

JavaSudy

day1

java的优势

- 跨平台
 - 什么是平台?
 - 指的是操作系统
 - 常见的操作系统?
 - Windows Unix Linux Solaris[Sun]
 - 什么是跨平台?
 - 用java开发出来的应用程序 不受底层操作系统的限制 可以运行在所有的操作系统上面
 - 底层的功臣?
 - JVM = java虚拟机 = 1 + 2 + 3
 - 秘书 + 保镖 + 翻译
 - 秘书 = **类加载器** = ClassLoader
 - 保镖 = **字节码校验器** = ByteCode Verifier
 - 翻译 = **解释执行器** = Interpreter
 - XXX.java[单词] -》 XXX.class[字节码文件] -》
- 安全 健壮
 - java的安全性体现在三个方面：
 - 少指针
 - 多异常[Exception]
 - 多自动的垃圾回收 -》 gc
 - 垃圾：一块不再访问的内存
 - 手动回收垃圾 -》 析构函数
- 免费 开源
- 简单
 - 语法简单：c++-- [取其精华 去其糟粕]
 - 糟粕：指针 运算符重载 手动的垃圾回收
 - 思想简单 = 面向对象的思想 = OO思想
 - 面向**过程**的思想：需要程序员站在计算机的角度去思考问题
 - 面向**对象**的思想：需要拿着代码去模拟现实生活
 - 类：一组类型相同事物高度抽象之后的集合概念
 - 创建对象的模板
 - 类好比工业生产中画的草图
 - 对象：类的一个具体的实例

- 范冰冰和人之间的关系：对象和类
 - 猫和HelloKitty之间关系：类和对象
 - 对象好比工业生产中 拿着草图生产出来一辆一辆具体的车子
 - 引用：对象的名字 `x = new XXX();`
 - 在java中每一个对象都需要有名字 一个对象同时可以拥有多个名字 如果一个对象在创建的时候没有名字的话 那么立刻会被gc回收掉 -》 零引用内存回收
 - 属性：对象有什么[一般是名词]
 - 方法：对象会什么[一般是动词]
 - 开发流程：
 - 先写测试类 -》 带有main方法的类
 - 再写实体类 -》 才有属性+方法
 - 定义属性：数据类型 属性名;
 - 定义方法：修饰符 + 返回类型 + 方法名(参数){...}
 - 在main方法测试
 - 创建对象：类名 对象名 = new 类名();
 - 属性赋值：对象名.属性 = 值;
 - 方法调用：对象名.方法名();
 - 动态更新
 - 对于核心类库里面提供的一些方法只保留一个指向的关系 当若干年后 核心类库升级了 我们的程序还是可以继续使用 这是挤兑c/c++的
-

安装jdk -> 下一步 下一步 下一步.....完成

配置环境变量 -> PATH=C:\Program Files\Java\jdk1.8.0_202\bin[我的电脑]

我的电脑 -》 右击 -》 属性 —》 高级 -》 环境变量 -》 PATH -> 编辑 -》 新建 -》 复制 -》 上移到第一个 -》

3个确定

day2

课程内容

- 搭建开发环境
 - 设置环境变量
 - 第一个程序HelloWorld
 - java中的软件包结构
-

搭建开发环境

- 安装jdk[www.oracle.com -> 下载相应的jdk版本]
 - SDK = software development kits = 软件开发工具包[目录/文件夹]
 - JDK = java + SDK = java软件开发工具包
 - jdk版本:
 - 永久修订版: 8 11 17
 - 非永久修订版
 - JRE = java runtime environment = java运行环境
 - 面试题: jdk和jre之间的区别?
 - jdk表示java软件开发工具包 如果一台电脑上面安装jdk之后 既可以编写代码 又可以运行代码
 - jre表示java运行环境 如果一台电脑上面安装jre之后 只能运行代码
 - 设置环境变量
 - | | 给谁使用 | 作用 |
|------|------|---------------------|
| PATH | 操作系统 | 让操作系统更加快捷的找到一个文件/命令 |
 - 为什么需要设置PATH?
 - 当我们在安装jdk的时候 假设装在默认盘符
 - 可是我们的代码可能写在其他的盘符 -> E:\课件 -> Test.java
 - 由于计算机看不懂.java文件 所以需要ctrl + 1翻译成.class文件
 - 当我们在按ctrl + 1的时候 底层操作系统开始找javac
 - 操作系统应该去哪里找javac 看环境变量PATH PATH指向哪里就去哪一个文件夹下面找
 - 所以我们需要设置环境变量 PATH指向jdk下面的bin目录
 - 如果在没有设置PATH的时候 PATH默认指向C:\Windows\System32文件夹
 - CLASSPATH 类加载器 让类加载器明确去哪里加载.class文件
 - 一般不需要设置 因为他有默认值 默认值: .[当前目录]
 - JAVA_HOME 其他的程序使用的
-

第一个程序HelloWorld

- 编译阶段: javac HelloWorld.java
 - 运行阶段: java HelloWorld
 - 如何在cmd里面编译运行
 - 进入cmd默认在C盘 假设我们的代码写在F:\课件 Test.java
 - 切换盘符 F:回车
 - 进入文件夹 cd 将所在的父目录拖拽进来 回车
 - 编译阶段: javac XXX.java
 - 运行阶段: java XXX
-

java中的软件包结构:

- 包结构 = 操作系统中的文件夹/目录
- 目录[包]的作用?
 - 按照文件的功能 性质进行分类 方便查找....
 - 给出不同的命名规范 从而解决重名的问题

- 如何打包
 - package 包 背包
 - 必须出现在文件的第一行
- 编译阶段:
 - 以前: javac XXX.java [没有包结构的代码]
 - 现在: javac -d . XXX.java [带有包结构的代码]
 - -d:自动创建包结构
 - .:当前文件夹下面创建
- 运行阶段:
 - 以前: java XXX [不带有包结构]
 - 现在: java 包结构.XXX [带有包结构]
- 在项目开发的时候 尽量避免包名叫
 - com0-com9:打印机设备
 - con:网络设备
 - nul:空设备

java中的数据类型

- 什么是数据类型?
 - 数据类型就是数据的单位 java是一个强类型的编程语言 所有的变量在第一次定义的时候必须有数据类型
- 数据类型分类:
 - 2种/无数种
 - **基本数据类型** 简单数据类型 [程序里面自带的数据类型]
 - 基本数据类型的分类:
 - **布尔类型: boolean** 只能使用true[真]/false[假]进行赋值
 - 涉及到判断
 - boolean x = true;
 - boolean y = false;
 - boolean z = 11;//c++可以 但是java语法出错
 - 字符类型: char 底层基于Unicode编码实现的 统一都是2个字节(2B) = 16个位(16bite)
 - 通常需要描述一个字/一个字母的时候 -> char
 - **char x1 = 'A';//指定字面值的赋值方式**
 - **char x2 = '男';//指定字面值的赋值方式**
 - char x3 = 55;//通过Ascii编码进行赋值
 - char x4 = '\u4e2d';//通过Unicode编码进行赋值
 - **char x5 = '\t';//转义字符**
 - **整数类型: 符号位1 + 数值位[n-1]**
 - byte short int long[L]
 - 8 16 32 64
 - 整数类型默认是int类型

- long类型结尾需要加上L/l 尽量写L
- 当数值超出int类型最大边界的时候 注意使用long类型
- 整数类型的赋值方式：
 - `int num1 = 55;`//十进制赋值
 - `int num2 = 055;`//八进制赋值
 - `int num3 = 0x55;`//十六进制赋值
 - `int num4 = 0b1100;`//二进制赋值
- byte:-128 127
- short:-32768 32767
- int:-2147483648 2147483647
- long:-XXXXXXXX XXXXXXXX
- 浮点类型：符号位 + 幂值位 + 数值位
 - float double[*]
 - 32 64
 - F/f D/d
 - float : 符号位1 + 幂值位8 + 数值位23
 - double:符号位1 + 幂值位11 + 数值位52
 - 拓展：BigDecimal精度比double类型更高
 - BigDecimal对象.add(BigDecimal对象)
 - BigDecimal对象.subtract(BigDecimal对象)
 - BigDecimal对象.multiply(BigDecimal对象)
 - BigDecimal对象.divide(BigDecimal对象)
 - `uu.money.add(new BigDecimal("500"))`
- 基本数据类型之间的转换
 - 小的数据类型可以**自动**变成大的数据类型 [向上转型]
 - 大的数据类型需要**强转**成小的数据类型 [向下转型]
 - short和char之间双向强转
 - byte【-128 127】和char【0-65535】之间双向强转
- 引用数据类型 自定义数据类型 [程序员自己开发的数据类型]

```

1  基本数据类型：
2      布尔类型：boolean -> true/false
3      字符类型：char -> 'a'/'中'/'\t'
4      整数类型：byte short int long -> 55
5      浮点类型：float double -> 45.5
6
7      boolean:判断
8      char:一个字/字母
9      int long:整数
10     double BigDecimal[引用数据类型]:小数
  
```

- **[+ - * / %]**
 - +:当+遇到String类型的时候 不再表示算数加法 而是表示**追加连接**
 - *:当数值超出int类型最大边界 注意使用long类型
 - /:整数/整数 = 整数
 - $7 / 2 = 3$
 - $7 / 2.0 = 3.5$
 - %:(**取模**)求余数 最终结果的符号位取决于参与运算的第一个数的符号
- **[+= -= *= /= %=...]**
 - 他们可以保证运算符的左边的数据类型不发生改变 -》隐式的强转
- **[++ -- 自增自减运算符]**
 - [a:取值 b:运算 x++:ab ++x:ba]
- **[> < >= <= != ==]**
 - [$>$ $<$ $>=$ $<=$]结果都是boolean类型
 - !:非 !boolean !true = false !false = true;
 - !=:判断左右两边的值是否不相等
 - `System.out.println(45 != 45);`//false
 - =和==之间的区别?
 - =: 赋值运算符 将右侧的值交给左侧的变量保管 `int x = 45;`
 - ==:判断左右两边的值是否相等 `gender == '女'`
- **&& || & | [逻辑运算符]**
 - 面试题: &&和&之间的区别?
 - 相同点: &&和&都是逻辑的与 相当于汉文中并且的意思 需要前后两个条件都成立才能执行
 - 不同点: &&叫做**逻辑的短路与** 有**短路特性**

- 短路特性：如果通过前面的条件可以得出最终结果的话 后面的条件直接舍弃 不在判断
- 18岁女生 `age == 18 && gender == '女'`
- 面试题：||和|之间的区别？
 - 相同点：||和|都是逻辑的或 相当于汉文中或者的意思 需要前后只要有一个条件成立就能执行
 - 不同点：||叫做**逻辑的短路或** 有短路特性

```

1  boolean    &&    boolean
2  true        判断
3  false       短路
4
5  boolean    ||    boolean
6  true        短路
7  false       判断

```

• & | ^ ~ [按位[二进制]运算符]

- 基本用法：

```

1  &:按位的与 上下二进制 都是1才能写1
2      System.out.println(9 & 5); //1
3
4      9=8 + 1 = 000...001001
5      5=4 + 1 = 000...000101  &
6      000...000001 -> 1
7
8  |:按位的或 上下二进制 只要有1就写1
9      System.out.println(9 | 5); //13
10     9=8 + 1 = 000...001001
11     5=4 + 1 = 000...000101  |
12     000...001101 -> 1 + 4 + 8 = 13
13
14 ^:按位的 异或 上下二进制不一样写1
15     9=8 + 1 = 000...001001
16     5=4 + 1 = 000...000101  ^
17     000...001100 -> 4 + 8 = 12
18
19 ~:按位的 取反 0->1  1->0
20     ~11101010
21     00010101

```

- 重点

```

1  &:在某些情况下取代%
2      某些情况 当我们拿着一个正数%上2的n次方数的时候 结果等价于
3      我们拿着这个正数 & 上2的n次方数-1的结果
4      x % 2(n) == x & 2(n)-1
5
6      System.out.println(100 % 8); //100-8-8-8-8-8-8-8-8-8-8=4
7      System.out.println(100 & 7); //二进制
8
9      27      %2      %4      %8      %16...%2(n)

```

```

10      1      3      3      11
11
12      27 &1      &3      &7      &15...&2(n)-1
13      1      3      3      11
14
15  ^:可以不借助第三块空间的方式交换两个变量的值
16      int x = 45;
17      int y = 90;
18      int z = x;
19      x = y;
20      y = z;
21
22      x = x ^ y;
23      y = x ^ y;
24      x = x ^ y;
25
26      能够使用^交换的数据类型: char byte short int long boolean

```

- [**>>** **<<** **>>>** **位 [二进制] 移运算符**]

```

1  <<相当于乘以 移动多少位相当于乘以2的几次方数
2  >>相当于除以 移动多少位相当于除以2的几次方数
3
4      *2      *4      *8      *16...*2(n)
5      <<1      <<2      <<3      <<4...<<n
6
7      /2      /4      /8      /16.../2(n)
8      >>1      >>2      >>3      >>4...>>n
9
10  >>和>>>之间的区别?
11      >>叫做 有符号右移 移动之后空缺的高位看原本的符号位
12      01110101 >> 2
13      00011101
14
15      10011111 >> 3
16      11110011
17
18      >>>叫做 无符号右移 移动之后空缺的高位直接补0
19      01110101 >>> 2
20      00011101
21
22      10011111 >>> 3
23      00010011

```

- **?: [三目运算符 三元运算符]**

- boolean ? 2 : 3
- 第一部分为true的话 那么执行第二部分
- 第一部分为false 的话 那么执行第三部分


```
1 + - * / %
2 ++ --
3 > < >= <= ! = != ==
4 && ||
5
6 += -= *= /= %=...
7 & ^
8 << >>
9 ? :
```

java中的变量：

- 面试题：java中的**变量**分为几种？
 - **成员变量 实例变量** 属性：定义在类体里面的变量
 - **局部变量 自动变量**：定义在方法体里面的变量
- 面试题：成员变量和局部变量之间的区别？
 - 定义的位置不同
 - 成员变量：定义在**类体**里面
 - 局部变量：定义在**方法体**里面
 - 作用范围不同
 - 成员变量：也叫属性 表示对象有什么 **依赖于对象存在** 当我们new对象的时候 属性也会被创建 当我们回收对象的时候 属性也会被回收
 - 局部变量：从定义的那一行开始 一直到所在的方法体执行结束之前可以访问的 一旦所在的方法体执行结束 局部变量立即消亡 **依赖于方法存在**
 - 默认值不同
 - 成员变量：即使不赋值 也有默认值
 - 局部变量：没有默认值的 要求在使用之前必须先赋值
- 成员变量和局部变量可以重名
 - 在重名的时候 如果直接访问这个变量 默认是局部变量
 - 如果想要访问成员变量 需要在变量的前面加上this.

java中的流程控制：

- 分支
 - **if else**

```
1  语法格式：
2      if(boolean){
3          执行语句；
4      }else if(boolean){
5          执行语句；
6      }else if(boolean){
7          执行语句；
8      }else{
9          执行语句；
10     }
11 学会使用if else的排他特性简化代码 不要下面的条件里面写出上面条件不成立的等价判断
```

拓展：

plus1:当if else里面只有一个语句的时候 {}可以不写

plus2:当if里面有return语句的时候 else单词可以不写

plus3:当if条件为true return true 当if条件为false return false

其实return回去的就是条件判断而已

plus4:不要拿着一个boolean类型的变量和true做连等比较

o switch case

```
1  switch: 开关          case: 情况
2
3  语法格式：
4      switch(参数){
5          case XXX : 执行语句;[break;]
6          case YYY : 执行语句;
7          case ZZZ : 执行语句;
8      }
9 学会使用break共享代码 将执行相同操作的代码放在一起共享
10
11 面试题: switch case的参数可以传哪些数据类型?
12     jdk1.0   char byte short int
13     jdk5.0   enum[枚举]
14     jdk7.0   String
```

day4

课程内容

- java中的循环
- 循环控制
- java中的数组

java中的循环

- **for[控制次数]**

- 语法格式:

```
1  for(初始化循环变量;循环执行的条件;循环之后的变化){
2      循环执行的代码;
3  }
```

- **while[控制条件]**

- 语法格式:

```
1  初始化循环变量;
2  while(循环执行的条件){
3      循环执行的代码;
4      循环之后的变化
5  }
```

- **do while**

- 语法格式:

```
1  赋值;
2  do{
3      执行语句;
4      ++/--;
5  }while(条件);
6
7  打印1-100
8      int x = 1;
9      while(x <= 100){
10         System.out.println(x);
11         x++;
12     }
13
14     int y = 1;
15     do{
16         Ssystem.out.println(y);
17         y++;
18     }while(y <= 100);
```

- **循环控制:**

- 循环嵌套: 一个循环定义在另一个循环里面

- 外层循环一个数字对应整个内层循环

- 循环控制:

- **continue**[继续]**:**跳过本次循环 跳到所在循环的第三部分
 - **break**[中断]**:**跳出所在的循环 跳到所在循环的结束部分

- 循环标签：
 - 当多个循环嵌套在一起 代码写在最里面循环 处理外层循环的话 需要给外层循环贴标签
 - 然后再内层循环里面continue/break + 标签名
-

java中的数组：

- 什么是数组？
 - **数组：容器 装类型相同 存储空间连续的元素**
 - 数组的基础应用：
 - 创建对象：
 - `int[] data = new int[5];`
 - `int[] data = new int[]{45,66,28,10};`
 - `int[] data = {34,55,92,10,88};`
 - 如何得到某一个元素
 - 数组对象[下标]
 - 如何得到数组大小：
 - 数组对象.length
 - 如何遍历数组
 - for + 下标
 - `for(int x = 0;x < data.length;x++){x:下标 data[x]:元素}`
 - foreach forin since jdk5.0
 - `for(数组类型 x : data){}` **[增强for循环]**
-

数组的复制

- **System.arraycopy(1,2,3,4,5);**
 - 1:要复制的老数组对象
 - 2:老数组的起始下标位置
 - 3:要复制到的目标数组
 - 4:目标数组的起始下标位置
 - 5:要复制的长度
 - **Arrays.copyOf(老数组对象,新数组长度);**
 - `import java.util.*;`
-

数组的排序

- 手动排序 **冒泡排序**

```

1  for(int x = 0;x < data.length -1;x++){//x:控制的是内层循环的次数
2      for(int y = 0;y < data.length -1 -x;y++){//y:控制的元素的下标
3          if(data[y] > data[y + 1]){
4              int z = data[y];
5              data[y] = data[y + 1];
6              data[y + 1] = z;
7          }
8      }
9  }
10

```

- 自动排序
 - **Arrays.sort(data);**
 - import java.util.*;
 - 只能**升序**排序

引用数据类型的数组

- 当我们创建完一个引用数据类型的数组的时候 其实里面一个元素都没有 里面装的都是null
- 为了防止出现空指针异常 所以在创建完数组之后 一定要给每块空间赋值

day5

课程内容

- 面向对象
 - 封装
 - 继承
 - 多态
- 方法重载
- 方法覆盖

面向对象的三大特点：封装 继承 多态

面向对象的四大特点：封装 继承 多态 抽象

封装：用private将不想被外界访问到的属性/方法隐藏起来

- private:私有的 只有本类可以使用的
- public:公共的 谁都可以使用的
- 封装的好处？
 - 类的开发者：数据得到了保护 从而更加的安全
 - 类的使用者：将精力放在核心业务逻辑上
- 如何封装？
 - 对于属性的封装：每个属性用private修饰 每个属性配套提供两个public修饰的方法
 - 一个是用来**给属性赋值**的方法 -》 **setter**
 - 一个是用来**获得属性值**的方法 -》 **getter**

- 对于方法的封装：判断哪些是核心方法
 - 核心方法 -》 public修饰
 - 给核心方法提供服务的方法 -》 private修饰

继承：用extends[派生]实现两个类[子类 父类]之间is a[是一个 是一种]的关系

- 继承是最简单的代码共享方式之一
- java中的类**只能单根继承**
 - class A{}
 - class B{}
 - class C extends A,B{}错误

多态：一个对象有不同的类型去定义它

- 作用一：创建对象 [了解]
 - 父类类型 = new 子类对象()
 - Person x = new Teacher();
 - Animal aa = new Cat();
 - 如果我们使用多态创建一个对象之后 这个对象只能调用子类/父类共有的属性/方法 一旦调用子类特有的属性/方法 报错
- 作用二：放在参数里面用于**解除代码之间的耦合度** [重点]
 - 耦合度：代码之间的相互依赖程度
 - 高内聚 低耦合

方法重载 方法过载 Overload

- 面试题：方法重载的条件？
 - 需要发生在同一个类体中
 - 方法名需要一模一样
 - 方法的参数需要不同
 - 参数类型不同
 - 参数个数不同
 - 参数顺序不同
- 方法重载的作用？

```
1 同时满足用户的不同需求...
2  System.out.println();                println()
3  System.out.println(45);               println(int)
4  System.out.println(45.5);             println(double)
5  System.out.println(1000L);            println(long)
6  System.out.println('a');              println(char)
7  System.out.println("etoak");          println(String)
8  System.out.println(true);             println(boolean)
```

方法覆盖 方法重写 Override

- 方法覆盖的条件？
 - 需要发生在**有继承关系的两个类中** 而且是在子类里面发生的
 - 子类在继承得到父类的某些方法之后 觉得父类的实现不好 于是在子类里面重新实现一下
- 方法覆盖注意的知识点：

```
1 public void test(int x) throws Exception {}
2
3 脑袋          躯干          尾巴
4
5 脑袋：访问权限修饰符 >= 父类的权限
6      Sun公司认为子类重新实现的方法应该更加优秀 更加优秀的方法应该给更多的人使用 这样才利于java的发展
7
8 躯干：返回类型 + 方法签名
9      jdk5.0之前      一模一样
10     jdk5.0开始
11 尾巴：异常处理部分 <= 父类的异常      范围
12     Sun公司认为子类重新实现的方法应该更加优秀 更加优秀的方法不应该有更多的隐患
13
```

- jdk5.0开始 方法覆盖的时候 可以加上@Override
 - @Override:注解 用来给机器看的 表示下面的方法一定要覆盖父类的某些方法
- jdk5.0开始 方法覆盖的时候 其实返回类型可以改变了
 - 方法覆盖的时候 返回类型可以变成父类方法返回类型的子类类型 -》 协变返回类型

构造方法

- ✓ 构造方法是**创建对象的方法**
- ✓ 构造方法是创建对象的时候调用的方法
- 构造方法的作用？
 - 构造方法语法的出现 是为了方便程序员的开发
 - **可以再创建对象的同时 直接给属性赋值**
 - 在java中只要是个类就一定有构造方法 即使我们没有写系统也会提供一个**默认的构造方法**
 - 默认的构造方法：**无参 空体 没有对属性赋值**
 - 如果我们想要一边创建对象 一边对属性赋值的话 需要自己写一个构造方法
 - 一旦我们写出自己的构造方法之后 **默认的构造方法将不再提供**
- 构造方法的特点？
 - 没有返回类型 连void都不能写 -》 修饰符 + 方法签名
 - **构造方法的名字需要和类名一模一样** -》 通常首字母大写
- 构造方法的首行：
 - **super()**:要执行本构造方法之前 先去执行父类的构造方法 具体执行父类的哪一个构造方法 看super()里面的参数类型

- **this()**:要执行本构造方法之前 先去执行本类的其他的构造方法 具体执行本类的哪一个构造方法看参数类型
- 面试题:
 - 构造方法能不能方法重载?
 - 可以
 - 构造方法能不能方法覆盖?
 - 不可以 构造方法不能被子类继承

day6

课程内容

- 扫描仪
- 变量共享
- 方法封装
- 参数传递

扫描仪：Scanner

- Scanner:文本扫描器 用来对用户输入的内容进行扫描
 - 读取字符串：**sc.next() / sc.nextLine()** /忽略空格/
 - 读取数字：**sc.nextInt()**
 - 读取BigDecimal:**sc.nextBigDecimal()**....

参数传递：

- java中只有值传递
- java中基本数据类型**传值** 引用数据类型**传地址**

day7

课程内容

- String类常见的面试题
- String类常用的20个方法

String类型常见的面试题：

- 1：new和不new之间的区别？
 - String x = "OK";
 - String y = new String("OK");
 - 不new的方式涉及到常量池查找机制
 - new的方式 **在堆里面开辟空间** 与此同时在**常量池**里面再次判断有没有
- 2：**String**和**StringBuffer/StringBuilder**之间的区别？
 - String str = new String("OK");

- StringBuffer buffer = new StringBuffer("OK");
 - StringBuffer和StringBuilder在第一次创建对象的时候 底层会多**预留16块缓冲区**
 - 缓冲区为了追加连接效率高
- 3: **StringBuffer**和**StringBuilder**之间的区别?
 - StringBuffer同一时间**允许一个线程**进行访问 效率较**低** 但是**不会出现并发错误**
 - StringBuilder同一时间**允许多个线程**进行访问 效率较**高** 但是**可能会出现并发错误**

String类常见的20个方法

- 和**长度**有关的方法

返回类型	方法签名	作用
int	length()	得到一个字符串的字符个数

- 和**数组**有关的方法

返回类型	方法签名	作用
byte[]	getBytes()	将一个字符串转换成 字节数组
char[]	toCharArray()	将一个字符串转换成 字符数组
String[]	split(String)	将一个字符串按照 指定的内容劈开

- 和**判断**有关的方法

返回类型	方法签名	作用
boolean	equals(String)	判断两个字符串的内容是否一模一样
boolean	equalsIgnoreCase(String)	忽略大小写的比较 两个字符串的内容是否一模一样
boolean	contains(String)	判断一个字符串里面是否 包含某个内容
boolean	startsWith(String)	判断一个字符串是否 以指定的内容开头
boolean	endsWith(String)	判断一个字符串是否 以指定的内容结尾

- 和**改变内容**有关的方法 [字符串里面所有的方法都**不会直接的处理原本的字符串** 而是将符合条件的字符串返回给我们 所以注意接收]

返回类型	方法签名	作用
------	------	----

返回类型	方法签名	作用
String	toUpperCase()	将一个字符串全部转换成 大写
String	toLowerCase()	将一个字符串全部转换成 小写
String	replace(String,String)	将字符串里面某个内容 全部替换成指定的内容
String	replaceAll(String,String)	将字符串里面某个内容全部替换成指定的内容 支持正则表达式
String	replaceFirst(String,String)	将字符串里面 第一次出现的某个内容 替换成指定的内容
String	trim()	去除字符串的前后空格
String	substring(int x,int y)	从下标x一直 截取 到下标y-1对应的元素
String	substring(int x)	从下标x一直 截取 到字符串的 最后

- 和**位置**有关的方法

返回类型	方法签名	作用
char	charAt(int x)	找到某个 下标对应的元素
int	indexOf(String)	找到某个内容 第一次出现的下标
int	lastIndexOf(String)	找到某个内容 最后一次出现的下标

day8

课程内容

- String练习
- **StringBuffer**常见的方法
- 面向对象的高阶特征
 - 访问权限修饰符
 - **static**修饰符
 - **单例模式**
 - final修饰符
 - abstract修饰符

面试题：方法重载和方法覆盖之间的区别？

- 含义不同
 - 方法重载：Overload
 - 方法覆盖：Override
- 发生的位置不同
 - 方法重载：需要发生在同一个类体中
 - 方法覆盖：需要发生在有继承关系的两个类中
- 对参数的要求不同
 - 方法重载：参数必须不同[类型 个数 顺序]
 - 方法覆盖：参数必须相同
- 对返回类型的要求不同
 - 方法重载：没有要求
 - 方法覆盖：分版本
 - jdk5.0之前 一模一样
 - jdk5.0开始 协变返回类型
- 作用不同
 - 方法重载：同时满足用户的不同需求
 - 方法覆盖：让一个方法变的更加优秀

- | | |
|---|------------------|
| 1 | 作用不同 |
| 2 | 方法重载：同时满足用户的不同需求 |
| 3 | 方法覆盖：让一个方法变的更加优秀 |

面试题：

- String和StringBuffer/StringBuilder之间的区别？
 - 创建方式不同
 - String类创建方式：new 不new
 - StringBuffer、StringBuilder创建方式：new
 - 能不能修改原本的字符串
 - String类里面所有的方法都**不会直接处理**原本的字符串
 - StringBuffer/StringBuilder类里面所有的方法都**可以直接的处理**原本的字符串
 - 有没有缓冲区
 - String类没有缓冲区 单词多长数组空间大小多大
 - StringBuffer/StringBuilder在创建对象的时候 底层会多预留16块缓冲区 追加连接效率高

StringBuffer/StringBuilder类里面常用的方法

- append():追加连接
 - reverse():反转字符串
-

访问权限修饰符：

- **public**:公共的 谁都可以使用的
- **protected**:受保护的 只有本包内可以访问 + **包外有继承关系的子类中可以访问**
- **default**:默认的 只有本包内可以访问
- **private**:私有的 只有本类可以访问的
- =====
- 各自能修饰哪些东西？

修饰符	类	成员[属性+方法]
public	T	T
protected	F	T
default	T	T
private	F	T

- 面试题：
A类里面定义一个public修饰的方法 在B类里面能用吗？
不一定 如果B类想要访问A类里面的方法 前提是需要先访问A类

static修饰符：

- static:静态的 修饰符 相当于汉文中的形容词
- 面试题：static能修饰什么？
- **属性：静态属性**：表示整个类型共享一份的属性 不是每个对象都有一份的属性 需要拿着类名去调用
 - **普通属性**：表示每个对象都有一份的属性 需要拿着对象去调用
- **System.out**
- **System.in**
- 面试题：java中变量和属性谁大？
 - 变量
- static为什么不能修饰局部变量？
 - **static修饰的变量要求类一加载就要在内存里面找到他 但是局部变量只有方法调用 代码执行到那一行的时候才能创建 类的加载永远在前面 方法调用永远在后面 这两个时间点赶不上一起的所以static不能修饰局部变量**
- 方法：静态方法 需要拿着类名去调用
 - 普通方法：需要拿着对象去调用
- 面试题：静态方法和普通方法谁调用简单？
 - 静态方法
- 既然静态方法调用简单 开发的时候 为什么不把一个类里面所有的方法都定义成静态的呢？
 - 静态方法里面只能直接的访问静态的成员

- **普通方法**里面既可以访问**静态成员**也可以访问**非静态成员**
 - **Math.random()**
 - **Arrays.sort()**
 - **System.arraycopy();**
 - **Arrays.copyOf**
 - **System.currentTimeMillis()**
 - 代码块：初始化一个普通属性的
 - 如果我们想要给静态属性赋值的话 需要使用静态代码块
 - 普通代码块：给普通属性赋值的 普通属性每个对象都有一份的属性 每创建一个对象执行一次 创建几个对象执行几次
 - 静态代码块：给静态属性赋值的 静态属性整个类型共享一份 静态代码当类第一次被加载的时候执行 而且只执行一次
-

模式：java中本没有模式的概念 用的程序员多了 于是有了模式

单例模式：控制一个类有且只有一个对象

- 醉汉式：
 - 私有化构造方法 防止外界随意的创建对象
 - 创建一个私有的 静态的 属于本类类型的对象
 - 提供一个公共的 静态的 返回本类对象的方法
 - 懒汉式 -》尚不完善
-

final修饰符：

- final 最终的 修饰符 相当于汉文中的形容词
 - 面试题：final能修饰哪些东西？
 - 类：最终类 "骡子类"
 - 特点：可以有父类 但是**不能有子类**
 - java中的String类和Math类都是最终类
 - String类是比较基础的类 要求所有的程序员使用的都是统一版本
 - Math类里面定义的都是数学里面的公理 定理 不能随意的修改
 - 方法：最终方法
 - 特点：可以被子类继承得到 但是**不能覆盖**
 - Sun公司不在乎我们通过继承得到最终方法 但是需要保证的是这个是最终版本 不能修改
 - 变量[属性 + 局部变量]：最终变量 常量
 - 特点：一旦赋值之后 就不能在修改值
 - final double π = 3.14;
-

abstract修饰符:

- **abstract**: 抽象的 修饰符 相当于汉文中的形容词
- 面试题: abstract能修饰哪些东西?
- 类: 抽象类 表示这个类型 **不形象 不具体**
 - 特点: **不能创建对象**
 - 面试题: 抽象类是类吗?
 - 是
 - 抽象类有构造方法吗?
 - 有, java中只要是个类就一定有构造方法
 - 抽象类是类 有构造方法 但是不能创建对象 抽象类里面的构造方法作用?
 - 给子类构造方法首行的super()使用的
- 方法: **抽象方法** 表示这个类型一定会这个方法 但是现在给不出具体的实现 待留给子类去实现
 - 一个类里面只要出现了抽象方法 那么这个类一定要变成抽象类
 - 抽象类里面既可以定义抽象方法又可以定义普通方法
- 面试题: **final**和**abstract**能不能同时修饰一个方法?
 - final修饰的方法 最终方法 可以被继承但是不能被覆盖
 - abstract修饰的方法 抽象方法 需要子类去覆盖
 - 矛盾 不能同时修饰一个方法

day9

课程内容

- java中的接口
- java中的Object类型

java中的接口:

- 接口: **interface** 相当于工业生产中的规范
- java中的第二大类型: [他们都可以在编译之后生成.class文件]
 - class interface enum @interface
 - 类 接口 枚举 注解
- 如何定义接口

```
1 interface XXX{
2     接口里面的属性默认加上三个修饰符: public static final
3     int x = 45;
4     接口里面的方法默认加上两个修饰符: public abstract
5     void test();
6 }
```

- 两两之间的关系:
 - 类和类: extends
 - 类和接口: implements
 - 接口和接口: extends
 - java中的类只能**单根继承**

- java中的接口允许多继承 多实现
- 接口和抽象类不能创建对象
- 方法覆盖的时候 可以加上@Override //注解, 检查
 - 类和类之间的方法覆盖 -》jdk5.0
 - 类和接口之间的方法覆盖 -》jdk6.0
- 面试题: 抽象类和接口之间的区别?
 - 这是java中的两大类型:
 - 抽象类: class
 - 接口: interface
 - 里面定义的属性不同
 - 抽象类: 默认都是普通属性 -》 default String x = "";
 - 接口: 默认都是静态的最终变量 -》 public static final
 - 里面定义的方法不同:
 - 抽象类: 既可以定义抽象方法也可以定义普通方法
 - 接口: 里面只能定义抽象方法 -》 public abstract
 - jdk8.0开始 接口里面可以出现普通方法了 但是必须加上static/default
 - jdk9.0开始 接口里面可以出现私有方法了
 - 继承关系不同
 - 抽象类: 只能单根继承
 - 接口: 允许多继承 多实现

Object类型: 鼻祖类 所有的类直接父类/间接父类

- clone(): "克隆"对象的方法
 - 面试题: 深克隆和浅克隆之间的区别?
 - 浅克隆仅仅克隆一个对象 里面的属性还是共享的状态
 - 深克隆将对象克隆 里面的属性也会克隆一份
- toString(): 制定一个对象打印显示的内容
 - 如果我们想要打印一个引用数据类型的对象的话 那么底层自动拿着对象.toString()
 - 如果一个类的toString()没有覆盖的话 和Object类保持一致

```

1  class Object{
2      public String toString(){
3          return getClass().getName() + "@" + 哈希码值的十六进制;
4      }
5  }
6
7  String类继承的得到toString()之后覆盖了toString()
8  字符串要求打印对象的时候显示 字符串的内容

```

- equals(): 制定一个类型的比较规则 当什么属性一样的话 把他们视为相等对象
 - 如果两个对象想要视为相等对象的话 需要比较equals()
 - equals()在没有覆盖的时候 和Object类保持一致

```

1  class Object{
2      public boolean equals(Object obj){
3          return this == obj; //比较地址
4      }
5  }
6  字符串在继承得到equals()方法的时候 方法覆盖 比较字符串的内容
7  所以我们自己写的类在没有覆盖equals()方法的时候比较地址

```

- 面试题：== 和 equals() 之间的区别？
 - ==是一个运算符 判断左右两边的值是否相等
 - 基本数据类型：比较数值
 - 引用数据类型：比较地址
 - equals():判断两个对象能不能视为相等对象 程序员可以按照自己的意愿 将内存里面不同的两个对象视为相等对象 只能引用数据类型
- hashCode():得到一个对象的散列特征码
 - 散列：将一大组数据分散为不同的小组
 - 注意：当此方法[equals]被重写时，通常有必要重写 hashCode 方法，以维护 hashCode 方法的常规协定，该协定声明相等对象必须具有相等的哈希码。
 - x.equals(y) == true
 - x.hashCode() == y.hashCode()

day10(阶段复习)

1:java的优势？

跨平台:一个软件可以在不同的操作系统上面运行

底层功臣: JVM = 类加载器 + 字节码校验器 + 解释执行器

```

1  安全 健壮
2      安全性:少指针[空指针异常]
3          运算符重载[+ & |]
4          多自动的垃圾回收[手动回收垃圾]
5      健壮:
6          代码会不会容易瘫痪
7
8  免费 开源
9
10 简单
11      语法简单:C++--
12      思想简单:面向对象的思想
13          类 -》 对象 -》 属性值
14
15 动态更新

```


2:搭建coreJava环境

官网下载jdk -> www.oracle.com

设置环境变量 -> PATH指向jdk下面的bin目录

安装开发软件

3:HelloWorld涉及到的面试题:

标识符的命名规范:

不能使用数字开头 可以用中文 英文 \$ _ 开头

不能使用java中的关键字/保留字[const/goto]

java中所有的类名都是合法的标识符 String Object

4:java中的打包语句? package

java中的导包语句? import

如果一个文件既需要打包 有需要导包? 先打包 然后再导包

5:java中的数据类型分为?

2种/无数种

基本数据类型

布尔类型:boolean

字符类型:char

整数类型:byte short int long

浮点类型:float double

引用数据类型

class

interface

enum

@interface

6:在java中2.0 - 1.1结果是多少?

2.0 - 1.1 = 0.8999999999

- 1 为什么会得到一个错误的结果?
- 2 在java中任何一个整数类型都可以用二进制精确的表达出来
- 3 但是不代表任何一个小数也可以在有限的位数里面表达出来
- 4 1.1在底层存储的时候 出现了数值位的截取 导致数字不正确
- 5 所以结果不正确
- 6
- 7 怎么解决?
- 8 数字扩大100倍 单位缩小100倍
- 9 BigDecimal

7:BigDecimal x = new BigDecimal("1.1");

参数为什么写字符串类型?

因为参数如果直接写1.1的话 需要先存储在计算机里面

1.1在存储的时候出现了数值位的截取 导致数字不正确

那么x里面存的值不正确

所以为了保护值不变 参数"1.1"

8:short x = 1;//short:-32768 32767

```
x = x + 1;//short = int
System.out.println(x);//报错

short x = 1;
x += 1;//x = (short)(x + 1);
System.out.println(x);//2
```

9:如何生成一个随机编码 由大写字母和两位数组成

```
String str=
""+(char)(Math.random() * 26 + 'A')+(int)(Math.random()*90+10)
```

10:&&和&之间的区别?

相同点:&&和&都是逻辑的与 相当于汉文中并且的意思
不同点:&&叫做逻辑的短路与 有短路特性

11:java中的变量分为哪些?

成员变量和局部变量

- 1 成员变量和局部变量之间的区别?
- 2 **a:**定义的位置不同
- 3 成员变量: 定义在类体里面
- 4 局部变量: 定义在方法体里面
- 5
- 6 **b:**作用范围不同
- 7 成员变量: 依赖于对象存在 当我们new对象的时候 底层创建属性
- 8 局部变量: 依赖于方法存在 当我们调用方法的时候 底层创建局部变量
- 9
- 10 **c:**默认值不同
- 11 成员变量: 即使不赋值 也有默认值
- 12 局部变量: 没有默认值 要求在使用之前必须先赋值
- 13
- 14 成员变量和局部变量可以重名
- 15 如果成员变量和局部变量重名的时候 直接访问这个变量 默认是局部
- 16 如果想要访问成员变量 需要在这个变量的前面加上**this.**
- 17 **this:**表示当前调用该方法的时候 不能出现在静态方法里面

12:switch case的参数可以穿哪些数据类型?

jdk1.0 char byte short int
jdk5.0 enum
jdk7.0 String

13:continue和break之间的区别?

continue:跳过本次循环 开始下一次 跳到所在循环的第三部分
break:跳出所在循环 跳到所在循环的结束部分

14: int[] data = {23,44,80,96,77};

```
//删除44
System.arraycopy(data,2,data,1,3);
data = Arrays.copyOf(data,data.length - 1);
```

15: 上面的数组升序排序

```
for(int x = 0;x < data.length -1;x++){
    for(int y = 0;y < data.length -1;y++){
        if(data[y] > data[y + 1]){
            int z = data[y];
            data[y] = data[y + 1];
            data[y + 1] = z;
        }
    }
}
```

16: 方法重载和方法覆盖之间的区别?

a: 含义不同

方法重载: Overload

方法覆盖: Override

- | | |
|----|--------------------------|
| 1 | b: 定义的位置不同 |
| 2 | 方法重载: 在同一个类体 |
| 3 | 方法覆盖: 需要发生在有继承关系的两个类中 |
| 4 | |
| 5 | c: 对参数的要求不同 |
| 6 | 方法重载: 参数必须不同[类型 个数 顺序] |
| 7 | 方法覆盖: 参数必须相同 |
| 8 | |
| 9 | d: 对返回类型的要求不同 |
| 10 | 方法重载: 没有要求 |
| 11 | 方法覆盖: 分版本 |
| 12 | jdk5.0之前 一模一样 |
| 13 | jdk5.0开始 协变返回类型 |
| 14 | |
| 15 | e: 作用不同 |
| 16 | 方法重载: 同时满足用户的不同需求 |
| 17 | 方法覆盖: 让一个方法变的更加优秀 |

17: 构造方法首行的super()/this()之间的区别?

super(): 要执行本构造方法之前 先去执行父类的构造方法

this(): 要执行本构造方法之前 先去执行本类的其他的构造方法

- | | |
|---|------------------------------|
| 1 | 出现在普通方法 |
| 2 | super.: 在子类调用父类的属性/方法 |
| 3 | this.: 调用本类的属性/方法 |

18:String类创建方式有几种？

```
String x = "OK";  
String y = new String("OK");
```

19:String x = new String("OK") 内存里面创建几个对象

2个
堆内存 + 常量池

20:String和StringBuffer/StringBuilder之间的区别？

创建方式不同

String:new和不new
StringBuffer、StringBuilder:new

```
1 能不能修改原本的字符串  
2      String:所有的方法都不会直接的处理原本的字符串  
3      StringBuffer/StringBuilder所有的方法都直接处理原本的字符串  
4  
5 有没有缓冲区  
6      StringBuffer和StringBuilder在第一次创建对象的时候 底层会  
7      多预留16块缓冲区
```

21:StringBuffer和StringBuilder之间的区别？

StringBuffer同一时间允许一个线程进行访问 效率较低 但是不会出现并发错误
StringBuilder同一时间允许多个线程进行访问 效率较高
但是可能会出现并发错误

22:String str = "swiss";//去重

//去重

```
String temp = "";  
char[] data = str.toCharArray();  
for(char x : data){  
    if(!temp.contains(x+"")){  
        temp += x;  
    }  
}  
  
//将第一个非重复元素打印出来  
char[] data = str.toCharArray();  
for(char x : data){  
    if(str.indexOf(x+"") == str.lastIndexOf(x+"")){  
        System.out.println(x);  
        break;  
    }  
}
```

23:static为什么不能修饰局部变量？

static修饰的变量要求类一加载就要在内存里面找到他
但是局部变量只有方法调用 代码执行到哪一行时才能创建
类的加载永远在前面 方法调用永远在后面
这两个时间点赶不上一起 所以static不能修饰局部变量

24:final和abstract能不能同时修饰一个方法？

不可以
final修饰的方法最终方法 不能被覆盖
abstract修饰的方法抽象方法 需要子类覆盖

25:抽象类和接口之间的区别？

抽象类：class 里面的属性都是普通属性 可以写抽象方法/普通方法
接口：interface 里面的属性都是静态的最终变量 只能写抽象方法
jdk8.0开始 接口里面可以出现普通方法 -> static / default

26:==和equals()之间的区别？

==是运算符 判断左右两边的值是否相等
基本数据类型：比较数值
引用数据类型：比较地址

- 1 equals() : 判断两个对象能不能视为相等对象
- 2 程序员可以按照自己的意愿 将内存里面不同的两个对象视为相等对象

day11(review after new year)

java的优势:

跨平台
java开发出来的软件可以运行在所有的操作系统上面
JVM = 类加载器 + 字节码校验器 + 解释执行器
安全 健壮
Object -> equals() -> 比较两个对象的地址
Student -> equals() -> 比较两个对象的姓名

```
1  学生对象.equals(null)
2  学生对象.equals(老师对象)
3
4  public boolean equals(Object obj){
5      if(obj == null)return false;
6      if(!(obj instanceof Student)) return false;
7      if(obj == this)return true;//a.equals(a)
8      return this.name.equals(((Student)obj).name);
9  }
10
11  免费 开源
12  简单
```

```
13  语法简单: C++--
14      少: 指针 运算符重载 手动的垃圾回收
15  思想简单: 面向对象的思想
16      类: 创建对象的模板    class
17      对象: 拿着类创建一个具体的实例
18      引用: 对象的名字
19      属性: 对象有什么
20      方法: 对象会什么
21
22  动态更新
```

搭建开发环境:

- 1: 安装jdk oracle
- 2: 设置环境变量
PATH:给操作系统使用
PATH=C:\Program Files\Java\jdk1.8.0_202\bin
CLASSPATH:给类加载器使用的
默认路径.
JAVA_HOME:其他的程序使用的 -》JDK安装目录
- 3: 安装开发软件

java中的软件包结构:

打包语句: package
导包语句: import

```
1  BigDecimal -> java.math    java.lang
```

```
1  50 -> 30实体类
2
3  打包的作用?
4      按照文件的功能 性质进行分类 方便查找
5      解决重名问题
```

java中的数据类型:

基本数据类型:

布尔类型:boolean -> true/false
字符类型:char -> 'a/中/'\t'
整数类型:byte short int* long* -> 45
浮点类型:float double -> 45.5 BigDecimal[引用]

```

1      转换规则：
2          小的数据类型可以直接变成大的数据类型
3          大的数据类型需要强转成小的数据类型
4
5          int num = (int)(Math.random() * 900 + 100);
6
7      引用数据类型：
8          String BigDecimal System....
9          Student Teacher....

```

java中的运算符：

+ - * / %
 +:算数加法 追加连接
 /:整数/整数 = 整数
 %:求余数

```

1      求1-100之间最大的8的倍数
2          System.out.println(100 / 8 * 8);12 * 8 = 96
3          System.out.println(100 - 100 % 8);
4
5          int max = 0;
6          for(int x = 8;x <= 100;x+=8){
7              max = x;
8          }
9          System.out.println(max);
10
11      += -= *= /= %=
12          short x = 1;//short取值范围： -32768    32767
13          x = x + 1;//short = int
14          System.out.println(x);//报错
15
16          short x = 1;
17          x += 1;
18          System.out.println(x);//2
19
20      ++ --
21          [自增自减运算符]
22          x++:先取值 再运算
23          ++x:先运算 再取值
24
25      > < >= <= ! = != ==
26          =:赋值运算符
27          ==:判断运算符
28              == -> 基本数据类型 -> 比较数值
29              == -> 引用数据类型 -> 比较地址
30              equals -> 引用数据类型 -> 比较内容
31
32      && || & |
33          &&: 并且
34          ||: 或者
35
36      & | ^ ~
37          &: 在某些情况下取代%

```

```

38         x % 2(n) == x & 2(n)-1
39
40         x % 8 -> x & 7
41
42     ^: 可以交换两个变量值:
43         可以交换的数据类型只有: boolean char byte short int long
44         int x = 45;
45         int y = 90;
46         int z = x;
47         x = y;
48         y = z;
49
50         x = x ^ y;
51         y = x ^ y;
52         x = x ^ y;
53
54         String x = "0";
55         String y = "K";
56         String z = x;
57         x = y;
58         y = z;

```

```

1  << >> >>>
2      <<相当于*      x * 2(n) == x << n
3                      x * 8 == x << 3
4      >>相当于/      x / 2(n) == x >> n
5
6  ? :
7      boolean ? 2 : 3

```

java中的变量:

成员变量 实例变量 属性:定义在类体里面

局部变量 自动变量:定义在方法体里面

```

1  int x = 45;
2
3  区别?
4  定义的位置不同
5      成员变量: 定义在类体里面
6      局部变量: 定义在方法体里面
7
8  作用范围不同
9      成员变量: 依赖于对象存在 只要对象没有消亡无论在哪里都可以访问
10     局部变量: 依赖于方法存在 代码执行到那一行的才能创建 方法执行完局部变量消亡
11
12  默认值不同
13     成员变量: 有默认值
14     局部变量: 没有默认值 要求使用之前必须先赋值
15
16     public static void main(String[] args){
17         int sum = 0;

```



```

18         for(int x = 1;x <= 100;x++){
19             sum = sum + x;
20         }
21         System.out.println(sum);
22     }

```

java中的分支:

if else

```

if(boolean){
    xxx
}else if(boolean){
    yyy
}else if(boolean){
    zzz
}else{
    zzz
}

```

```

1  switch case
2      switch(参数){
3          case XXX : .....;break;
4          case YYY : .....;break;
5          default : .....;
6      }
7
8      参数的类型:
9          jdk1.0    char byte short int
10         jdk5.0    enum
11         jdk7.0    String

```

java中的循环:

for

while

do while

```

1  循环控制:
2      循环嵌套: 一个循环定义在另一个循环里面
3      a:for(1-10){
4          for(1-10){
5              continue/break a;
6          }
7      }
8
9      循环控制:
10     continue: 跳过本次循环 开始下一次 跳到所在循环的第三部分
11     break: 跳出所在循环 跳到所在循环的结束部分
12
13     循环标签:
14     当多个循环嵌套在一起 代码写在最里面循环 想要处理外层循环
15     需要给外层循环贴标签 然后再内层循环里面continue/break + 名

```

java中的数组：

创建对象：

```
int[] data = new int[5];  
int[] data = new int[]{45,66,92,10};  
int[] data = {23,445,'a',67,90};
```

```
1  得到数组里面的某一个元素：  
2      data[下标]  
3      下标： 0 到 数组总长度-1  
4  
5  得到数组大小：  
6      data.length  
7  
8  如何遍历数组：  
9      for + 下标  
10     for(int x = 0;x < data.length;x++){  
11         System.out.println(data[x]);  
12     }  
13  
14     foreach  
15     for(数组类型 x : data){  
16         System.out.println(x);  
17     }
```

数组复制：

```
System.arraycopy(老数组,下标,新数组,下标,长度);  
新数组 = Arrays.copyOf(老数组,新数组长度)
```

数组排序：

冒泡排序：

```
for(int x = 0;x < data.length -1;x++){  
    for(int y = 0;y < data.length -1-x;y++){  
        if(data[y] > data[y + 1]){  
            int z = data[y];  
            data[y] = data[y + 1];  
            data[y + 1] = z;  
        }  
    }  
}
```

```
1  自动排序：  
2      Arrays.sort(data);
```

封装：

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

```

1 public String getName(){
2     return name;
3 }

```

继承: extends

```
class Animal{...}
```

```
class Cat extends Animal{}
```

java中的类只能单根继承 一个类只能继承一个类

一个类实现多个接口

多态:

创建对象: Person x = new Student();

放在参数里面用于解除代码之间的耦合度:

```
class Object{
```

```

1         public boolean equals(Object obj){
2             return this == obj;
3         }
4     }

```

方法重载 Overload

条件:

需要发生在同一个类体中

方法名需要一模一样

方法的参数需要不同[类型 个数 顺序]

```

1 作用:
2     同时满足用户的不同需求
3
4 例子:
5     System.out.println();
6     System.out.println(45);
7     System.out.println(45.5);
8     System.out.println("String");
9
10    String:
11    substring(int x)
12    substring(int x,int y)

```

方法覆盖 Override

条件:

子类继承得到父类的方法觉得父类的实现不好 于是在子类里面重新实现一下

```

1 修饰符 >= 父类的修饰符
2 返回类型
3     jdk5.0之前 一模一样
4     jdk5.0开始 斜变返回类型
5 方法签名 一模一样
6 异常 <= 父类的异常

```

构造方法：

创建对象的同时直接给属性赋值

```
1  class Student{
2      String name;
3      int age;
4      public Student(String name,int age){
5          this.name = name;
6          this.age = age;
7      }
8  }
9
10  首行：
11  super():要执行本构造方法之前 先去执行父类的构造方法
12      默认找父类的无参构造方法
13      如果父类没有无参构造方法：
14          1: 提供父类的无参构造方法
15          2: 在super()里面传参数 指定他找父类的哪一个构造方法
16  this():要执行本构造方法之前 先去执行本类的其他的构造方法
17      具体执行本类的哪一个构造方法看参数类型
```

参数传递：

基本数据类型传值

引用数据类型传地址

String类：

String x = new String("OK");内存里面创建几个对象？

2个 堆内存 + 栈内存

```
1  String类创建方式有几种？
2      2种  String x = ""; String y = new String("");
3
4  String和StringBuffer/StringBuilder之间的区别？
5      StringBuffer/StringBuilder在创建对象的时候 底层多预留16块缓冲区
6      目的为追加连接效率高
7      StringBuffer/StringBuilder创建对象的方式只有一种 -> new
8      StringBuffer/StringBuilder所有的方法都可以直接的处理原本的字符串
9      buffer.append("xxx");
10
11  StringBuffer和StringBuilder之间的区别？
12      StringBuffer同一时间允许一个线程进行访问 效率较低 但是不会出现并发错误
```

String类常用的方法：

indexOf(String) -> int

length() -> int

equals(String) -> boolean

equalsIgnoreCase(String) -> boolean

toCharArray() -> char[]

charAt(int) -> char

replace(String,String) -> boolean

substring(int x) -> String

split(String) -> String[]
replaceAll(String,String) -> String
toUpperCase() -> String
endsWith(String) -> boolean
startsWith(String) -> boolean
getBytes() -> byte[]
replaceFirst(String,String) -> String
lastIndexOf(String) -> int
trim() -> String
substring(int x,int y) -> String
toLowerCase() -> String
contains(String) -> boolean
intern() -> 得到字符串常量池的地址

StringBuffer:

append():追加连接
reverse():翻转字符串

修饰符:

访问权限修饰符:

public > protected > default > private

类成员 成员 类成员 成员

```
1  static : 静态的
2      属性:整个类型共享一份属性
3
4      方法:只能直接的访问静态的属性 如果想要访问非静态的属性 需要创建对象 拿着对象去调用
5
6      代码块:
7      {}:当创建对象的时候执行 创建几个对象执行几次
8      static {}:静态代码块 当类第一次被加载的时候执行 从头到尾执行一次
9
10     static为什么不能修饰局部变量?
11         static修饰的变量要求类一加载就要在内存里面找到他
12         但是局部变量只有方法调用 代码执行到那一行的时候才能创建
13         类的加载在前面 方法调用在后面 所以static不能修饰局部变量
14
15     abstract : 抽象的
16         类:抽象类 不能创建对象
17         方法:抽象方法 不能有方法体
18
19     final : 最终的
20         类:最终类 不能有子类 但是可以有父类
21         方法:最终方法 不能被覆盖 但是可以被继承
22         变量:最终变量 一旦赋值之后不能在修改值
23
24     单例模式:
25
26     class Sun{
27         private Sun(){
28         private static Sun only = new Sun();
29         public static Sun getOnly(){
30             return only;
```

```
31     }
32 }
```

接口: interface

```
interface XXX{
    int x = 45;//public static final
    void test();//public abstract
    jdk8.0开始 接口里面可以出现普通方法 -》 static/default
    jdk9.0开始 接口里面可以出现私有方法
}
```

```
1  class YYY implements XXX{
2      @Override
3      public void test(){
4          ....
5      }
6  }
```

Object类:

toString():制定一个对象打印显示的内容

```
System.out.println(stu); stu.toString()
Object -> toString()
public String toString(){
    return 类名@XXX;
}
```

```
1  equals():判断两个对象能不能视为相等对象[逻辑相等]
2      Object-》equals()
3      public boolean equals(Object obj){
4          return this == obj;//比较地址
5      }
6
7  hashCode():生成对象的哈希码值 HashSet / HashMap
8      Object -> hashCode()
9      public int hashCode(){
10         return 地址;
11     }
```

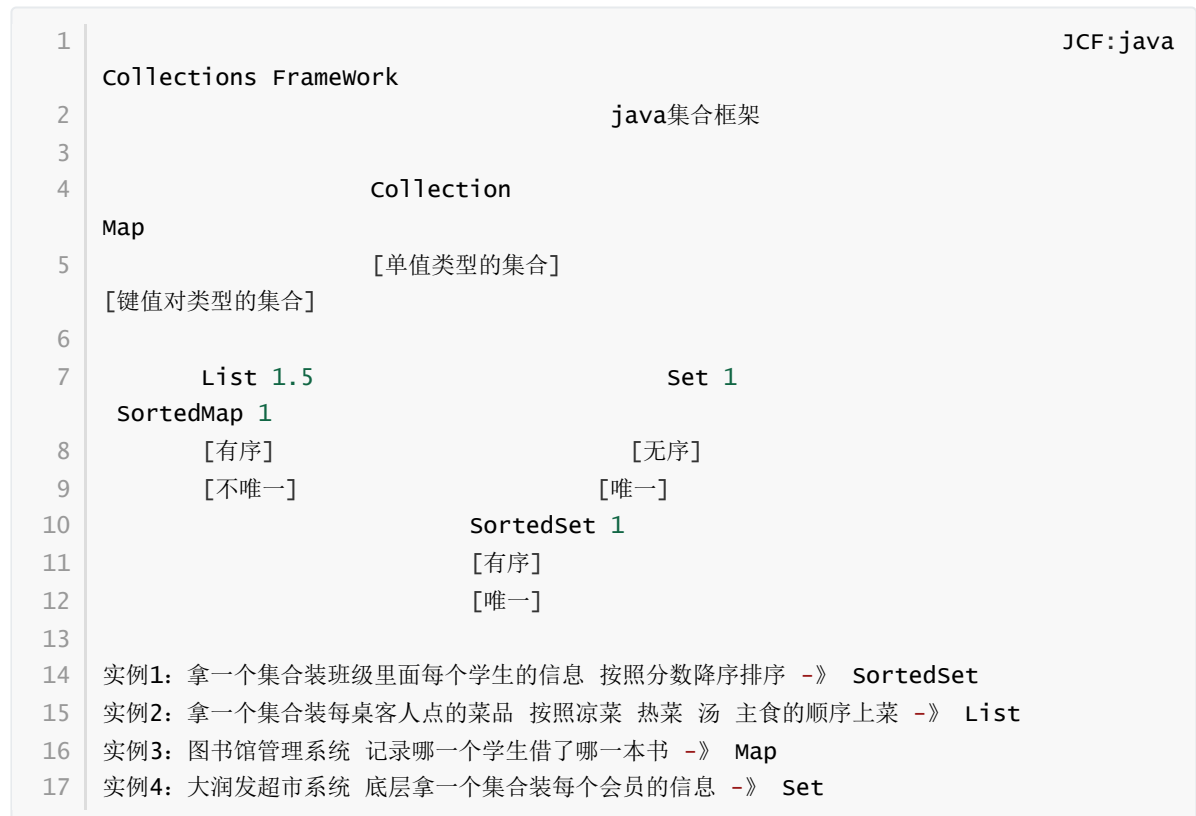
day12

课程内容

- 集合之ArrayList集合

集合：容器 装类型不同的元素 没有个数限制

数组：容器 装类型必须一样元素 有个数限制



什么是单值类型的集合？

- 每次往集合里面添加一个元素
- Word文档 -》 单词校验功能 apple ae
- Word文档软件 -》 集合 -》 装大量的常见的英语单词 apple animal cat dog...

什么是键值对类型的集合？

- 每次往集合里面添加一对 key[主键]=value[值]
- 金山词霸 -》 中英文翻译 -》 apple -> 苹果
- 金山词霸 -》 集合 -》 apple=苹果 cat=猫

List集合：接口 里面定义的都是规范

- ArrayList
 - 特点：有序 不唯一
 - 数据结构：数组
 - String -> char[] String str = "etoak"; //e t o a k -> [] -> str
 - ArrayList -> Object[] ArrayList -> a b c -> a b c -> [] -> list
- ArrayList0:包装类
 - 作用一：由于ArrayList底层基于Object[]实现的 所以只能装引用数据类型 为了保证集合可以装基本数据类型 所以需要将基本数据类型转换成包装类 装进集合里面
 - int x = 45;
 - Integer y = x; //自动打包 封箱
 - Integer a = new Integer(77);
 - int b = a; //自动解包 拆箱

- 作用二：包装类里面提供一个方法 将字符串**转换**成对应的基本数据类型
 - String str = "45";
 - int x = Integer.parseInt(str);
- ArrayList1：基本用法及特点：
 - 特点：有序[添加顺序] 不唯一
 - 创建对象
 - ArrayList list = new ArrayList();//5.0之前 默认往集合里面装的都是Object类型的对象
 - ArrayList<泛型> list = new ArrayList<泛型>();5.0开始 可以加泛型
 - ArrayList<泛型> list = new ArrayList<>();7.0开始 后面的泛型自动推断
 - 添加元素：
 - list.add(元素);一次添加一个元素
 - Collections.addAll(集合,元素,元素...);一次添加多个元素
 - 面试题：Collection和Collections之间的区别？
 - Collection是所有单值集合统一的父接口：interface
 - Collections是集合的工具类：class
 - list.set(下标,元素);修改某个下标对应的元素
 - 得到集合大小
 - list.size()
 - 得到某一个元素
 - list.get(下标)
 - 判断集合里面是否包含某个元素：
 - list.contains(Object 元素)
 - 遍历集合对象
 - for + 下标
 - foreach
 - 迭代器
- ArrayList2:如何删除元素：
 - list.remove(int 下标)
 - list.remove(Object 元素)
 - 一个remove()只能删除一个元素
- ArrayList3:
 - ArrayList类里面指定元素删除的方法 能不能删除成功 底层尊重**equals()**比较原理
 - ArrayList类里面判断元素是否存在的方法 底层尊重**equals()**比较元素
- ArrayList4:
 - 谁主张谁举证
 - 要被删除的对象会主动调用他自己类的equals()和集合里面的每个元素作比较
- ArrayList5:
 - 当我们使用迭代器在遍历集合的时候 **不允许对集合的整体进行添加/删除操作** 否则触发**CME异常**
 - CME = ConcurrentModificationException = 并发修改异常
 - 如果需求想要我们一边遍历 一边删除的话 使用迭代器自己的删除方法：car.remove()
 - foreach底层基于**迭代器**实现的

CME检测原理

- 1 - 每一个集合底层都有一个变量 叫`modCount`记录对集合的处理次数
- 2 - 当我们通过集合获得迭代器的时候 底层会将`modCount`的值拷贝到迭代器自己的属性里面
- 3 - 迭代器自己的属性: `expectedModCount`
- 4 - 每次`car.next()`触发`modCount`和`expectedModCount`的值的比较
- 5 - 如果两个值不同步 那么出现CME异常

- ArrayList6:构造方法

- ArrayList类里面常用的构造方法
- `ArrayList list = new ArrayList(int 数组容量);`数组空间 ≥ 0
- `ArrayList list = new ArrayList();`//默认10块空间
- 无论底层开辟多大空间 都可以存储无数个元素 集合会**自动扩容**
- 项目开发的时候尽量避免扩容:
 - 创建新数组对象
 - 将老数组里面的元素复制到新数组里面
 - 改变引用指向
 - 回收老数组对象
 - 继续添加元素
- 扩容: `list.ensureCapacity(int 容量)`: 将集合大小直接扩大为XXX块空间
- 缩容: `list.trimToSize()`:将集合大小**缩小为元素个数**

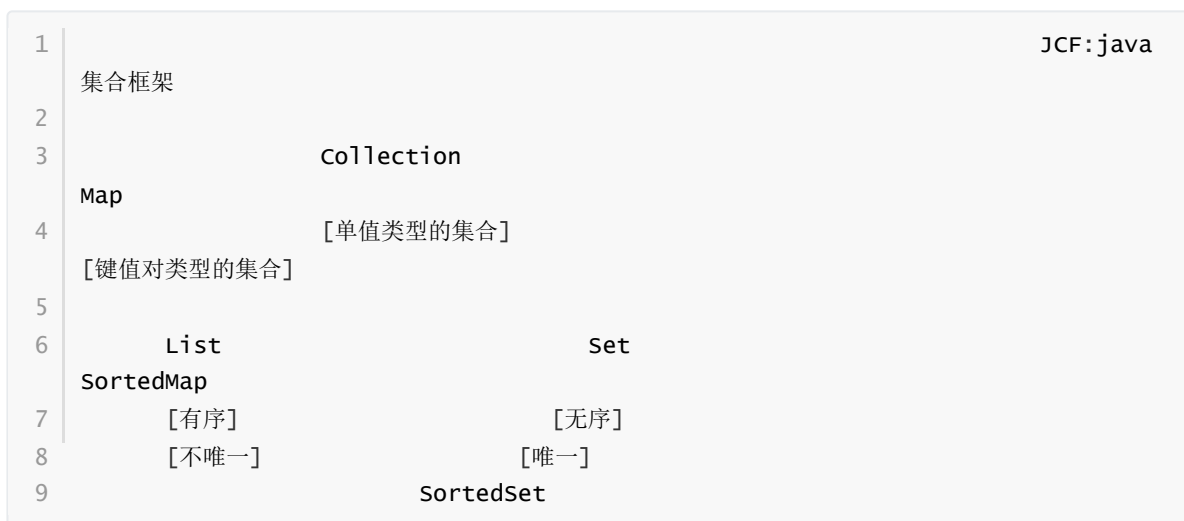
- Vector
- LinkedList
- Stack

Day13 (通讯录项目练习)

day14

课程内容

HashSet集合



Set接口 里面都是抽象方法

- **HashSet**

- 特点：无序 唯一
- 数据结构：**哈希表**实现的


- **HashSet1:基本用法及特点**

- 创建对象：
 - `HashSet<泛型> set = new HashSet<>();`
- 添加元素：
 - `set.add(元素);`
 - `Collections.addAll(集合,元素...);`
- 得到集合大小：
 - `set.size()`
- 判断集合里面是否存在某个元素：
 - `set.contains(元素)`
- 删除元素：
 - `set.remove(元素);`
- 遍历集合：
 - `foreach + 迭代器`

- **HashSet2:即使不是内存里面的同一个对象也有可能被视为相等对象 从而只能往集合里面添加一次**

- 当我们想要将两个不同的对象视为相等对象的话 HashSet集合里面需要覆盖 **`hashCode()+equals()`**
- `hashCode()`:计算哈希码值 可以决定一个对象去到哪一个小组
- `equals()`:当一个对象去到某个小组之后 发现这个小组里面有一个元素的哈希码值和新来元素的哈希码值一样 那么就拿着`equals()`详细比较

- **HashSet3:HashSet添加元素的时候 其实完整流程：**

- `hashCode() == equals()`
- 如图：
- 

- **HashSet4: 当HashSet遇到重复元素的时候 舍弃的是新来的元素**
 - HashSet底层的**add(元素) remove(元素) contains(元素)**底层尊重**hashCode() == equals()**
- **HashSet5:当我们使用迭代器在遍历集合的时候 不要对集合的整体进行添加/删除操作 否则触发CME异常**
 - 如果在遍历的时候想要删除的话 `car.remove()`
- **HashSet6:**
 - 当一个对象已经添加进**HashSet集合**之后 不要随意的修改那些参与生成哈希码值的属性值
 - 如果在添加之后想要修改的话 需要**删除+修改+添加**
 - 如果这个属性没有参与生成哈希码值 **直接修改**
- **HashSet7:**
 - `HashSet set = new HashSet(int 分组组数,float 加载因子);`
 - 分组组数程序员可以随意的指定 但是最终一定会变成**2的n次方数**
 - `HashSet set = new HashSet();`
 - 默认分16个小组 加载因子0.75F 阈值: 12

day15

课程内容

TreeSet集合



SortedSet接口:

- **TreeSet**
 - 特点: 有序 唯一
 - 数据结构: **二叉树**
- **TreeSet1:基本用法及特点:**
 - 创建对象: `TreeSet<泛型> set = new TreeSet<>();`
 - 添加元素: `set.add(元素); Collections.addAll(集合,元素,元素...);`
 - 得到集合大小: `set.size()`

- 判断集合包含: `set.contains(元素)`
 - 删除元素: `set.remove(元素)`
 - 得到第一个元素: `set.first()`
 - 得到第一个元素并且删除: `set.pollFirst()`
 - 得到最后一个元素: `set.last()`
 - 得到最后一个元素并且删除: `set.pollLast()`
- TreeSet2:任何一个引用数据类型如果想要装进TreeSet集合的话 都需要这个类型会排序 有比较规则
 - `class 类型 implements Comparable<类型>{}`
 - `public int compareTo(参数 x){}`
- TreeSet3:如果我们想要按照多个属性综合排序的话 优先比较什么属性 就描述当这个属性不一样的话...
- TreeSet4:当我们在覆盖`compareTo()`尽量保证方法都会返回0 否则
 - `add(元素)`: 不能保证唯一性
 - `remove(元素)`: 永远删除失败
 - `contains(元素)`: 永远返回false
 - 如果`compareTo()`在没有机会返回0的时候 想要删除的话: `car.remove()`
- TreeSet5:当我们使用迭代器在遍历集合的时候 不允许对集合的整体进行添加/删除 否则触发CME异常
 - 如果需求要求一边遍历 一边删除的话: `car.remove()`
- TreeSet6:当一个对象已经被装进TreeSet集合的话 不要随意的修改那些参与排序的属性值
 - 如果在添加之后想要修改的话: 删除 + 修改 + 添加
- TreeSet7:
 - 第一个添加进TreeSet集合里面的元素在某个时间段内是跟节点 随着元素的增多 根节点可能会进行改变 底层会进行旋转修复 目的是为了平衡效率高

TreeSet8(比较器):

- 如果我们想要脱离一个类制定他的比较规则的话 需要写比较器
- `class XXX implements Comparator<>{}`

单例模式:

```
1 private BJQ() {}
2 private static BJQ bb = new BJQ();
3 public static BJQ getBB() {
4     return bb;
5 }
```

@Override

- `public int compare(参数1,参数2){}`

day16

课程内容

- Collections类里面常用的方法

- 单值集合总复习

Collections:单值集合的工具类

- Collections类里面专门用来处理List集合的方法：
 - Collections.sort(List集合): 自然排序 按照泛型类的排序规则对List集合里面的元素进行排序
 - Collections.sort(List集合,比较器): 定制排序 按照比较器的规则对List集合里面的元素进行排序
 - Collections.reverse(List集合): 反转List集合里面的元素
 - Collections.shuffle(List集合): 打乱List集合里面的顺序
- Collections类里面对所有单值集合都提供的方法：
 - Collections.addAll(集合,元素,元素....);
 - Collections.max(单值集合): 按照泛型类的规则排序找队列里面最后一个元素
 - Collections.max(单值集合,比较器): 按照比较器的规则排序找队列里面的最后一个元素
 - Collections.min(单值集合): 按照泛型类的规则排序找队列里面第一个元素
 - Collections.min(单值集合,比较器): 按照比较器的规则排序找队列里面的第一个元素

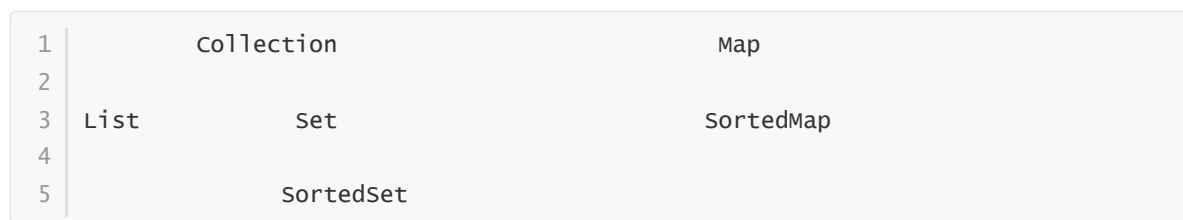
=====

===

面试题 (List集合)

1:请画出集合框架

JCF:java集合框架



2:Collections和Collection之间的区别

Collection是**所有单值集合的父接口** -》 interface

Collections是**集合的工具类**： -》 class

3:模拟实现Collections.addAll(集合,"","",""...");

```

1 public static void addAll(ArrayList<String> jihe,String ... data){
2     for(String x : data){
3         System.out.println(x);
4     }
5 }
```

4:模拟实现ArrayList集合的contains()底层实现

```

1 public boolean contains(Object obj){
2     for(int x = 0;x < 集合.size();x++){
3         if(obj.equals(data[x])){
4             return true;
5         }
6     }
7     return false;
8 }

```

5:ArrayList和LinkedList之间的区别?

```

1 ArrayList底层基于数组实现的
2     优点:查找遍历效率高
3     缺点:添加 删除元素效率低

```

```

1 LinkedList底层基于链表实现的
2     优点:添加 删除元素效率高
3     缺点:查找 遍历效率低

```

6:ArrayList和Vector之间的区别?

a:同步特性不同

ArrayList同时允许多个线程进行访问 效率高 但是可能会有并发错误

Vector同时允许一个线程进行访问 效率低 但是不会出现并发错误

b:扩容机制不同

ArrayList分版本

jdk6.0及之前 $x * 3 / 2 + 1$

jdk7.0及之后 $x + (x >> 1)$

Vector分构造方法

Vector(int) -> 2倍扩容

Vector(int,int) -> 定长扩容

c:出现的版本不同

ArrayList since jdk1.2

Vector since jdk1.0

7:考虑到ArrayList开发的时候可能会出现并发错误 那么有哪些类可以取代?

Vector

List list = Collections.synchronizedList(ArrayList对象)

8:请写出ArrayList的add()方法底层的流程

当我们想要往ArrayList集合里面添加元素的时候 **直接**将新来的元素放在**集合的最后一块空间**

请写出ArrayList的remove(元素)底层的流程

当我们想要指定一个元素进行删除的时候 底层拿着被删除的元素和集合里面的每一个元素做**equals()**

请写出ArrayList的contains(元素)底层的流程

当我们想要判断ArrayList集合里面是否包含元素的时候

底层拿着被包含的元素和集合里面的每一个元素做**equals()**

请写出HashSet的add(元素)底层的流程

当我们想要往HashSet集合添加元素的时候 底层拿着新来的元素
和老元素做`hashCode() == equals()`比较

请写出HashSet的remove(元素)底层的流程

当我们想要从HashSet集合里面删除元素的时候 底层拿着被删除的元素
和集合里面的每个元素做`hashCode() == equals()`比较

请写出HashSet的contains(元素)底层的流程

当我们想要判断HashSet集合里面是否存在某个元素的时候 底层拿着被包含的元素和集合里面的每一个元素做`hashCode() == equals()`

请写出TreeSet的add(元素)底层的流程

当我们想要往TreeSet集合里面添加元素的时候 底层拿着新来元素和老元素
做`compareTo()/compare()`比较

请写出TreeSet的remove(元素)底层的流程

当我们想要从TreeSet集合里面删除元素的时候 底层拿着被删除的元素和老元素做
`compareTo()/compare()`比较

请写出TreeSet的contains(元素)底层的流程

当我们想要判断TreeSet集合里面是否存在某个元素的时候 底层拿着被包含的元素和老元素做
`compareTo()/compare()`比较

9:学过的单值集合有哪些 他们的底层都是基于什么数据结构实现的

ArrayList 数组
LinkedList 链表
Vector 数组
Stack 栈
HashSet 哈希表
LinkedHashSet 链表+哈希表
TreeSet 二叉树

10:简述CME的原理

- 每一个**集合底层**都有一个变量 叫**modCount**记录对集合的处理次数
- 当我们通过集合获得迭代器的时候 底层会将modCount的值**拷贝到迭代器自己的属性里面**
- 迭代器自己的属性: **expectedModCount**
- 每次**car.next()**触发modCount和expectedModCount的值的比较
- **如果两个值不同步** 那么出现CME异常

11:分别写出ArrayList HashSet TreeSet构造方法的参数分别可以传什么

- ArrayList:
 - 可以接收一个 **Collection 类型的对象**, 来初始化 ArrayList, 这样 ArrayList 中的元素就是 Collection 中的元素的副本。
 - 也可以接收一个 **int 类型的整数**, 表示 ArrayList 的初始容量。
- HashSet:
 - 可以接收一个 Collection 类型的对象, 来初始化 HashSet, 这样 HashSet 中的元素就是 Collection 中的元素的副本。
- TreeSet:

- 可以接收一个 Collection 类型的对象，来初始化 TreeSet，这样 TreeSet 中的元素就是 Collection 中的元素的副本。
- 也可以接收一个 **Comparator 对象**，用来排序 TreeSet 中的元素。

需要注意的是，无论是 ArrayList，HashSet 还是 TreeSet，都有一个**默认**的**无参构造方法**，用来创建空的集合。

day17

课程内容

- Collection集合复习
- Map集合

Collections工具类复习：

- 对List集合提供服务的方法
 - Collections.**sort(List对象)**:自然排序 按照List集合里面的泛型规则排序
 - Collections.**sort(List对象,比较器)**:定制排序 按照比较器规则排序
 - Collections.**shuffle(List集合)**:打乱集合顺序
 - Collections.**reverse(List集合)**:翻转集合里面的元素
- 对所有单值集合服务的方法：
 - Collections.**addAll(集合,元素,元素,元素....)**;
 - Collections.**max(集合)**:按照泛型类排序得到集合里面的最大值**[最后]**
 - Collections.**max(集合,比较器)**: 按照比较器排序找最大值
 - Collections.**min(集合)**:按照泛型类排序找最小值**[第一个元素]**
 - Collections.**min(集合,比较器)**:按照比较器排序找最小值

1	JCF: java集合框架	
2		
3	Collection	Map
4	[单值类型的集合]	[键值对类型的集合]
5		
6	List	Set
7	SortedMap	
8	[有序]	[无序]
9	[不唯一]	[唯一]: hashCode==equals()
10		SortedSet
11		[有序] compareTo()/compare() [唯一] compareTo()/compare()

Map:键值对类型的集合 每次往集合里面添加两个元素 Key[主键]=Value[值]

- 主键要求唯一
- HashMap
- TreeMap

Map1:所有Map集合通用的方法

- 创建对象: `HashMap<主键,值> map = new HashMap<>();`
- 添加元素: `map.put(主键,值);`
- 合并集合: `map1.putAll(map2)`
- 得到集合大小: `map.size()`
- 通过主键得到值对象: `map.get(主键)`
- 判断主键是否存在: `map.containsKey(主键)`
- 判断值是否存在: `map.containsValue(值)`
- 删除元素: `map.remove(主键)`

Map集合通用的遍历方式:

- 通过Map集合得到所有的主键视图
 - `Set<主键> set = map.keySet();`
 - 遍历set集合得到所有的主键
 - 通过主键得到值对象: `map.get(主键)`
- 通过Map集合得到所有的值视图
 - `Collection<值> cs = map.values();`
 - 遍历cs集合得到所有的值对象
- 通过Map集合得到所有的记录[主键+值]
 - `Set<Map.Entry<主键,值>> set = map.entrySet();`
 - 遍历set集合得到每一条记录: `x[主键+值]`
 - 通过记录x得到主键: `x.getKey()`
 - 通过记录x得到值: `x.getValue()`
 - 通过记录修改值: `x.setValue()`

HashMap: (原理)

底层基于**哈希表**实现的 **默认分16个小组** 每对元素[主键+值]应该添加进哪一个小组里面 **根据主键的哈希码值决定** 哈希码值 **% 分组组数**看余数 如果来到某个小组之后发现这个小组里面有一个元素的主键的哈希码值和新来对象主键的哈希码值一样 需要拿着**`== equals()`**继续比较

- HashMap底层的
 - `put(主键,值)`
 - `get(主键)`
 - `remove(主键)`
 - `containsKey(主键)`
 - 底层尊重**`hashCode() == equals()`**比较机制
- Map当主键遇到重复元素的时候 **主键不变 值替换**

面试题：HashMap和Hashtable之间的区别？

- 同步特性不同
 - Hashtable同一时间允许一个线程进行访问 **效率较低** 但是**不会出现并发方法错误**
 - HashMap同一时间允许多个线程进行访问 **效率较高** 但是**可能会出现并发方法错误**
 - jdk5.0开始 集合的工具类里面提供一个方法 将线程不安全的HashMap变成线程安全的Map集合 所以Hashtable渐渐被淘汰了
 - Map<> map = Collections.synchronizedMap(HashMap对象);
- 对null的处理不同
 - Hashtable无论是主键还是值对象 **都不能传null** 一旦输入null触发空指针异常
 - HashMap无论是主键还是值对象**都可以传null** 但是由于**主键是唯一的** 所以**只能传一个null**
- 分组组数不同
 - Hashtable默认分**11个小组** 程序员可以随意的指定
 - HashMap默认分**16个小组** 程序员可以随意的指定 但是最终**一定会变成2的N次方数**（为了计算机处理效率高）
- 出现的版本不同
 - Hashtable since jdk1.0
 - HashMap since jdk1.2

面试题：HashMap Hashtable ConcurrentHashMap之间的区别？

-Hashtable底层所有的方法都加上synchronized 只要有一个线程进入哈希表会将这个哈希表全部加锁 所以效率很低 但是安全

- HashMap底层为了追求高效性 所以**不加锁** 可能会有多个线程同时访问一个小组的风险

- ConcurrentHashMap为了追求高效性和安全性 将锁的力度降低 当一个线程进入哈希表里面某个小组之后 **仅仅是对这一个小组进行加锁** 如果新来的线程也想要访问该小组 需要等待 如果新来的线程想要访问没有加锁的小组 直接访问

面试题：HashMap在高并发的情况下 不安全 有哪些方法可以取代？

- Hashtable
- ConcurrentHashMap
- Map<> map = Collections.synchronizedMap(HashMap对象)

面试题：如何计算每个元素应该放进HashMap/HashSet哪一个小组？

- 通过key.hashCode()得到主键的哈希码值：h
- 通过调用哈希算法进行进一步处理： $h = h \wedge (h \ggg 16)$ 进行高位16运算
- 通过 $h \% 16$ 看余数 余数为几就去哪一个小组
- 为什么将hashCode右移16位再进行异或运算？
 - 可以将hashCode()的高位和低位的值进行混合做异或运算 这样低位的信息加上高位的信息也就是计算小组的时候将高位16的值也参与进来了 参与进来的元素越多 重码的概率越低
- 为什么HashMap/HashSet分组最终一定会变成2的n次方数？
 - 当长度是2(n)的时候 $hash \% 分组组数$ 等价于 $hash \& (分组组数-1)$ 效率更高

- **HashMap在jdk7.0前后底层变化?**

- HashMap底层基于哈希表实现的
- jdk7.0之前 哈希表底层基于数组 + 链表组成
 - 数组: 装表头信息 可以快速的为每个元素定位应该去到哪一个小组
 - 链表: 组内用链表结构存储
 - 为了添加/删除元素效率高
 - 采用头插法
- jdk8.0及之后 哈希表底层基于数组 + 链表 + 二叉树组成
 - 数组: 装表头信息 可以快速的为每个元素定位应该去到哪一个小组
 - 链表: 组内用链表存储 方便添加/删除元素
 - 当**小组链表长度** > 8 && **数组长度** > 64 由**链表**变成**二叉树**
 - 当**小组二叉树**个数 < 6 二叉树变成**链表**
 - 采用尾插法

TreeMap (原理)

底层基于**二叉树**实现的 每对元素[主键+值]应该被放进左子树/右子树 底层拿着**新来元素主键**和**老元素主键**比较compareTo()/compare()

[> 0]新来的一对元素放在**右子树**

[= 0]主键不变 **值替换**

[< 0]新来的一对元素放在**左子树**

TreeMap集合底层的:

```
put(主键,值)
get(主键)
remove(主键)
containsKey(主键)
```

底层尊重compareTo()/compare()比较

day18

课程内容

- jdk8.0新特性:
 - 接口
 - Optional类型
 - Lambda表达式
 - Stream流
-

jdk8.0新特性接口:

- jdk8.0之前 接口里面只能出现抽象方法
- jdk8.0开始 接口里面可以出现普通方法 --> 必须加上static / default
- jdk9.0开始 接口里面可以出现私有方法

```
1 //jdk8.0之前 接口里面只能出现抽象方法
2 interface Collection{
3     boolean add(Object obj); //默认添加abstract/default
4     boolean remove(Object obj);
5     boolean contains(Object obj);
6     public static void forEach(){
7         dy();
8         dy();
9         dy();
10        dy();
11    }
12
13    private static void dy(String str){
14        System.out.println(str);
15    }
16 }
17
18 class ArrayList implements Collection{
19     @Override
20     public boolean add(Object obj){
21         ...
22     }
23     @Override
24     public boolean remove(Object obj){
25         ...
26     }
27     @Override
28     public boolean contains(Object obj){
29         ...
30     }
31 }
```

Optional类型: jdk8.0出现的新类 专门用来解决空指针异常

- 如何将一个对象放进Optional容器类里面
 - String str = ...;
 - Optional<泛型> op = **Optional.of(str)**;
 - //如果参数为null直接of()出现空指针异常
 - Optional<泛型> op = **Optional.ofNullable(str)**;
 - 参数为null的话 容器里面什么都没有
- 如何判断容器里面是否存在值:
 - op.**isPresent()** -> boolean
- 如何从容器里面取值:
 - op.**get()** --> 泛型
- 如果容器里面有值 那么直接使用容器里面的值 如果容器里面没有值 那么使用默认值

- 对象类型 `x = Optional.ofNullable(对象).orElse(默认值)`

Lambda表达式: jdk8.0出现的新语法 可以让代码变得更加的简单

- jdk8.0开始出现了一个新的操作符 -> 箭头操作符
 - 将整个语句分为两个部分
 - **左侧**: 要覆盖方法的**参数列表**
 - **右侧**: **lambda体** 具体的**执行步骤**
- Lambda表达式出现的格式:
 - `() -> void`
 - `(参数) -> void`
 - `(参数) -> boolean`
 - `(参数) -> 类型`
- lambda表示式需要注意:
 - 左侧如果参数列表只有一个参数的话 **()可以省略** 但是如果参数列表**无参 多参** 那么**()不能省略**
 - **左侧参数列表里面的数据类型可以省略的** jdk7.0开始 类型可以**自动推断**
 - 右侧lambda体里面如果只有一个语句的话 那么**return和{}可以省略** 如果 **多个语句的话 return和{}不能省略**
- Lambda表达式使用前提是调用的方法**参数是函数式接口**
 - 函数式接口: 接口里面最多只能有一个**抽象方法**
- 四大函数式接口: jdk8.0新增接口
 - **Consumer[消费型接口]**
 - 抽象方法: **accept(参数) : void**
 - **Predicate[断言型接口]**
 - 抽象方法: **test(参数) : boolean**
- jdk8.0开始 所有的单值集合新增
 - **foreach((x) -> void)**: 遍历单值集合里面所有的元素
 - **forEach((k,v) -> void)**: 遍历键值对集合里面所有的元素
 - **removeIf((x) -> boolean)**: 删除集合里面符合条件的元素
- 方法调用: 语法糖
 - 对象::普通方法
 - 类名::静态方法
 - 类名::普通方法

day19

课程内容

- Optional类型
- Lambda表达式
- Stream流
- 时间类

- Date jdk1.0
- Calendar jdk1.2
- LocalDateTime jdk8.0
 - LocalDate:当前的日期 [年月日]
 - LocalTime:当前的时间[时分秒]

```
1 Optional类:
2   将一个对象装进Optional容器里面
3   Optional<泛型> op = Optional.of(对象);
4   Optional<泛型> op = Optional.ofNullable(对象);
5
6   判断容器里面是否存在值:
7   op.isPresent() -> boolean
8
9   从容器里面取值:
10  op.get()
11  如果容器里面有值 那么直接将值取出来
12  如果容器里面没有值 会出现NoSuchElementException
13
14  如果容器里面有值 那么使用容器里面的值 如果容器里面没有值 那么使用默认值
15  Optional.ofNullable(对象).orElse(默认值);
```

```
1 Lambda表达式:    (参数类型) -> lambda体
2  如果想要使用Lambda表达式 想以函数式接口为前提
3  函数式接口: 接口里面只有一个抽象方法
4
5  注意:
6  如果参数列表里面只有一个参数的时候 ()可以不写 如果无参/多参 ()不能省略
7  参数列表里面的参数类型可以省略的 jdk7.0开始 类型可以自动推断
8  lambda体里面如果只有一个语句的时候 return {}可以省略
9  lambda体里面有多条语句的话 return {}不能省略
10
11  四大函数式接口:
12  Consumer -> 消费型接口
13      抽象方法: accept(x) -> void
14  Predicate -> 断言型接口
15      抽象方法: test(x) -> boolean
16  Function -> 函数型接口
17      抽象方法: apply(T) -> R
18  Supplier -> 供给型接口
19      抽象方法: get() -> T
20
21  方法调用: 对Lambda简写
22  对象::普通方法
23      覆盖: accept(x) -> void
24      调用: println(x) -> void
25      forEach(System.out::println);
26
27  类名::静态方法
28
29  类名::普通方法
30  必须满足公式:
```

```
31 我们要覆盖的方法的参数列表比我们调用方法的参数列表多1个
32 且将要覆盖的方法的第一个参数拿出来 应该是调用方法的对象
33 覆盖: apply(String) -> String
34 调用: toUpperCase() -> String
35 map((str) -> str.toUpperCase())
36 map(String::toUpperCase)
37
```

Stream类: jdk8.0新类 专门用来处理集合/数组 可以让代码变得更加的简洁/优雅

- 如何把集合里面的元素放进流里面
 - `Stream<泛型> ss = 集合对象.stream()`
- 如何将数组里面的元素放进流里面
- `Stream<泛型> ss = Arrays.stream(数组对象)`
- Stream类里面中间方法:
 - **filter(Predicate)**:过滤集合里面符合条件的元素
 - **limit(long n)**:截断流 只要集合里面的前n个元素
 - **skip(long n)**:丢弃前n个元素 将剩下的元素放在新集合里面
 - **map(Function)**:参数接收一个函数 会将该函数应用到每个对象身上
 - **distinct()**:将集合里面的重复元素去除 -》尊重hashCode() + equals()
 - **sorted()**:自然排序 按照泛型类排序规则排序
 - **sorted(比较器)**:定制排序 按照比较器的排序进行排序
 - **generate(Supplier)**:创建无限流对象
- Stream类里面的终端方法:
 - **forEach(Consumer)**:遍历集合里面的每个元素 -> void
 - **anyMatch(Predicate)**:判断集合里面是否有一个元素符合条件 -> boolean
 - **allMatch(Predicate)**:判断集合里面是否所有元素都符合条件 -》 boolean
 - **noneMatch(Predicate)**:判断集合里面是否所有元素都不符合条件-》 boolean
 - **reduce(BiFunction)**:将集合里面的多个值归约成一个值 ->Optional
 - **count()**:得到流里面的元素个数 -》 long
 - **findFirst()**:得到流里面的第一个元素 -> Optional
 - **max(Comparator)**:得到流里面的最大值[最后一个元素] -》 Optional
 - **min(Comparator)**:得到流里面的最小值[第一个元素] -》 Optional
 - **collect(Collectors对象)**: 将流里面的元素收集起来
 - `Collectors.toList()` -> List集合
 - `Collectors.toSet()` -> Set集合
 - `Collectors.toMap()` -> Map集合
- 注意:
 - Stream类里面所有的中间方法必须配合终端方法使用 如果只调用中间方法 不调用终端方法的话 那么中间不会执行 -> 惰性求值

Date类: 时间类 jdk1.0 不安全 java.util.*

- `Date dd = new Date();`得到当前的时间
- `Date dd = new Date(long time);`得到指定的时间
- 直接打印Date对象的时候 显示的效果可能和我们平时的习惯不同 所以格式化
- SimpleDateFormat:格式化日期
 - **将Date转换成String**
 - `Date dd = new Date();`
 - `SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");`
 - `String time = sdf.format(dd);`
 - **将字符串转换成Date类型**
 - `String str = "2022-5-18 23:12:56";`
 - `SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");`
 - `Date dd = sdf.parse(str);`

Calendar:日历类 since jdk1.2 不安全 java.util.*;

- 如何得到当前的时间 由于Calendar是抽象类 所以不能new对象
 - `Calendar cc = Calendar.getInstance();`
- 如何得到当前时间的年月日
 - `System.out.println(cc.get(1));`
 - `System.out.println(cc.get(Calendar.YEAR));`
 - `1 = Calender.Year[年]`
 - `2 = Calender.MONTH[月]`
 - `5 = Calender.DAY_OF_MONTH[号]`
 - `7 = Calender.DAY_OF_WEEK[周几]`
 - `11 = Calender.HOUR_OF_DAY[24小时制]`
 - `12 = Calender.MINUTE[分钟]`
 - `13 = Calender.SECOND[秒数]`
- 如何延后时间/提前时间
 - `cc.add(2,3);`//3个月后
 - `cc.add(Calendar.MONTH,3);`//3个月后
- 如何格式化日历类:
 - `Calendar cc = Calendar.getInstance();`
 - `SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");`
 - `String time = sdf.format(cc.getTime());`

LocalDateTime: jdk8.0出现的新的时间类 安全 java.time.*

- LocalDate:得到日期[年月日]
- LocalTime:得到时间[时 分 秒]
- 如何得到当前的时间: 年月日时分秒
 - `LocalDateTime ldt = LocalDateTime.now();`
- 如何得到指定的时间
 - `LocalDateTime ldt1 = LocalDateTime.of(2022,5,18,23,40,50);`

- 如何得到年 月 日 时 分 秒
 - `ldt.getYear()`
 - `ldt.getMonth()`
 - `ldt.getDayOfMonth()`
 - `ldt.getDayOfWeek()`
 - `ldt.getHour()`
 - `ldt.getMinute()`
 - `ldt.getSecond()`
- 如何添加日期
 - `ldt.plusYears(数量)`
 - `ldt.plusMonths(数量)...`
- 比较两个时间前后
 - `LocalDateTime`对象1.**`isBefore(LocalDateTime对象2)`** : 1时间是否在2时间之前
 - `LocalDateTime`对象1.**`isAfter(LocalDateTime对象2)`** : 1时间是否在2时间之后
- **Duration类: 记录两个时间的间隔**
 - `Duration du = Duration.between(LocalDateTime对象1, LocalDateTime对象2)`
 - `du.toDays()`: 间隔天数
 - `du.toHours()`: 间隔小时
- **DateFormatter类: 格式化 `LocalDate`类 `LocalTime`类 `LocalDateTime`类**
 - 最下面

总结: `Date` / `Calendar` / `LocalDate` / `LocalTime` / `LocalDateTime`

```

1  得到某个日期:
2      Date:jdk1.0  java.util
3          Date dd = new Date(); //当前时间
4          Date dd = new Date(long 毫秒数); //指定日期
5
6      Calendar:jdk1.2  java.util
7          Calendar cc = Calendar.getInstance(); //当前日期
8  -----
9
10     LocalDate:jdk8.0  java.time
11         LocalDate dd = LocalDate.now(); //当前年月日
12         LocalDate dd = LocalDate.of(2022, 5, 18); //指定年月日
13
14     LocalTime:jdk8.0  java.time
15         LocalTime ll = LocalTime.now(); //当前时分秒
16         LocalTime ll = LocalTime.of(23, 12, 50); //指定时分秒
17
18     LocalDateTime:jdk8.0  java.time
19         LocalDateTime ldt = LocalDateTime.now(); //当前年月日时分秒
20         LocalDateTime ldt = LocalDateTime.of(2022, 10, 28, 5, 20, 45); //指定的
    年月日时分秒
  
```

```

1  分别得到年 月 日 时 分 秒
2  Date:
3      得到年月日 时分秒的方法全部过时
4
5  Calendar类:
  
```

```
6      get(参数)
7      参数:
8          1 = Calendar.YEAR
9          2 = Calendar.MONTH    从0开始计算
10         5 = Calendar.DAY_OF_MONTH
11         7 = Calendar.DAY_OF_WEEK    周天算周一
12         11 = Calendar.HOUR_OF_DAY
13         12 = Calendar.MINUTE
14         13 = Calendar.SECOND
15
16     LocalDate类:
17         getYear()
18         getMonth()
19         getDayOfWeek()
20         getDayOfMonth()
21
22     LocalTime类:
23         getHour()
24         getMinute()
25         getSecond()
26
27     LocalDateTime类:
28         getYear()
29         getMonth()
30         getDayOfWeek()
31         getDayOfMonth()
32         getHour()
33         getMinute()
34         getSecond()
```

```
1  如何添加时间
2
3  Date类:
4      添加时间的方法全部过时
5
6  Calendar类:
7      cc.add(谁, 数量)
8      3年后
9          add(1, 3)
10         add(Calendar.YEAR, 3)
11
12  LocalDate类:
13      plusYears(数量)
14      plusMonths(数量)
15      plusWeek(数量)
16      plusDays(数量)
17
18  LocalTime类:
19      plusHours(数量)
20      plusMinute(数量)
21      plusSecond(数量)
22
23  LocalDateTime类:
24      plusYears(数量)
25      plusMonths(数量)
```

```
26 plusWeek(数量)
27 plusDays(数量)
28 plusHours(数量)
29 plusMinute(数量)
30 plusSecond(数量)
```

```
1 如何将Date转换成字符串
2      Date dd = new Date();
3      SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
4      String time = sdf.format(dd);
5
6 如何将Calendar转换成字符串
7      Calendar cc = Calendar.getInstance();
8      SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
9      String time = sdf.format(cc.getTime());
10
11 -----
12 -----
13 如何将LocalDate转换成字符串
14      LocalDate ld = LocalDate.now();
15      DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy-MM-dd ");
16      String time = df.format(ld);
17
18 如何将LocalTime转换成字符串
19      LocalTime lt = LocalTime.now();
20      DateTimeFormatter df = DateTimeFormatter.ofPattern("HH:mm:ss");
21      String time = df.format(lt);
22
23 如何将LocalDateTime转换成字符串
24      LocalDateTime ldt = LocalDateTime.now();
25      DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy-MM-dd
26      HH:mm:ss");
26      String time = df.format(ldt);
```

```
1 如何将字符串转换成Date类
2      String str = "2023-02-08 19:42:50";
3      SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
4      Date dd = sdf.parse(str);
5
6 -----
7 -----
6 如何将字符串转换成LocalDate类:
7      String str = "2023-02-18";
8      DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy-MM-dd");
9      LocalDate ld = LocalDate.parse(str,df);
10
11 如何将字符串转换LocalTime类:
12      String str = "19:45:20";
13      DateTimeFormatter df = DateTimeFormatter.ofPattern("HH:mm:ss");
14      LocalTime lt = LocalTime.parse(str,df);
15
16 如何将字符串转换成LocalDateTime类:
17      String str = "2022-11-23 09:23:59";
```

```
18     DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
19     LocalDateTime ldt = LocalDateTime.parse(str,df);
```

day20

课程内容

- 枚举

枚举: enum

- java中有四大类型: class interface **enum** @interface
- 枚举: 当一个类的对象是有限个 确定的 这个类就适合写成**枚举类**

```
1  有些类希望可以创建无数个对象 -》 普通类
2      class Student{}
3
4  有些类希望只能创建一个对象 -》 单例模式
5      class BJQ{
6
7      }
8
9  有些类希望可以创建有限个对象 确定的 -》 枚举类
10     class Day{}//周几 -》 应该有7个对象
11     class Color{}//交通灯的颜色 -》 应该有3个对象
12     class OrderState()//订单状态 -》 待付款 代发货 待收货 待评价 售后/退款
13     class Season{}//季节 -》 应该有4个对象
14
15
```

- 枚举类里面的构造方法默认都是私有的
- 枚举类里面要求类一开始就要定义对象 **对象和对象之间,隔开 最后一个对象后面可以加;** 如果最后一个对象后面还有语句;不能省略 如果最后一个对象后面没有语句了 那么;可以省略
- 枚举类都**默认继承Enum类** 所以不能继承其他的类 但是**可以实现多个接口**
- 枚举类里面的toString()默认打印枚举对象的名字
- 枚举类里面常用的方法
 - 将枚举类对象放在**数组**里面
 - 枚举类名[] data = 枚举类名.values();
 - 将枚举类对象放在**集合**里面
 - EnumSet<枚举类名> set = EnumSet.allOf(枚举类名.class);
 - 将字符串转换成**枚举对象**
 - 枚举类名 x = 枚举类名.valueOf(字符串对象)
 - 将枚举对象转换成**字符串**
 - String x = 枚举对象.name();

- 得到某个**枚举对象的下标**
 - `int x = 枚举对象.ordinal();`

Date格式(LocalDateTime)

`Date dd = new Date();`

打印对象: 月 周几 小时 分钟 秒数 年份

比较时间前后:

`Date对象1.after(Date对象2)`

`Date对象1.before(Date对象2)`

```
1  格式化:
2  Date dd = new Date();
3  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
4  String time = sdf.format(dd);
5
6  String str = "2022-08-11 23:11:56";
7  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
8  Date dd = sdf.parse(str);
```

Calendar

`Calendar cc = Calendar.getInstance();`

打印对象: 一大堆 Year Month Day_of_month

`cc.add(Calendar.YEAR,数量)`

`cc.get(Calendar.MONTH)`

```
1  比较时间前后:
2      Calendar对象1.after(Calendar对象2)
3      Calendar对象1.before(Calendar对象2)
4
5  格式化:
6  Calendar cc = Calendar.getInstance();
7  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
8  String time = sdf.format(cc.getTime());
```

LocalDateTime

`LocalDate`

`LocalTime`

```
1  LocalDateTime ldt = LocalDateTime.now()
2  打印对象: 年-月-日T时:分:秒.毫秒数
3  ldt = ldt.plusYears(5L)
4  ldt = ldt.plusMonths(6L);
5
6  ldt.getYear();
7  ldt.getMonth()
8
9  比较时间前后:
10 LocalDateTime对象1.isAfter(LocalDateTime2)
11 LocalDateTime对象1.isBefore(LocalDateTime2)
12
```

```

13 | 间隔时间:
14 | Duration dd = Duration.between(LocalDateTime对象,LocalDateTime对象);
15 | dd.toDays()
16 | dd.toHours()
17 |
18 | 格式化:
19 | LocalDateTime ldt = LocalDateTime.now()
20 | DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
21 | dtf.format(ldt);

```

day21

课程内容

- 注解

java中的四大类型:

- class interface enum @interface
- 类 接口 枚举 注解

什么是注解?

- 注解其实就是一个标记 给已有的代码提供补充信息

java中的三大内置注解:

- **@Override**:表示下面的方法一定要覆盖父类的某些方法
- **@Deprecated**:表示下面的类/方法/属性...已经过时了
- **@SuppressWarnings**:表示忽略某种警告
 - **deprecation**:过期警告
 - **unchecked**:未经检查的警告

java中的四大元注解: 给其他的注解提供补充信息

- **@Target**: 表示被修饰的注解 可以作用在谁的上
 - 只能从**ElementType枚举类**里面取值
 - **TYPE**:四大类型
 - **CONSTRUCTOR**:构造方法....
- **@Retention**:表示被修饰的注解生命周期
 - 只能从**RetentionPolicy枚举类**里面取值
 - **SOURCE**:编译
 - **CLASS**:该注解可以生成在.class文件里面 但是不能被反编译
 - **RUNTIME**:该注解可以生成在.class文件里面 但是可以被反编译
- **@Documented**:表示被修饰的注解可以生成在api文档里面
- **@Inherited**:表示被修饰的注解可以被继承

