->{return str.length() > 3;}, (str)->{return
str.contains("jack");});
          method("jackloverose", str->str.length() > 3, str->str.contains(str));
    }
    public static void method(String str, Predicate<String> p1, Predicate<String>
p2) {
        boolean flag = p1.and(p2).test(str);
        System.out.println(flag);
    }
}
```

3-or 方法

default Predicate<T> or(Predicate<? super T> other)

与 and 的“与”类似，默认方法 or 实现逻辑关系中的“或”。

- **predicate.or(anotherPredicate).test(str)**

Demo:

```
public class TestDemo {
   public static void main(String[] args) {
        method(str->str.length()>2, str->str.contains("a"), "jack");
    }
    public static void method(Predicate<String> p1, Predicate<String> p2, String
str) {
        boolean flag = p1.or(p2).test(str);
        System.out.println(flag);
    }
}
```

4-negate 方法

default Predicate<T> negate()

剩下的“非”（取反）也会简单。

它是执行了 test 方法之后，对结果 boolean 值进行”!”取反而已，一定要在 test 方法调用之前调用 negate 方法

- **predicate.negate().test(str)；**

Demo:

```
public class TestDemo {
   public static void main(String[] args) {
        method(str->str.contains("a"), "jack");
    }
    public static void method(Predicate<String> p, String str) {
        boolean flag = p.negate().test(str);
        System.out.println(flag);
    }
}
```

24-2-2_Function(转换接口)

java.util.function.Function<T,R>接口 - 转换接口

用来根据一个类型的数据得到另一个类型的数据，前者称为前置条件，后者称为后置条件。有进有出，所以称为“函数 Function”。

1-apply 方法

Function 接口中最主要的抽象方法为：**R apply(T t)，根据类型 T 的参数获取类型 R 的结果**

Demo:

```
public class TestDemo {
   public static void main(String[] args) {
        method(i->String.valueOf(i), 123);
    }
    public static void method(Function<Integer, String> f, Integer i) {
        String str = f.apply(i);
```

```
        str += "jack";
        System.out.println(str);
    }
}
```

2-andThen 方法

default <V> Function<T,V> andThen(Function<? super R,? extends V> after)

Function 接口中有一个默认的 andThen 方法，用来进行组合操作。

该方法同样用于“先做什么，再做什么”的场景，和 Consumer 中的 andThen 差不多

· function.addThen(anotherFunction).apply(t)

Demo:

```
public class TestDemo {
    public static void main(String[] args) {
        method("123", str->Integer.parseInt(str)*10, i->String.valueOf(i));
    }
    public static void method(String str, Function<String, Integer> f1,
Function<Integer, String> f2) {
        String s = f1.andThen(f2).apply(str);
        System.out.println(s);
    }
}
```

24-3_Stream 流(流水线操作)

24-3-1_Stream 流特点介绍

- stream 流不能存储数据
- stream 流是单向的，不能重复操作
- stream 流不会改变源数据
- stream 流中的部分操作是延迟的

24-3-2_Stream 流的分类

延迟方法：Stream 流中只要返回值是 Stream 的就是延迟方法

终结方法：Stream 流中只要返回值不是 Stream 的就是终结方法

24-3-3_Stream 流的使用步骤和场景

当使用一个流的时候，通常包括三个基本步骤：**获取一个数据源（source）→ 数据转换→执行操作获取想要的结果**

如果有一个集合或者数组，希望通过一系列的筛选得到一个结果，那么就可以使用 stream 流比较优化

Stream 流通过集合或者数组获取数据源---->数据转换(筛选)--->得到结果

24-3-4_Stream 流的常用方法

1-获取流(延迟方法)

获取一个流非常简单，有以下几种常用的方式：

- 所有的 Collection 集合都可以通过 stream 默认方法获取流；

• Stream 接口的静态方法 of 可以获取数组对应的流。

List 和 Set 集合获取流的方法：
default Stream<E> stream()
数组获取流的方式：
static <T> Stream<T> of(T... values)
map 集合获取流的方式：
Stream<T> map.keySet.stream()
Stream<T> map.values.stream()
Stream<T> map.entrySet.stream()

Demo:

```
public class TestDemo {
   public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        Set<String> set = new HashSet<>();
        Integer[] arr = new Integer[10];
        // List集合获取流
        Stream<String> listStream = list.stream();
        // Set集合获取流
        Stream<String> setStream = set.stream();
        // 数组获取流
        Stream<Integer> arrStream = Stream.of(arr);

        // Map集合获取流 - 三种方式
        Map<String, String> map = new HashMap<>();
        Stream<String> mapKeyStream = map.keySet().stream();
        Stream<String> mapValuesStream = map.values().stream();
        Stream<Map.Entry<String, String>> mapEntryStream = map.entrySet().stream();
   }
}
```

2-filter(延迟方法)

filter 过滤操作
将一个流转换成另一个子集流
Stream<T> filter(Predicate<? super T> predicate);
filter 方法的返回值是一个新的 Stream 流
filter 方法的参数是一个 Predicate 函数的接口 -→该接口是一个判断接口
Predicate 接口中的抽象方法是 boolean test(T t)，该方法是用来判断一个类型的对象是否符合某个规则
解析：
把 Stream 流中的元素传递给 Predicate 接口,使用接口中的 test 方法进行判断是否符合规则,如果 test 方法
返回的结果是 true,那么该元素就保留在新的流中,如果 test 方法返回的是 false,那么该元素就不保留在新的流中

Demo:

```
public class TestDemo {
   public static void main(String[] args) {
      List<String> listA = new ArrayList<>();
        // 添加元素
        listA.add("张无忌");
        listA.add("周芷若");
        listA.add("赵敏");
```

```
        listA.add("张全蛋");
        listA.add("张三丰");

        // 未使用Stream流的做法
        // 找出所有姓张的名字存储到集合B中
        ArrayList<String> listB = new ArrayList<>();
        for (String name : listA) {
            if (name.startsWith("张")) {
                listB.add(name);
            }
        }
        // 找出姓名长度为3的元素
        ArrayList<String> listC = new ArrayList<>();
        for (String name : listB) {
            if (name.length() == 3) {
                listC.add(name);
            }
        }
        System.out.println(listC);
        // 使用Stream流的做法
        listA.stream().filter(str -> str.startsWith("张"))
            .filter(str -> str.length() == 3).forEach(System.out::println);
    }
}
```

3-forEach(终结方法)

void forEach(Consumer<? super T> action);

forEach 方法常用 filter 方法搭配使用

解析：

forEach 方法没有返回值

forEach 方法的参数是 Consumer 函数式接口 --->消费接口

Consumer 函数式接口中的抽象方法是 void accept(T t)该方法就是用来消费一个类型的数据

把 Stream 流中的元素传递给 Consumer 接口，使用 Consumer 接口中的 accept()方法对该元素进行消费

Demo:

```
public class TestDemo {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("张无忌", "张三丰", "张翼德", "张伟", "张全蛋
");
        stream.filter(str->str.startsWith("张")).forEach(System.out::println);
    }
}
```

4-count(终结方法)

long count() 统计个数

```
public class TestDemo {
    public static void main(String[] args) {
        // 统计姓张的个数
        Stream<String> stream = Stream.of("张无忌", "张三丰", "张翼德", "张伟", "张全蛋
", "赵敏", "周芷若");
```

```
        long num = stream.filter(str -> str.startsWith("张")).count();
        System.out.println(num);
    }
}
```

5-limit(延迟方法)

Stream<T> limit(long maxSize)

limit 方法可以对流进行截取，只取用前几个

Demo:

```
public class TestDemo {
    public static void main(String[] args) {
        // 取 姓张的前3个名字
        Stream<String> stream = Stream.of("张无忌", "张三丰", "张翼德", "张伟", "张全蛋
", "赵敏", "周芷若");
        stream.filter(str->str.startsWith("张
")).limit(3).forEach(System.out::println);
    }
}
```

6-skip(延迟方法)

Stream<T> skip(long n)

如果希望跳过前几个元素，可以使用 skip 方法获取一个截取之后的新流

Demo:

```
public class TestDemo {
    public static void main(String[] args) {
        // 取姓张的名字，从第3个开始
        Stream<String> stream = Stream.of("张无忌", "张三丰", "张翼德", "张伟", "张全蛋
", "赵敏", "周芷若");
        stream.filter(str->str.startsWith("张
")).skip(2).forEach(System.out::println);
    }
}
```

7-map(延迟方法)

<R> Stream<R> map(Function<? super T, ? extends R> mapper);

如果需要将流中的元素映射到另一个流中，可以使用 map 方法

解析：

map 方法的返回值是一个新的 Stream

map 方法的参数是一个 Function 函数式接口--->该函数式接口是一个转换接口

R apply(T t)把 T 类型转换为 R 类型

把 Stream 流中的元素传递给 Function 接口，使用 Function 接口中的 apply()方法，对元素进行转换，转换之后存储到新的 Stream 流中

Demo:

```
public class TestDemo {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("100","200","300","400");
        // 把str转换为Integer类型
        stream.map(str ->
```

```
Integer.parseInt(str)).forEach(str->System.out.println(str+1));
   }
}
```

8-concat(延迟方法)

static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)

如果有两个流，希望合并成为一个流，那么可以使用 Stream 接口的静态方法 concat

备注：这是一个静态方法，与 java.lang.String 当中的 concat 方法是不同的。

Demo:

```
class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                '}';
    }
}
public class TestDemo {
   public static void main(String[] args) {
       /*
            需求:
                1. 第一个队伍只要名字为3个字的成员姓名;
                2. 第一个队伍筛选之后只要前3个人;
                3. 第二个队伍只要姓张的成员姓名;
                4. 第二个队伍筛选之后不要前2个人;
                5. 将两个队伍合并为一个队伍;
                6. 根据姓名创建Person对象;
                7. 打印整个队伍的Person对象信息。
        */
        //第一支队伍
        ArrayList<String> one = new ArrayList<>();
        one.add("迪丽热巴");
        one.add("宋远桥");
        one.add("苏星河");
        one.add("石破天");
        one.add("石中玉");
        one.add("老子");
        one.add("庄子");
```

```
        one.add("洪七公");

        //第二支队伍
        ArrayList<String> two = new ArrayList<>();
        two.add("古力娜扎");
        two.add("张无忌");
        two.add("赵丽颖");
        two.add("张三丰");
        two.add("尼古拉斯赵四");
        two.add("张天爱");
        two.add("张二狗");

        // 1. 第一个队伍只要名字为3个字的成员姓名;
        // 2. 第一个队伍筛选之后只要前3个人;
        Stream<String> streamF = one.stream().filter(str -> str.length() ==
3).limit(3);
        // 3. 第二个队伍只要姓张的成员姓名;
        // 4. 第二个队伍筛选之后不要前2个人;
        Stream<String> streamS = two.stream().filter(str -> str.startsWith("张
")).skip(2);
        // 5. 将两个队伍合并为一个队伍;
        // 6. 根据姓名创建Person对象;
        // 7. 打印整个队伍的Person对象信息。
        Stream<String> streamNew = Stream.concat(streamF, streamS);
        streamNew.map(str -> new Person(str)).forEach(System.out::println);
    }
}
```

24-4_并发流

24-4-1_串行流转并发流

S parallel();　把 串行流 转换为 并发流

只需要在流上调用一下无参数的 parallel 方法，那么当前流即可变身成为支持并发操作的流，返回值仍然为 Stream 类型

Demo:

```
public class TestDemo {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("张三丰", "张无忌", "张翠山", "张天爱");
        // 把Stream流转换为并发流
        stream.parallel().forEach(System.out::println);
    }
}
```

24-4-2_通过集合获取并发流

default Stream<E> parallelStream() {...}

在通过集合获取流时，也可以直接调用 parallelStream 方法来直接获取支持并发操作的流

Demo:

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

```
        ArrayList<String> one = new ArrayList<>();
          one.add("迪丽热巴");
          one.add("宋远桥");
          one.add("苏星河");
          one.add("石破天");
          one.add("石中玉");
          one.add("老子");
          one.add("庄子");
          one.add("洪七公");
          // 集合转换为并发流
          one.parallelStream().forEach(System.out::println);
    }
}
```

24-4-3_Stream 流中的数据收集到集合中

Stream 流提供 collect 方法

<R,A> R collect(Collector<? super T,A,R> collector)

要想传递 Collector 对象,就需要使用 Collectors 中的静态方法:

- public static <T> Collector<T, ?, List<T>> toList()：转换为 List 集合。
- public static <T> Collector<T, ?, Set<T>> toSet()：转换为 Set 集合。

Demo:

```
public class TestDemo {
   public static void main(String[] args) {
      // 把一个Stream流 里面的数据 转换到 集合中\数组中
        Stream<String> stream = Stream.of("张三丰", "张无忌", "张翠山", "张天爱");

        // 把stream流中的数据收集到List集合中
        List<String> list = stream.collect(Collectors.toList());
        System.out.println(list);

        // 把stream流中的数据收集到Set集合中
        Set<String> set = stream.collect(Collectors.toSet());
        System.out.println(set);
    }
}
```

24-4-4_ Stream 流中的数据收集到数组中

Stream 提供 toArray 方法来将结果放到一个数组中，由于泛型擦除的原因，返回值类型是 Object[]的

扩展：解决泛型数组问题

<A> A[] toArray(IntFunction<A[]> generator);

toArray()方法的参数是 IntFunction 函数式接口,所以就传一个 lambda 表达式,该 lambda 方案是创建数组

Demo:

```
public class TestDemo {
   public static void main(String[] args) {
      // 把一个Stream流 里面的数据 转换到 集合中\数组中
        Stream<String> stream = Stream.of("张三丰", "张无忌", "张翠山", "张天爱");
        // 把stream流中的数据收集到数组中
        Object[] objs = stream.toArray();
        for (Object obj : objs) {
```

```
            String str = (String)obj;
            System.out.println(str);
        }
        //  把stream流中的数据收集到数组中    解决泛型数组问题
        String[] arr = stream.toArray(String[]::new);
    }
}
```

24-5_函数式接口大总结

常用的函数式接口
讲函数式接口 其实就是在说 见到了 这样的接口应用 比如用在返回值上比如用在参数列表上
实际上就是在说 这里要实现的功能是什么
而我们知道 Lambda 就是用来实现 具体代码的

Supplier<T>接口
生产型
T get()
可以理解为
Lambda 表达式 表达的就是 生产(返回)一个对象
Consumer<T>接口
消费型
void accept(T t)
可以理解为
Lambda 表达式 就是 消费(操作)一个对象

Predicate<T>接口
条件判断接口
就是对某种类型的数据 进行判断的
boolean test(T t)
可以理解为
Lambda 表达式 就是 用来判断 t 对象 是不是符合某些条件

Function<T,R>接口
转换接口
就是将 T 对象变为 R 对象
R apply(T t) 根据类型 T 的参数 获取类型 R 的结果
可以理解为
Lambda 表达式 就是用来完成 T 类型转换为 R 类型的操作方式

只出不进 Supplier
只进不出 Consumer
有进有出 Function
要是条件 Predicate

Stream 类似于 生产流水线
特点 理解为 管道
1：流水线作业
2：内遍历

对一列元素进行操作的
想使用

1：获取流
2：拼接模型
3：按照要求得到结果

流的来源
集合 数组 转换为 Stream 流

将 Stream 中数据 收集到集合或者数组中

Stream 中的方法

延迟方法
Stream<T> filter(Predicate<T> p) 通过该方法 将一个流转换成另一个子集流
参数是一个函数式接口 使用 Lambda 表达式 表达式完成的事情就是条件判断

Stream<T> limit(long n) 对流进行截取 只取前 n 个

Stream<T> skip(long n) 跳过前 n 个元素 截取一个新流

<R> Stream<R> map(Function<T,R> lambda) 需要将流中元素 映射到另一个流中 使用 map 方法
参数是一个函数式接口 可以将当前流中的 T 类型数据 转换为 R 类型 的流
Lambda 中就是完成转换功能的

终结方法
long count() 获取流中的个数 但是这个功能做完 该流程结束
forEach(Consumer<T> lambda) 逐一遍历
参数是函数式接口 会将每一个流元素交给该函数进行处理 交给 Lambda 表达式

静态方法
of(...) 添加数据 形成一个管道
static Stream<T> concat(Stream<T> a,Stream<T> b)
合并流

收集方法
将我们的 流 转换成 List 集合
collect(Collectors.toList())
转换成 Set 集合
collect(Collectors.toSet())

Object[] toArray()
转换成指定数组
T[] toArray(len->new T[len])
获取流
Collection stream()方法

Stream 不是集合元素 JDK8 中的一个新特性 被我们称为 流

也不是数据结构 不保存数据 它是有关算法和计算操作的 更像高级别版本的迭代器
Stream 单向 不可往复 数据只能遍历一次 就好比流水 从前面流过了 一去不复返

不同的地方在于 支持并行化操作

串行方式

一个一个依次执行

并行 并发

在一个时间段内 多个操作并发的执行