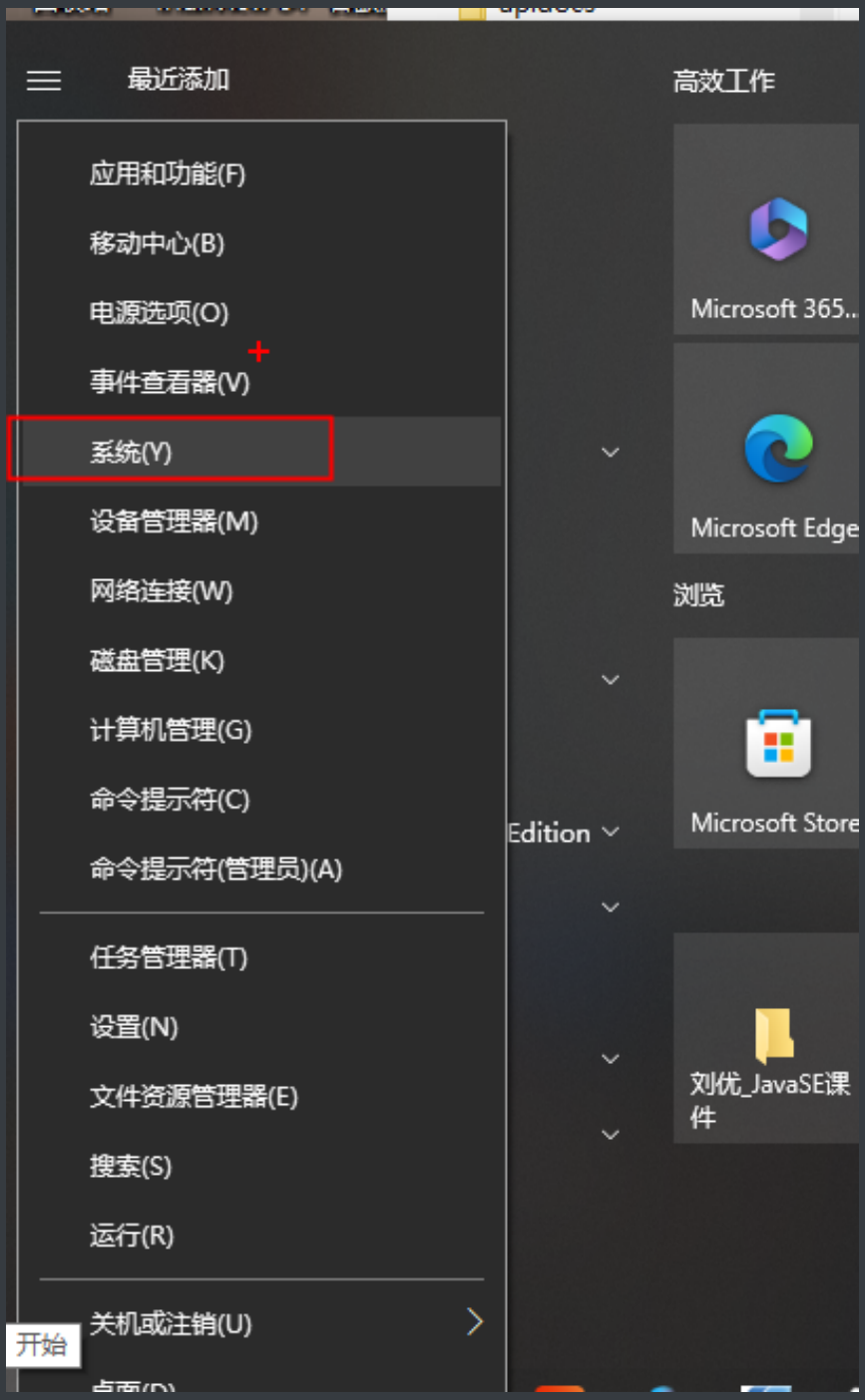
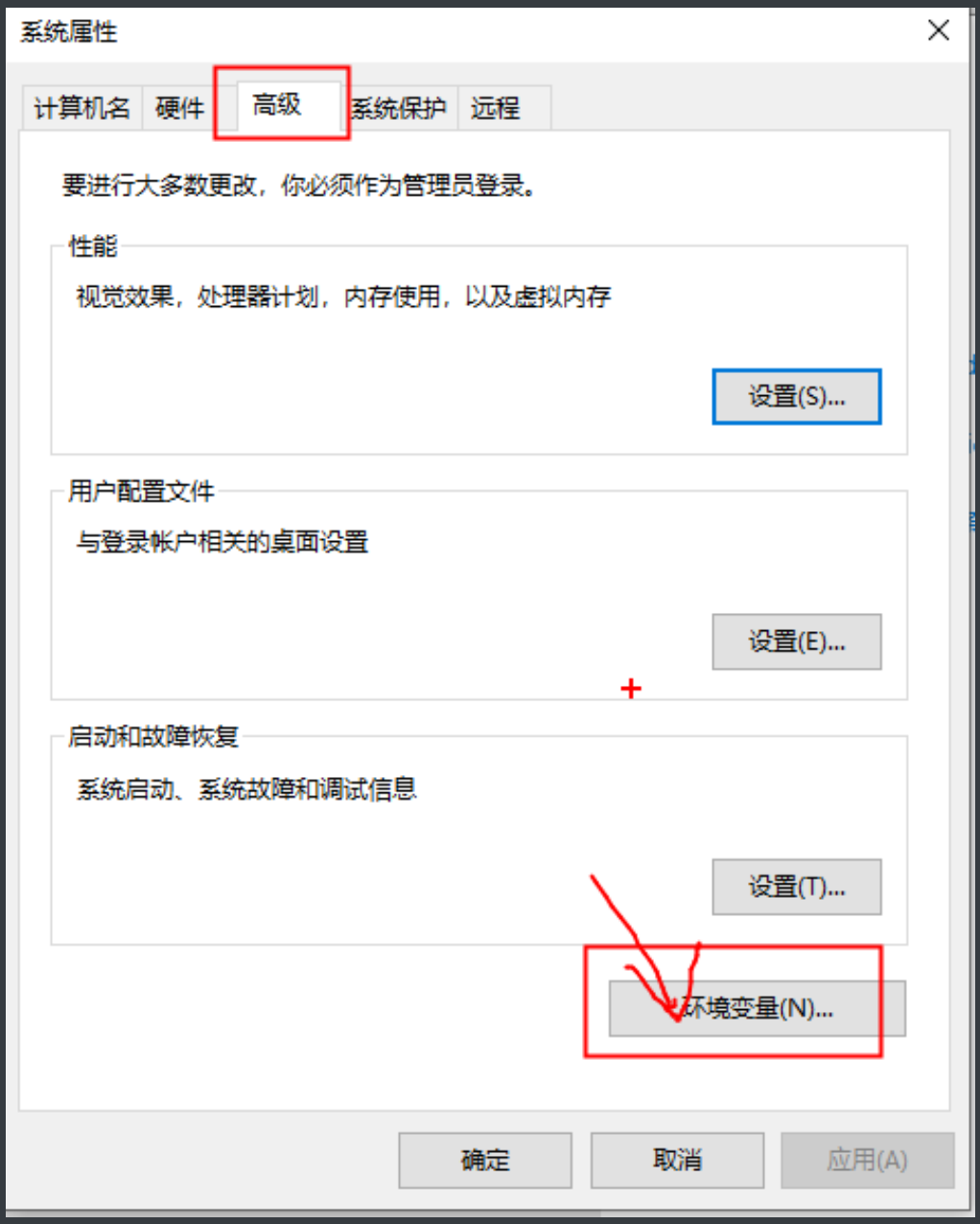
JDK安装

路径：一条路，这条路通向某个文件或目录，它是一个地址

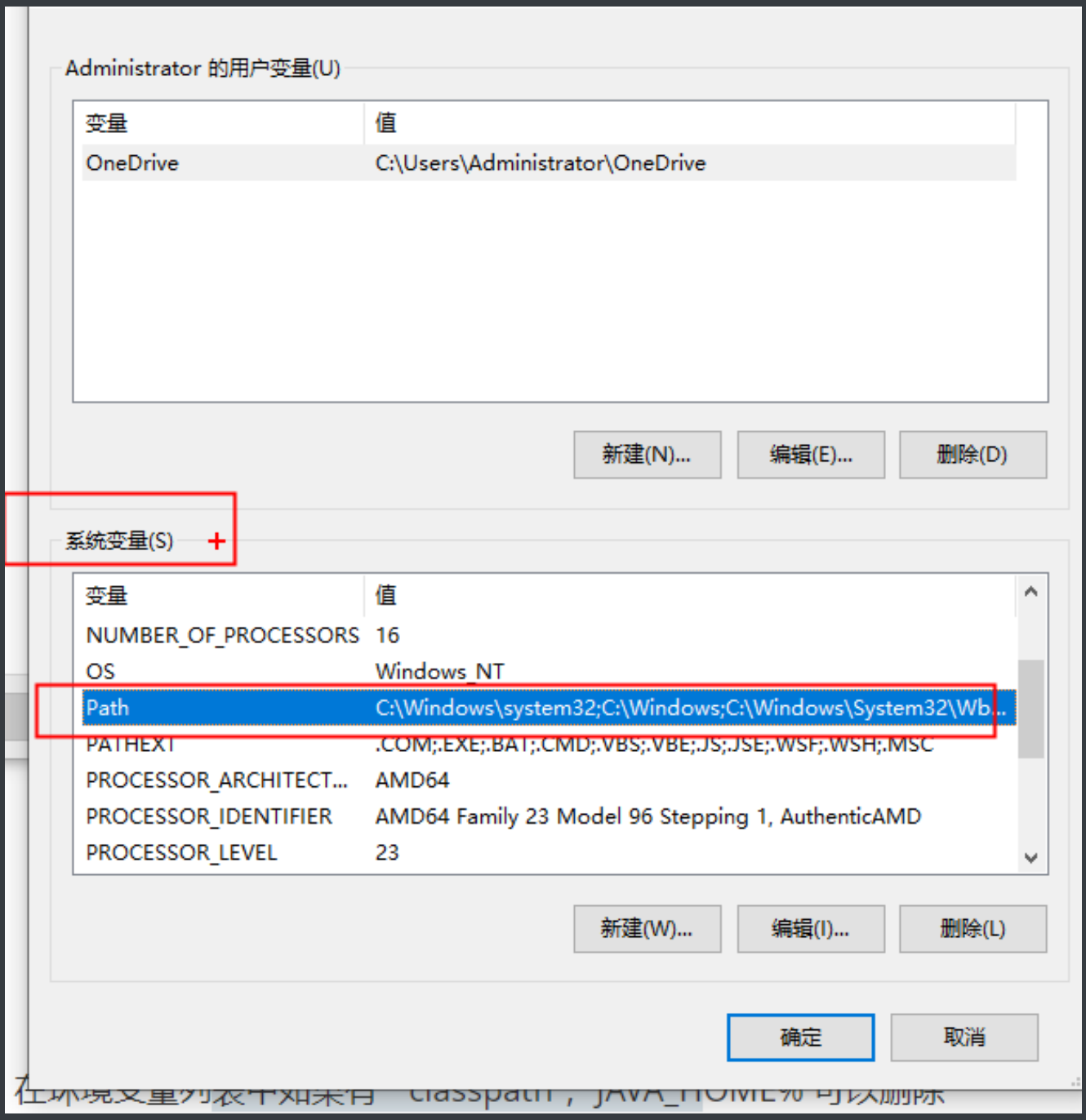
D:\Atguigu\02_Program\jdk-17.0.10\bin

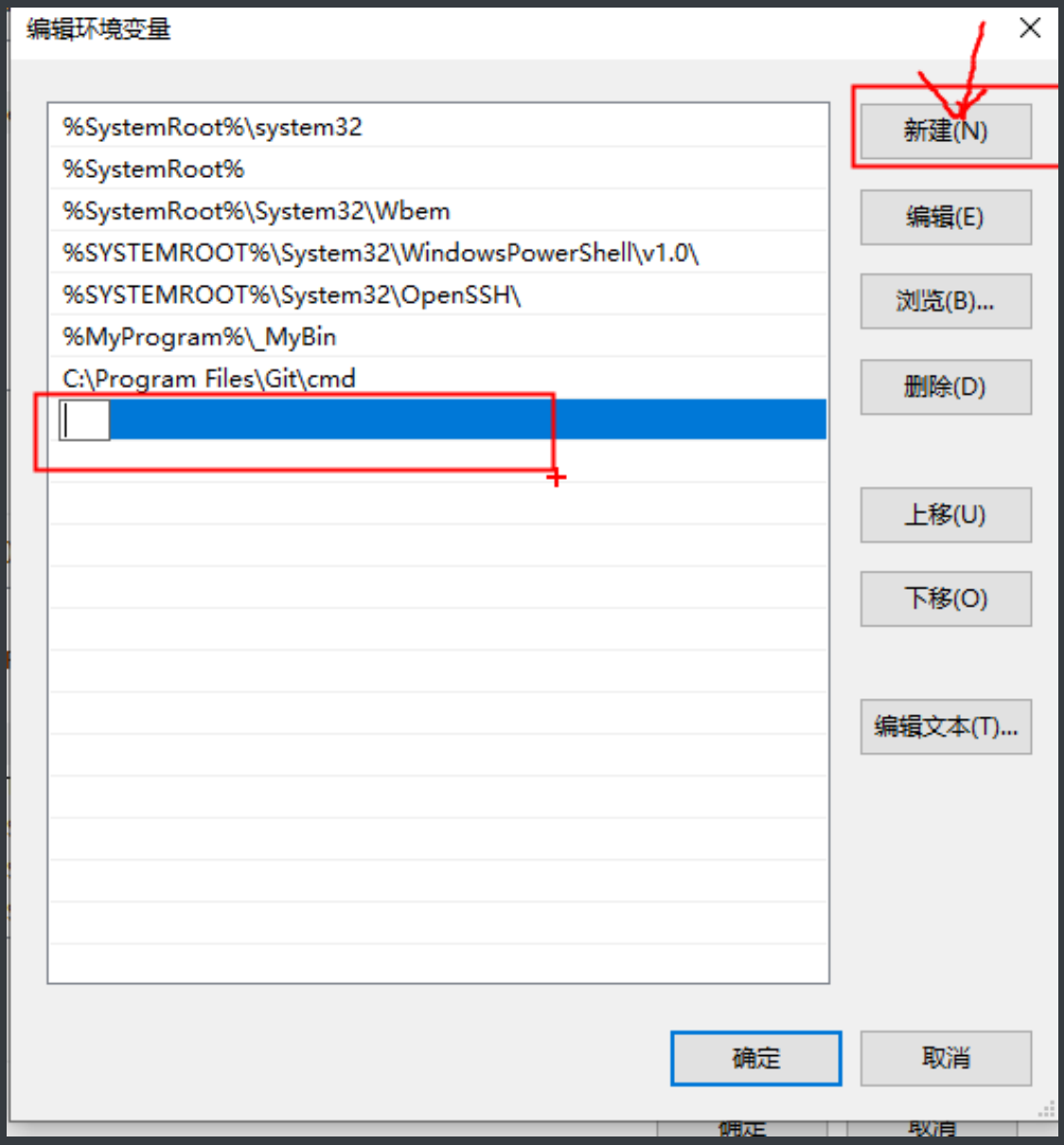

最近添加
高效工作
应用和功能(F)
移动中心(B)
电源选项(O)
事件查看器(V)
系统(Y)
设备管理器(M)
网络连接(W)
磁盘管理(K)
计算机管理(G)
命令提示符(C)
命令提示符(管理员)(A)
任务管理器(T)
设置(N)
文件资源管理器(E)
搜索(S)
运行(R)
关机或注销(U)
开始
Microsoft 365..
Microsoft Edge
浏览
Microsoft Store
Edition
刘优_JavaSE课件

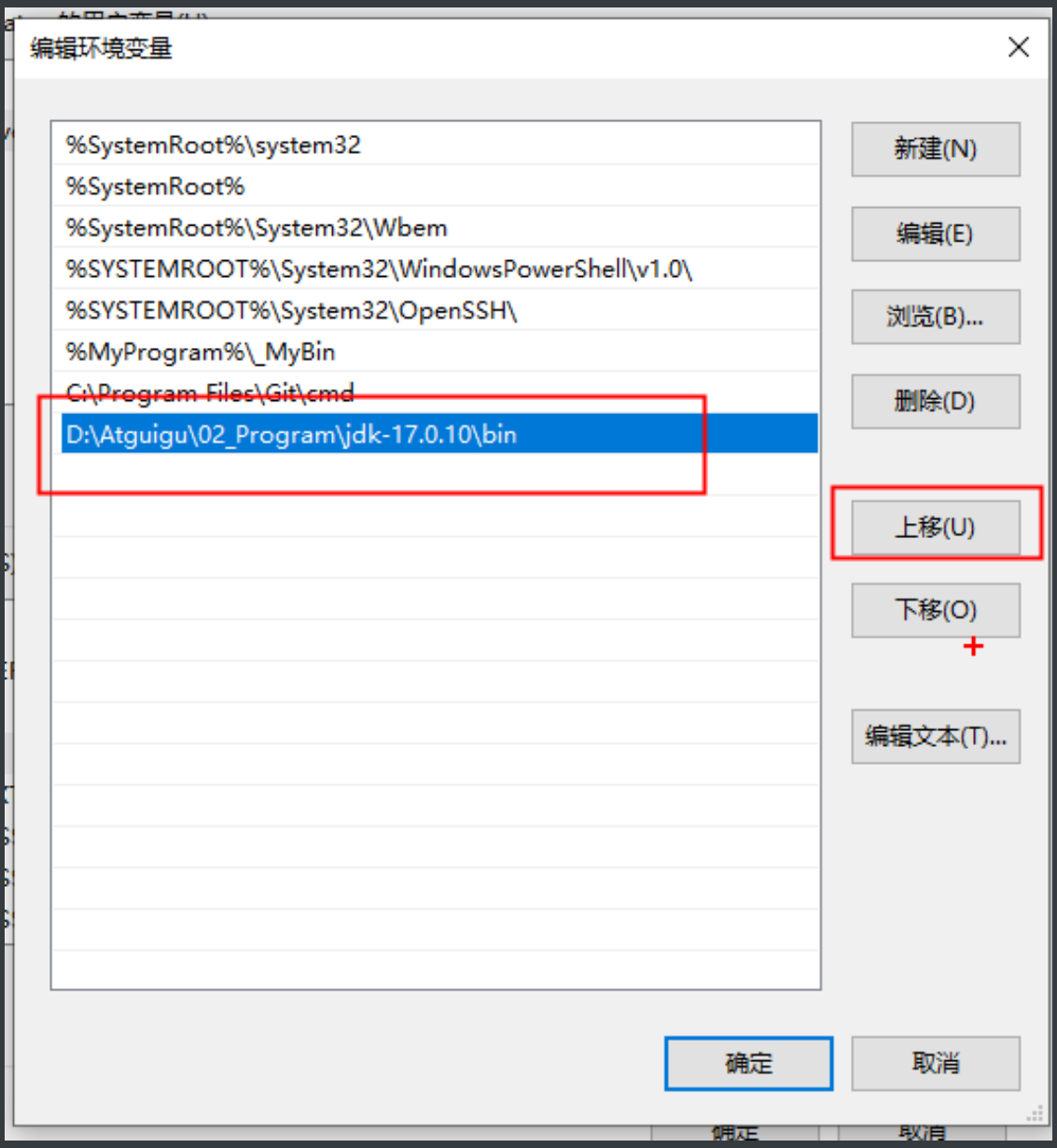

主页
设置
屏幕
声音
通知和操作
专注助手
电源和睡眠
电池
存储
平板电脑
多任务处理
关于
复制
更改产品密钥或升级 Windows
阅读适用于我们服务的 Microsoft 服务协议
阅读 Microsoft 软件许可条款
相关设置
BitLocker 设置
设备管理器
远程桌面
系统保护
高级系统设置
重命名这台电脑
获取帮助
提供反馈

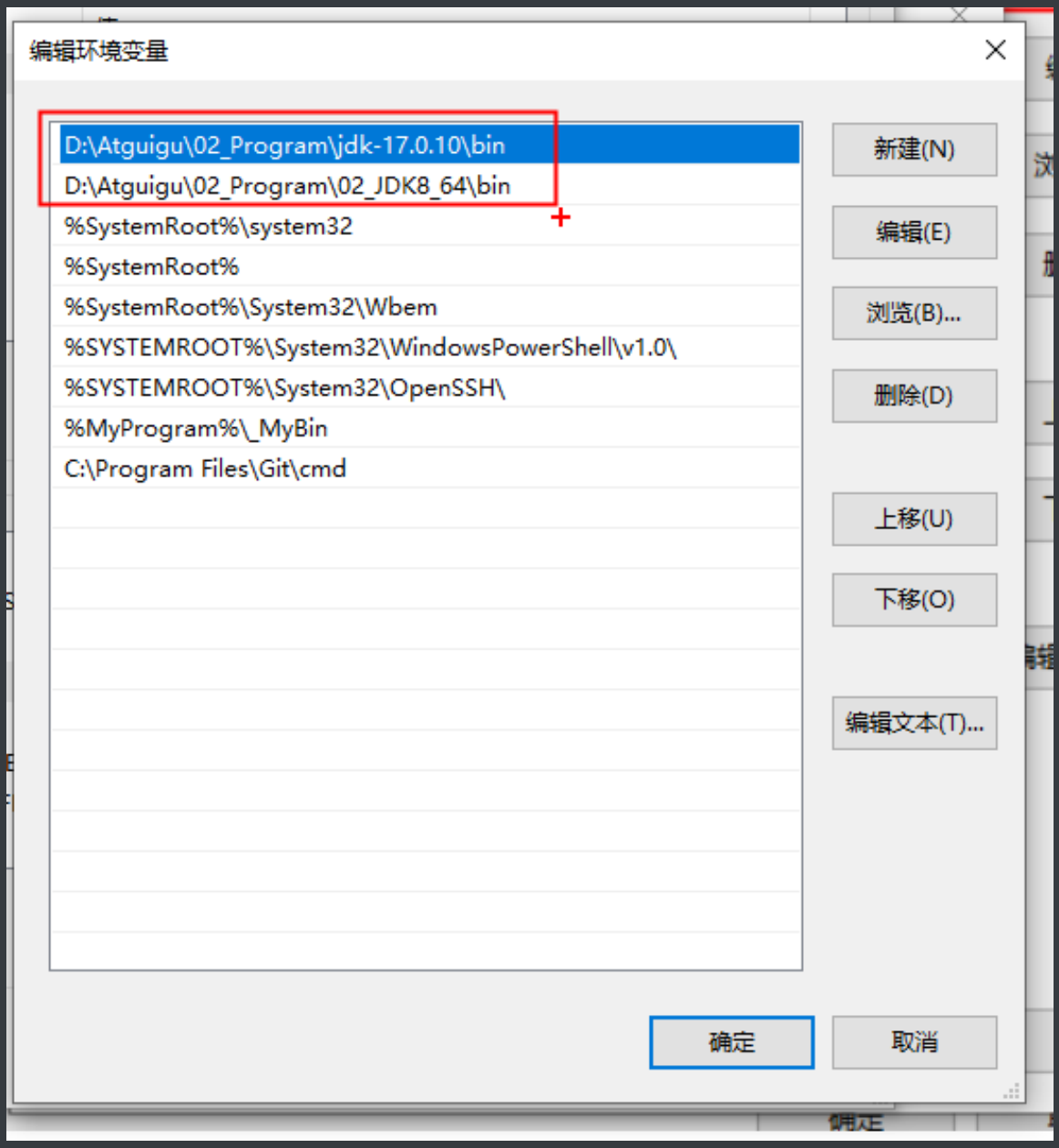


在环境变量列表中如果有 " classpath", "JAVA_HOME% 可以删除


Administrator 的用户变量(U)
变量
值
OneDrive
C:\Users\Administrator\OneDrive
新建(N)...
编辑(E)...
删除(D)
系统变量(S)
变量
值
NUMBER_OF_PROCESSORS
16
OS
Windows_NT
Path
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wb...
PATHEXT
.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECT...
AMD64
PROCESSOR_IDENTIFIER
AMD64 Family 23 Model 96 Stepping 1, AuthenticAMD
PROCESSOR_LEVEL
23
新建(W)...
编辑(I)...
删除(L)
确定
取消

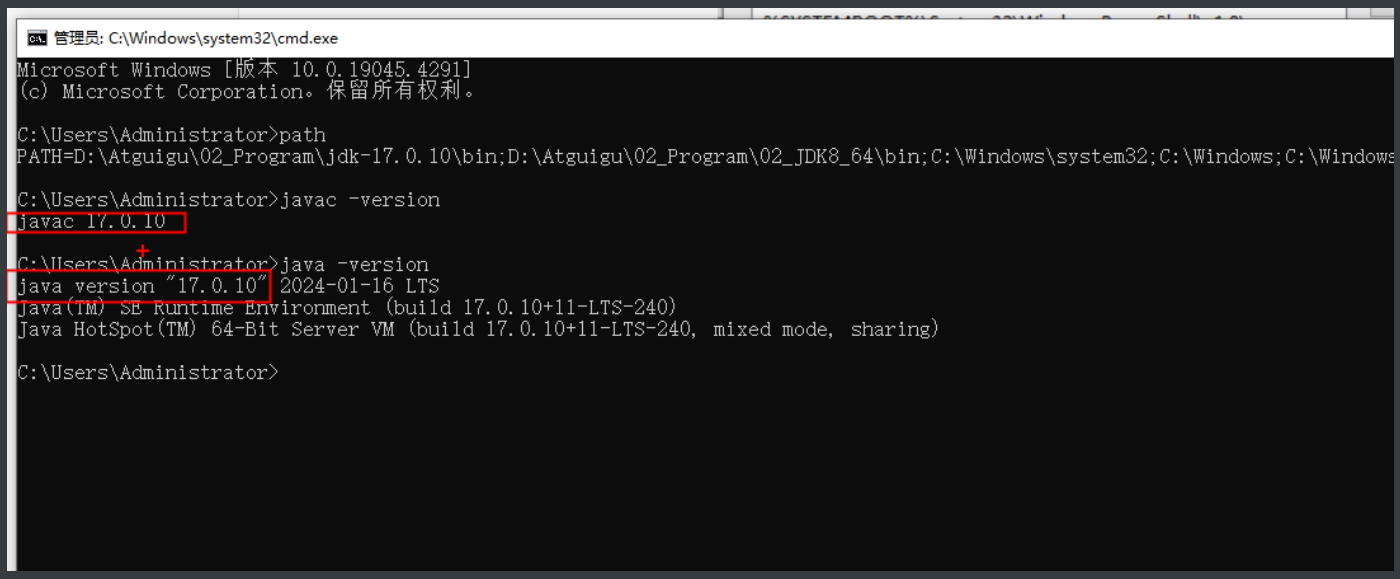

编辑环境变量
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
%MyProgram%_MyBin
C:\Program Files\Git\cmd
新建(N)
编辑(E)
浏览(B)...
删除(D)
上移(U)
下移(O)
编辑文本(T)...
确定
取消


编辑环境变量
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
%MyProgram%_MyBin
C:\Program Files\Git\cmd
D:\Atguigu\02_Program\jdk-17.0.10\bin
新建(N)
编辑(E)
浏览(B)...
删除(D)
上移(U)
下移(O)
编辑文本(T)...
确定
取消



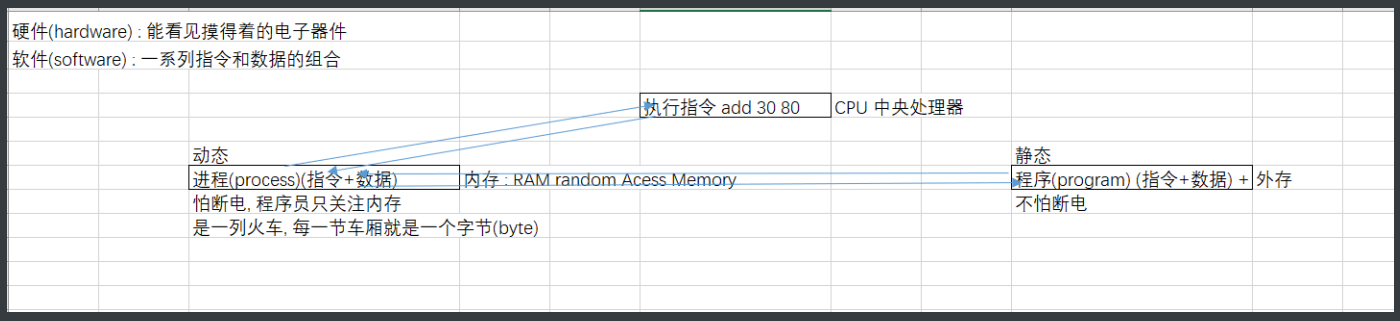
验证

win + r => cmd => 回车



第1章 基础常识

计算机构成



软件 : (software)

应用软件 (Application) : 为了满足某种需求的程序

系统软件 (OS) 操作系统 : 管理所有的硬件和软件

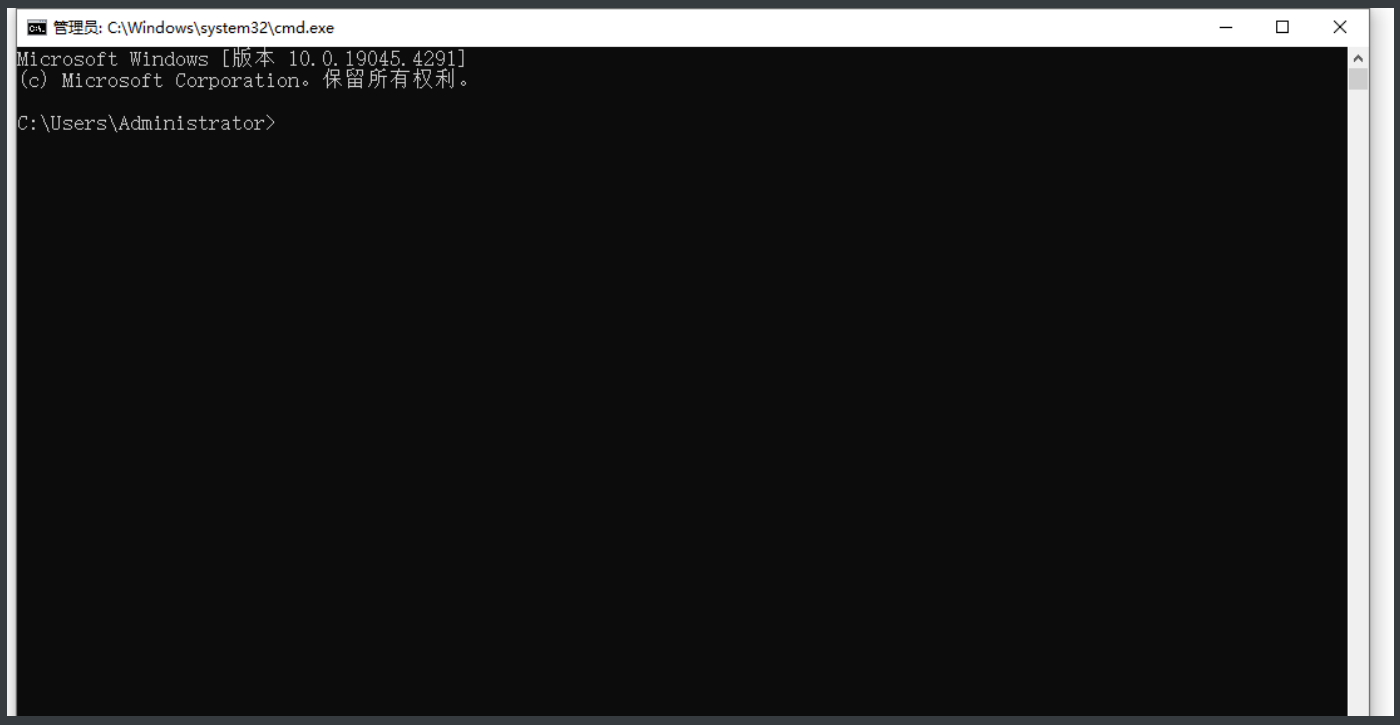
人机交互 : 人通常使用OS来操作所有硬件和软件 的过程

图形化界面 : GUI, 直观.

命令行界面 : CLI, 不直观

win+r => cmd => 回车

DOS Disk Operating System 命令窗口, 核心任务就是文件管理



路径(path) : 一条路, 这条路通向一个文件或目录(directory)

C:\Users\Administrator> : 左面的这个目录称为当前工作目录.

查看当前目录下的内容(子目录和子文件)

dir

切换工作目录

cd(change directory) 目标目录名

切换盘符

目标盘符:

在当前目录下创建新子目录

md (make directory)

md aa bb cc dd 其中的 aa bb cc dd 称为命令行参数, 作用是给命令传递进一步信息.

练习 : 创建目录

D:\Atguigu\05_Code\JavaSE\day01

2个特殊目录

. 目录表示就是当前目录

.. 目录表示的就是当前目录的父目录

相对路径 : 以.或..目录为开始的路径 . 简单, 容易出易.

.\atguigu\05_code\javase\day01

绝对路径 : 以根目录为开始的路径 , 复杂, 绝不出错

D:\Atguigu\05_Code\JavaSE\day01

跨越盘符直接跳转到指定目录

cd /d D:\Atguigu\05_Code\JavaSE\day01

cd ..\..\..\..\ 以相对路径的方向向上跳转

删除空目录

rd(remove directory)

退出窗口

exit

命令的本质就是程序

内部命令 : OS内置的命令, 不需要依赖文件, 可以直接执行

外部命令 : 存储在硬盘上面的可以执行的文件, xxx.exe, 需要在执行时依赖path环境变量.

javac.exe 称为编译器

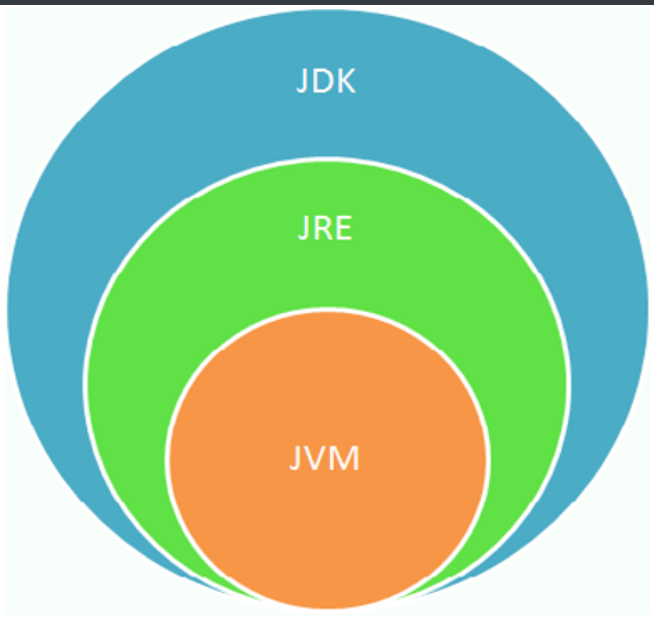
java.exe 称为运行器

语言发展

1. 打孔机 : 直接以二进制形式写程序, 面向所有硬件和软件
2. 汇编 : 面向CPU指令集.
3. 高级语言 : 面向OS开发，OS变化会导致程序出问题
4. 虚拟机语言 : 面向VM开发, VM的规范永不改变.

语言特点

1. 简单 : 相对C/C++而言.
2. 面向对象 : 关注的是具有功能的对象.
3. 分布式 : 基于网络的多主机协作.
4. 健壮 : 强类型机制(只要是数据一定有类型的约束), 异常处理机制, 垃圾(内存中本来不再被使用,但是被占用的空间)的自动收集. 引用(reference) 安全的指针.
5. 安全 : 有类加载器负责所有类的加载和检查.
6. 跨平台 : 跨操作系统, 不同的OS上都有VM
7. 性能好 : java是编译型, 比解释型速度快
8. 多线程 : 最大化利用cpu, 高并发. 多任务.



- JRE = JVM + Java SE标准类库
- JDK = JRE + 开发工具集（例如Javac编译工具等）

开发java程序的步骤

1. 编写源程序

    1) 在特定的目录下面新建文本文档, 改名为Xxx.java

    2) 右键这个文件, "edit with notepad++" . 使用NotePad++编辑此文件, 在里面写代码

2. 编译源程序让其生成执行的程序

    3) 进入命令行窗口, 以刚才的文件的目录为工作目录, 进行切换

    4) 执行 javac Xxx.java , 编译的结果就是产生一个新文件 Xxx.class

```
C:\Users\Administrator>cd /d d:\Atguigu\05_Code\javaSE\day01

d:\Atguigu\05_Code\JavaSE\day01>dir
 驱动器 D 中的卷是 Data
 卷的序列号是 E3FE-C981

 d:\Atguigu\05_Code\JavaSE\day01 的目录

2024/04/23  11:56    <DIR>          .
2024/04/23  11:56    <DIR>          ..
2024/04/23  12:01               159 Hello.java
               1 个文件            159 字节
               2 个目录 1,347,090,251,776 可用字节

d:\Atguigu\05_Code\JavaSE\day01>javac Hello.java

d:\Atguigu\05_Code\JavaSE\day01>dir
 驱动器 D 中的卷是 Data
 卷的序列号是 E3FE-C981

 d:\Atguigu\05_Code\JavaSE\day01 的目录

2024/04/23  12:03    <DIR>          .
2024/04/23  12:03    <DIR>          ..
2024/04/23  12:03               485 Hello.class
2024/04/23  12:01               159 Hello.java
               2 个文件            644 字节
               2 个目录 1,347,088,089,088 可用字节

d:\Atguigu\05_Code\JavaSE\day01>
```

3. 执行程序

   5) 仍然在此窗口中, 继续使用命令 java Xxx // 不要加上.class了

```
2024/04/23  12:03    <DIR>          .
2024/04/23  12:03    <DIR>          ..
2024/04/23  12:03               485 Hello.class
2024/04/23  12:01               159 Hello.java
               2 个文件            644 字节
               2 个目录 1,347,088,089,088 可用字节

d:\Atguigu\05_Code\JavaSE\day01>java Hello
这里可以随便写, 因为这是字符串内容, 把我打印在屏幕上吧..

d:\Atguigu\05_Code\JavaSE\day01>
```

第1个程序的分析

```
// 注释，单行注释 comment，作用是不参与程序的执行(编译时所有的注释全部删除)，仅用作说明提醒
// 我也想注释，只能作用于当前行
/*
  多行注释，适用于注释内容多的场景.
  这一行也可以被注释
  多行注释绝不可以嵌套
*/
/**
  文档注释，会被工具识别，用于生成文档说明.
*/


/*
  public : 公共的
  class : 用于声明或定义类的关键字
  Hello : 是具体的类名

  类 : java程序的基本单位. 一个java程序至少得是一个类
  {} 成对出现，用于表示一个范围或一个块

  类 = 类头(类签名) + 类体(类名后面的一对{}及其中的内容)

  类中包含方法，方法中包含语句
  方法必须要隶属于类，语句必须隶属于方法.

  主类 : 包含了主方法的类
  非主类 : 没有包含主方法的类

  一个java程序的本质就是一个主类

  公共类 : 有public修饰的类，类名必须和源文件名一致. 一个源文件中最多只能有一个公共类. 将来可以被跨包使用
  非公共类 : 没有public修饰的类. 一个源文件中可以有任意多个非公共类.
```

```
*/

/*
public static void test2() {

}
*/

public class Hello {

  /*
    这是一个方法 : 方法是java程序中的独立的功能单位
    public : 公共的
    static : 静态的
    void : 无返回值的
    main : 称为方法名
    (String[] args) : 称为参数

    方法 = 方法头(方法签名) + 方法体(参数后面的一对{}及其中的内容)

    主方法 : 也称为入口方法, 最特殊的方法, 特殊在程序的运行总是从main方法开始的.
    主方法写法永远固定, 记住它
  */
  public static void main(String[] args) {
    // 这是一个语句Statement, 语句是java程序中最小执行单位, 语句必须要以;结尾.
    // 语句会被编译成指令, JVM负责执行的指令.
    System.out.println("这里可以随便写, 因为这是字符串内容, 把我打印在屏幕上
吧..");
    System.out.println("jlaskdjflkajsdf");

  }

  /** 方法冲突了
  public static void main(String[] args) {
    System.out.println("main 2() .....");
  }
```

```
    */

    // 方法并没有执行.
    public static void test() {
      System.out.println("test()...");
    }


}

/* 同一个目录下，不可以有多个同名类
class Hello {

}*/

// 非主类，没有入口. 每个类都一定会有一个独立的.class文件
class Hello2 {

    public static void test2() {
      System.out.println("test2()...");
    }
}

// 主类，可以直接执行
class Hello3 {

    public static void main(String[] args) {
      System.out.println("test3 main()...");
    }
}
```

执行java程序过程：

java 类名

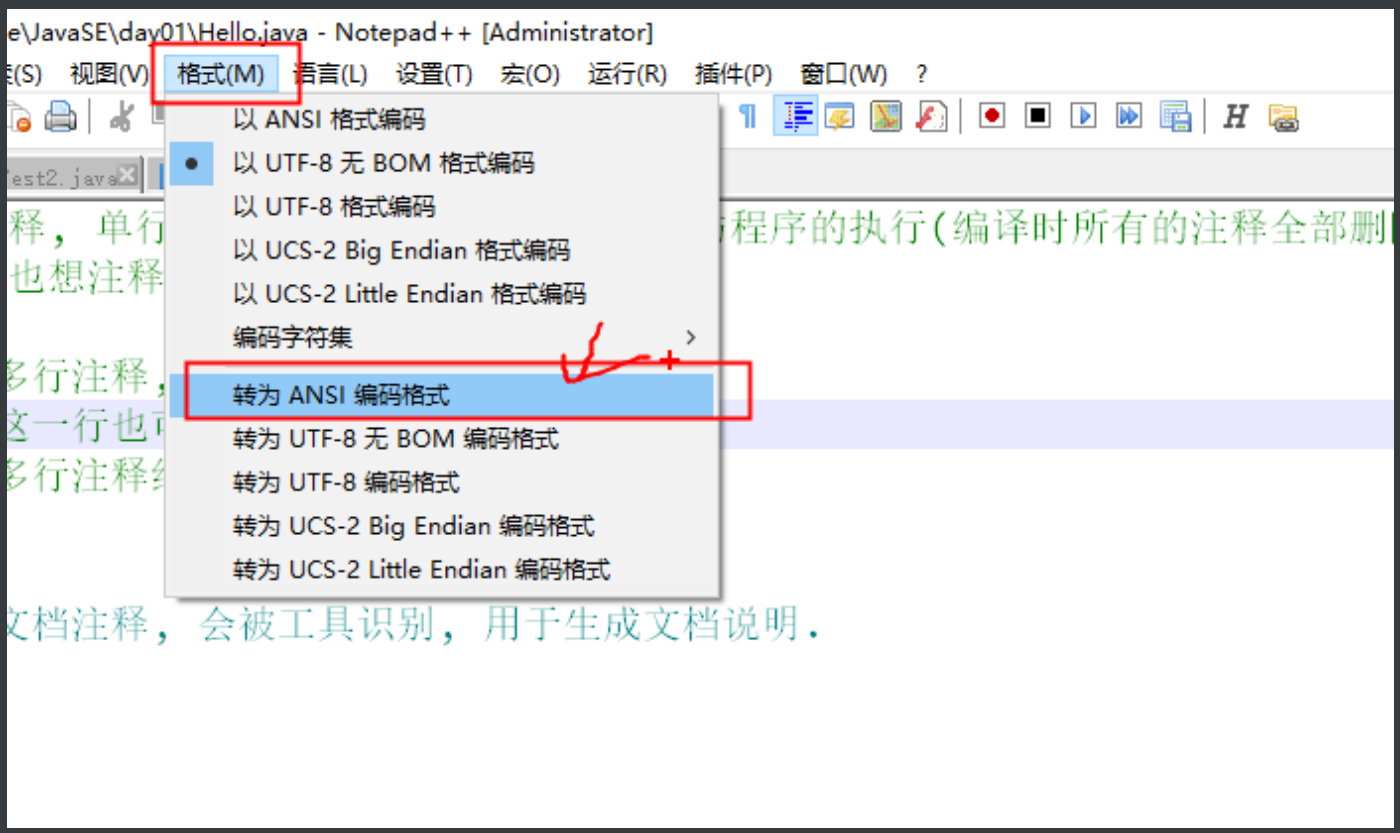
java 瞬间创建一个JVM

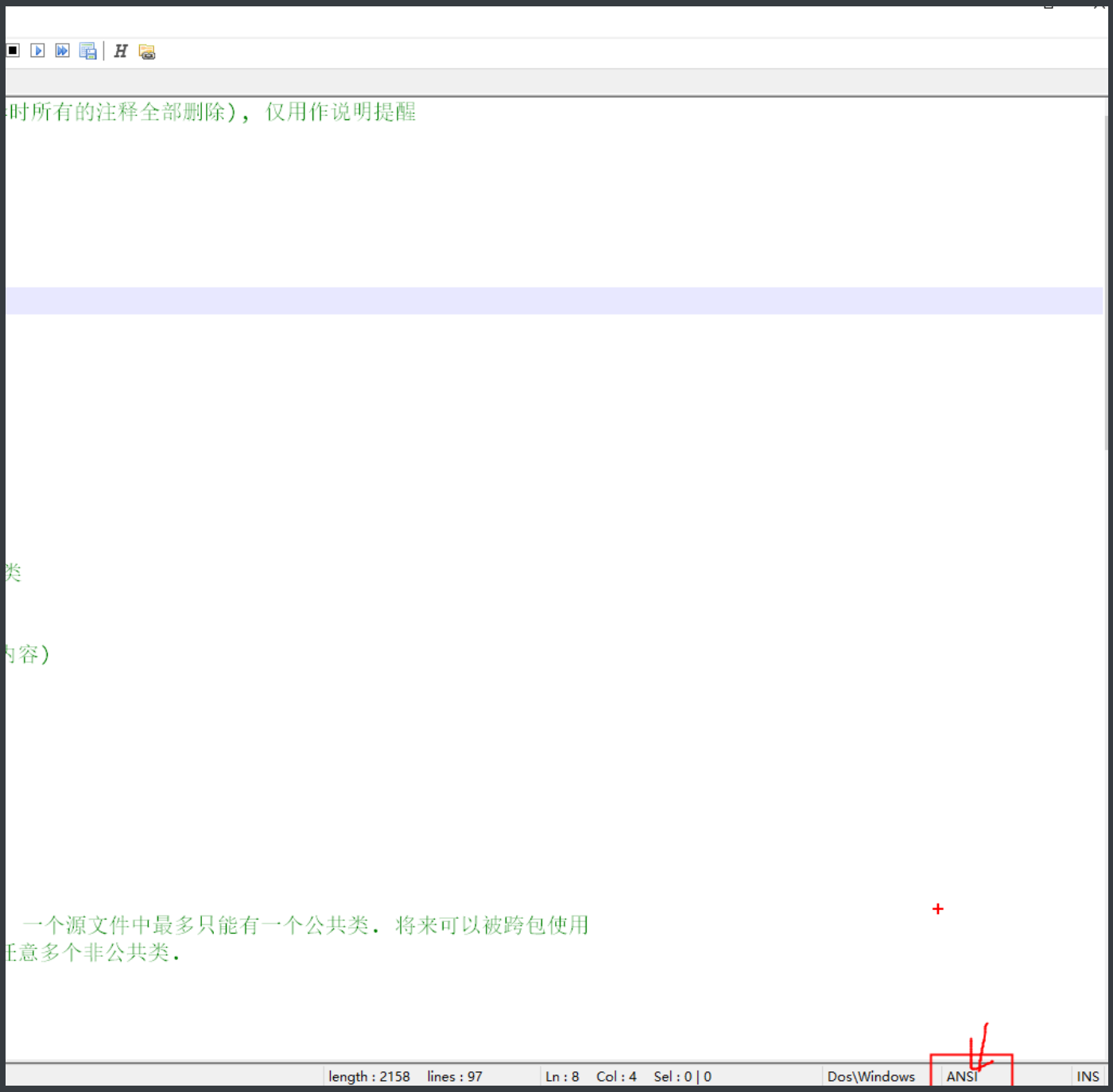
根据后面的类名, 找到 类名.class文件, 并由类加载器负责把它读入到JVM中

检查一下这个类中是否有main方法, 如果没有直接报错提醒, 并销毁JVM.

如果有main方法, 执行main方法, 从第1个语句开始执行, 一直到最后一个语句. 执行完以后, 程序就结束了, JVM销毁.

转换文本文件的编码：





也可以选择以指定的编码方式处理

javac -encoding utf8 源文件名

小结

Java源文件以“java”为扩展名。源文件的基本组成部分是类（class），如本类中的Hello, Hello2, Hello3类。

一个源文件中最多只能有一个public类。其它非公共类的个数不限，如果源文件包含一个public类，则文件名必须按该类名命名。

Java应用程序的执行入口是main()方法。它有固定的书写格式：**public static void main(String[] args**) {...}

Java语言严格区分大小写。

Java方法由一条条语句构成，每个语句必须以“;”结束。

大括号都是成对出现的，缺一不可。

NotePad++ 使用

ctrl + a : 全选

shift + tab 选中的内容左移

tab 选中的内容右移, 插入一个缩进

ctrl + c : 复制

ctrl + v : 粘贴

ctrl + x : 剪切

ctrl + s : 保存

ctrl + z : 撤消

ctrl + d : 快速复制当前行

ctrl + l : 快速删除当前行

第2章 进制与变量

关键字

起特殊作用的单词.

数据类型相关 : class 定义类

byte : 1字节整数

short : 2字节整数

int : 4字节整数

long : 8字节整数

float : 4字节浮点数

double : 8字节的浮点数

char : 2个字节的字符

boolean : 1字节的布尔型(只允许 true和false)

标识符

用于标识区分某个东西的符号, 简单地理解就是名字.

命名规则 : 必须遵守

1) 组成52字母, 10个数字字符.

2) 数字不能开头

3) 不能直接使用关键字, 但是可以包含

4) 大小写不一样, 长度最长65535

5) 不能包含空格

命名规范 : 建议遵守

1) 包名, 全部小写, 多个单词用.隔开. 如 : my.packagename.sample

2) 类名 : 首字母大写, 后面的所有单词的首字母都大写 如 : MyClassNameSample

3) 变量名和方法名 : 首字母小写 , 后面的所有单词的首字母都大写 如 : myVariableNameSample

4) 常量名 : 全部大写, 多个单词用_隔开 如 : MY_CONST_NAME_SAMPLE

编程风格 : 更加个人化

1) 合适的注释

2) 在运算符两边加上空格

```
a+b+c/d*e

a + b + c / d * e
```

3) 方法之间尽量加空行

4) {}尽量使用行尾风格

变量(是我们学习IT的最重要概念, 没有之一) :

内存中的一块被命名的且有特定数据类型约束的空间, 此空间中可以保存一个数据类型约束的数据, 此空间中的数据可以随意改变!!!

数据类型的作用

1) 决定空间大小

2) 决定空间中可以保存什么数据

3) 决定了空间中的数据可以做什么.

数据类型 :

整数

byte

short

int

long

浮点数

float

double

字符

char

布尔

boolean

变量声明 :

数据类型 变量名;
int n1;

变量赋值

变量名 = 值;

n1 = 20;

变量使用

System.out.println(变量名);

变量使用细节

```
public class VariableTest {

  public static void main(String[] args) {
    // 变量声明 数据类型 变量名;
    int n1; // 在内存中开辟了一块空间，共4个字节，再用n1符号和此空间进行映射，对此空间的使用都是通过变量名来完成
    // 变量赋值
    n1 = 25; // 把右面的值25写入到n1符号映射的内存空间中.
```

```
        System.out.println(n1); // 根据n1符号，找到它映射的内存空间，并把里面的值25复制一个副本，实际打印的就是副本值25

        int n2 = n1; // 先声明n2，开辟4字节空间，用n2映射，  再处理右面，根据 n1符号找到它映射的空间，把里面的值25复制出来，再把这个副本的值25写入n2符号所映射的内存空间中去
        //System.out.println(n2);

        n1 = n1 + n2; // 根据 n1符号,把空间中值25load出来，根据n2符号把空间中的值25也load出来，求和得到结果 50 ，再把50这个数据写入n1符号映射的空间

        System.out.println(n1);

    }
}
```

每日一考_day02

1. 写出java语言的8个特性
    1. 简单, 相比C/C++
    2. 跨平台, JVM有不同平台的版本.
    3. 分布式 : 基于网络
    4. 健壮 : 强类型(只要是数据,必须要有数据类型约束), 异常处理, GC自动清理垃圾(把空间标记为可用). 引用(安全的指针)
    5. 面向对象 : 关注的是具有功能的对象.
    6. 性能好 : 编译型比解释型效率高
    7. 多线程 : 多任务
    8. 安全 : 有了类加载器的安全检查.

2. 描述一下语句,类,和方法之间的关系.

   类中包含方法, 方法中包含语句

   类 : java程序的基本单位

   方法 : java程序中的独立的功能单位

   语句 : 最小执行单位.

   方法必须隶属于类

   语句必须隶属于方法

3. 什么是主类, 什么是公共类, 公共类有什么注意点?

   主类 : 包含了入口方法的类

   公共类 ：被public修饰的类

   公共类的类名必须要和源文件名一致.

4. 变量是什么?

   内存中的一块被命名的且被特定的数据类型约束的空间, 此空间中可以保存一个数据类型约束的数据, 此空间中的数据可以在数据类型约束的范围内随意改变.

5. 数据类型的作用有哪些(3点)

   1. 决定了空间大小
   2. 决定了空间中可以保存的数据
   3. 决定了空间中的数据可以做什么

变量声明：

数据类型 变量名;

变量按照数据类型来分：

基本数据类型(primitive)：内存空间中保存的是数据自身(itself)

数值型(number)

整数: byte, short, int long, char

浮点数：float double

布尔型(boolean)

数值型(number)												
	整数											
		byte	-128~127	10101010								
		short	-32768~32767	10101010	10101010							
		int	-20多亿~20多亿	10101010	10101010	10101010	10101010					
		long	-900多亿亿~900多亿亿	10101010	10101010	10101010	10101010	10101010	10101010	10101010	10101010	
	浮点数											
		float	-10^38~10^38	10101010	10101010	10101010	10101010					
		double	-10^308~10^308	10101010	10101010	10101010	10101010	10101010	10101010	10101010	10101010	
布尔型												
		boolean	true(1), false(0)	10101010								
字符型												
		char	? ~ ?	10101010	10101010							

引用数据类型(reference)：内存空间中保存的是别的数据(对象)的内存地址(address), 某个字节的编号. 0地址编号称为 null, 空指针.

转义字符	说明
\b	退格符
\n	换行符
\r	回车符
\t	制表符
\"	双引号
\'	单引号
\\	反斜线

变量注意事项

```
public class VariableTest1 {

  public static void main(String[] args) {
    // 变量 : 内存中的一块空间, 此空间中可以保存一个数据, 且可以变化.
    // 声明 : 数据类型 变量名;
    int n1;
    n1 = 200; // 赋值, 反人类思维, 从右向左, 把右面的确定的值 写入 到左面的变量名映射的内存空间中. istore_x
    System.out.println(n1); // 读值, 把变量名映射的空间中的值复制出来. iload_x

    /** 注意1 : 变量必须要有数据类型和变量名
    n2 = 300;
    int = 400;
    */
    int n2 = 300;

    /** 注意2 : 必须先声明后使用
    System.out.println(n3);
```

```
        int n3 = 500;
        */

        /** 注意3 : 变量有其数据范围
        byte b1 = 10;
        byte b2 = 300;
        */

        // 变量声明
        /** 注意4 : 变量在同一范围内不能重复声明
        short n1;
        */

        {
          // n4只能在这个范围内使用
          int n4 = 800;
          System.out.println(n4);
        }

        /** 注意5 : 变量有其作用范围，作用范围由其声明语句所隶属的一对{}决定
        System.out.println(n4); // 超出了n4的作用范围
        */

        {
          //short n2;
          // 这个n4和上面的{}里面的n4完全不是一回事
          short n4;
        }

        /** 注意6 : 变量必须经过初始化(initialize)后才能使用!!
        int n5;

        System.out.println(n5); // ?
        */

        int n6;
```

```
        n6 = 10; // 出生后的第一次赋值
        n6 = 20; // 普通赋值
        n6 = 30;

        System.out.println(n6);

        // 这是最好的做法. 声明的同时就马上初始化.
        int n7 = 3000;

    }
}
```

不同类型混合赋值

```
public class VariableTest2 {

    // 兼容性只和数据的范围有关

    // 左面变量 = 右面量值, 如果左面的变量的数据类型的范围大于等于右面的量值的数据类型的范围时, 可以直接完成
    // 左面变量 = (目标类型)右面量值, 如果左面的变量的数据类型的范围小于右面的量值的数据类型的范围时, 必须强制转换才可以
    // 兼容性从小到大
    // byte < short < int < long < float < double

    // 变量 : 可以变化的量
    // 常量 : 不可以变化的量, 常量包括 字面量 30, 'a', false, true, 342.32, "alskdjfalksdjf" 和有final修饰的量
    // 浮点数字面量默认使用double类型
    // 整数字面量默认使用的是int类型
    public static void main(String[] args) {
        // 变量的混合赋值
        byte b1 = 20;
```

```
        byte b2 = b1;

        short s1 = b1;
        s1 = 200;
        /*
        b2 = 30; // 编译器对于常量来说是放心的.
        b2 = s1; // 编译器对于变量来说是担心的.
        */

        //b2 = s1;
        b2 = (byte)s1; // 强制类型转换, 把右面的量值的高位字节直接丢弃. 是有风险的, 右面的量值大于左面的变量的范围时.
        System.out.println(b2);

        // 左面是一个字面量, 本质是常量, 内存空间中的数据不允许改变, 赋值会失败.
        // 50 = 50;
        // 赋值操作的左值 永远是 变量
        int i1 = s1;

        int i2 = 50;
        short s2 = (byte)i2;
        long l1 = i2;
        // 这个操作会出问题, 左面的变量是int型, 右面的量是long型
        // i2 = l1;
        i2 = (short)l1;

        long l2 = 32_4234_0000L; // L后缀的作用是提醒编译器, 请不要再使用4字节空间来保存这个整数字面量, 请使用8字节空间来存这个数


        double d1 = .999;
        //l1 = d1;
        l1 = (long)d1; // 浮点数的强制类型转换, 精度的丢失会更高, 不会四舍五入. 直接丢弃小数.
        System.out.println(l1); // 0
```

```
    float f1 = (float)2.3;
    float f2 = 2.3F; // F后缀的作用是提醒编译器，此浮点数字面量不要再使用默认的double型，而使用float类型

    f1 = l1;
    // l1 = f1; long 无法兼容float，因为float范围更大
    l1 = (long)f1;

    // s3是常量，编译器不用担心 它会变成不可控数据
    final short s3 = 10;
    byte b3 = 10; // 没有问题
    byte b4 = s3; //
    byte b5 = (byte)200;

  }
}
```

不同类型混合运算

```
public class VariableTest3 {

  /*
    任意非long整数运算的结果一定是int型
    因为它们用的都是 iXXX指令
    long型数据参与时会升级成 lXXX指令
    float型数据参与时会升级为 fXXX指令
    double型数据参与时会升级为 dXXX指针

    更多的类型参与混合运算时，会先找到范围最大的那个类型，所有操作数都会升级为大类型，最终结果一定是范围最大的那个类型
  */
  public static void main(String[] args) {
    byte b1 = 10;
    short s1 = 20;
```

```
    int i1 = 30;
    long l1 = 40;
    float f1 = 1.1f;
    double d1 = 2.2;

    //s1 = b1 + s1;
    s1 = (short)(b1 + s1); // 右面是int型
    //i1 = l1 + i1; // 右面是long型
    i1 = (int)(l1 + i1); // ladd
    i1 = (int)l1 + i1; // iadd

    l1 = (long)(f1 + l1);
    d1 = d1 + f1;

    d1 = f1 + b1 + s1;
    d1 = d1 + f1 + i1 + l1;

  }
}
```

char数据类型

```
class CharTest2 {

  public static void main(String[] args) {
    char c1 = ' ';
    char c2 = '\r'; // 回车，光标回到行首
    char c3 = 13;
    char c4 = '\n'; // 换行，光标进入新行
    char c5 = 10;
    char c6 = '\t'; // 制表符 码值为9
    char c7 = '\b'; // 回退符 码值为8

    System.out.println("abc\b\b123456"); // a123456
```

```
        System.out.println("abcdefg\r345"); // 345defg
        System.out.println("abcd\n12345");

        //char c8 = ''; // 非法
        char c8 = 0;
        String s1 = ""; // 空串
    }
}

public class CharTest {

    public static void main(String[] args) {
        // char占用2个字节，用于保存一个字符，字符字面量使用''包围
        // 0~65535 ，字符数据类型底层就是非long整数，注意 ： 它没有负数. char和byte,short不兼容，char可以被int兼容
        char c1 = 'a'; // 编译器在处理时先查找表，'a' => 97，把97写入到c1变量中
        char c2 = 'b';
        char c3 = 'A'; // 65
        char c4 = '3'; // 51
        char c5 = '我';
        char c6 = '你';

        System.out.println(c1); // 打印时，虽然存的是97，但是它的类型是char，要反向查表 根据97 => 'a'，最后打印的是字符
        System.out.println(c2);
        System.out.println(c3);
        System.out.println(c4);
        System.out.println(c5);
        System.out.println(c6);

        System.out.println((int)c1);
        System.out.println((int)c2);
        System.out.println((int)c3);
        System.out.println((int)c4);
        System.out.println((int)c5);
        System.out.println((int)c6);
```

```
    char c7 = 20000;
    System.out.println(c7);
    c7 = (char)(c7 + 500); // 非long整数运算了.
    System.out.println(c7);
  }
}
```

boolean数据类型

```
public class BooleanTest {

  public static void main(String[] args) {
    // boolean占用1个字节, 底层使用1表示true, 0表示false
    boolean b1 = true;
    boolean b2 = false;
    boolean b3 = b1;

    System.out.println(b1);
    System.out.println(b2);
    System.out.println(b3);

    // boolean数据类型和其他基本数据类型全部不兼容, 强转也不行.
    //boolean b4 = (boolean)1;

  }
}
```

String字符串

```
/*
```

```
  字符串 ： 是引用型，所以它的变量将会保存字符串对象的地址
  字符串对象是内容不可以改变的对象.
  字符串可以和任意数据拼接，拼接的结果是产生一个新的字符串对象，其内容就是原来的串和新内容的合成效果

  基本数据类型 => String
  String s = "" + xxx; // "xxx"
*/

class StringTest2 {

  public static void main(String[] args) {
    //String str1 = (String)4;        //判断对错:
    String str1 = "" + 4;
    String str2 = 3.5f + "";            //判断str2对错:
    System.out.println(str2);        //输出: 3.5
    System.out.println(3+4+"Hello!");      //输出: 7Hello
    System.out.println("Hello!"+3+4);      //输出: Hello34
    System.out.println("Hello!"+(3+4));      //输出: Hello
    System.out.println('a'+1+"Hello!");    //输出:  98Hello
    System.out.println((char)('a'+1)+"Hello!");    //输出:  98Hello
    System.out.println("Hello"+(char)('a'+1));            //输出:

  }
}

public class StringTest {


  public static void main(String[] args) {
    String s1 = "abcd";
    System.out.println(s1);

    s1 = s1 + 200; // "abcd200"
    boolean b = false;
    s1 = s1 + b; // "abcd200false"
```

```
        s1 = s1 + 3.22;
        s1 = s1 + '好';
        s1 = s1 + "别的内容";

        System.out.println(s1);

        String s2 = ""; // 空串，有对象,但是没有内容，更好用
        String s3 = null; // 空，什么也没有

        s2 = s2 + 999;
        s3 = s3 + 999;

        System.out.println(s2);
        System.out.println(s3);

        // int => String
        int n1 = 39284;
        //String s4 = n1;
        String s4 = "" + n1;
        System.out.println(s4);
    }
}
```

进制

285471

2 * 10 ^ 5 +

8 * 10 ^ 4 +

5 * 10 ^ 3+

4 * 10 ^ 2 +

7 * 10 ^ 1+

1 * 10 ^ 0

权值计算是以10为底的n次方, 这样的数称为10进制数

X进制 : 权值计算时以X为底的n次方, X进制没有X, 逢X进1

二进制 : 权值以2为底的n次方, 没有2, 逢2进1

十六进制 : 权值是以16为底的n次方, 没有16, 逢16进1

0x123

1 * 16 ^ 2 + // 256+32+3

2 * 16 ^ 1 +

3 * 16 ^ 0 = 291

0b0010 1101

0 * 2 ^ 7 // 128

0 * 2 ^ 6 // 64

1 * 2 ^ 5 // 32

0 * 2 ^ 4 // 16

1 * 2 ^ 3 // 8

1 * 2 ^ 2 // 4

0 * 2 ^ 1 // 2

1 * 2 ^ 0 // 1

32 + 8 + 4 + 1 = 45

练习 :

0x6211

6 * 16 ^ 3 + // 24576

2 * 16 ^ 2 + // 529

1 * 16 ^ 1 +

1 * 16 ^ 0 = 25105

char c = '\u6211'; // '我' : 25105

0b0110 1101

2^6 // 64

2^5 // 32

2^3 // 8

2^2 // 4

2^0 // 1

= 96 + 8 + 4 + 1

= 109

无论是什么进制其实是一 回事.

0x67

6*16+7 = 103

0b0110 0111

2^6 + 2^5 + 2^2 + 2^1 + 2^0 = 64 + 32 + 7 = 103

103

十进制	二进制	十六进制
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5

6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

一个16进制数正好可以对应4个位的2进制数.

2个16进制数正好对应8bit, 就是一个字节byte

1011 0100 => 0xB4

0xEF => 1110 1111

8421求和

0110 1101 0111 1011 1110 1101 1001 0111 =>

0x6D7BED97

0x8526CA4D

1000 0101 0010 0110 1100 1010 0100 1101

练习：

0x9AC2DB57

1001 1010 1100 0010 1101 1011 0101 0111

1101 1110 1011 0101 0111 0110 1010 0100

0xDEB576A4

计算机中数据的所有的处理及存储都是以二进制补码形式处理的.

一个数在计算机使用它的最高位(最左面)为符号位, 符号位如果为0表示这个数是正数, 符号位如果为1表示这个数是负数.

正数的补码就是它自身.

负数的补码是它的相反数全部按位取反, 再加1得到.

0011 1101 是一个正数

0x3D => 16*3+13=>61, 所以0011 1101就是+61在计算机中的样子

-10的补码

10 => 0000 1010

全部取反 => 1111 0101

+1 => 1111 0110

1111 0110就是-10

1011 0110 => 是一个负数的补码, 是多少?

-1 => 1011 0101

取反 => 0100 1010 => 0x4A => 16*4+10 => 74

所以 1011 0110是-74

分析 :

0101 1011 => 是一个正数 => 0x5B => 16*5+11 = 91

1100 0110 => 是一个负数, 负多少?

-1 => 1100 0101

取反 => 0011 1010 => 0x3A => 16*3+10 = 58

1100 0110 是 -58

byte型整数最大值 :

0111 1111 => 0x7F => 7*16+15 => 127

1111 1111 => 是一个负数 负多少? 负数最大

-1 => 1111 1110

取反 => 0000 0001 => 1

1111 1111是-1

byte型最小值 :

1000 0000 => 是一个负数, 负多少

-1 => 0111 1111

取反 => 1000 0000 => 0x80 => 16*8 => 128

所以1000 0000是 -128

0000 0000 => 正数最小

short型最大值

0111 1111 1111 1111 => 0x7FFF

short型最小值

1000 0000 0000 0000 => 0x8000

int型最大

0111 1111 1111 1111 1111 1111 1111 1111 => 0x7FFFFFFF

int型最小

1000 0000 0000 0000 0000 0000 0000 0000 => 0x80000000

long型最大

0x7FFFFFFF_FFFFFFFF

long型最小

0x80000000_00000000

char型最大

1111 1111 1111 1111 => 0xFFFF

char型最大

0000 0000 0000 0000 => 0x0000

最大值最小值代码

```
public class IntegerRanger {

  public static void main(String[] args) {
    byte bMax = 0b00000000_00000000_00000000_01111111; // 0x7F
    //byte bMin = -0b00000000_00000000_00000000_10000000; // 0x80
    byte bMin = (byte)0b00000000_00000000_00000000_10000000; // 0x80

    System.out.println("bMin : " + bMin);
    System.out.println("bMax : " + bMax);

    short sMax = 0b00000000_00000000_01111111_11111111; // 0x7FFF
    short sMin = (short)0b00000000_00000000_10000000_00000000;
    System.out.println("sMin : " + sMin);
    System.out.println("sMax : " + sMax);

    int iMax = 0x7FFFFFFF;
    int iMin = 0x80000000;

    System.out.println("iMax : " + iMax);
    System.out.println("iMin : " + iMin);

    long lMax = 0x7FFFFFFF_FFFFFFFFL;
    long lMin = 0x80000000_00000000L;

    System.out.println("lMax : " + lMax); //922 3372 0368 5477 5807
    System.out.println("lMin : " + lMin);
  }
}
```

每日一考_day03

1. 变量按照数据类型来分, 分为基本数据类型变量和引用数据类型变量, 请写出基本数据类型变量和引用数据类型变量的区别.

   数据类型的作用: 1) 决定空间大小 2) 决定了空间中可以保存什么数据 3) 决定了数据可以做什么

   基本数据类型(primitive) : 变量存储的是数据值本身(自己)

   引用数据类型(reference) : 变量存储的是地址值(别人)

2. 列出变量使用注意事项(6点), 简单说明

   1）使用前要先声明；先声明后使用, 只有声明了才有空间可以使用.

   2）使用前进行初始化赋值；因为之前遗留的数据不可靠.

   3. 变量有其数据范围

   4）变量有其作用范围，声明语句所属{}内；

   4）变量有明确的的数据类型和变量名

   5）在同一个范围内, 变量不能重复声明

3. 基本数据类型有8种, 写出8种基本数据类型, 并用的16进制形式写出所有整数数据类型的最小值和最大值

   数值型(number) :

   整数 :

   1）byte 1 0b1000 0000 => 0b0111 1111 : 0x80 0x7F

   0b1111 1111 : 负数最大值, 0000 0000 : 正数最小值

   2) short 2 0x8000 0x7FFF

   3) int 4 0x80000000 0x7FFFFFFF

   4) long 8 0x8000000000000000 0x7FFFFFFFFFFFFFFF

   5) char 2 0x0000 0xFFFF

   浮点数

   6) float 4

7) double 8

数值型之间可以随意互转 :

左值 = 右值，左值的数据类型的范围大于等于右值的数据类型的范围时

左值 = (目标类型)右值，左值的数据类型的范围小于右值的数据类型的范围时, 必须使用强制类型转换

double > float > long > int > short > byte

char(没有负数)

布尔型 : 没有范围一说, 只允许2个固定的值, true, false

8）boolean 1

4. 判断下列的带( )行的对错

int i1 = 20;

short s1 = i1; (F)

char c1 = 'a' // 97; (T)

char c2 = '我' - '你'; (T) // 编译器对待常量是放心的.

char c3 = (char)(c1 - 32); //可以达到小写变大写的效果( ) 对 编译器对待变量是小心的

float f1 = i1;(T)

long l1 = 234234239933;(F)

f1 = l1 * 20;(T)

double d1 = .342;

d1 = i1 * f1 * l1;(T)

l1 = f1 / 10000;(F)

boolean b1 = (boolean)1;(F)

5. 相同字面量十六进制表示的数比十进制要大, 对或错? 为什么?

0x30, 30， 0x30大

0x8, 8 一样大

错,

表示的数的绝对是大 ， 因为十六进制的权值大。

0x80, 80 ：

如果0x80是一个非byte型整数. 0x80大于80

如果0x80是一个byte型整数, 0x80小于80

变量 :

内存中的一块被命名且有特定数据类型约束的空间,

此空间中可以保存一个数据类型范围内的数据, 此空间中的数据可以在数据范围内随意改变.

常量 :

内存中的一块空间, 此空间中可以保存一个数据, 此空间中的数据不能改变.

变量分类

变量按照数据类型来分 :

1) 基本数据类型(primitive) : 存自己

2) 引用数据类型(reference) : 存对象地址(对象在内存中的某个字节的编号) (主流是64位, 所以是8字节)

类, 接口, 枚举, 注解, 数组

String s1 = "abc";

变量按照声明语句位置来分：

1) 局部变量(local)：声明在方法中的变量，范围小，寿命短.

2) 成员变量(member)：声明在类中方法外的变量，范围大，寿命长

全局的，必须要有访问控制修饰符控制它

寿命：实例变量(和对象一样长)，类变量(和类模板一样长)

位运算

<< 一个二进制数左移一些位，右面会补相应的0，左面移出去的位丢弃.

'>> 有符号右移，一个二进制数右移一些位，右面移出丢的丢弃，左面补符号位

'>>> 无符号右移，一个二进制数右移一些位，右面移出丢的丢弃，左面补0

& 按位与，2个二进制数对应的位上 只要有0结果就是0，除非全是1结果才是1，结果变小

| 按位或，2个二进制数对应的位上 只要有1结果就是1，除非全是0，结果才是0，结果变大.

^ 按位异或,2个二进制数对应的位上，只要不一样就是1，一样就是0

~ 是一个单目(一元)，让一个二进制数的所有位全部取反，正数变负，负数变正.

```
class BitOperator2 {

  public static void main(String[] args) {
    int n1 = 0x5B;
    int n2 = 0xDA;
```

```
    // & : 只要有0就是0
    // 0000 0000 0000 0000 0000 0000 0101 1011 &
    // 0000 0000 0000 0000 0000 0000 1101 1010 =
    // 0000 0000 0000 0000 0000 0000 0101 1010 => 0x5A => 5*16+10 =>
90
    System.out.println(n1 & n2); // 90

    // | : 只要有1就是1
    // 0000 0000 0000 0000 0000 0000 0101 1011 |
    // 0000 0000 0000 0000 0000 0000 1101 1010 =
    // 0000 0000 0000 0000 0000 0000 1101 1011 => 0xDB => 13*16+11 =>
219
    System.out.println(n1 | n2);

    // ^ : 只要不一样就是1
    // 0000 0000 0000 0000 0000 0000 0101 1011 ^
    // 0000 0000 0000 0000 0000 0000 1101 1010 =
    // 0000 0000 0000 0000 0000 0000 1000 0001 => 0x81 => 8*16+1 =>
129
    System.out.println(n1 ^ n2);

    // 0000 0000 0000 0000 0000 0000 0101 1011 ~
    // 1111 1111 1111 1111 1111 1111 1010 0100 => 这就是结果, 这是负多少
    // -1 : 1111 1111 1111 1111 1111 1111 1010 0011
    // 取反 : 0000 0000 0000 0000 0000 0000 0101 1100 => 0x5C =>
5*16+12= 92
    System.out.println(~n1);  // -92
  }
}

public class BitOperator {

  public static void main(String[] args) {
    // 左移, 右面补0
```

```
        // 有符号右移，左面补符号位，无符号右移，左面补0
        int n = 0x6E; // 0000 0000 0000 0000 0000 0000 0110 1110
        System.out.println(n);
        System.out.println(n << 3); // 0 0000 0000 0000 0000 0000 0110
    1110 000
        //System.out.println(n << 3); // 0000 0000 0000 0000 0000 0011
    0111 0000 => 0x370 => 3*256+7*16 = 880
        System.out.println(n >> 5); // 0000 0000 0000 0000 0000 0000 0000
    0011 => 3

        System.out.println(-1 >> 3); // 有符号右移 -1
        System.out.println(-1 >>> 1); // 无符号右移
      }
    }
```

算术运算符

+	**正号 取一个数自身**	**+3**	**3**
-	负号 取一个数相反数	b=4; -b	-4

+	**加**	**5+5**	**10**
-	减	6-4	2
*	乘	3*4	12
/	除 2个整数相除, 结果还是整数,小数直接丢弃	5/3	1

int x=3510;

x = x / 1000 * 1000; // x:3000

%	**取模, 取余, 2个数相除, 商丢弃, 取余数**	**7%5**	**2**
	左面如果是负数, 结果有可能是负数, 右面如果是负数, 则忽略负号		

5 % -2 => 1

-5 % 2 => -1

8 % 5 : 3

9 % 5 : 4

10 % 5 : 0

11 % 5 : 1

% 典型应用 :

M % N : 结果就是让一个未知数M落入一个已知的N的范围内. [0~N-1]

M % N : 结果如果为0, 说明M可以被N整除的, 如果不为0, 说明M不能被N整除.

M % 2 : 结果如果为0, 说明M是偶数, 如果结果非0 说明M是奇数

++ : 自增1

```
int a = 10;

a++; // 11

a = 10;

++a; // 11
```

前加加和后加加对于a没有区别.

++a : 前加加, 先加后用

a++ : 后加加 : 先用后加

-- : 自减1

```
int a = 10;

a--; // 9

a = 10;

--a; // 9
```

自加自减代码

```
class OperatorTest5 {

  public static void main(String[] args) {
    // 赋值运算符，从右向左，因为它的优先级最低，右面没有搞定，不能向左走.
    int a;
    // 赋值运算表达式也有它自己的值，这个值就是它的最右值
    //a = 20;
    //System.out.println(a); // 打印a变量中的值的副本

    System.out.println(a = 20); // 打印赋值表达式本身的值.

    //int b,c; // 这样的写法不推荐!!!!

    int b;
    int c;

    a = b = c = 100;  // 连续赋值，从右向左.
```

```
    short s = 3;
    //s = s + 2; // 右面会真的升级为int型
    s += 2; // 累操作是安全的.
    s *= 2.8;


    byte b2 = 127;
    b2 += 2; // -127

    System.out.println(b2);

  }
}

class OperatorTest4 {

  public static void main(String[] args) {
    int n = 10;
    // 前加加, 前减减, 没有临时空间, 后加加,后减减, 有临时空间的概念
    n = n++ + n++ + ++n + n-- +  --n + n--;

    System.out.println("n : " + n); // 69

    System.out.println('*' + '\t' +'*'); // 3个整数相加
    System.out.println("*" + '\t' +'*'); // 产生新串

  }
}

class OperatorTest3 {

  public static void main(String[] args) {
    int n = 10;
    int m = n++;
    /*
    3: iload_1
```

```
        4: iinc          1, 1
        7: istore_2
    */

    n = 10;
    m = ++n;
    /*
    11: iinc          1, 1
    14: iload_1
    15: istore_2
    */

    for (int i = 0; i < 10; i++) {} // 先用后加，效率稍差
    for (int i = 0; i != 10; ++i) {} // 先加后用，效率稍高，极致效率
  }
}

class OperatorTest2 {

  public static void main(String[] args) {
    int n = 10;
    n = n++;

    /*
    3: iload_1
        4: iinc          1, 1
        7: istore_1
     */
    System.out.println("n : " + n);

    n = 10;
    n = ++n;

    //23: iinc          1, 1
    //26: iload_1
    //27: istore_1
```

```
    System.out.println("n : " + n);
  }
}

public class OperatorTest {

  public static void main(String[] args) {
    int n = 10;
    // 赋值符号的优先级是最低的，所以赋值总是最后执行.
    int m = n++; // m:10 n : 11 后加加，先用后加，1）把n中的老值，放入临时空
间，2）再把n自加成11，3）把临时空间中的值10写入m
    System.out.println("m : " + m + ", n : " + n);

    /*
    n = 10;
    m = ++n; // m:11 n : 11 前加加，先加后用
    System.out.println("m : " + m + ", n : " + n);
    */

  }
}
```

累操作：不会引起数据类型的变化，使用起来很方便

比较运算符：

结果总是一个boolean.

boolean虽然简单，但是有大用途，控制分支，控制循环.

比较大小操作,只适用于基本数据类型中的数值型之间(number)

其他的包括 boolean及所有对象都不能比较大小

<	**小于 4<3 false**
>	大于 4>3 true
<=	小于等于 4<=3 false
>=	大于等于 4>=3 true

3 < x < 6 , 因为左面的3 < x会产生一个boolean, 而boolean不能比较大小.

3 < x == false 这是可以的.

相等或不等操作适用于所有数据类型之间, 但是如果比较的是2个引用或2个对象时, 比较的就是2个地址值.

==	**相等于 4==3 false**
!=	不等于 4!=3 true

逻辑运算符：

只能适用于boolean类型之间, 结果总是一个新的boolean.

& 逻辑与, && 短路与

他们都是并且的逻辑，a 并且 b. 只要有false, 结果一定是false. 只要有假就是假, 除非全是真结果才真.

2者区别在于 & 就是按位与

&&称为短路与：

a && b . 如果a是true, 需要继续查看b, 如果b是true, 结果为true, 如果b为false, 结果为false

a && b . 如果a是false, 结果100%是false, 完全不需要再处理b了, 称为短路. 提升效率, 逻辑更严密

| 逻辑或, || 短路或

他们都是或者的逻辑, a 或者 b. 只要有true, 结果 一定是true, 只要有真就是真, 除非全是假结果才假.

|| 称为短路或

a || b . 如果a是false, 需要继续查看b. 如果b是false, 结果为false, 如果b是true, 结果就是true.

a || b . 如果a是true, 结果100%就是true, 完全不需要再处理b, 称为短路.

三元运算符, 三目运算符

表达式1和表达式2的数据类型完全一致.

变量 = 布尔表达式 ? 表达式1 : 表达式2;

```
// 获取两个数中的较大数
class ThreeOperator2 {

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

```
    int n1 = Integer.parseInt(args[0]); // 把命令行参数中的第1个字符串转换为真int值
    int n2 = Integer.parseInt(args[1]); // 把命令行参数中的第2个字符串转换为真int值

    int max = n1 > n2 ? n1 : n2;

    System.out.println("max : " + max);
  }
}

class ArgsTest {

  // 在这里会真的用到3个命令行参数时, 执行主类时后面必须跟3个或以上的参数列表.
  public static void main(String[] args) {
    System.out.println(args[0]); // 打印第1个命令行参数字符串
    System.out.println(args[1]); // 打印第2个命令行参数字符串
    System.out.println(args[2]); // 打印第3个命令行参数字符串

    // int => String
    int n1 = 234234;
    //String s1 = n1; // int不能直接赋值给String
    String s1 = "" + n1;
    // String => int
    String s2 = "98234";
    //int n2 = s2; // String不能直接赋值给int
    int n2 = Integer.parseInt(s2); // 把String内容转换为真的int值
    System.out.println("n2 : " + n2);
  }
}

// 获取两个数中的较大数
public class ThreeOperator {

  public static void main(String[] args) {
    int n1 = 100;
```

```
        int n2 = 520;

        int max = n1 > n2 ? n1 : n2;

        System.out.println("max : " + max);
    }
}
```

每日一考_day04

1. 列出变量的使用注意事项(至少6点)

   1.变量必须先声明

   2.变量必须命名有数据类型

   3.变量需要初始化

   4.变量具有作用域 声明语句存在的{}

   5.同一个作用域名称不能相同

   6.变量存放的数据是可在其数据范围内变化

2. 变量分类有两种分法, 第一种是按数据类型来分,另外一种是按照声明位置来分, 每一种又各分为哪些种类型. 各有什么特点?

   基本数据类型 数值和布尔　存的是自己的值

   引用数据类型 引用的对象地址，堆内存中的实际对象存储的位置 （首地址）8字节

   按照声明位置来分

   局部变量：声明语句属于方法, 范围小, 寿命短

   成员变量：类里面所有方法体都可以，特殊的修饰符private 其他类也可以访问 范围大, 寿命长

3. 计算下列结果, 分析过程, 只需要计算到十六进制形式即可.

0110 1011

0101 1101

0100   1001

0X49

0111 1111

0X7F

0011 0110

0X36

```
byte a = 0x6B;
byte b = 0x5D;
System.out.println(a & b);
System.out.println(a | b);
System.out.println(a ^ b);
```

4. 运算符%的作用是什么? 有什么实际的应用?

   %两数相除取余数

   1. M % N [0-N-1] 让一个未知数据M落入已知范围N之内
   2. M % N 如果结果为0, 说明M被N整除.
   3. M % 2 如果结果为0, 说明M是偶数, 结果如果是非0, 说明M是奇数

5. 判断:
   if else if else if else 语句中, 如果同时有多个条件都为true, 那么将会有多个语句块被执行

   错的 当第一个执行以后后面就不会去执行

第3章 流程控制

分支：

根据 条件, 选择性地执行某段代码的功能

```
if (布尔表达式) {
    语句块;  // 当布尔表达式为true时执行.
}
```

2 分支

```
if (布尔表达式) {
    语句块; // 当布尔表达式为true时执行
} else {
    语句块2; // 当布尔表达式为false时执行
}
```

多分支

```
if (布尔表达式1) {
    语句块1;
} else if (布尔表达式2) {
    语句块2; // 布尔表达式1为false 并且 布尔表达式2为true时
```

} else if (布尔表达式3) {

    语句块3; // 布尔表达式1为false, 并且 布尔表达式也为false 并且 布尔表达式3 为true.

} else if (布尔表达式4) {

    语句块4;

} else {

    // 上面的所有条件都为false时执行.

}

分支代码

```
class IfTest5 {

  public static void main(String[] args) {
    int n = 10;

    // 条件的顺序按重要程度排列
    if (n == 1) {
      System.out.println("n == 1");
    } else if (n >= 2 && n == 10) { // 先决条件
      System.out.println("n >= 2"); // 它的执行不再简单
      System.out.println("n == 10"); // 先决条件和直接条件是并且的关系
    } else if (n == 3) {
      System.out.println("n == 3");
    } else if (n == 5) {
      System.out.println("n == 5");
    } else {
      System.out.println("else ...");
```

```
    }

    System.out.println("after if else if else if ....");
  }

}

class IfTest4 {

  public static void main(String[] args) {
    int n = 10;

    // 条件的顺序按重要程度排列
    if (n == 1) {
      System.out.println("n == 1");
    } else if (n >= 2) { // 先决条件
      System.out.println("n >= 2");
      if (n == 10) {
        System.out.println("n == 10"); // 先决条件和直接条件是并且的关系
      }
    } else if (n == 3) {
      System.out.println("n == 3");
    } else if (n == 5) {
      System.out.println("n == 5");
    } else {
      System.out.println("else ...");
    }

    System.out.println("after if else if else if ....");
  }

}

class IfTest3 {

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

```
    int n = 20;
    // 多个语句块之间绝对互斥
    if (n == 1) {
      System.out.println("n == 1");
    } else if (n == 2) {
      System.out.println("n == 2");
    } else if (n == 3) {
      System.out.println("n == 3");
    } else if (n == 5) {
      System.out.println("n == 5");
    } else if (n == 10) {
      System.out.println("n == 10");
    } else {
      System.out.println("else ...");
    }

    System.out.println("after if else if else if ....");
  }

}

class IfTest2 {

  public static void main(String[] args) {
    /*
    if (布尔) {
      语句块1;
    } else {
      语句块2;
    }*/
    int n = 5;
    // 语句块之间是绝对互斥. 一定是一个执行另一个不执行.
    if (n == 5) {
      System.out.println("n == 5");
    } else {
      System.out.println("n != 5");
```

```
    }

    System.out.println("after if else ...."); // 无条件执行
  }
}

public class IfTest {

  public static void main(String[] args) {
    /*
    if (布尔) {
      语句块; 如果布尔为true
    }*/
    int n = 4;

    if (n == 3) {
      System.out.println("n == 3"); // 有条件执行.
    }

    System.out.println("after if ..."); // 无条件执行语句
  }
}
```

switch

适用于变量的等值性判断分支时.

```
int n =
        // 多个语句块之间绝对互斥
        if (n == 1) {
            System.out.println("n == 1");
        } else if (n == 2) {
            System.out.println("n == 2");
        } else if (n == 3) {
```

```
            System.out.println("n == 3");
        } else if (n == 5) {
            System.out.println("n == 5");
        } else if (n == 10) {
            System.out.println("n == 10");
        } else {
            System.out.println("else ...");
        }
```

switch 的底层逻辑就是对某个变量中可能的值的所有情况的穷举和列表.

case 后面必须是常量,

常量包含2种,

一种是字面量, 所见即所得. 900, 20, 'a', '你', "驾照苯甲川基苯并呋喃酮" ,

另一种就是final修饰的量

switch (必须是变量) : 变量的类型是非long整数. 字符串, 枚举

```
switch (用于分支变量) {
    case 常量1 : // if (变量 == 常量1)
        语句块;
        break; // 中断整个switch结束 , 瞬间跑到switch后面的语句
    case 常量2 : // else if (变量 == 常量2)
```

```
        语句块;
        break;
    case 常量N :
        .....
    default : // 相当于 else 作用
        语句块;
        break;
}
```

分支代码

```
class SwitchTest2 {

  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    // 如果在case中省略了break, 会导致 出现穿透(fall through), 如果某个case一旦进入了, 后面的case就形同虚设
    // 直接到遇到break或结束为止, 适当的穿透和break配合可以达到分组的效果.
    switch (n) {
      case 1 :
        System.out.println("case 1...");
        //break; // 中断整个switch
      case 2 :
        System.out.println("case 2...");
        break;
      case 3 :
        System.out.println("case 3...");
```

```
        //break;
      case 5 :
        System.out.println("case 5...");
        break;
      case 10 :
        System.out.println("case 10...");
        //break;
      default :
        System.out.println("default");
        break;
    }

    System.out.println("after switch");
  }
}

public class SwitchTest {

  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);

    /*
    if (n == 1) {
      System.out.println("n == 1");
    } else if (n == 2) {
      System.out.println("n == 2");
    } else if (n == 3) {
      System.out.println("n == 3");
    } else if (n == 5) {
      System.out.println("n == 5");
    } else if (n == 10) {
      System.out.println("n == 10");
    } else {
      System.out.println("else ...");
    }
    */
```

```java
    /*
  switch 的底层逻辑就是对某个变量中可能的值的所有情况的穷举和列表.
  case 后面必须是常量,
    常量包含2种,
      一种是字面量, 所见即所得. 900, 20, 'a', '你', "驾照苯甲川基苯并呋喃酮" ,
      另一种就是final修饰的量
  switch (必须是变量) : 变量的类型是非long整数. 字符串, 枚举
  */
    switch (n) {
      case 1 :
        System.out.println("case 1...");
        break; // 中断整个switch
      case 2 :
        System.out.println("case 2...");
        break;
      case 3 :
        System.out.println("case 3...");
        break;
      case 5 :
        System.out.println("case 5...");
        break;
      case 10 :
        System.out.println("case 10...");
        break;
      default :
        System.out.println("default");
        break;
    }

    System.out.println("after switch");

  }
}
```

循环

在条件满足的情况下, 反复地执行某段代码的功能

循环的组成部分

1) 初始化语句 (init) : 作准备工作, 初始化工作只执行一次

2) 循环条件语句(test) : 最重要的, 决定生死

3) 循环体(body) : 被反复执行的语句块

4) 迭代语句(alter) : 让循环条件趋于假.

while 循环

```
初始化语句;
while (循环条件) {
    循环体;
    迭代语句;
};
```

do while 循环

初始化语句;

do {

    循环体;

    迭代语句;

} while (循环条件); // 后面的;必须加上

while 循环是先判断后循环, 循环次数: 0 ~N次

do while 循环是 先循环再判断, 循环次数 : 1 ~ N次

for 循环 : 最严谨, 最好的循环

for (初始化语句A; 循环条件B; 迭代语句C) {

    循环体D;

}

A B D C B D C B D ..... C B D .... B

迭代语句C是每次循环的开始..

循环代码

```
/**
  while和 do while 的正常写法应该是
```

```
  while (循环条件) {
    循环体;
  }

  do {
    循环体 ;
  } while (循环条件);

  // 算法 : 规律, 尤其在循环中的规律. 大概率要和因子i有关系.

  内循环次数 = k * i + b; // k称为斜率, 如果k为正数, 内循环次数增加的趋势, 而k为负数, 内循环次数减少的趋势.

打印5行的等腰3角形
        i       空格      *
    *     0     4     1
   ***      1     3     3
  *****     2           2     5
 *******    3     1       7
*********     4     0     9

*/
class LoopTest14 {

  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      for (int j = 0; j < 10 - i; j++) {
        System.out.print("*");
      }
      for (int j = 0; j < i; j++) {
        System.out.print("#");
      }
      System.out.println();
    }
  }
}
```

```java
class LoopTest13 {

  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      for (int j = 0; j < -2 * i + 30; j++) { // 内循环次数 = -1 * i +
10;
        System.out.print("*");
      }
      System.out.println();
    }
  }
}

class LoopTest12 {

  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      for (int j = 0; j < 3 * i + 8; j++) { // 内循环的次数随着i的变化而变
化.  内循环次数 = 1 * i + 0;
        System.out.print("*");
      }
      System.out.println();
    }
  }
}

class LoopTest11 {


  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    int m = Integer.parseInt(args[1]);

    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
```

```
        System.out.print("J:" + j + "   ");
      }
      System.out.println(); // 有换行.
    }
  }
}

class LoopTest10 {

  public static void main(String[] args) {
    for (int i = 0; i < 5; i++) { // 外循环控制行
      for (int j = 0; j < 3; j++) { // 内循环控制列
        System.out.print("j : " + j + "      "); // 3次 * 5趟 = 15
      }
      System.out.println(); // 5次, 直接受外循环控制, 这是一个纯粹的换行
    }
  }
}

class LoopTest9 {

  public static void main(String[] args) {
    // 求100以内能被7整除的数的平均值
    int sum = 0;
    int count = 0; // 计数器
    for (int i = 0; i < 100; i++) {
      if (i % 7 == 0) {
        System.out.println("i : " + i); // 有条件执行
        sum += i; // 有条件求和
        count++; // 有条件计数
      }
    }
    System.out.println("sum : " + sum);
    System.out.println("count : " + count);
    int avg = sum / count;
    System.out.println("avg : " + avg);
```

```
  }
}

class LoopTest8 {

  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);

    int result = 0;
    for (int i = 0; i < n; i++) { // 循环次数 : 条件右面的值
      result += i;
    }

    System.out.println("result : " + result);
    //System.out.println("i : " + i); 不要乱访问循环中的i
  }
}

class LoopTest7 {

  public static void main(String[] args) {
    int result = 0;
    /*
    for (初始化语句; 循环条件; 迭代语句) {
      循环体;
    }*/
    //int i = 1;

    //i = 3;

    //for (; i <= 5; i++) { // 把i声明在外面, 万一被别人修改了, 循环就会紊乱...
    for (int i = 1; i <= 5; i++) { // 循环次数 : 条件右面的值 - 初始值, 如果有=再加1
      result += i;
    }
```

```
    System.out.println("result : " + result);
    //System.out.println("i : " + i); 不要乱访问循环中的i
  }
}

class LoopTest6 {

  public static void main(String[] args) {
    //while (1 == 1) { // 死循环, 真的结束不了.
    boolean flag = true;
    while (flag) { // 无限循环
      System.out.println("1 == 1");
    }

    do {
      System.out.println("ddo odododod.");
    } while (flag);

    System.out.println("after loop");
  }
}

class LoopTest5 {

  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);

    int result = 0;

    int i = 0;
    do {
      result += i;
      i++;
    } while (i < n); // 循环次数 : 条件右面的值
```

```
    System.out.println("result : " +  result);
    System.out.println("i : " +  i);
  }
}

class LoopTest4 {

  public static void main(String[] args) {
    /*初始化语句;
    do {
      循环体;
      迭代语句;
    } while (循环条件); // 不要丢了;
    */
    int result = 0;

    int i = 1;
    do {
      result += i;
      i++;
    } while (i <= 5); // 循环次数 : 条件右面的值 - 初始值, 如果有=再加1

    System.out.println("result : " +  result);
    System.out.println("i : " +  i);
  }
}

class LoopTest3 {

  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);

    int result = 0;
    int i = 0;
    while (i < n) { // 循环次数 : 循环条件右面的值
      result += i;
```

```
      i++;
    }
    System.out.println("result : " + result); // 求 1 + 2 + 3... + 5和
    System.out.println("i : " + i);
  }
}

class LoopTest2 {

  public static void main(String[] args) {
    int result = 0;
    int i = 25; // 初始化语句, 因为i称为循环因子(factor), 循环因子的变量名永远是i
    while (i <= 125) { // 循环次数 : 循环条件右面的值 - 因子的初始值, 如果有=, 再加1次.
      result += i; // 循环体
      i++; // 迭代语句
    }
    System.out.println("result : " + result); // 求 1 + 2 + 3... + 5和
    System.out.println("i : " + i);
  }
}

public class LoopTest {

  public static void main(String[] args) {
    /*
    初始化语句;
    while (循环条件) {
      循环体;
      迭代语句;
    }*/
    int result = 0;
    int i = 1; // 初始化语句, 因为i称为循环因子(factor), 循环因子的变量名永远是i
    while (i <= 5) { // 循环条件
```

```
            result += i; // 循环体
            i++; // 迭代语句
        }
        System.out.println("result : " + result); // 求 1 + 2 + 3... + 5和
        System.out.println("i : " + i);
    }
}
```

每日一考_day05

1. 变量分类有两种分法, 第一种是按数据类型来分,另外一种是按照声明位置来分,

   每一种又各分为哪些种类型. 各有什么特点?

   基本数据类型 (保存数据自身)

   数值型 : 所有算术运算, 比较大小的运算, 比较相等或不等

   整数 : byte, short, char, int , long. : 支持位运算

   浮点数 : float, double

   布尔型 : 比较相等或不等运算, 逻辑运算

   boolean

   引用型 (保存对象地址) , 占用8字节(64位jdk) 内存地址 : 每个字节的空间都有唯一编号.

   局部变量 : 声明在方法中的变量 , 范围小, 寿命短

   成员变量 : 声明在类中方法外的变量 范围大, 寿命长

2. 在switch结构中, switch()括号中的要求具体是什么? case后面的要求又是什么?

   switch (变量) ; 数据类型必须是非long整数, 字符串, 枚举

case； 后面必须是常量, 常量有2种, 字面量, final量

break；不是必须的, 但是没有break会形成穿透.

3. while循环和do while循环的区别是什么

   while 循环0-n次, 先判断后循环

   do while至少循环一次, 先循环后判断

4. for循环的结构是什么? 执行流程是如何?

   for （初始声明A； 条件判断B； 迭代语句C）{

   循环体 D

   }

   ABD CBDCBD.....B

   迭代语句是每次循环的开始

5. 写程序, 打印一个倒直角三角形.

```
public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    // 打印n行的倒直角三角形
    for(int i = 0; i < n; i++) {

        for(int j = 0; j < -1 * i + n; j++ ){
             System.out.print("*");
        }
        System.out.println(" ");
    }
}
```

特殊控制语句

```
/**
  break : 中断语句块的执行, 包括switch块和循环和其他语句块
  break的破坏力度大 ,像一场大事故.

  continue : 中断循环中的某些次执行, 直接继续执行下一次循环.
  continue的破坏力度小, 像小意外.

  break和continue后面不可以直接跟语句, 因为永远无法执行到.

  while和do while 用一个布尔控制, 适用于循环次数不确定的循环, 取决 于外部条件.
  for 适用于循环次数确定的循环.
*/

class LoopTest13 {

  public static void main(String[] args) {
    // 死循环, 编译器认为后面语句无法执行的循环
    //while (true);
    boolean flag = true;
```

```
    while (flag);
    while (true) {
      if (true) {
        break;
      }
    }

    //do {} while (true); // 死
    do {} while (flag); //

    //for (;;); // 死循环
    //for (int i = 0; ; i++); // 死循环
    //for (int i = 0; true; i++); // 死循环
    for (int i = 0; flag; i++); // 无限
    for (int i = 0; i < 100;); // 无限
    for (int i = 0; i < 100; i--); // 有限

    System.out.println("after loop");
  }
}

class LoopTest12 {

  public static void main(String[] args) {
    // 列出100以内的所有质数
    l1 : for (int i = 2; i < 100; i++) {
      for (int j = 2; j < i; j++) {
        if (i % j == 0) {
          // 制造意外，让后面的打印语句废掉
          continue l1;
        }
      }
      System.out.println(i + "是质数");
    }
  }
}
```

```
class LoopTest11 {

  public static void main(String[] args) {
    l1 : for (int i = 0; i < 5; i++) {
      l2 : for (int j = 0; j < 3; j++) {
        if (j % 2 != 0) {
          //continue; // 默认中断离我最近的循环的当次循环，直接离我最近循环的下一次
          continue l1; // 中断标签指示的循环的当次循环，直接进入标签指示的循环的下一次.
        }
        System.out.println("j : " + j);
      }
      System.out.println("i : " + i);
    }
    // j : 0
    // j : 0
  }
}

class LoopTest10 {

  public static void main(String[] args) {
    for (int i = 0; i < 100; i++) {
      if (i % 2 == 0) {
        continue; // 出了小意外
        //System.out.println("hello");
      }
      // 无条件处理数据
      System.out.println("i : " + i);
    }

    /*
    for (int i = 0; i < 100; i++) {
      if (i % 2 != 0) { // 有条件处理数据
```

```
        System.out.println("i : " + i);
      }
    }*/
  }
}

class LoopTest9 {

  public static void main(String[] args) {
    l1 : {
      System.out.println("hello1");
      if (true) {
        break l1; // 中断的命名的块
        //System.out.println("kljkajlsdf");
      }
      System.out.println("hello2");
    }
  }
}

// 练习 : 从命令行参数接收一个n, 使一个本来应该循环100次的循环, 实际循环n次.
class LoopExer2 {

  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    for (int i = 0; i < 100; i++) {
      System.out.println("i : " + i);
      if (i == n - 1) {
        break;
      }
    }
    if (n > 100) {
      for (int i = 0; i < n - 100; i++) {
        System.out.println("i : " + i);
      }
    }
```

```
  }
}
class LoopTest8 {

  public static void main(String[] args) {
    // 列出100以内的所有质数
    for (int i = 2; i < 100; i++) {
      boolean flag = true; // 假定i是质数
      for (int j = 2; j < i; j++) { // 尝试找反例
        if (i % j == 0) { // 找到了反例
          flag = false;
          break; // 提前中断对i是不是质数的判定，因为它肯定不是质数
        }
      }

      if (flag) {
        System.out.println(i + "是质数");
      }
    }
  }
}

class LoopTest7 {

  public static void main(String[] args) {
    l1 : for (int i = 0; i < 5; i++) { // l1的作用是标识此循环的名称
      l2 : for (int j = 0; j < 3; j++) { // l2的作用是标识 内循环
        System.out.println("j : " + j);
        if (j == 1) {
          //break; // 中断离我最近循环，就近原则.
          break l1; // 中断的就是标签标识的循环
        }
      }
      System.out.println("i : " + i);
    }
  }
```

```java
}

class LoopTest6 {

  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      System.out.println("i : " + i);
      if (i == 3) {
        break;
      }
    }
    System.out.println("after loop");
  }
}


class LoopTest5 {

  public static void main(String[] args) {
    // 练习 : 列出100~200之间所有质数. 扩展 : 外循环因子用i, 内循环因子用j
    for (int i = 100; i < 200; i++) {
      boolean flag = true; // 假定i是质数.
      // 尝试推翻, 从2~i-1中把所有数都列出来
      for (int j = 2; j < i; j++) {
        // 如果 i 被某个 j 整除了, 说明i不是质数
        if (i % j == 0) {
          // 推翻假设
          flag = false;
        }
      }
      if (flag) { // 如果flag维持了true的状态, 说明没有反例.
        System.out.println(i + "是质数");
      }
    }
  }
}
```

```
class LoopTest4 {

  public static void main(String[] args) {
    // 练习 : 列出100~200之间所有质数. 扩展 : 外循环因子用i, 内循环因子用j
    // 列出100以内的所有质数
    for (int j = 2; j < 100; j++) {
      // 判断一个数j是否是质数, 质数 : 只能被1和自身整除的数, 从2~j-1中把所有数都列出来, 并且用这些数尝试整除j
      boolean flag = true; // 假定j是质数.
      // 尝试推翻, 从2~j-1中把所有数都列出来, 找反例
      for (int i = 2; i < j; i++) {
        // i就是用于检测的数据, 如果j能被某个i整除.
        if (j % i == 0) { // 说明j不是质数
          flag = false; // 推翻假设, j肯定不是质数
        }
      }
      if (flag) { // flag如果一直为true, 那就说明在上面的for循环中的if语句从来没有真的
        System.out.println(j + "是质数");
      }
    }
  }
}

class LoopTest3 {

  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    // 判断一个数n是否是质数, 质数 : 只能被1和自身整除的数, 从2~n-1中把所有数都列出来, 并且用这些数尝试整除n
    boolean flag = true; // 假定n是质数.
    // 尝试推翻, 从2~n-1中把所有数都列出来, 找反例
    for (int i = 2; i < n; i++) {
      // i就是用于检测的数据, 如果n能被某个i整除.
      if (n % i == 0) { // 说明n不是质数
```

```
        flag = false; // 推翻假设，n肯定不是质数
        break; // 提前中断
      }
    }
    if (flag) { // flag如果一直为true，那就说明在上面的for循环中的if语句从来没有真的
      System.out.println(n + "是质数");
    }
  }
}

/*
打印n行的等腰3角形，选做，打成空心的.
        i       空格      *
    *     0     4     1
   ***      1     3     3
  *****     2           2     5
 *******    3     1       7
*********     4     0     9
*/
class LoopTest2 {

  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n - 1 - i; j++) { // 控制空格的输出
        System.out.print(" ");
      }
      for (int j = 0; j < 2 * i + 1; j++) { // 控制*号的输出
        //if (第一行或最后行或第一列或最后列) {
        if (i == 0 || i == n - 1 || j == 0 || j == 2 * i) {
          System.out.print("*");
        } else {
          System.out.print(" ");
        }
      }
```

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

  class LoopTest1 {

    public static void main(String[] args) {
      for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 4 - i; j++) { // 控制空格的输出
          System.out.print(" ");
        }
        for (int j = 0; j < 2 * i + 1; j++) { // 控制*号的输出
          System.out.print("*");
        }
        System.out.println();
      }
    }
  }
```

第4章 方法

```
  /*
    方法(method) : 也称为函数(function), 是程序中的一个独立的功能单位.

    声明 :

      修饰符 返回值类型 方法名(数据类型1 形参1, 数据类型2 形参2, 数据类型3 形参
  3,....) {
```

```
        语句块，作用主要是加工数据
        return 返回值;
    }
```

返回值类型 ： 描述此方法的功能的最后成果数据是什么.
方法名 ： 标识符
形式参数 ： 有数据类型约束的一个变量，声明方法时它里面是没有确定的值的. 虽然它的值不确定，但是并不影响功能性.
　　方法的功能性的完成需要外部数据的支持，没有数据绝对不行，但是数据是多少又不影响功能.
返回值 ： 在方法中对数据加工后的一个成果数据，如果返回值类型为void时，表明此方法没有返回值
return 的作用就是把返回值数据返回给调用者

方法 = 方法头(方法签名，方法使用说明书API) + 方法体(真正的执行东西);
方法绝对不可以嵌套声明，因为方法必须隶属于类的. 方法之间都是平级的并列的关系.

方法必须要经过调用(invoke)，才能真的执行，否则就是一个摆设.
方法调用，是一个语句 ：

```
方法名(实参1，实参2，...); // 实参列表必须要和形参列表完全一致.
```

方法的返回值，只有一次机会接收这个值，就是在调用的同时 ： 如果错过了就错过了.

```
变量 = 方法调用本身;
```

有参有返回 ： 有输入有输出
有参无返回 ： 有输入无输出
无参有返回 ： 无输入有输出
无参无返回 ： 无输入无输出

注　意:
　　没有具体返回值的情况，返回值类型用关键字void表示，那么该函数中的return语句如果在最后一行可以省略不写。
　　定义方法时，方法的结果应该返回给调用者，交由调用者处理。
　　方法中只能调用方法，不可以在方法内部定义方法。
　　方法的返回值只有一次机会接收，就是在调用时
　　如果在方法中又调用了方法本身，称为递归调用

```java
*/

public class MethodTest {

  public static void test() { // 无参无返回
    System.out.println("test()...");
    //return;
  }


  // 写一个方法，完成一个求2个整数的和的功能
  public static int add(int a, int b) {
    // 提供者
    System.out.println("add(int a, int b)...");
    short c = (short)(a + b);
    return c; // 说明书中声明的是int类型为返回值，实际给的是short，没有问题.
  }

  // 方法调用流程 :
  // 1) 方法调用者中把实参传递给形参，值参的本质就是赋值 形参 = 实参
  // 2) 现场保护，流程进入被调用方法内部
  // 3) 执行被调用方法中的所有语句
  // 4) 把返回值放入临时空间中后，被调用方法彻底结束，所有的局部变量全部死掉
  // 5) 流程回到调用者方法中，调用者方法接收到临时空间中的返回值后 现场恢复
  // 6) 继续执行调用者方法中的其他语句

  public static void main(String[] args) {
    System.out.println("main begin...");

    // 使用者
    short a = 20;
    short b = 100;
    // 实参和形参之间不一定是完全匹配，只要能兼容即可.
    int x = add(a, b); // 调用!!!，20和100称为实参，作用是给形参传递数据
    //System.out.println(c); 在这里绝不可跨方法访问方法中的局部变量
```

```
        System.out.println(x);

        /*
        System.out.println(add(90, 320)); // 嵌套调用, 执行顺序是由内向外.

        // 方法没有返回值时, 不可以接收它的调用
        //int y = test(); // 无参方法调用时也必须要有一个().
        test(); // 无参方法调用时也必须要有一个().

        //System.out.println(test()); 如果没有返回值, 也不能打印方法调用.
        */

        System.out.println("main end...");
    }

}
```

重载

```
/*
  重载(overload) : 就是过载, 同一个类中有多个同名方法, 但是参数列表必须不同, 形成重载.
  参数不同体现为类型不同, 个数不同, 顺序不同.
  重载和方法的返回值类型无关!!!

  多个重载的方法是通过调用者中的实参来加以区分.

  实参和形参匹配时有一个度, 优先精准匹配, 再兼容性匹配.

  重载的目的 : 让调用者调用简单, 而且方便. 重载的多个方法的功能性是相近的.
*/
public class OverloadTest {

  /*
```

```
  public static int add(int a, int b) {
    System.out.println("add(int a, int b)...");
    return a + b;
  }*/

  public static double add(double a, double b) {
    System.out.println("add(double a, double b)...");
    double c = a + b;
    return c;
  }

  public static double add(int a, double b) {
    System.out.println("add(int a, double b)...");
    return a + b;
  }

  /* 此方法是和上面的方法是冲突的关系, 名字的顺序不同, 不能有任何作用.
  public static double add(int b, double a) {
    System.out.println("add(int b, double a)");
    return b + a;
  }*/

  public static double add(double a, int b) {
    System.out.println("add(double a, int b) ...");
    return a + b;
  }

  public static int add(int a, int b, int c) {
    System.out.println("add(int a, int b, int c).");
    return a + b + c;
  }

  public static void main(String[] args) {
    //System.out.println(add(35, 88)); // (int, int) 此方法会和上面的2个
方法都匹配, 出现不确定性!!!!
    System.out.println(add(9.2, 3.323));
```

```
    System.out.println(add(22, 3.323));
    System.out.println(add(22.324324, 8324));

    System.out.println(add(9, 3, 7));

    System.out.println(100); // int
    System.out.println(false); // boolean
    System.out.println(3.22); // double
    System.out.println('A'); // char
    System.out.println("alksdjflka"); // String
    System.out.println(342.2f); // float

  }
}
```

连环调用

```
public class MethodTest2 {

  public static void test(int a) {
    System.out.println("test(int a) : " + a); // 4
    //add(5, 6); 危险, 形成间接递归调用!!!!
    //while (true);
  }

  public static int add(int a, int b) {
    test(a); // 连环调用!!!
    System.out.println("add(int a, int b)..."); // 3
    int c = a + b;
    return c;
  }

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

```
        System.out.println("main begin"); // 1

        int a = 10;
        int b = 20;
        int c = add(a, b);
        System.out.println(c);

        System.out.println("main end"); // 2
    }
}
```

每日一考_day06

1. 简述break语句和continue语句的作用.

break是结束程序当前的循环，continue是跳过程序当前运行进入下一次运行；

我们在使用可以给程序加标签，用break加标签结束标签所指定的程序；

用continue加标签跳过标签所指定的程序当前运行进入下一次执行；

2. 声明定义一个方法的语法格式是什么？解释每部分的作用.

方法的声明：权限修饰符（有四种权限修饰符，方法一般只使用其中两种，用来表明被调用时可见性的大小）

返回值类型（用来表明该方法被执行后会得到一个什么样的结果）

方法名（方法声明和调用时所提供的名称，最好见名知意）

形参列表（数据类型1 形参1，.....）接收调用时传递的实参

方法的功能性的完成时, 需要外部数据的支持时, 我们就使用形式参数.

3. 如何调用一个带有形参的方法?

在同一类中，方法名（对应的实参）

4. 方法的返回值是什么? 如何使用这个返回值?

返回值类型（用来表明该方法被执行后会得到一个什么样的结果）；

在调用者方法中，方法的返回值就是 方法调用 本身

变量 = 方法名(实参列表);

返回值可以直接输出，也可以用对应的变量来接收；

方法只有一次机会得到对应的执行结果，那就是在方法调用时

5. 什么是方法重载, 为什么要方法重载?

方法重载：两同一不同：同一类同一个方法名，不同指形参列表不同

形参列表不同分为三种：个数、类型、顺序

作用：提高方法兼容性，方便调用者使用

第5章 数组1

```
/**
  数组 ： 一组相同类型的数据的组合， 实现对这些数据的统一管理， 统一管理就是通过循环来实现的.
  数组 == 循环

  任意数据类型都可以创建数组， 包括基本数据类型和引用数据类型

  符合Random Access 特性.

  数组是引用型， 数组的数据是对象.

  数组的声明 :
    元素数据类型[] 数组名;
    int[] arr1; // 推荐这样的写法.
    int arr2[];

  数组对象的创建 :
    数组名 = new 元素数据类型[元素个数];
    数组名中保存的就是数组对象的首地址.

    // arr引用变量中保存了数组对象的首地址
```

```
    arr1 = new int[8]; // 在内存中创建一个数组对象，并把它的地址写入arr1.

    数组对象一旦创建，它的所有元素值都会被JVM自动初始化为0

  数组元素的定位 ：
    数组名(首地址) + 偏移量(索引)
    偏移量也称为下标，脚标，索引 等....
    数组名[下标] = 值; // 写入
    System.out.println(数组名[下标]); // 读取

  数组的有效索引范围 ：[0 ~ length - 1]
  数组的长度永远无法改变，因为随意改变长度会导致内存的使用出问题.
  数组的元素类型也不能随意改变.

  所以数组是最稳定的数据结构.

  数组算法 ：
    求和，求平均值.
    找最大值，最小值
    反转
    排序
  高级算法 ：
    复制，缩减，扩容，删除，取子数组，添加元素，选择排序，快速排序.
*/
class ArrayTest11 {

  public static void main(String[] args) {
    int[] arr = new int[8];
    for (int i = 0; i < arr.length; i++) {
      arr[i] = (int)(Math.random() * 30);
    }
    // 遍历
    for (int i = 0; i < arr.length; i++) {
      System.out.print(arr[i] + " ");
    }
    System.out.println();
```

```
    // 0 1 2  3  4  5  6 7
    //10 0 24 28 11 29 2 7
    //找出能被7整除的数的最大值
    int max7 = 0x80000000; // 称为标记值，标记值一定是非法数据!!!
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] % 7 == 0) { // 如果某个数满足了条件
        if (arr[i] > max7) { // 再比较和刷新
          max7 = arr[i];
        }
      }
    }
    if (max7 == 0x80000000) { // 如果它仍是维持了标记值，说明if没有进去过
      System.out.println("没有能被7整除的数");
    } else {
      System.out.println("max7 : " + max7);
    }
  }
}

class ArrayTest10 {

  public static void main(String[] args) {
    int[] arr = new int[8];
    for (int i = 0; i < arr.length; i++) {
      arr[i] = (int)(Math.random() * 30);
    }
    // 遍历
    for (int i = 0; i < arr.length; i++) {
      System.out.print(arr[i] + " ");
    }
    System.out.println();
    // 0 1 2  3  4  5  6 7
    //10 0 24 28 11 29 2 7
    // 最大值
    int max = 0x80000000; // 不允许使用任意元素作为初始值. int最小值 1000
0000 0000 0000 0000 0000 0000 0000
```

```
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] > max) {
        max = arr[i];
      }
    }
    System.out.println("最大值 : " + max);
    // 最小值
    int min = 0x7FFFFFFF;
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] < min) {
        min = arr[i];
      }
    }
    System.out.println("最小值 : " + min);
  }
}

class ArrayTest9 {

  public static void main(String[] args) {
    int[] arr = new int[8];
    for (int i = 0; i < arr.length; i++) {
      arr[i] = (int)(Math.random() * 30);
    }
    // 遍历
    for (int i = 0; i < arr.length; i++) {
      System.out.print(arr[i] + " ");
    }
    System.out.println();
    // 0 1 2  3  4  5  6 7
    //10 0 24 28 11 29 2 7
    // 找出最大值.
    int max = arr[0]; // 假定第1个元素的值最大.
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] > max) { // 如果某元素的值 比 max 还大.
        max = arr[i]; // 刷新max为某元素.
```

```
      }
    }
    System.out.println("最大值 : " + max);
    // 找出最小值
    int min = arr[0]; // 假定第1个元素的值最小
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] < min) {
        min = arr[i];
      }
    }
    System.out.println("最小值 : " + min);
  }
}

class ArrayTest8 {

  public static void main(String[] args) {
    int[] arr = new int[8];
    for (int i = 0; i < arr.length; i++) {
      arr[i] = (int)(Math.random() * 30);
    }
    // 遍历
    for (int i = 0; i < arr.length; i++) {
      System.out.print(arr[i] + " ");
    }
    System.out.println();
    // 0 1  2  3  4 5  6  7
    //30 14 30 25 7 26 7 21
    //求所有能被7整除的数的平均值
    int sum7 = 0;
    int count7 = 0;
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] % 7 == 0) { // 判断的是元素是否满足条件，并不是下标
        sum7 += arr[i]; // 有条件求和
        count7++; // 有条件计数
      }
```

```
    }
    // 数据的筛选时要考虑没有数据的情况
    if (count7 == 0) {
      System.out.println("没有能被7整除的数");
    } else {
      int avg7 = sum7 / count7;
      System.out.println("avg7 : " + avg7);
    }
  }
}

class ArrayTest7 {

  public static void main(String[] args) {
    int[] arr = new int[8];
    for (int i = 0; i < arr.length; i++) {
      arr[i] = (int)(Math.random() * 30 + 1);
    }
    // 遍历
    for (int i = 0; i < arr.length; i++) {
      System.out.print(arr[i] + " ");
    }
    System.out.println();
    //0  1 2 3 4  5  6 7
    //4 16 7 1 25 12 3 14
    //求和
    int sum = 0;
    for (int i = 0; i < arr.length; i++) {
      sum += arr[i];
    }
    System.out.println("sum : " + sum);
    int avg = sum / arr.length;
    System.out.println("avg : " + avg);

  }
}
```

```java
class RandomTest {

  public static void main(String[] args) {
    // 获取随机数. 返回一个[0~1) 随机浮点数
    double rand1 = Math.random();
    System.out.println(rand1);
    // 获取100以内的随机浮点数
    double rand2 = Math.random() * 100;
    System.out.println(rand2);
    // 获取100以内的随机整数
    int rand3 = (int)(Math.random() * 100);
    System.out.println(rand3);
    // 练习 : 获取[20~80)随机整数
    int rand4 = (int)(Math.random() * (80 - 20) + 20);
    System.out.println(rand4);
  }
}

class ArrayTest6 {

  public static void main(String[] args) {
    int[] arr = new int[18];
    for (int i = 0; i < arr.length; i++) {
      arr[i] = 2 * i + 1;
    }
    // 遍历
    for (int i = 0; i < arr.length; i++) {
      System.out.print(arr[i] + " ");
    }
    System.out.println();
    // 元素求和, 求平均值.
    int sum = 0;
    for (int i = 0; i < arr.length; i++) {
      // sum += i; // 累加循环因子.
      sum += arr[i]; // 累加元素值
```

```
    }
    System.out.println("sum : " + sum);
    int avg = sum / arr.length;
    System.out.println("avg : " + avg);
  }
}

class ArrayTest5 {

  public static void main(String[] args) {
    // 数组的创建和初始化的方式, 3种

    // 1) 动态方式 , 适合于大量数据
    int[] arr1 = new int[3];
    arr1[0] = 9;
    arr1[1] = 38;
    arr1[2] = 99;

    // 2) 静态方式1, 元素值一步到位. 适用于数据量小.
    int[] arr2 = new int[] {1, 3, 2000, 30, 40, 80}; // []中绝对不可以加上长度

    // 3) 静态方式2 , 写法更简单, 元素值一步到位, 使用场景 : 声明和初始化必须在同一行语句上.
    int[] arr3 = {3, 9, 7, 8, 2, 1, 300, 400, 22};

    int[] arr4;
    //arr4 = {3, 4, 2, 9};
    arr4 = new int[]{3, 4, 2, 9};

  }
}

class ArrayTest4 {

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

```
    char[] arr = new char[26];
    for (int i = 0; i < arr.length; i++) {
      arr[i] = (char)('a' + i);
    }
    // 遍历
    for (int i = 0; i < arr.length; i++) {
      System.out.print(arr[i] + " ");
    }
    System.out.println();
  }
}

class ArrayTest3 {

  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    int[] arr = new int[n];
    // 获取数组的长度
    System.out.println("arr.length : " + arr.length);

    // 数组的赋值.
    for (int i = 0; i < arr.length; i++) {
      //System.out.println("i : " + i);
      arr[i] = 2 * i + 1;
    }

    /*
    arr[0] = 1;
    arr[1] = 3;
    arr[2] = 5;
    arr[3] = 7;
    arr[4] = 9;
    arr[5] = 11;
    arr[6] = 13;
    arr[7] = 15;
    */
```

```
    // 数组的遍历，把它的所有元素都过一遍.
    for (int i = 0; i < arr.length; i++) {
      System.out.println(arr[i]);
    }

    /*
    System.out.println(arr[0]);
    System.out.println(arr[1]);
    System.out.println(arr[2]);
    System.out.println(arr[3]);
    System.out.println(arr[4]);
    System.out.println(arr[5]);
    System.out.println(arr[6]);
    System.out.println(arr[7]);
    */
  }
}

class ArrayTest2 {

  public static void main(String[] args) {
    int[] arr = new int[7];

    //arr = new double[8];

    // 获取数组的长度
    System.out.println("arr.length : " + arr.length);
    //arr.length = 8;
    //arr.length = 4;

    // arr[-1] = 22; 绝对禁止

    arr[0] = 1;
    arr[1] = 3;
    arr[2] = 5;
```

```
        arr[3] = 7;
        arr[4] = 9;
        arr[5] = 11;
        arr[6] = 13;

        // 编译不出问题，但是运行时出了问题，称为异常!!! 数组越界异常
        //arr[7] = 15; // ArrayIndexOutOfBoundsException: Index 7 out of
    bounds for length 7

        System.out.println(arr[0]);
        System.out.println(arr[1]);
        System.out.println(arr[2]);
        System.out.println(arr[3]);
        System.out.println(arr[4]);
        System.out.println(arr[5]);
        System.out.println(arr[6]);
        //System.out.println(arr[7]);
      }
    }

    public class ArrayTest {

      public static void main(String[] args) {
        int[] arr; // arr是引用型，它里面用于保存一个地址，此时啥地址也没有
        arr = new int[5]; // 在堆内存中创建一个真正的数组对象，包括有5个元素，每个
    元素都是int类型的.
        // arr引用变量中保存了数组对象的首地址
        arr[2] = 20; // 根据 首地址+偏移2个单位，定位到第3个格子
        arr[0] = 3;
        arr[arr[0]] = arr[2];
        arr[4] = arr[0] + arr[2];

        // 3 , 0, 20, 20, 23

        System.out.println(arr[0]);
        System.out.println(arr[1]);
```

```
        System.out.println(arr[2]);
        System.out.println(arr[3]);
        System.out.println(arr[4]);

        //int a; // 栈中的局部变量并没有自动初始化，而堆中的成员变量是有自动初始化.
        //System.out.println(a);
    }

}
```

每日一考_day07

1. 写出递归方式求n!, 简单加以说明

```
    // 递归 : 分解成一个小问题和相同类型的子问题.
    public static int jie(int n) {
        if  (n == 1) { // 问题规模最小的情况
            rentu 1;
        }
        return n * jie(n-1); // 分解成规模小一点的子问题
    }
```

2. 数组是什么? 什么类型的数据可以创建数组?

   数组是存储一组相同类型的数据的组合，实现统一(用循环) 管理.

   任何数据类型都可以创建数组。 包括 基本数据类型和引用数据类型

3. 如何声明并创建数组? 有几种方式?

   int[] arr = {1,2,3,4}; // 静态方式, 使用场景 : 声明和初始化赋值在同一行语句

   int[] arr = new int[] {1,2,3,4}; // 静态方式2 ,

   int[] arr;

arr = new int[] {1, 2,3,4};

int[] arr2 = new int[9]; // 元素值都是0 动态方式, 场景 : 适用于处理大数据

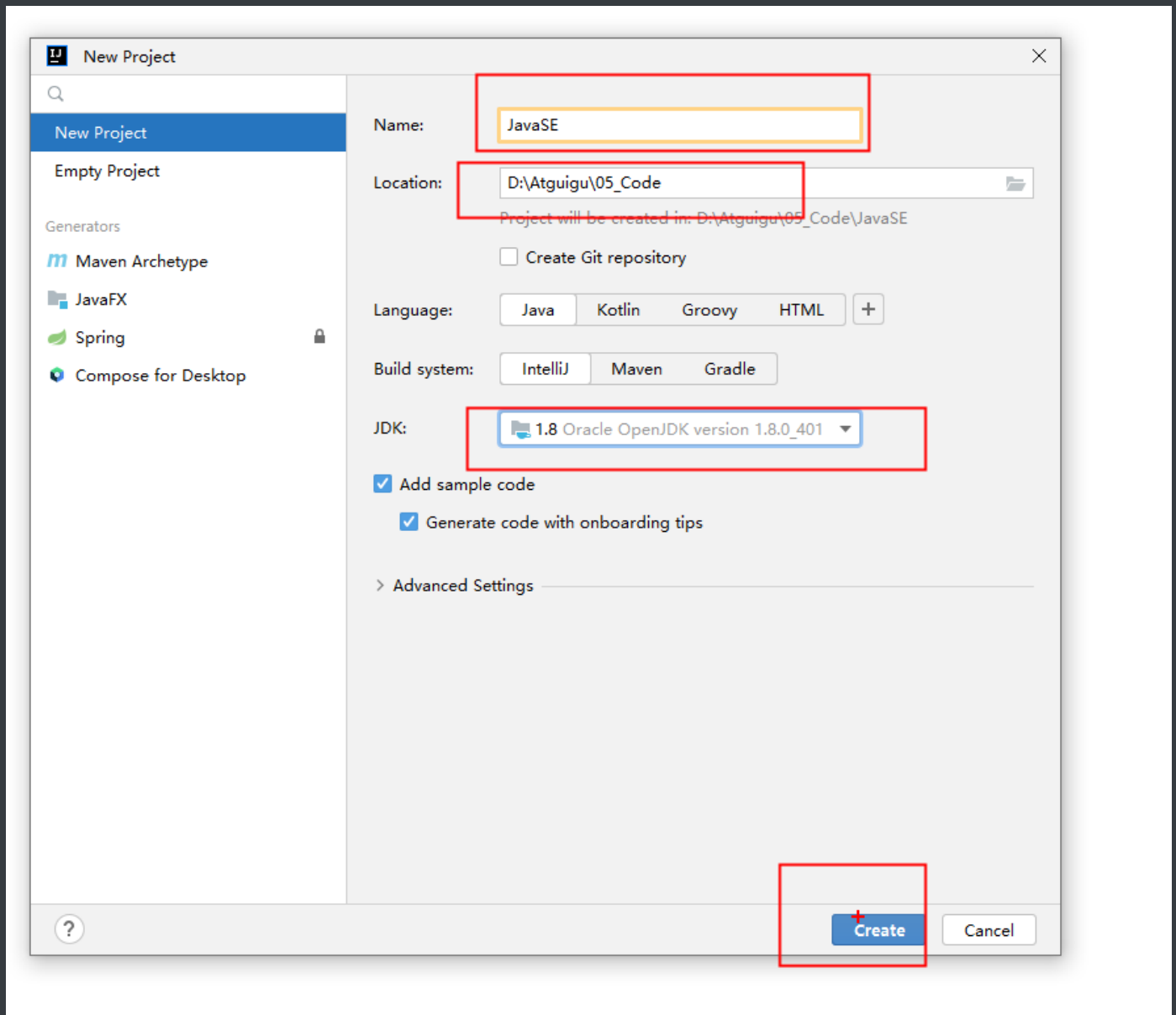
4. 判断:
    1. 数组的长度可以随意变化.(错)
    2. 数组的元素类型和数组类型是一回事.（错）元素是个体, 数组是群体
    3. 数组的访问是通过数组名.下标实现的.（错）arr.2 = 30;
    4. 数组的访问方式是数组名(首地址) + 偏移算出来的.（对）arr[2] = 30;
    5. 数组的元素类型可以随意变化.（错·）

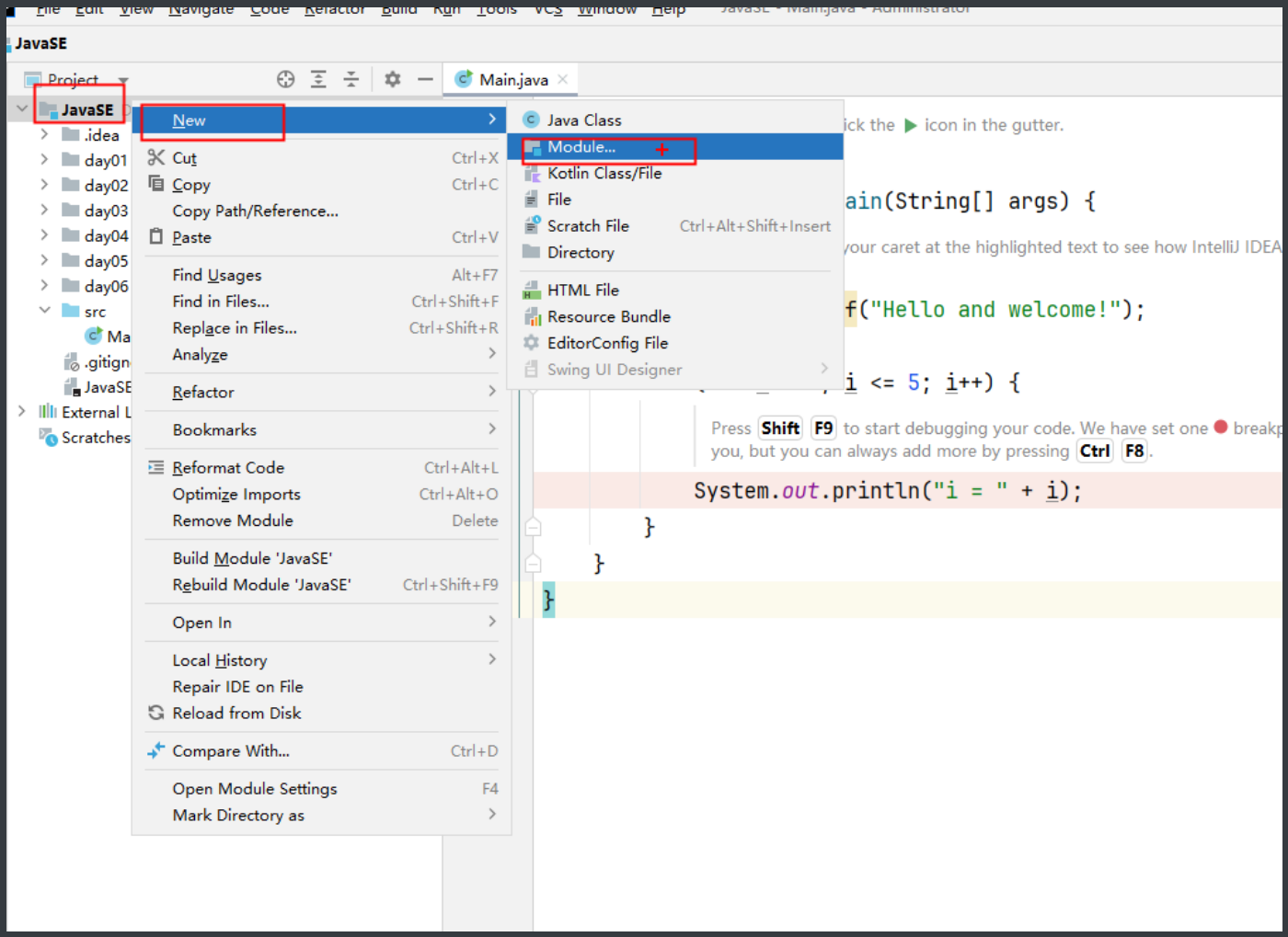
5. 完成代码

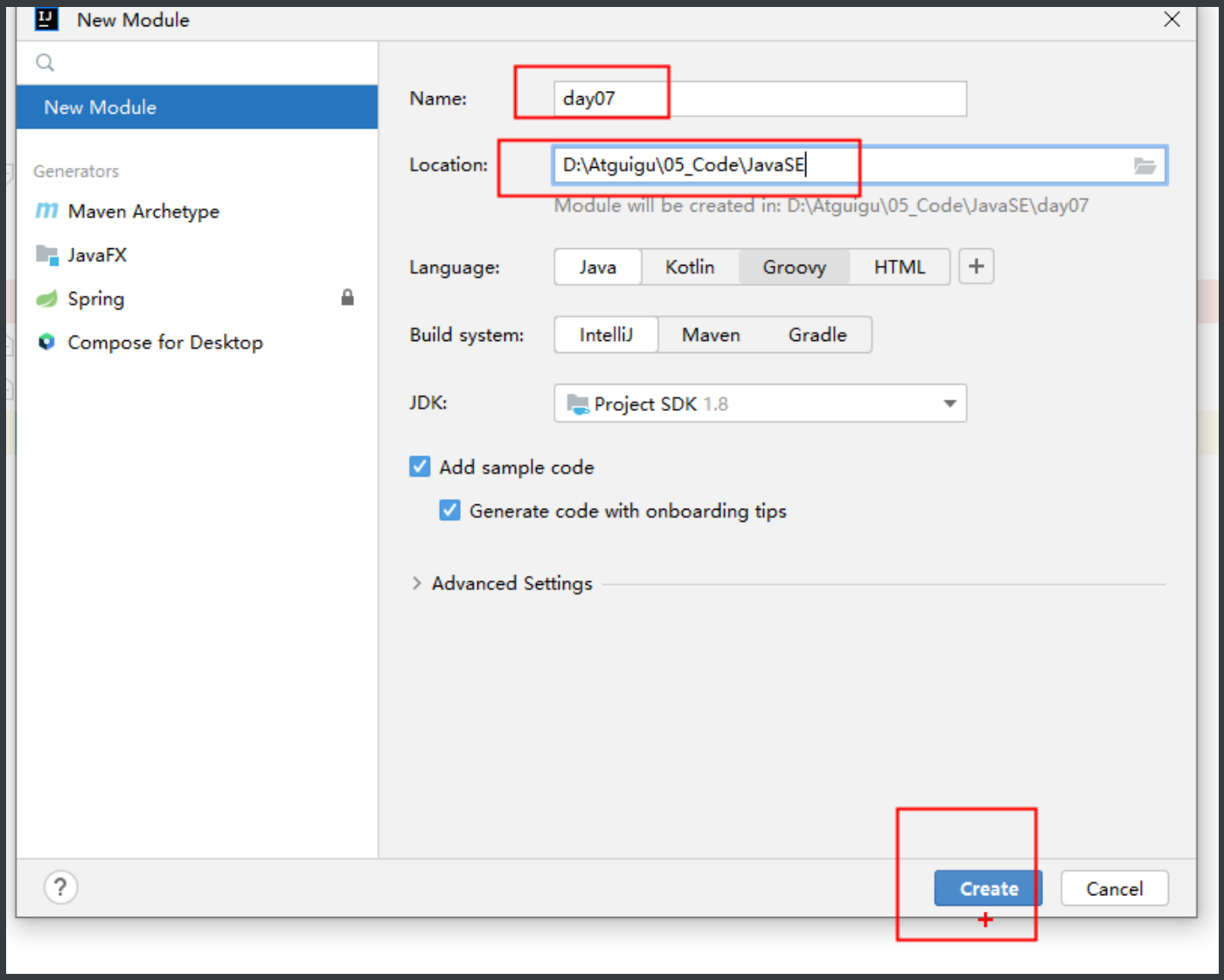
```
public static void main(String[] args) {
    int[] arr = {9, 2, 3, 1, 30, 25, 80, 40};

    // 找出能被7整除的数的最大值
    int max = 0x8000_0000; // 1000 ...0 00, 标记值 : 非法
    for (int i = 0; i < arr.length;i++){
        if(arr[i] % 7 == 0){
            if(arr[i] > max){ // 比较并刷新
                max = arr[i];
            }


        }
    }
    if(max != 0x8000_0000){
        System.out.println(max);
```

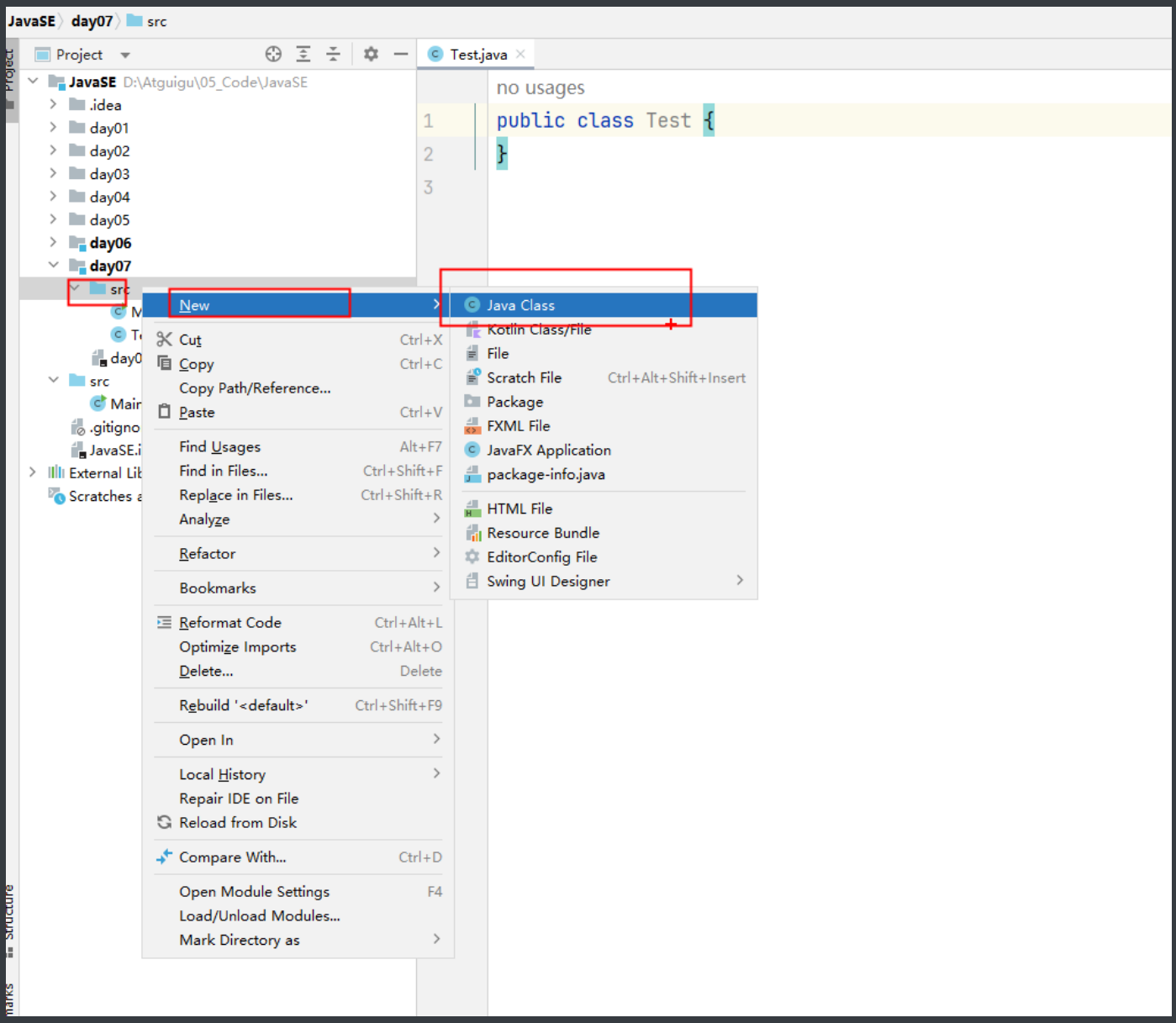
```
        }else{
            System.out.println("没有能被七整除的数");
        }
    }
```

idea使用




JavaSE
Project
Main.java
JavaSE
.idea
day01
day02
day03
day04
day05
day06
src
External L
Scratches
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path/Reference...
Paste Ctrl+V
Find Usages Alt+F7
Find in Files... Ctrl+Shift+F
Replace in Files... Ctrl+Shift+R
Analyze
Refactor
Bookmarks
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Remove Module Delete
Build Module 'JavaSE'
Rebuild Module 'JavaSE' Ctrl+Shift+F9
Open In
Local History
Repair IDE on File
Reload from Disk
Compare With... Ctrl+D
Open Module Settings F4
Mark Directory as
Java Class
Module...
Kotlin Class/File
File
Scratch File Ctrl+Alt+Shift+Insert
Directory
HTML File
Resource Bundle
EditorConfig File
Swing UI Designer
ick the ▶ icon in the gutter.
ain(String[] args) {
your caret at the highlighted text to see how IntelliJ IDEA
f("Hello and welcome!");
i <= 5; i++) {
Press Shift F9 to start debugging your code. We have set one breakp
you, but you can always add more by pressing Ctrl F8.
System.out.println("i = " + i);
}
}
}


New Module
New Module
Generators
Maven Archetype
JavaFX
Spring
Compose for Desktop
Name:
day07
Location:
D:\Atguigu\05_Code\JavaSE
Module will be created in: D:\Atguigu\05_Code\JavaSE\day07
Language:
Java
Kotlin
Groovy
HTML
Build system:
IntelliJ
Maven
Gradle
JDK:
Project SDK 1.8
Add sample code
Generate code with onboarding tips
Advanced Settings
Create
Cancel



main : 生成主方法

sout : 打印语句

soutv : 打印变量名及它的值

cltr + d : 快速复制行

cltr + x : 快速剪切行

ctrl + c :

shift + 回车： 直接进入新行

alt + 回车 : 快速修正小问题

ctrl + p : 列出方法的形参列表

debug

F7 进入细节

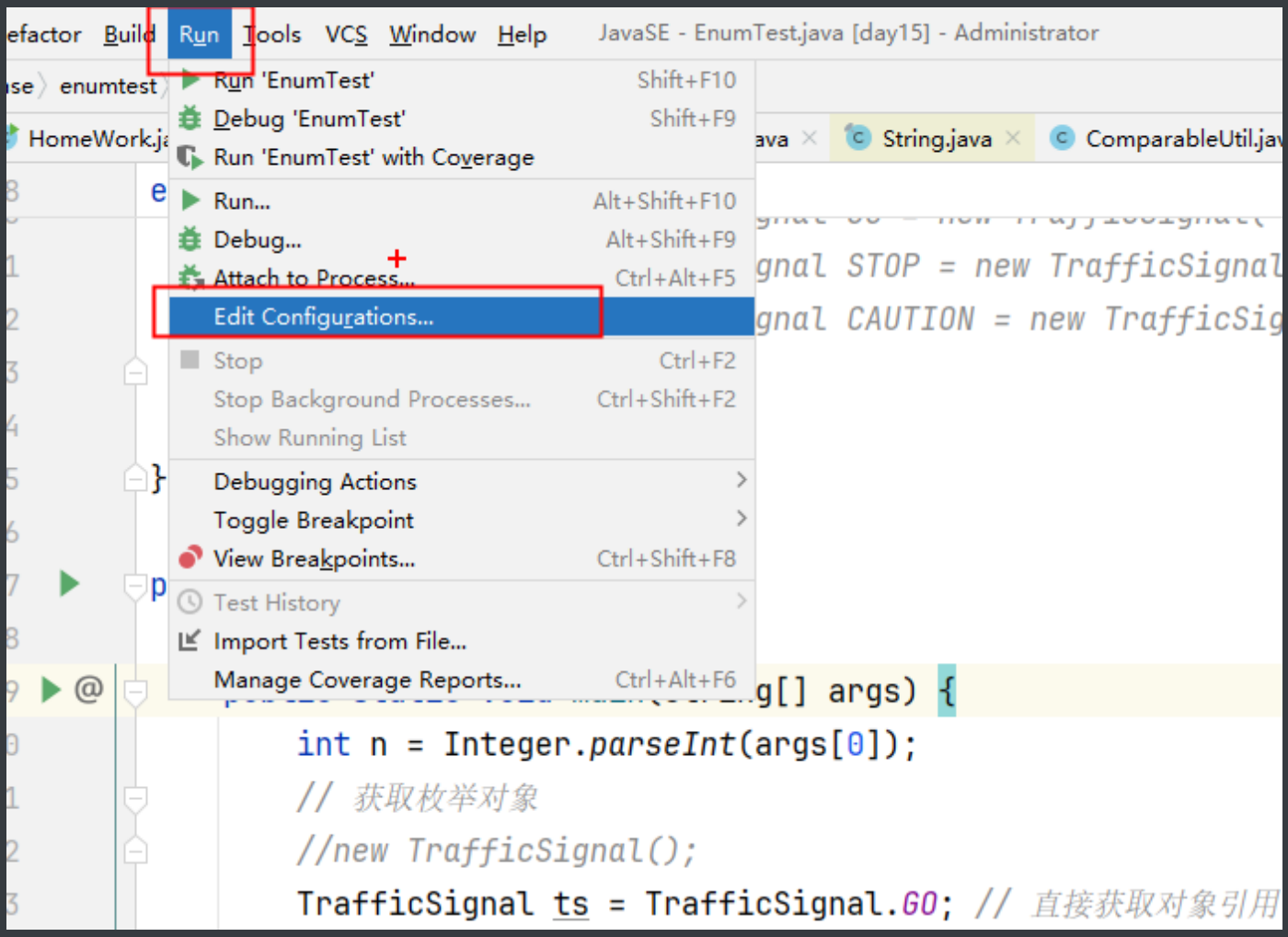
alt + shift + F7 强行进入细节

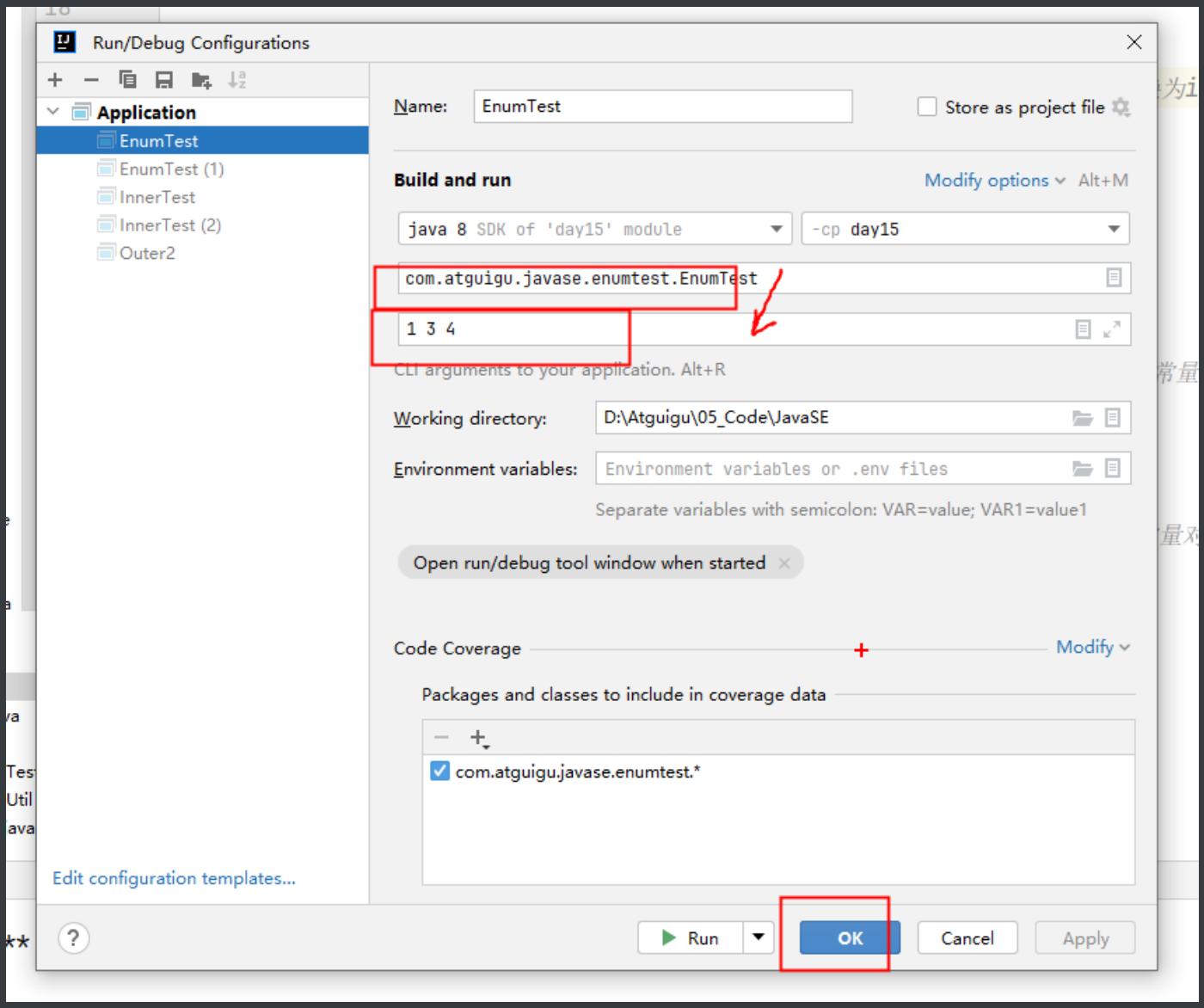
F8 直接执行当前行

shift + F8 : 直接跳出当前方法

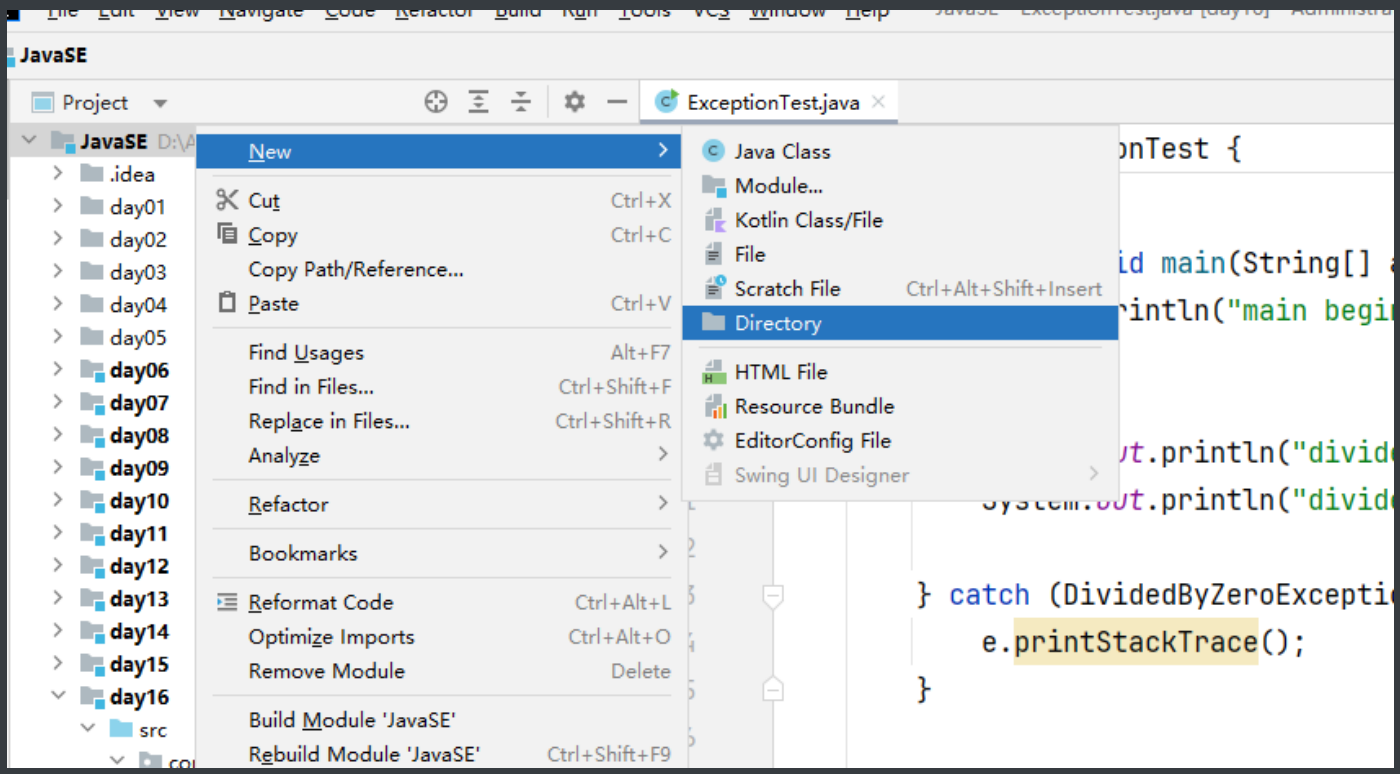
F9 直接继续执行

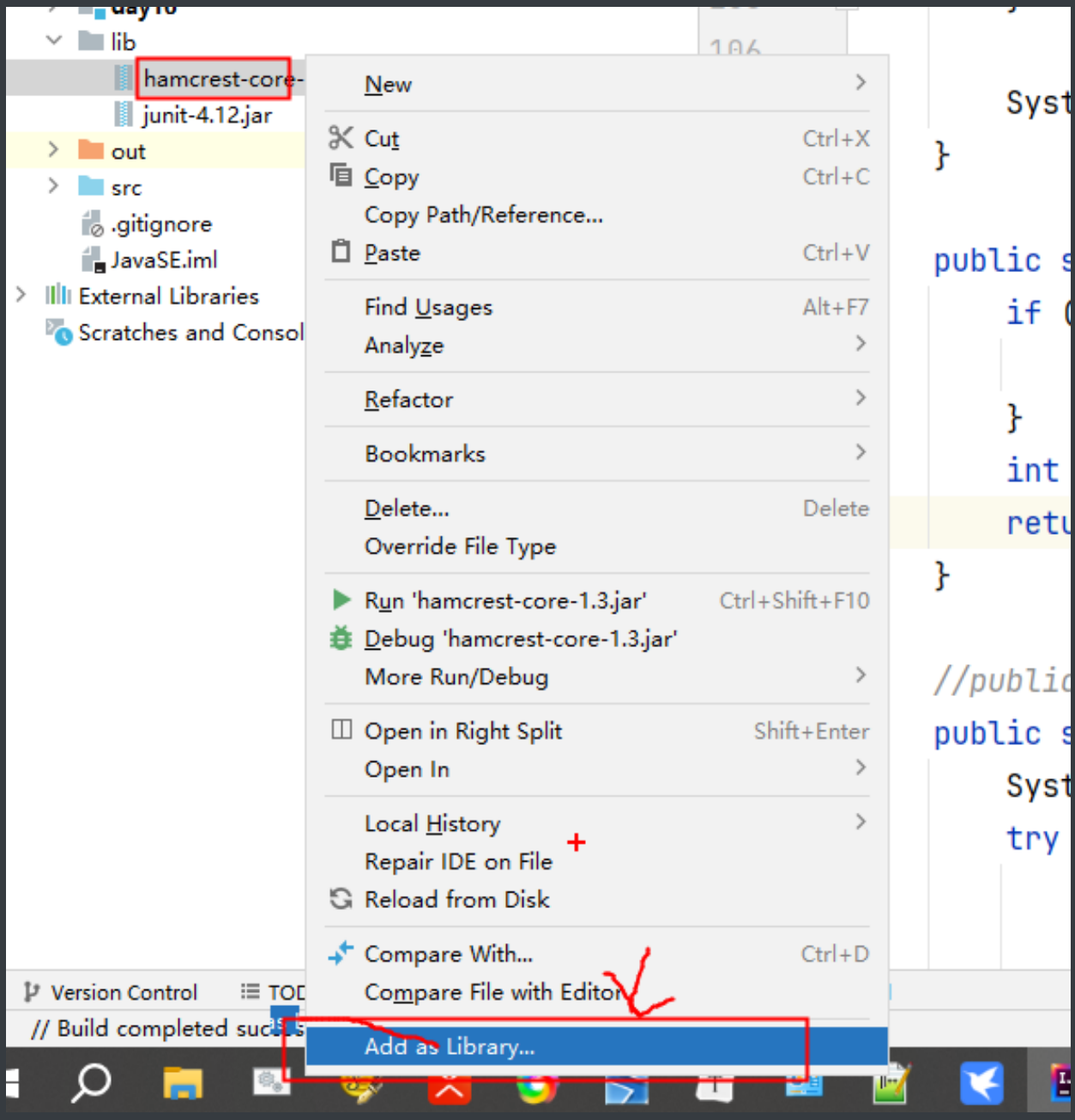
处理命令行参数

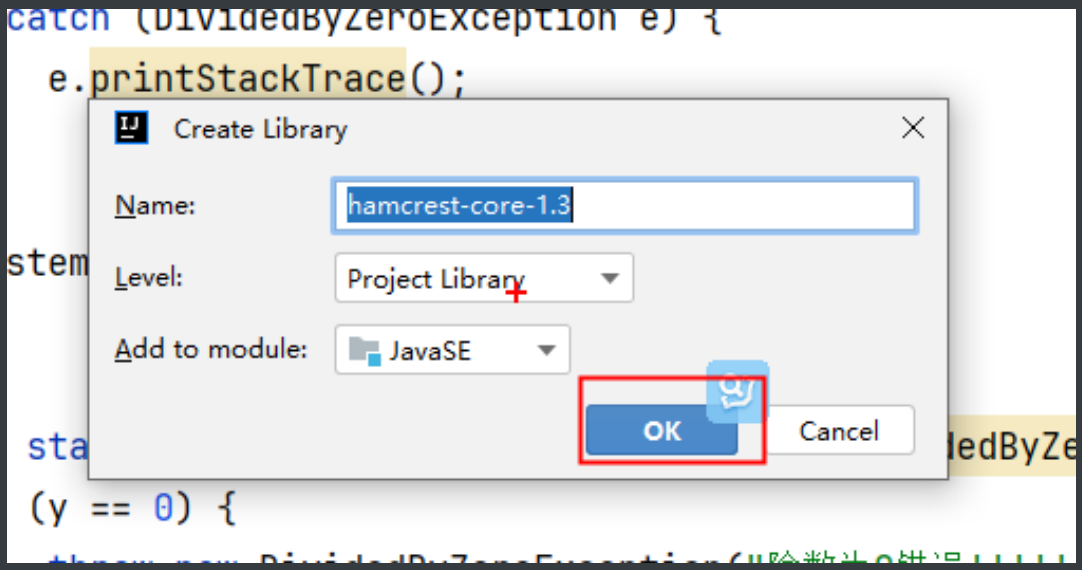

efactor Build Run Tools VCS Window Help
JavaSE - EnumTest.java [day15] - Administrator
Run 'EnumTest' Shift+F10
Debug 'EnumTest' Shift+F9
Run 'EnumTest' with Coverage
Run... Alt+Shift+F10
Debug... Alt+Shift+F9
Attach to Process... Ctrl+Alt+F5
Edit Configurations...
Stop Ctrl+F2
Stop Background Processes... Ctrl+Shift+F2
Show Running List
Debugging Actions
Toggle Breakpoint
View Breakpoints... Ctrl+Shift+F8
Test History
Import Tests from File...
Manage Coverage Reports... Ctrl+Alt+F6
String.java
ComparableUtil.jav
int n = Integer.parseInt(args[0]);
// 获取枚举对象
//new TrafficSignal();
TrafficSignal ts = TrafficSignal.GO; // 直接获取对象引用

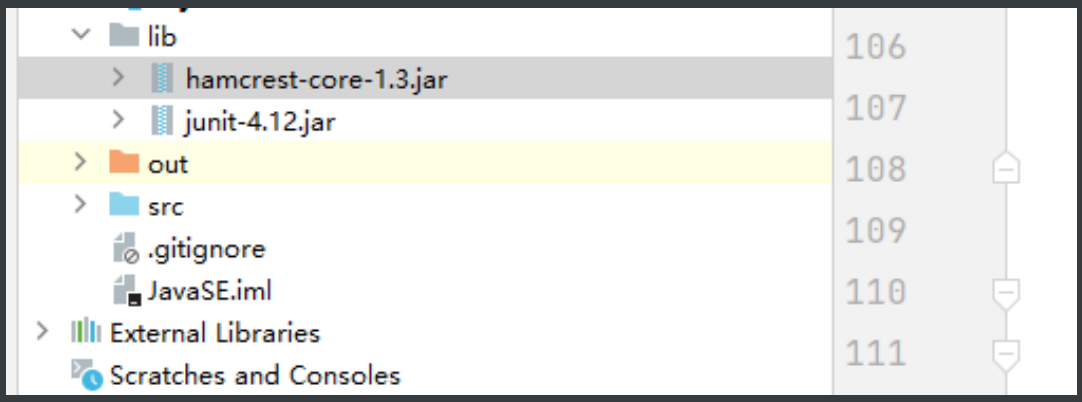

Run/Debug Configurations
Application
EnumTest
EnumTest (1)
InnerTest
InnerTest (2)
Outer2
Name: EnumTest
Store as project file
Build and run
Modify options Alt+M
java 8 SDK of 'day15' module
-cp day15
com.atguigu.javase.enumtest.EnumTest
1 3 4
CLI arguments to your application. Alt+R
Working directory: D:\Atguigu\05_Code\JavaSE
Environment variables: Environment variables or .env files
Separate variables with semicolon: VAR=value; VAR1=value1
Open run/debug tool window when started
Code Coverage
Modify
Packages and classes to include in coverage data
com.atguigu.javase.enumtest.*
Edit configuration templates...
Run
OK
Cancel
Apply

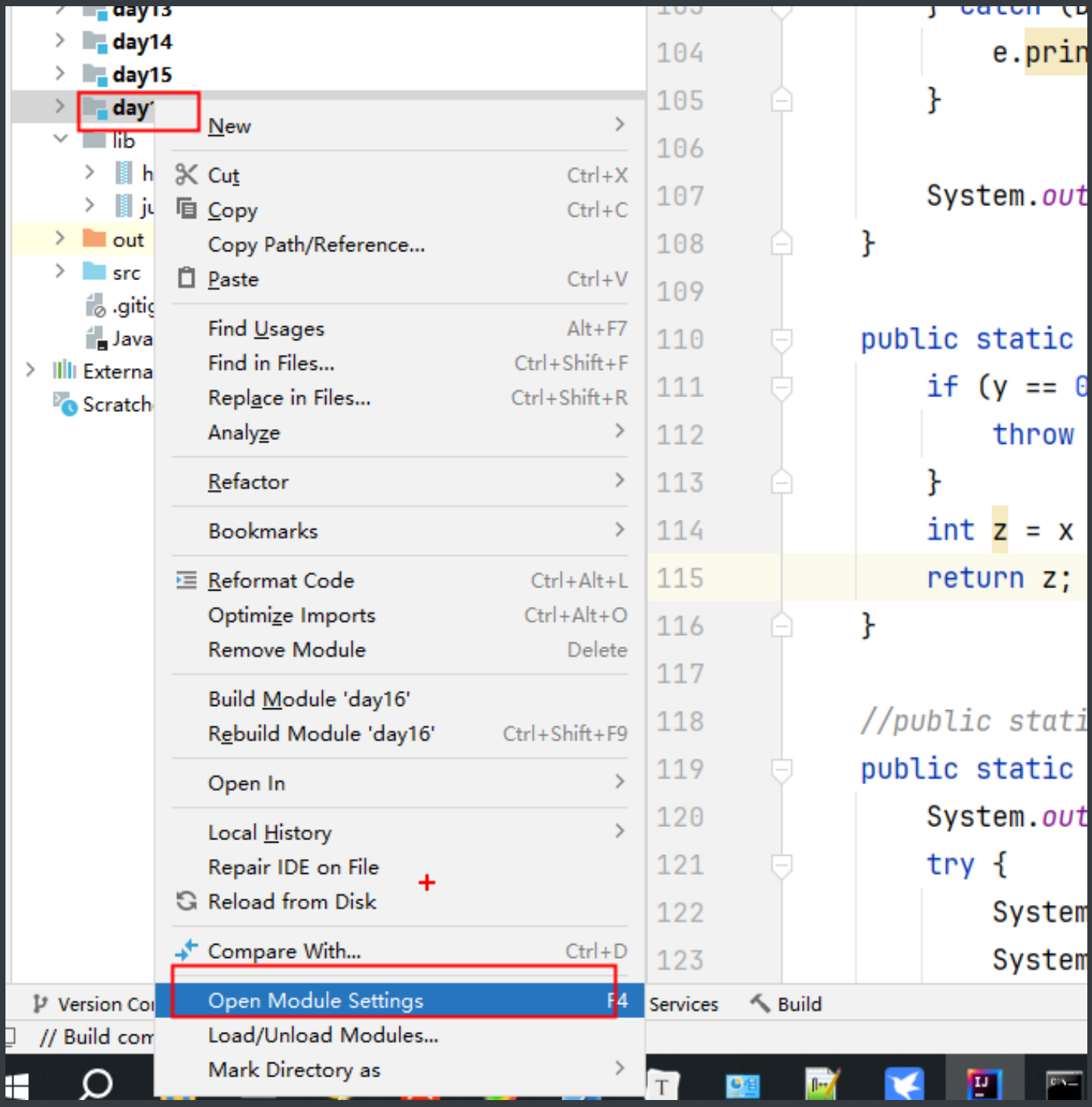
导入第3方的jar

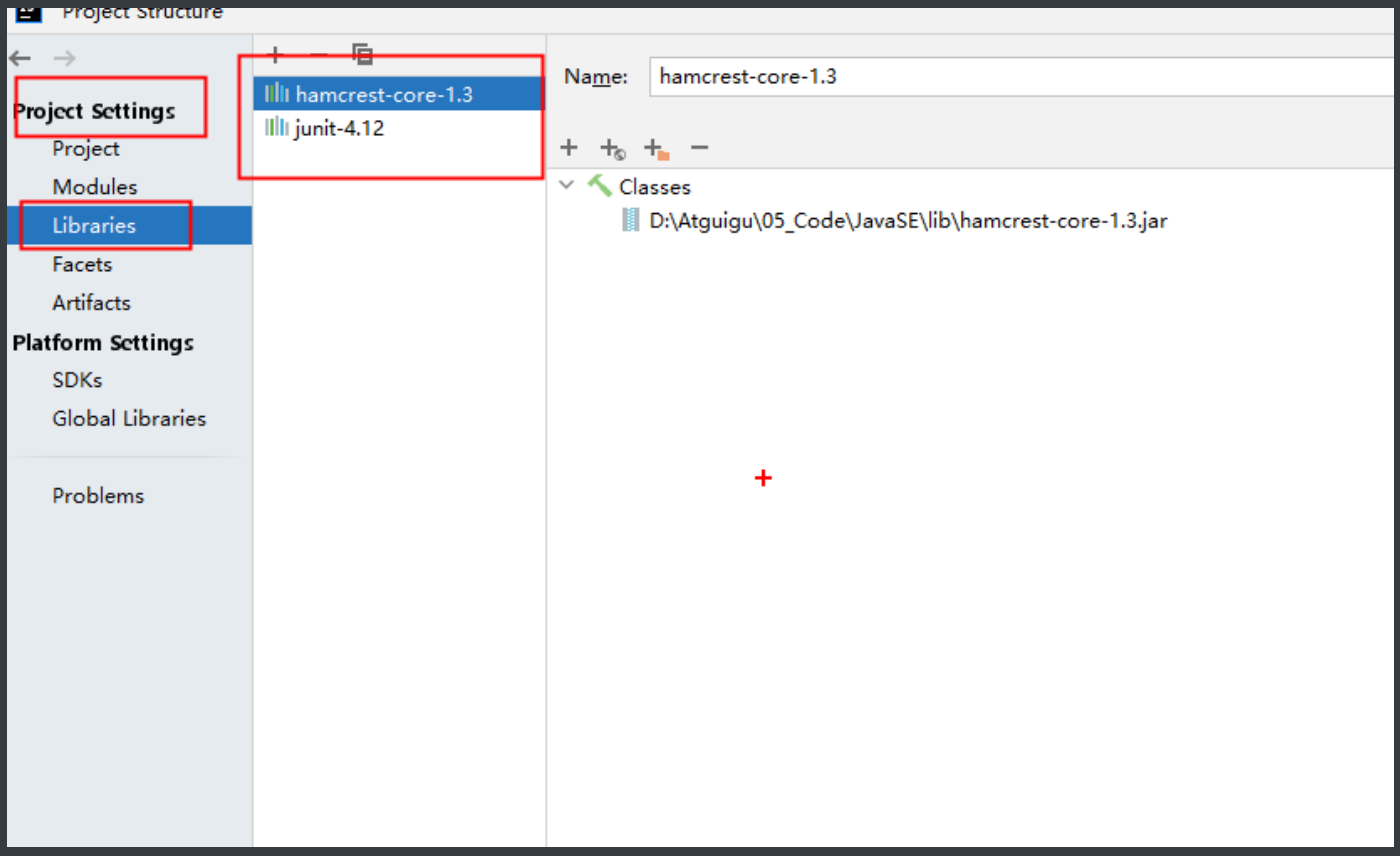

JavaSE
Project
ExceptionTest.java
JavaSE
.idea
day01
day02
day03
day04
day05
day06
day07
day08
day09
day10
day11
day12
day13
day14
day15
day16
src
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path/Reference...
Paste Ctrl+V
Find Usages Alt+F7
Find in Files... Ctrl+Shift+F
Replace in Files... Ctrl+Shift+R
Analyze
Refactor
Bookmarks
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Remove Module Delete
Build Module 'JavaSE'
Rebuild Module 'JavaSE' Ctrl+Shift+F9
Java Class
Module...
Kotlin Class/File
File
Scratch File Ctrl+Alt+Shift+Insert
Directory
HTML File
Resource Bundle
EditorConfig File
Swing UI Designer
onTest {
id main(String[]
println("main begi
ut.println("divide
System.out.println("divide
} catch (DividedByZeroExceptio
e.printStackTrace();
}

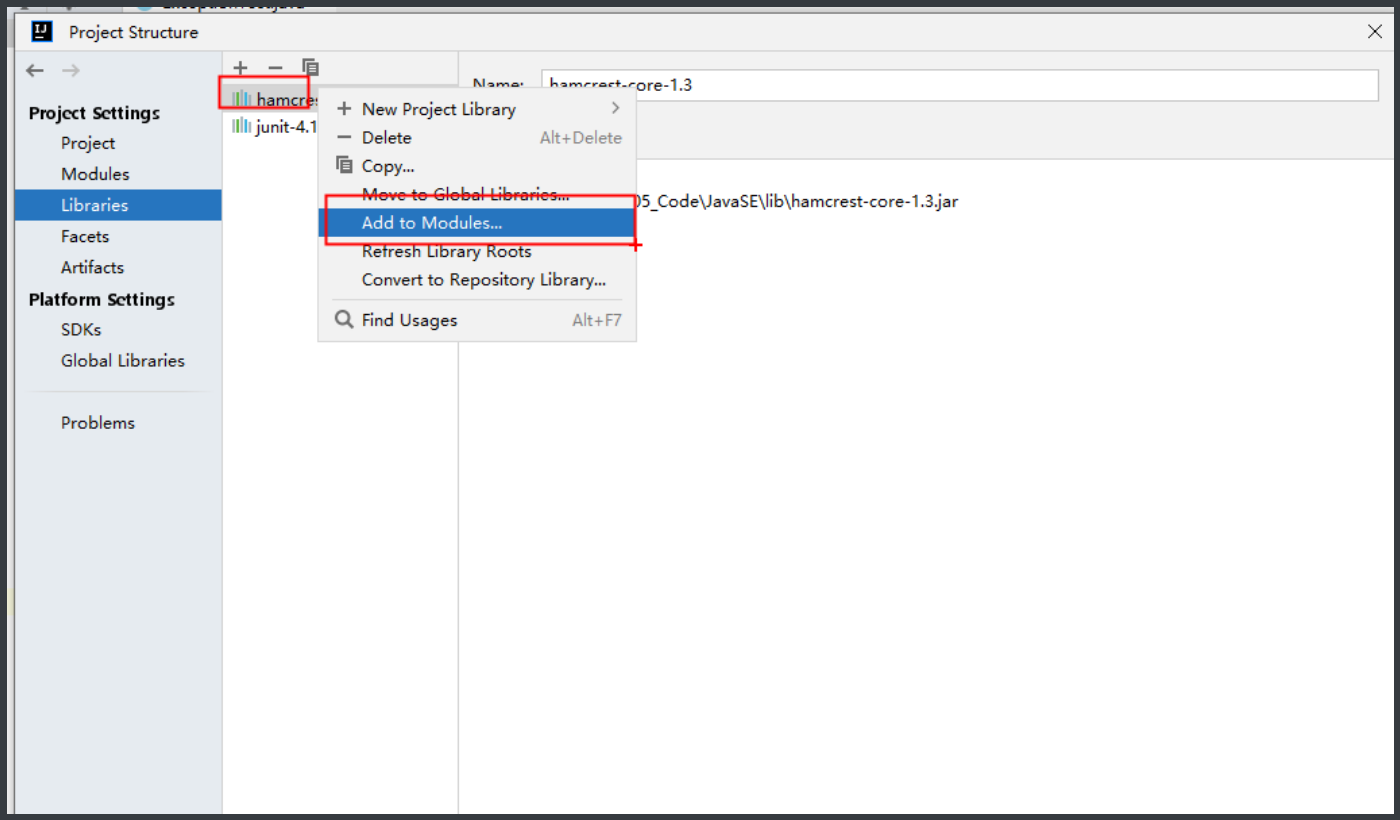

lib
hamcrest-core-
junit-4.12.jar
out
src
.gitignore
JavaSE.iml
External Libraries
Scratches and Consol
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path/Reference...
Paste Ctrl+V
Find Usages Alt+F7
Analyze
Refactor
Bookmarks
Delete... Delete
Override File Type
Run 'hamcrest-core-1.3.jar' Ctrl+Shift+F10
Debug 'hamcrest-core-1.3.jar'
More Run/Debug
Open in Right Split Shift+Enter
Open In
Local History
Repair IDE on File
Reload from Disk
Compare With... Ctrl+D
Compare File with Editor
Add as Library...
Version Control
// Build completed
public
int
retu
//public
try

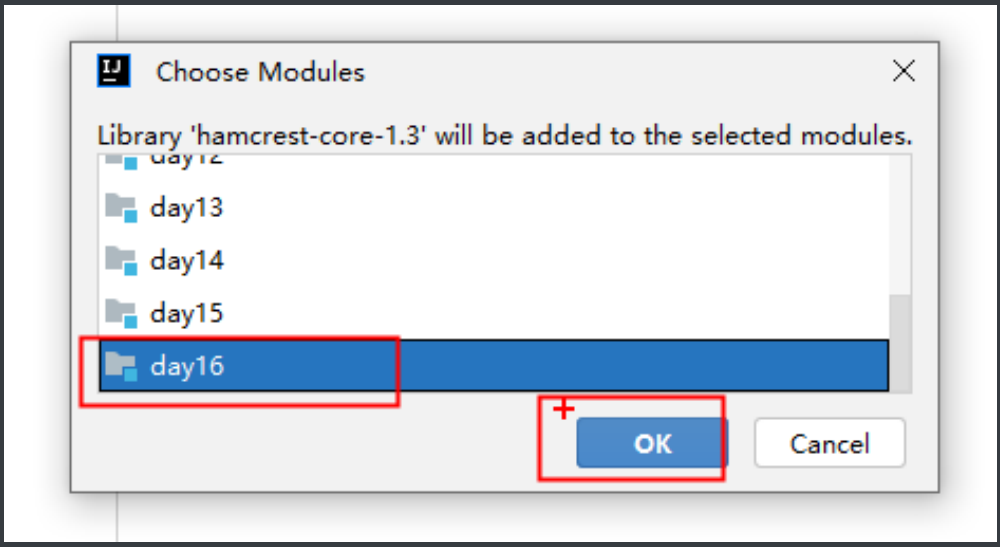

catch (DividedByZeroException e) {
e.printStackTrace();
Create Library
Name:
hamcrest-core-1.3
Level:
Project Library
Add to module:
JavaSE
OK
Cancel
(y == 0) {


lib
hamcrest-core-1.3.jar
junit-4.12.jar
out
src
.gitignore
JavaSE.iml
External Libraries
Scratches and Consoles
106
107
108
109
110
111


day13
day14
day15
day
lib
out
src
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path/Reference...
Paste Ctrl+V
Find Usages Alt+F7
Find in Files... Ctrl+Shift+F
Replace in Files... Ctrl+Shift+R
Analyze
Refactor
Bookmarks
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Remove Module Delete
Build Module 'day16'
Rebuild Module 'day16' Ctrl+Shift+F9
Open In
Local History
Repair IDE on File
Reload from Disk
Compare With... Ctrl+D
Open Module Settings F4
Load/Unload Modules...
Mark Directory as
104 e.prin
105 }
106
107 System.out
108 }
109
110 public static
111 if (y == 0
112 throw
113 }
114 int z = x
115 return z;
116 }
117
118 //public stati
119 public static
120 System.out
121 try {
122 System
123 System
Services
Build


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
hamcrest-core-1.3
junit-4.12
Name:
hamcrest-core-1.3
Classes
D:\Atguigu\05_Code\JavaSE\lib\hamcrest-core-1.3.jar


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
New Project Library
Delete
Alt+Delete
Copy...
Move to Global Libraries...
Add to Modules...
Refresh Library Roots
Convert to Repository Library...
Find Usages
Alt+F7
05_Code\JavaSE\lib\hamcrest-core-1.3.jar



第6章 面向对象1

3条主线

1.java类及类的成员(member)

1. 属性(field)
2. 方法(method)
3. 构造器(constructor)
4. 语句块
5. 内部类

2.面向对象的三大特征

1. 封装(encapsurlation)
2. 继承(Inheritance)
3. 多态(polymorphism)

3.其它关键字

this, package, import, super, extends, implements, static, final, abstract, interface .....

面向对象代码

```
/**
 * 类 ： 某种事物的描述，是对象的模板
 *       属性 ： 描述事物的特征
 *       方法 ： 描述事物的行为
 *
 * 对象 ： 这类事物中的一个个体(实例)
 *
 * 成员变量 ： 在堆内存中，寿命长，有自动初始化值0
 * 局部变量 ： 在栈内存中，寿命短，没有自动初始化，必须由程序员手工初始化.
 *
 * 匿名对象
 *       我们也可以不定义对象的引用，而直接调用这个对象的方法。这样的对象叫做匿名对象。
 *       如: new Person().shout();
 *
 * 使用情况
 *       如果对一个对象只需要进行一次方法调用，那么就可以使用匿名对象。
 *       我们经常将匿名对象作为实参传递给一个方法调用。
 *
 * private 可以修饰所有成员，表示此成员不允许外部直接使用.
 * 只能在本类中使用.
 *
 * 为属性赋值
 * public void setXxx(xxx) {
 *
 * }
```

```
 * 获取属性值
 * public xxx getXxx() {
 *
 * }
 *
 * 封装 : 成员私有化，使得使用者外部类只能通过方法单间访问成员，在方法中加入逻辑判断，进而保护对象内部的数据.
 *
 * this关键字用于表示对象，必须用在方法中，特指的就是此方法的调用者对象. 是变化的.
 *
 * 成员之三 : 构造器(constructor)
 *       作用 : 创建对象时为对象作初始化工作的特殊方法. 也称为初始化器，因为它的名字就是 <init>
 *
 * 构造器(构造方法) 特点 :
     *   1) 方法名必须和类名一致，规范中唯一允许首字母大写.
     *   2) 不声明返回值类型，连void也没有
     *   3) 不能被一些关键字修饰，static，final，abstract，....
     *   4) 不能像普通方法一样随意调用，只能在创建对象时调用仅有一次.
 *
 * 如果在类中并没有写任何构造器时，类中仍然有一个构造器(编译器帮助我们添加的)
 * 称为缺省构造器 : 1) 修饰符和类一致，2) 无参，3) 无语句.
 *
 * 在类中随意添加了构造器，编译器就不再添加缺省构造器了.
 *
 * 同一个类中可以有多个构造器，形成重载，参数必须不同.
 *
 * 注意 :
 *       this(...);必须在构造器中的第一行.
 *       必须要保证有一个构造器中没有this(...)，否则会形成无限递归.
 *
 * Java语言中，每个类都至少有一个构造器
 * 默认构造器的修饰符与所属类的修饰符一致
 * 一旦显式定义了构造器，则系统不再提供默认构造器
 * 一个类可以创建多个重载的构造器
```

```
 * 父类的构造器不可被子类继承
 */
public class Teacher {

    // 成员变量 : 有JVM自动初始化的值0
    // 对象属性或实例变量
    private String name = "某老师"; // 显式赋值, 作用于所有对象.
    private int age = 22;
    private String gender ; // 如果属性没有显式初始化赋值, 也有0

    /**
     * 无参构造器, 所有属性都是无法变化.
     * 缺点 : 创建出来的对象都长的一样.
     * 优点 : 使用简单, 最重要的构造器.
     */
    public Teacher() { // 无参构造器 : public void <init>()
        this("佟刚", 40, "男"); // 必须在第一行的效果就是它的执行一定是先于本构造器的其他代码.
        /*
        this.name = "佟刚";
        this.age = 40;
        this.gender = "男";
        */
        System.out.println("Teacher()...");
    }

    /*
    public Teacher(String name, int age) { // 有参构造器 public void <init>(String name, int age) {}
        this(name, age, "男"); // 连环调用了3参构造器
        //this.name = name;
        //this.age = age;
        //this.gender = "男";
        System.out.println("Teacher(String name, int age)...");
    }
    */
```

```
    /**
     * 全参构造器, 所有属性都能变化的初始化
     * 优点 : 创建出来的对象的所有属性都是个性化且一步到位.
     * 缺点 : 使用复杂
     */
    public Teacher(String name, int age, String gender) {
        this.name = name;
        this.setAge(age);
        this.gender = gender;
        System.out.println("Teacher(String name, int age, String
gender)...");
    }

    /*
    public void Teacher() { // 普通方法
        System.out.println("public void Teacher() {}");
    }*/

    // 间谍方法, 用于间接访问age属性
    // 为属性赋值, setAge, 有参, 无返回
    public void setAge(int age) {
        if (age < 0 || age > 130) { // 如果参数中的数据是非法数据, 可以直接
让方法弹栈
            return;
        }
        // 栈中的数据向堆中的变量赋值的过程!!
        // 如果局部变量和成员变量出现重名时, 必须加上this限定来区分成员变量.
        this.age = age;
    }

    // 获取属性值 getAge, 无参, 有返回
    public int getAge() {
        return this.age;
    }
```

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

    public String getName() {
        return name;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public String getGender() {
        return gender;
    }

    // 成员方法
    // 对象方法或实例方法
    public void lesson() {
        System.out.println(this.say() + "老师在上课"); // 成员互访
    }

    public void eat(String some) {
        System.out.println(name + "老师在吃" + some);
    }

    // 返回对象的详细信息字符串, 对象的所有属性值的拼接长串
    public String say() {
        return "姓名 : " + this.name + ", 年龄 : " + this.age + ", 性别
: " + gender;
    }

}


public class TeacherTest {
```

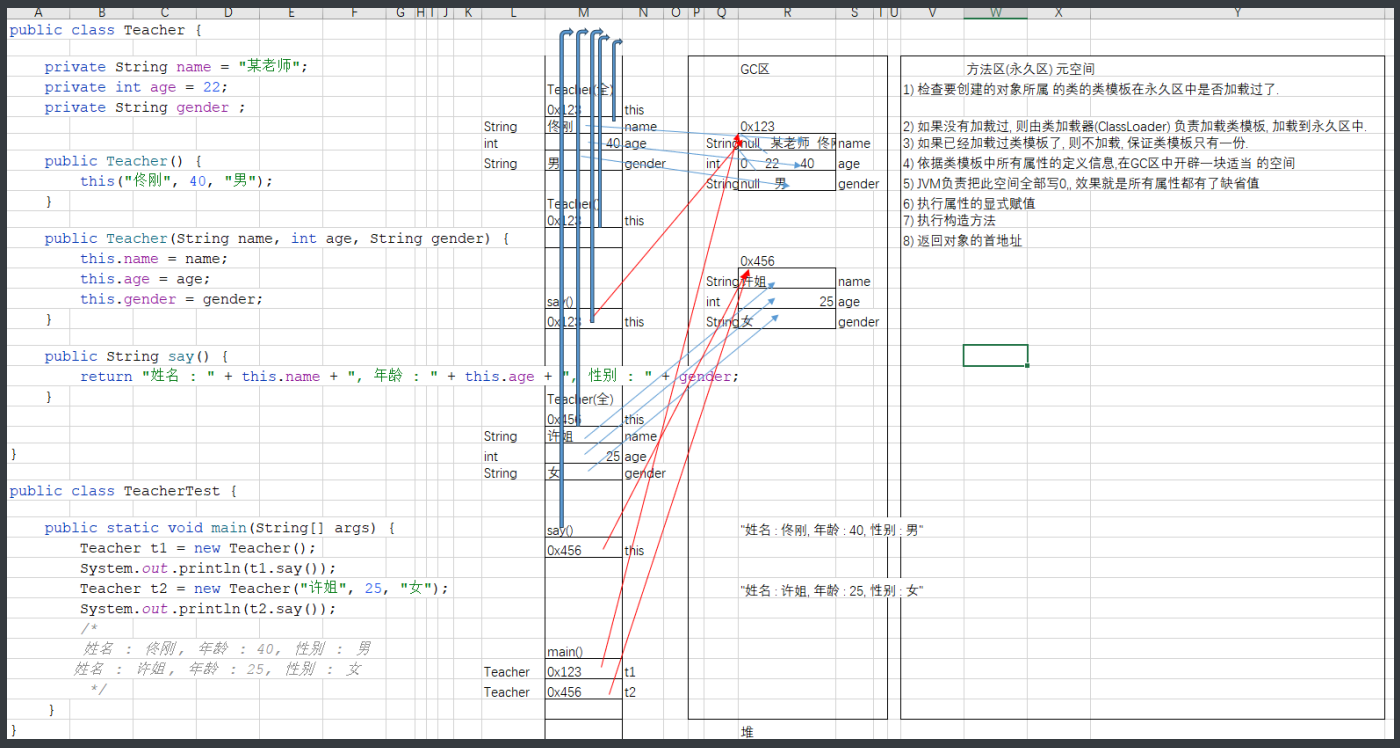
```java
    public static void main(String[] args) {
        Teacher t1 = new Teacher();
        System.out.println(t1.say());
        Teacher t2 = new Teacher("许姐", 25, "女");
        System.out.println(t2.say());
    }

    public static void main5(String[] args) {
        Teacher t1 = new Teacher();
        /*
        t1.setName("佟刚");
        t1.setAge(40);
        t1.setGender("男");
        */
        System.out.println(t1.say());

        //Teacher t2 = new Teacher("大海", 30);
        //System.out.println(t2.say());

        Teacher t3 = new Teacher("许姐", 20, "女");
        System.out.println(t3.say());
    }

    /*
    public static void main4(String[] args) {
        Teacher t = new Teacher();
        //t.name = "许姐";
        t.setName("许姐");
        //t.age = -200; // age属性是私有的, 在测试类中不能直接访问了.
        t.setAge(20); // 间接访问属性
        //t.gender = "女";
        t.setGender("女");

        //System.out.println(t.name);
        System.out.println(t.getName());
```

```
        //System.out.println(t.age);
        System.out.println(t.getAge()); // 间接访问属性
        //System.out.println(t.gender);
        System.out.println(t.getGender());

        System.out.println(t.say());
        System.out.println(t.say());
        System.out.println(t.say());
        System.out.println(t.say());
        System.out.println(t.say());

        //t.Teacher();
    }
    */

    /*

    public static void main4(String[] args) {
        Teacher t1 = new Teacher();
        t1.name = "佟刚";

        Teacher t2 = new Teacher();
        t2.age = 30;

        System.out.println(t1.say());
        System.out.println(t2.say());
    }

    public static void main3(String[] args) {
        Teacher t1 = new Teacher();
        Teacher t2 = new Teacher();
        t1.name = "佟刚";
        t1.age = 40;
        t1.gender = "男";

        t2.name = "许姐";
```

```
        t2.age = 25;
        t2.gender = "女";

        System.out.println(t1.say());
        System.out.println(t2.say());

        System.out.println("************************");
        // 引用交换
        Teacher tmp = t1;
        t1 = t2;
        t2 = tmp;

        System.out.println(t1.say());
        System.out.println(t2.say());
    }

    public static void main2(String[] args) {
        Teacher t1 = new Teacher();
        Teacher t2 = new Teacher();
        t1.name = "佟刚";
        t1.age = 40;
        t1.gender = "男";

        t2.name = "许姐";
        t2.age = 25;
        t2.gender = "女";

        System.out.println(t1.say());
        System.out.println(t2.say());

        //t1 = null; // 赋值为null也可以导致它指向的对象变成垃圾
        t1 = t2; // 极具危险性，无情地刷新t1中的地址，原来的地址消失，它原来指向的对象也没有引用指向了. 原来的对象将会变成垃圾对象
        // 垃圾对象 : 不再有引用指向的对象.
```

```
        t1.age = 20; // 虽然通过t1修改对象属性，但是它和t2是一回事，所以它的修改会影响到t2
        System.out.println(t2.say());
    }
    public static void main1(String[] args) {
        Teacher t; // 声明一个Teacher类类型的一个引用型变量，用于保存对象地址
        t = new Teacher(); // 在堆内存中创建一个Teacher类型的对象，也称为Teacher类的实例化. 并把对象的首地址写入t引用变量.
        // 对象的使用必须通过对象的引用来完成
        t.name = "佟刚";
        t.age = 40;
        t.gender = "男";

        System.out.println(t.name);
        System.out.println(t.age);
        System.out.println(t.gender);

        t.lesson();
        t.eat("饼子");
        System.out.println(t.say());
    }

     */
}
```

创建对象过程



每日一考_day09

1　什么是类, 什么是对象, 什么是实例?

类是对某种事物的描述，是一个模板

对象是某类事物实实在在的存在体，也称为实例

2　类中有哪些成员(3个)? 各有什么作用? 成员意味着什么? 什么是封装 ?

属性(field)，方法(method)，构造器(constructor)

属性描述事物的特征

方法描述事物的行为

构造器创建对象时为对象做初始化工作, 特殊 :

1) 不能像普通方法一样随意调用.

2) 方法名必须和类名一致

3) 不能有返回值. 甚至连void也没有

4) 不能被一些关键字修饰.

成员意味着可以互访

封装：成员私有化，使得使用者外部类只能通过方法访问成员，可在方法中加入逻辑判断来保护对象内部属性

3 Teacher t1 = new Teacher(); // A

```
Teacher t3 = t1;
Teacher t2 = new Teacher(); // B
t1 = t2;
t2 = t3;
```

t1 = null; // 导致B对象变成垃圾对象

t2 = null;

```
以上代码中创建了几个对象?
A. 1个 B.2个  C.3个  D.4个
```

B

4 创建一个对象的步骤(7步)

1.检查类模板在永久区内是否被加载过

2.若没有则使用类加载器加载，若已经加载则确保类模板的唯一性

3.依据类模板中所有属性的定义信息(修饰符, 数据类型, 属性名) ， 在GC区中开辟一块适当的空间

4.Jvm负责将此空间全部写0, 所有属性都自动初始化为0

5.执行属性的显式赋值语句

6.执行构造方法

7.返回对象的首地址

5 什么是垃圾对象? 垃圾对象会被立刻清理吗? 如何清理垃圾?

不被引用的对象为垃圾对象

不会

GC清理垃圾

JVM回收空间，将其空间标记为可用状态.

GC判定一个对象是垃圾对象???

对象可达性判断 : 从root引用追踪, 如果某个对象在树中不存在引用, 就认为这个对象是垃圾对象.

类模板：

保存在永久区(方法区, 元空间 meta space), meta data 元

类模板中包含 :

所有的属性的定义信息

所有的方法的代码. (不同的对象调用相同方法时, 如何区分呢? 就是通过方法的调用者对象this来区分)

现实中是先有对象, 后有类

计算机中是先有类, 后有对象.

封装 :

1) 成员私有化, 保护

2) 功能性, 该我做的事情, 义不容辞, 不该我做的事情, 绝不多管闲事.

对象传递

```
public class Teacher {

    private String name;
    private int age;
    private String gender;

    public Teacher() {}
```

```
    public Teacher(String name, int age, String gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }

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

    public String getName() {
        return name;
    }

    public void setAge(int age) {
        if (age > -1 && age < 130) {
            this.age = age;
        }
    }

    public int getAge() {
        return age;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public String getGender() {
        return gender;
    }

    public String say() {
        return "姓名 : " + this.name + ", 年龄 : " + age + ", 性别 : " +
gender;
```

```
    }

    public void lesson(Computer com) {
        System.out.println(name + "老师在使用电脑[" + com.say() + "]上
课");
    }

    public void eat(String some) {
        System.out.println(name + "老师在吃" + some);
    }

}

package com.atguigu.javase.javabean;

/**
 * 描述计算机
 */
public class Computer {

    private double cpu;
    private int memory;
    private String keyboard;

    public Computer() {}

    public Computer(double cpu, int memory, String keyboard) {
        this.cpu = cpu;
        this.memory = memory;
        this.keyboard = keyboard;
    }

    public void setCpu(double cpu) {
        this.cpu = cpu;
    }
```

```
    public double getCpu() {
        return cpu;
    }

    public void setMemory(int memory) {
        this.memory = memory;
    }

    public int getMemory() {
        return memory;
    }

    public void setKeyboard(String keyboard) {
        this.keyboard = keyboard;
    }

    public String getKeyboard() {
        return keyboard;
    }

    public void powerOn() {
        System.out.println("开机中.....");
    }

    public String say() {
        return "CPU : " + cpu + ", 内存 : " + memory + ", 键盘 : " +
keyboard;
    }

}



public class TeacherTest {

    public static void main(java.lang.String[] args) {
```

```
        Teacher t = new Teacher("张三", 28, "男");

        Computer c = new Computer(3.2, 16, "小米");
        t.lesson(c); // 对象传递


    }
}
```

JavaBean

```
/**
 * javabean ： 咖啡豆 ， 是一种java语言的可重用组件.
 * 类的要求 ：
 *       1）类是公共的 ： 方便此类在任意位置使用
 *       2）有公共无参构造器 ： 方便地创建对象
 *       3）有属性，一个对象就能携带多个数据
 */
```

package

```
/**
 * package ： 给此源文件中的所有类打入包目录中，必须写在源文件中的最上面，作用于所有类.
 * 包名规范 ： 字母全小写，多个单词用.隔开
 * package 包名.子包名.子子包名.子子子包名;
 * package 机构类型.机构名称.项目名称.模块名称;
 * package com.atguigu.javase.javabean;
 *
 * 一旦使用了package语句，会带来2个麻烦
```

```
 *      1) 编译此源文件时, 必须使用 -d选项
 *          javac -d 目标目录 源文件.java
 *      2) 再在其他包中使用本类时, 必须使用本类的全限定名称(fully qualified name)
 *              包名.子包名.子子包名.子子子包名.类名
 *
 */
/**
 * import : 导入其他包的类, 必须指明它的全限定类名
 * 一旦导入了某个类, 再在本文件中使用此类时就可以使用简单类名.
 * java.lang包是最特殊的包, 此包中的所有类的使用都可以简单化.
 */
```

对象关联

```
package com.atguigu.javase.javabean;

/**
 * 对象关联 : 一个对象拥有另外一个对象
 * 如何关联 : 把另一个对象作为我的属性即可.
 * 为什么要关联 : 为了在本类中方便地频繁地使用一个对象.
 * 需要处理 :
 *      1) 全参构造
 *      2) 提供get/set方法
 *      3) 修正say()方法
 */
public class Teacher {
    // 练习 Person, name, height, weight, public void wangzhe(Phone p)
    // 类Phone, os, screen, memory
    // 新加一个方法 public vod taobao().
    // 在测试类中创建对象并调用这2个方法.
    private String name;
    private int age;
    private String gender;
```

```
    private Computer myComputer; // 拥有

    public Teacher() {}

    public Teacher(String name, int age, String gender, Computer
myComputer) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.myComputer = myComputer;
    }

    public void setMyComputer(Computer myComputer) {
        this.myComputer = myComputer;
    }

    public Computer getMyComputer() {
        return myComputer;
    }

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

    public String getName() {
        return name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int getAge() {
        return age;
    }
```

```
    public void setGender(String gender) {
        this.gender = gender;
    }

    public String getGender() {
        return gender;
    }

    public String say() {
        return "姓名 : " + name + ", 年龄 : " + age + ", 性别 : " +
gender + ", 我的电脑 : " + myComputer.say();
    }

    //public void lesson(Computer c) {
    public void lesson() {
        System.out.println(name + "老师在使用电脑[" + myComputer.say() +
"]上课");
    }

    public void film() {
        System.out.println(name + "老师在使用电脑[" + myComputer.say() +
"]看电影");
    }

    public void eat(String some) {
        System.out.println(name + "老师在吃" + some);
    }

}


package com.atguigu.javase.javabean;

public class Computer {

    private double cpu;
```

```
    private int memory;

    public Computer() {}
    public Computer(double cpu, int memory) {
        this.cpu = cpu;
        this.memory = memory;
    }

    public void setCpu(double cpu) {
        this.cpu = cpu;
    }

    public double getCpu() {
        return cpu;
    }

    public  void setMemory(int memory) {
        this.memory = memory;
    }

    public  int getMemory() {
        return memory;
    }

    public String say() {
        return "CPU : " + cpu + ", 内存 : " + memory;
    }

}



package com.atguigu.javase.test;

import com.atguigu.javase.javabean.Computer;
```

```
import com.atguigu.javase.javabean.Teacher;

public class TeacherTest {

    public static void main(String[] args) {
        Computer c = new Computer(3.8, 16);
        Teacher t = new Teacher("张伟", 20, "男", c); // 99.99%情况下就是通过构造器关联
        /*
        B b = new B();
        A a = new A(b);
        */

        //t.lesson(c1); // 对象传递.
        t.lesson();
        t.film();
    }
}
```

第7章 数组2

基本类型数组

```
package com.atguigu.javase.array;

public class ArrayTest {

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

```
        int[] arr = new int[5];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = (int) (Math.random() * 20);
        }
        // 遍历
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
        /*
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] > arr[j]) {
                    int tmp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = tmp;
                }
            }
        }*/
        // 选择排序，以位置为主导
        for (int i = 0; i < arr.length - 1; i++) {
            // 以i下标位置为基准位置，目标就是把包括基准位置在内到后面的所有数的数列中的最小值放入基准位置
            int minIndex = i; // 假定基准位置的值最小
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j; // 记录了更小的值的下标.
                }
            }
            // 循环结束 后，minIndex就记录了最小值下标
            int tmp = arr[i];
            arr[i] = arr[minIndex]; // 最小值归位
            arr[minIndex] = tmp;
        }
        // 遍历
        for (int tmp : arr) {
```

```
            System.out.print(tmp + " ");
        }
        System.out.println();
        // 把学生数组按分数选择排序
    }

    public static void main7(String[] args) {
        int[] arr = new int[8];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = (int) (Math.random() * 20);
        }
        // 遍历
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
        // 冒泡
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 0; j < arr.length - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    int tmp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = tmp;
                }
            }
        }
        // 遍历
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
    }

    public static void main6(String[] args) {
        int[] arr = new int[8];
        for (int i = 0; i < arr.length; i++) {
```

```
            arr[i] = (int) (Math.random() * 20);
        }
        // 遍历
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
        //0  1 2  3  4 5  6 7
        //16 2 12 11 0 19 1 2
        //取子数组，取所有奇数.
        // 1) 创建新数组,和老数组一样长，防止全是有效元素
        int[] newArr = new int[arr.length];
        // 2) 声明变量计数器，初始值为0. 它是算法灵魂，有2个功能 1) 本职工作:新数组中已经插入的元素的计数 2) 新数组插入有效元素的下标指示器
        int count = 0;
        // 3) 遍历老数组
        for (int i = 0; i < arr.length; i++) {
            // 4) 如果某元素满足了某种条件
            if (arr[i] % 2 != 0) {
                // 5) 把满足条件的元素插入到新数组中，下标由计数器的当前值决定
                newArr[count] = arr[i];
                // 6) 调整计数器，计数器++，有2个效果 1) 计数 2) 下标指示器向右移动一个单位
                count++;
            }
        }
        // 7) 把新数组缩减为计数器个元素的结果数组
        int[] resultArr = new int[count];
        for (int i = 0; i < count; i++) {
            resultArr[i] = newArr[i];
        }

        for (int tmp : resultArr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
```

```java
        /*
        int[] a1 = new int[0]; // 空数组 , 不能存储元素及获取元素.
        int[] a2 = null; // 空 , 会出现 空指针异常

        String s1 = "";
        String s2 = null;
        */
    }

    public static void main5(String[] args) {
        int[] arr = new int[8];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = (int) (Math.random() * 20);
        }
        // 遍历
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
        //0  1 2  3  4 5  6 7
        //16 2 12 11 0 19 1 2
        //扩容 1.5倍
        // 1) 创建新数组, 长度比原来长
        int[] newArr = new int[(int)(arr.length * 1.5)];
        // 2) 依次复制元素到新数组中
        for (int i = 0; i < arr.length; i++) {
            newArr[i] = arr[i];
        }
        // 3) 老引用指向新数组
        arr = newArr; // 老数组对象变垃圾
        // 遍历
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
```

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

    public static void main4(String[] args) {
        int[] arr = new int[8];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = (int) (Math.random() * 20);
        }
        // 遍历
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
        //0  1 2  3  4 5  6 7
        //16 2 12 11 0 19 1 2
        // 数组缩减
        // arr.length = 4;
        // 1) 创建新数组, 容量比老数组小
        int[] newArr = new int[arr.length / 2];
        // 2) 依次把老数组中的前几个元素复制到新数组相应的位置中
        for (int i = 0; i < newArr.length; i++) {
            newArr[i] = arr[i];
        }
        // 3) 老引用指向新数组,
        arr = newArr; // 此时老数组对象就不再有引用指向了, 变成了垃圾对象

        // 遍历, 新数组
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
    }

    public static void main3(String[] args) {
        int[] arr = new int[8];
        for (int i = 0; i < arr.length; i++) {
```

```
            arr[i] = (int) (Math.random() * 20);
        }
        // 遍历
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
        //0  1 2  3  4 5  6 7
        //16 2 12 11 0 19 1 2
        //数组的复制，下面的写法不是复制.
        //int[] copy = arr;
        //copy[0] = 100;
        // 1) 创建新数组
        int[] copy = new int[arr.length];
        // 2) 依据把所有老数组中的元素都复制到新数组的对应位置中
        for (int i = 0; i < arr.length; i++) {
            copy[i] = arr[i];
        }

        //copy[0] = 100; // 不会影响 arr
        // 遍历，副本数组
        for (int tmp : copy) {
            System.out.print(tmp + " ");
        }
        System.out.println();

        // 遍历
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
    }

    public static void main2(String[] args) {
        int[] arr = new int[8];
        for (int i = 0; i < arr.length; i++) {
```

```
            arr[i] = (int)(Math.random() * 20);
        }
        // 遍历
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
        //0  1 2  3  4 5  6 7
        //16 2 12 11 0 19 1 2
        // 找最大值下标
        int maxIndex = 0;
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] > arr[maxIndex]) {
                maxIndex = i;
            }
        }
        System.out.println("最大值 arr[" + maxIndex + "] : "  +
arr[maxIndex]);
        // 反转
        for (int i = 0; i < arr.length / 2; i++) {
            // 交换i和length - 1 - i
            int tmp = arr[i];
            arr[i] = arr[arr.length - 1 - i];
            arr[arr.length - 1 - i] = tmp;
        }
        // 遍历
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
    }

    public static void main1(String[] args) {
        int[] arr = new int[8];
        // 经典for循环, 适合于数组的赋值或和下标操作相关的
        for (int i = 0; i < arr.length; i++) {
```

```
            arr[i] = (int)(Math.random() * 20);
        }
        // 遍历
        for (int i = 0; i < arr.length; i++) {
            int tmp = arr[i];
            //tmp *= 10; // 并不会导致数组内部元素的值的变化. 修改的是栈中的数
据
            //arr[i] *= 10; // 修改会对数组产生影响, 修改的是堆中的数据
            System.out.print(tmp + " ");
        }
        System.out.println();

        // 增强型for循环, 遍历数组是安全的, 因为它是只读式
        // for (元素数据类型 临时变量 : 数组名) {
        //      访问临时变量.
        // }
        for (int tmp : arr) {
            System.out.print(tmp + " ");
        }
        System.out.println();
        // 求平均值
        int sum = 0;
        for (int tmp : arr) {
            sum += tmp;
        }
        int avg = sum / arr.length;
        System.out.println("avg = " + avg);
        // 找最大值, 最大值
        int max = arr[0];
        int min = arr[0];
        for (int tmp : arr) {
            if (tmp > max) {
                max = tmp;
            }
            if (tmp < min) {
                min = tmp;
```

```
            }
        }
        System.out.println("max = " + max);
        System.out.println("min = " + min);
    }
}
```

引用数组

```
package com.atguigu.javase.array;

import com.atguigu.javase.javabean.Student;

/**
 * 完美数组 : 没有空洞
 * 结构良好 : 有空洞，都排列在有效元素后面，随时可以缩减成完美数组
 * 结构不好 : 空洞和有效元素混乱排列.
 *
 * 容量capacity : 数组长度，存储上限
 * 计数器(有效元素个数) size : 数组中真实的有效对象的个数.
 * 容量 == 计数 : 称为数组是满的状态.
 *
 */
public class StudentArrayTest {

    public static void main(String[] args) {
        int capacity = 10; // 容量
        int size = 0;
        Student[] stuArr = new Student[capacity];
        String[] name1 = {"张", "王", "李", "赵", "杨", "刘", "马", "宋",
"陈"};
        String[] name2 = {"伟", "芳", "刚", "华", "琳", "丽", "军", "庆",
"英", "梅", "旭", "梦"};
```

```
        for (int i = 0; i < capacity; i++) {
            int id = i + 1;
            int index1 = (int) (Math.random() * 10000) % name1.length;
            int index2 = (int) (Math.random() * 10000) % name2.length;
            String name = name1[index1] + name2[index2];
            int grade = (int) (Math.random() * 6 + 1);
            double score = (int) (Math.random() * 101);
            stuArr[i] = new Student(id, name, grade, score);
            size++;
        }
        // 此时数组是完美的, size == capacity
        for (Student tmp : stuArr) {
            System.out.println(tmp.say());
        }

System.out.println("****************************************");
        // 冒泡排序
        for (int i = 0; i < capacity - 1; i++) {
            for (int j = 0; j < capacity - 1 - i; j++) {
                if (stuArr[j].getScore() > stuArr[j + 1].getScore()) {
                    Student tmp = stuArr[j];
                    stuArr[j] = stuArr[j + 1];
                    stuArr[j + 1] = tmp;
                }
            }
        }

        // 此时数组是完美的, size == capacity
        for (Student tmp : stuArr) {
            System.out.println(tmp.say());
        }

System.out.println("****************************************");
    }

    public static void main7(String[] args) {
```

```
        int capacity = 10; // 容量
        int size = 0;
        Student[] stuArr = new Student[capacity];
        String[] name1 = {"张", "王", "李", "赵", "杨", "刘", "马", "宋",
"陈"};
        String[] name2 = {"伟", "芳", "刚", "华", "琳", "丽", "军", "庆",
"英", "梅", "旭", "梦"};
        for (int i = 0; i < capacity; i++) {
            int id = i + 1;
            int index1 = (int) (Math.random() * 10000) % name1.length;
            int index2 = (int) (Math.random() * 10000) % name2.length;
            String name = name1[index1] + name2[index2];
            int grade = (int) (Math.random() * 6 + 1);
            double score = (int) (Math.random() * 101);
            stuArr[i] = new Student(id, name, grade, score);
            size++;
        }
        // 此时数组是完美的, size == capacity
        for (Student tmp : stuArr) {
            System.out.println(tmp.say());
        }

 System.out.println("****************************************");
        // 扩容
        capacity *= 1.5;
        Student[] newArr = new Student[capacity];
        for (int i = 0; i < size; i++) {
            newArr[i] = stuArr[i];
        }
        stuArr = newArr;

        // 此时数组是结构良好
        // 遍历数组
        for (Student student : stuArr) {
            if (student != null) {
                System.out.println(student.say());
```

```
            } else {
                System.out.println(student);
            }
        }
        // 尾部插入
        stuArr[size] = new Student(20, "小明", 2, 88);
        size++; // 调整计数器

        System.out.println("****************** 尾部插入
**********************");
        // 遍历数组
        for (Student student : stuArr) {
            if (student != null) {
                System.out.println(student.say());
            } else {
                System.out.println(student);
            }
        }
        // 从中间删除某个元素
        int index = 3;
        // 把目标下标位置赋值为null
        stuArr[index] = null;
        // 依次从目标下标开始,把右面的所有有效元素左称
        for (int i = index; i < size - 1; i++) {
            stuArr[i] = stuArr[i + 1];
        }
        // 之前的最后有效元素会重复, 手工置空,
        stuArr[size - 1] = null;
        // 调整计数器
        size--;
        System.out.println("***************************** 删除目标下标["
+ index +  "] 后 *************");
        for (Student student : stuArr) {
            if (student != null) {
                System.out.println(student.say());
            } else {
```

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

        Student s2 = new Student(30, "小丽", 5, 99);
        int index2 = 5;
        // 把s2插入到目标下标index2位置处
    }

    // 找出所有3年级同学
    public static void main6(String[] args) {
        Student[] stuArr = new Student[10];
        String[] name1 = {"张", "王", "李", "赵", "杨", "刘", "马", "宋",
"陈"};
        String[] name2 = {"伟", "芳", "刚", "华", "琳", "丽", "军", "庆",
"英", "梅", "旭", "梦"};
        for (int i = 0; i < stuArr.length; i++) {
            int id = i + 1;
            int index1 = (int) (Math.random() * 10000) % name1.length;
            int index2 = (int) (Math.random() * 10000) % name2.length;
            String name = name1[index1] + name2[index2];
            int grade = (int) (Math.random() * 6 + 1);
            double score = (int) (Math.random() * 101);
            stuArr[i] = new Student(id, name, grade, score);
        }
        for (Student tmp : stuArr) {
            System.out.println(tmp.say());
        }

 System.out.println("****************************************");
        // 找出所有3年级同学
        // 1) 创建新数组
        Student[] newArr = new Student[stuArr.length];
        // 2) 声明灵魂变量计数器
        int count = 0;
        // 3) 遍历数组
```

```
        for (Student tmp : stuArr) {
            // 4) 如果找到了元素
            if (tmp.getGrade() == 3) {
                // 5) 保存到新数组中, 下标由计数器控制
                newArr[count] = tmp;
                // 6) 计数器++
                count++;
            }
        }
        // 7) 缩减出来
        Student[] resultArr = new Student[count];
        for (int i = 0; i < count; i++) {
            resultArr[i] = newArr[i];
        }

        for (Student student : resultArr) {
            System.out.println(student.say());
        }
    }

    public static void main5(String[] args) {
        Student[] stuArr = new Student[10];
        String[] name1 = {"张", "王", "李", "赵", "杨", "刘", "马", "宋",
"陈"};
        String[] name2 = {"伟", "芳", "刚", "华", "琳", "丽", "军", "庆",
"英", "梅", "旭", "梦"};
        for (int i = 0; i < stuArr.length; i++) {
            int id = i + 1;
            int index1 = (int) (Math.random() * 10000) % name1.length;
            int index2 = (int) (Math.random() * 10000) % name2.length;
            String name = name1[index1] + name2[index2];
            int grade = (int) (Math.random() * 6 + 1);
            double score = (int) (Math.random() * 101);
            stuArr[i] = new Student(id, name, grade, score);
        }
        for (Student tmp : stuArr) {
```

```
            System.out.println(tmp.say());
        }

System.out.println("****************************************");
        // 扩容1.5倍
        // 1) 创建新数组
        Student[] newArr = new Student[stuArr.length * 3 / 2];
        // 2) 复制元素
        for (int i = 0; i < stuArr.length; i++) {
            newArr[i] = stuArr[i];
        }
        // 3) 修改引用
        stuArr = newArr;
        for (Student tmp : stuArr) {
            if (tmp == null) {
                System.out.println(tmp);
            } else {
                System.out.println(tmp.say());
            }
        }

System.out.println("****************************************");
    }

    public static void main4(String[] args) {
        Student[] stuArr = new Student[10];
        String[] name1 = {"张", "王", "李", "赵", "杨", "刘", "马", "宋",
"陈"};
        String[] name2 = {"伟", "芳", "刚", "华", "琳", "丽", "军", "庆",
"英", "梅", "旭", "梦"};
        for (int i = 0; i < stuArr.length; i++) {
            int id = i + 1;
            int index1 = (int) (Math.random() * 10000) % name1.length;
            int index2 = (int) (Math.random() * 10000) % name2.length;
            String name = name1[index1] + name2[index2];
            int grade = (int) (Math.random() * 6 + 1);
```

```
            double score = (int) (Math.random() * 101);
            stuArr[i] = new Student(id, name, grade, score);
        }
        for (Student tmp : stuArr) {
            System.out.println(tmp.say());
        }

System.out.println("****************************************");
        // 缩减成一半
        // 创建新数组，容量比原来小
        Student[] newArr = new Student[stuArr.length / 2];
        // 依次复制元素
        for (int i = 0; i < newArr.length; i++) {
            newArr[i] = stuArr[i];
        }
        // 老引用指向新数组，丢弃老对象
        stuArr = newArr;
        for (Student tmp : stuArr) {
            System.out.println(tmp.say());
        }

System.out.println("****************************************");
    }

    public static void main2(String[] args) {
        Student[] stuArr = new Student[10];
        String[] name1 = {"张", "王", "李", "赵", "杨", "刘", "马", "宋",
"陈"};
        String[] name2 = {"伟", "芳", "刚", "华", "琳", "丽", "军", "庆",
"英", "梅", "旭", "梦"};
        for (int i = 0; i < stuArr.length; i++) {
            int id = i + 1;
            int index1 = (int) (Math.random() * 10000) % name1.length;
            int index2 = (int) (Math.random() * 10000) % name2.length;
            String name = name1[index1] + name2[index2];
            int grade = (int) (Math.random() * 6 + 1);
```

```
            double score = (int) (Math.random() * 101);
            stuArr[i] = new Student(id, name, grade, score);
        }
        for (Student tmp : stuArr) {
            System.out.println(tmp.say());
        }

System.out.println("****************************************");
        // 浅复制即可
        Student[] copy = new Student[stuArr.length];
        for (int i = 0; i < stuArr.length; i++) {
            copy[i] = stuArr[i];
        }
        for (Student tmp : copy) {
            System.out.println(tmp.say());
        }
    }

    public static void main3(String[] args) {
        Student[] stuArr = new Student[10];
        String[] name1 = {"张", "王", "李", "赵", "杨", "刘", "马", "宋",
"陈"};
        String[] name2 = {"伟", "芳", "刚", "华", "琳", "丽", "军", "庆",
"英", "梅", "旭", "梦"};
        for (int i = 0; i < stuArr.length; i++) {
            int id = i + 1;
            int index1 = (int) (Math.random() * 10000) % name1.length;
            int index2 = (int) (Math.random() * 10000) % name2.length;
            String name = name1[index1] + name2[index2];
            int grade = (int) (Math.random() * 6 + 1);
            double score = (int) (Math.random() * 101);
            stuArr[i] = new Student(id, name, grade, score);
        }
        for (Student tmp : stuArr) {
            System.out.println(tmp.say());
        }
```

```
 System.out.println("****************************************");
        // 使用下标方式找出全校谁最菜.
        int minIndex = 0; // 假定第1位同学最菜
        for (int i = 0; i < stuArr.length; i++) {
            if (stuArr[i].getScore() < stuArr[minIndex].getScore()) {
                minIndex = i;
            }
        }
        System.out.println("最菜同学 : stuArr[" + minIndex + "] : " +
stuArr[minIndex].say());
        // 反转
        for (int i = 0; i < stuArr.length / 2; i++) {
            // 交换i下标位置和length - 1 - i下标位置
            Student tmp = stuArr[i];
            stuArr[i] = stuArr[stuArr.length - 1 - i];
            stuArr[stuArr.length - 1 - i] = tmp;
        }
        for (Student tmp : stuArr) {
            System.out.println(tmp.say());
        }

 System.out.println("****************************************");
    }

    public static void main1(String[] args) {
        Student[] stuArr = new Student[10];
        String[] name1 = {"张", "王", "李", "赵", "杨", "刘", "马", "宋",
"陈"};
        String[] name2 = {"伟", "芳", "刚", "华", "琳", "丽", "军", "庆",
"英", "梅", "旭", "梦"};
        for (int i = 0; i < stuArr.length; i++) {
            int id = i + 1;
            int index1 = (int)(Math.random() * 10000) % name1.length;
            int index2 = (int)(Math.random() * 10000) % name2.length;
            String name = name1[index1] + name2[index2];
```

```
            int grade = (int)(Math.random() * 6 + 1);
            double score = (int)(Math.random() * 101);
            stuArr[i] = new Student(id, name, grade, score);
        }
        // 遍历
        // for (元素数据类型 临时变量 : 数组名) {
        //      访问临时变量.
        // }
        for (Student tmp : stuArr) {
            System.out.println(tmp.say());
        }

System.out.println("****************************************");
        // 找出全校谁最牛
        Student maxStudent = stuArr[0];
        for (Student tmp : stuArr) {
            if (tmp.getScore() > maxStudent.getScore()) {
                maxStudent = tmp;
            }
        }
        System.out.println("最牛同学 : " + maxStudent.say());
        // 找出4年级最菜同学.
        Student minStudent4 = null;
        for (Student tmp : stuArr) {
            if (tmp.getGrade() == 4) {
                // 第1位4年级同学直接刷值
                if (minStudent4 == null) {
                    minStudent4 = tmp;
                } else {
                    if (tmp.getScore() < minStudent4.getScore()) {
                        minStudent4 = tmp;
                    }
                }
            }
        }
        if (minStudent4 != null) {
```

```
            System.out.println("4年级最菜同学 : " + minStudent4.say());
        } else {
            System.out.println("没有4年级同学");
        }
    }
}
```

数组综合算法

```
package com.atguigu.javase.arraytest;

import com.atguigu.javase.javabean.Student;

public class StudentArrayTest {

    public static void main(String[] args) {
        // 数组分类 : 完美数组, 结构良好.
        // 容量 : 数组的长度
        int capacity = 10;
        // 计数 : 实际保存的元素个数
        int size = 0;
        Student[] stuArr = new Student[capacity];
        for (int i = 0; i < capacity; i++) {
            int id = i + 1;
            String name = "某同学" + (i + 1);
            int grade = (int)(Math.random() * 6 + 1);
            double score = (int)(Math.random() * 101);
            stuArr[size] = new Student(id, name, grade, score); // 尾部
插入
            size++;
        }
        // 此时数组状态 : 完美的
        // 遍历
```

```
        for (Student student : stuArr) {
            System.out.println(student.say());
        }
        System.out.println("**********************************");
        // 找最大值
        Student maxStudent = stuArr[0];
        for (Student student : stuArr) {
            if (student.getScore() > maxStudent.getScore()) {
                maxStudent = student;
            }
        }
        System.out.println("最牛 : " + maxStudent.say());
        // 有条件最大值, 3年级最牛同学的下标
        int maxIndex3 = -1; // 3年级最牛同学不存在
        for (int i = 0; i < stuArr.length; i++) {
            if (stuArr[i].getGrade() == 3) {
                if (maxIndex3 == -1) {
                    maxIndex3 = i;
                } else {
                    if (stuArr[i].getScore() >
stuArr[maxIndex3].getScore()) {
                        maxIndex3 = i;
                    }
                }
            }
        }
        if (maxIndex3 == -1) {
            System.out.println("没有3年级同学");
        } else {
            System.out.println("3年级最牛[" + maxIndex3 + "] : " +
stuArr[maxIndex3].say());
        }
        // 扩容 1.5倍
        capacity *= 1.5;
        Student[] newArr = new Student[capacity];
        for (int i = 0; i < size; i++) {
```

```
            newArr[i] = stuArr[i];
        }
        // 老引用指向新数组
        stuArr = newArr;
        System.out.println("********************扩容
**********************");
        // 遍历
        for (Student student : stuArr) {
            if (student == null) {
                System.out.println(student);
            } else {
                System.out.println(student.say());
            }
        }
        System.out.println("********************尾部插入
**************");

        // 尾部插入
        Student s1 = new Student(11, "小明", 5, 80);
        stuArr[size++] = s1;
        // 遍历
        for (Student student : stuArr) {
            if (student == null) {
                System.out.println(student);
            } else {
                System.out.println(student.say());
            }
        }
        System.out.println("***************反转 ******************");
        // 反转
        for (int i = 0; i < size / 2; i++) {
            // 交换i和size-1-i
            Student tmp = stuArr[i];
            stuArr[i] = stuArr[size - 1 - i];
            stuArr[size - 1 - i] = tmp;
        }
```

```
        // 遍历
        for (Student student : stuArr) {
            if (student == null) {
                System.out.println(student);
            } else {
                System.out.println(student.say());
            }
        }

        // 取子数组，取没有及格的同学
        // 1) 创建新数组，长度和老的一样
        Student[] newArr2 = new Student[size];
        // 2) 声明控制新数组的下标和计数器2
        int count = 0; // 算法灵魂
        // 3) 遍历老数组，找到满足条件的元素后
        for (int i = 0; i < size; i++) {
            if (stuArr[i].getScore() < 60) {
                // 4) 插入新数组中的计数器2，并调用计数器2
                newArr2[count++] = stuArr[i];
            }
        }
        // 5) 把新数组缩减成计数器2个元素
        Student[] caiStudentArr = new Student[count];
        for (int i = 0; i < count; i++) {
            caiStudentArr[i] = newArr2[i];
        }
        System.out.println("*************** 没有及格的
*****************");
        for (Student student : caiStudentArr) {
            System.out.println(student.say());
        }

        // 删除元素
        int indexToRemove = 4;
        for (int i = indexToRemove; i < size - 1; i++) {
            stuArr[i] = stuArr[i + 1];
```

```
        }
        // 之前的最后有效元素重复了
        stuArr[--size] = null;
        System.out.println("************* 删除下标[" + indexToRemove +
"] 后");
        // 遍历
        for (Student student : stuArr) {
            if (student == null) {
                System.out.println(student);
            } else {
                System.out.println(student.say());
            }
        }


        // 中间插入元素
        Student s2 = new Student(12, "小丽", 2, 90);
        int indexToInsert = 2;
        for (int i = size; i > indexToInsert; i--) {
            stuArr[i] = stuArr[i - 1];
        }
        stuArr[indexToInsert] = s2;
        ++size; // 调整计数器
        System.out.println("************* 向下标[" + indexToInsert + "]
插入新元素 后");
        // 遍历
        for (Student student : stuArr) {
            if (student == null) {
                System.out.println(student);
            } else {
                System.out.println(student.say());
            }
        }

        // 排序
        // 冒泡
```

```
        for (int i = 0; i < size - 1; i++) {
            for (int j = 0; j < size - 1 - i; j++) {
                if (stuArr[j].getScore() > stuArr[j + 1].getScore()) {
                    Student tmp = stuArr[j];
                    stuArr[j] = stuArr[j + 1];
                    stuArr[j + 1] = tmp;
                }
            }
        }
        System.out.println("************** 冒泡排序 后");
        // 遍历
        for (Student student : stuArr) {
            if (student == null) {
                System.out.println(student);
            } else {
                System.out.println(student.say());
            }
        }
        // 乱序
        for (int i = 0; i < size; i++) {
            int index1 = (int)(Math.random() * 20000) % size;
            int index2 = size - 1 - index1;
            Student tmp = stuArr[index1];
            stuArr[index1] = stuArr[index2];
            stuArr[index2] = tmp;
        }
        System.out.println("************** 乱序 后");
        // 遍历
        for (Student student : stuArr) {
            if (student == null) {
                System.out.println(student);
            } else {
                System.out.println(student.say());
            }
        }
        // 选择
```

```
        // 是位置为主导
        for (int i = 0; i < size - 1; i++) {
            // 以下标i为基准位置，目标是把包括基准位置在内到后面的所有数中的最小值放入基准位置
            int minIndex = i; // 假定基准位置的值最小
            for (int j = i + 1; j < size; j++) {
                if (stuArr[j].getScore() < stuArr[minIndex].getScore()) {
                    minIndex = j;
                }
            }
            // minIndex就保存了最小值下标
            Student tmp = stuArr[i];
            stuArr[i] = stuArr[minIndex]; // 最小值归位
            stuArr[minIndex] = tmp;
        }

        System.out.println("************** 选择排序 后");
        // 遍历
        for (Student student : stuArr) {
            if (student == null) {
                System.out.println(student);
            } else {
                System.out.println(student.say());
            }
        }
    }
}
```

二维数组

```
package com.atguigu.javase.arraytest;
```

```java
/**
 * 二维数组 : 一维数组的数组, 它的元素都是1维子数组, 本质就是引用数组
 */
public class ArrayArrayTest {

    public static void main(String[] args) {
        // 动态方式
        int[] arr1 = new int[10];

        // 静态方式1 : 适用数据量小
        int[] arr2 = new int[]{3, 4, 5, 9};
        arr2 = new int[] {9, 109};

        // 静态方式2, 使用受限 : 只能用于声明和初始化赋值在同一行语句
        int[] arr3 = {3, 2, 1, 9, 10};
        //arr3 = {1, 2};

        // 动态方式
        int[][] arrarr1 = new int[5][9];

        // 静态方式1
        int[][] arrarr21 = new int[][]{}; // 彻底的空2维数组
        int[][] arrarr22 = new int[][]{{}, {}, {}, {}}; // 拥有4个空的一维子数组的二维数组
        int[][] arrarr23 = new int[][] {{2, 3, 4}, {0, 10}, {7, 2, 8, 9, 3}};

        // 静态方式2 : 使用受限
        int[][] arrarr31 = {};
        int[][] arrarr32 = {{}, {}};
        int[][] arrarr33 = {{1}, {2}, {3, 4}};

        // 声明写法
        int[][] arrarr4; // 推荐的写法
        int[] arrarr5[];
        int arrarr6[][];
```

```
    }

    // 练习 : 创建一个拥有6个元素的二维数组, 每个子数组的长度是随机的[3~8]个元素
, 里面保存100以内的随机整数, 找出其中的最大值和最小值.
    public static void main2(String[] args) {
        int[][] arrarr = new int[6][];
        for (int i = 0; i < arrarr.length; i++) {
            arrarr[i] = new int[(int)(Math.random() * 6 + 3)];
            for (int j = 0; j < arrarr[i].length; j++) {
                arrarr[i][j] = (int)(Math.random() * 100);
            }
        }
        // 遍历
        for (int[] arr : arrarr) {
            for (int tmp : arr) {
                System.out.print(tmp + " ");
            }
            System.out.println();
        }
        System.out.println("**********************************");
        int max = arrarr[0][0];
        int min = arrarr[0][0];
        for (int i = 0; i < arrarr.length; i++) {
            for (int j = 0; j < arrarr[i].length; j++) {
                if (arrarr[i][j] > max) {
                    max = arrarr[i][j];
                }
                if (arrarr[i][j] < min) {
                    min = arrarr[i][j];
                }
            }
        }
        System.out.println("max = " + max);
        System.out.println("min = " + min);
        // 找出各组的最大值
```

```
        int[] maxArr = new int[ararr.length];
        for (int i = 0; i < ararr.length; i++) {
            maxArr[i] = ararr[i][0]; // 假定每组的第1个元素最大
            for (int j = 0; j < ararr[i].length; j++) {
                if (ararr[i][j] > maxArr[i]) {
                    maxArr[i] = ararr[i][j];
                }
            }
        }
        for (int i = 0; i < maxArr.length; i++) {
            System.out.println(maxArr[i]);
        }
    }

    public static void main1(String[] args) {
        int[][] ararr = new int[7][]; // 创建一个2维数组对象，它拥有7个子数组，但是此时全是空洞
        for (int i = 0; i < ararr.length; i++) {
            ararr[i] = new int[4 + i]; // 必须创建1维子数组对象创建好，并存入2维数组中
            for (int j = 0; j < ararr[i].length; j++) {
                ararr[i][j] = (int)(Math.random() * 50);
            }
        }
        // 遍历，
        for (int i = 0; i < ararr.length; i++) {
            for (int j = 0; j < ararr[i].length; j++) {
                System.out.print(ararr[i][j] + " ");
            }
            System.out.println();
        }
        // 求平均值
        int sum = 0;
        int count = 0;
        for (int[] arr : ararr) {
            for (int tmp : arr) {
```

```
                sum += tmp;
                count++;
            }
        }
        int avg = sum / count;
        System.out.println("avg = " + avg);
    }
}
```

第8章 面向对象2

继承

```
package com.atguigu.javase.inheritance;

/**
 * 继承 : 从现有类创建衍生子类. 现有类称为父类(时间), 基类(子类要以父类为基础),
超类(在子类中使用super关键字限定父类成员)是官方叫法.
 * 子类继承父类的所有成员. (构造器除外)
 * 继承表达所有权, 有所有权不一定意味着直接使用权, 原因是保护 .
 *
 * 子类继承父类的私有成员(表达 所有权), 但是没有直接使用权
 * 必须由父类提供公共的get/set方法, 用于在子类中继承后再间接访问.
 *
 * 继承的本质 : 子类是(is)完全的且更强大的父类
 *
 * java中只支持单继承, 不支持多重继承(一个子类有多个直接父类)
 * java也支持多层继承, 一个子类有一个直接父类, 但是又有多个间接父类.
```

```
 *
 * java中最重要的概念，没有之一.
 * 重写(override) :
 *      在子类中根据需要重写从父类继承来的方法，因为父类的方法不能满足子类的需要.
 *
 * 重写的要求 :
 *      1) 子类中的方法和父类中的方法的签名完全一致， (返回值类型,方法名 参数列表[参数个数，顺序一致，类型一致]完全一致)
 *      2) 子类重写方法的访问控制修饰符要大于等于父类被重写方法
 *      3) 父类方法不能被一些关键字修饰
 *      4) ....
 *
 * super : 表示超级的意思，子类中特指父类继承的成员限定.
 *
 * this表示的是当前方法的调用者对象整体
 * super仅仅是一个限定符，用于限定从父类继承的成员.
 *
 * this.属性 可以访问属性，从子类开始向上检索，有就近原则
 * super.属性 可以访问属性，从直接父类开始向上检索，有就近原则
 *
 * this.方法() 可以调用方法，从子类开始向上检索，有就近原则
 * super.方法() 可以调用方法，从直接父类开始向上检索，有就近原则
 *
 *
 * // 注解annotation : 就是注释，是一种特殊的注释，特殊在于可以被编译器和运行时JVM识别.
 * @Override 特别用于提醒方法覆盖的条件是否满足，请编译器帮忙作检查，如果有问题，请编译出错!!!
 *
 * 四种访问权限修饰符
 * private          本类
 * default          本类       本包其他类
 * protected        本类       本包其他类        其他包子类
 * public           全局
 *
 * 构造器虽然不能被子类继承，但是必须要被子类调用到，且默认调用的是无参构造器
```

```
 * 在子类中使用super(...) 实现对于父类构造器的显式调用, 并且此语句必须在第一行.
 *
 * 子类中所有的构造器默认都会调用父类中空参数的构造器
 * 子类构造器中的第一行要么是super(...) 要么是this(...)
 *      super(...) 直接调用父类构造器
 *      this(...)  间接调用父类构造器
 * 结论 : 子类构造器一定会"先"调用父类构造器,
 */
public class Person {

    private String name;
    private int age;
    private String gender;

    public Person() {
        System.out.println("Person()..."); // 1
    }

    public Person(String name, int age, String gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        System.out.println("Person(全)..."); // 2
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }
```

```
    public void setAge(int age) {
        this.age = age;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public void eat(String some) {
        System.out.println("吃" + some);
    }

    public void sleep() {
        System.out.println("睡...");
    }

    public String say() {
        return "姓名 : " + name + ", 年龄 : " + age + ", 性别 : " +
gender;
    }
}



package com.atguigu.javase.inheritance;

/**
 * extends 是扩展的意思.
 */
public class Student extends Person {
```

```
    /*
    String name;
    int age;
    String gender;
    */
    //int age;  // 子类中的同名属性和父类中的属性是共存关系.
    String school; // 特有成员.

    public Student() {
        //super(); // 调用父类的无参构造器, 这个语句是编译器一定会加的. 是缺省
行为
        //super("小明", 18, "男"); // 显式调用父类构造器
        this("小明", 18, "男", "atguigu"); // 本类重载的构造器的连环调用!!!
        /*
        this.setName("小明");
        this.setAge(18);
        this.setGender("男");
        */
        //this.school = "atguigu";
        System.out.println ("Student()..."); // 3
    }

    // 全参
    public Student(String name, int age, String gender, String school)
{
        /*
        super(); // 子类中所有的构造器默认都会调用父类中空参数的构造器
        this.setName(name);
        this.setAge(age);
        this.setGender(gender);
        */
        super(name, age, gender); // 显式调用父类的全参构造器
        this.school = school;
        System.out.println("Student(全)..."); // 4
    }
```

```
    /*
    public void eat(String some) {
        System.out.println("吃" + some);
    }

    public void sleep() {
        System.out.println("睡...");
    }

    public String say() {
        return "姓名 : " + name + ", 年龄 : " + age + ", 性别 : " +
gender;
    }*/

    public void study() {
        //System.out.println(name + "学生在学习"); // 成员互访，name是从父
类继承来的，是我的，但是不能直接使用
        System.out.println(getName() + "学生在学习"); // 成员互访，name是
从父类继承来的，是我的，可以间接使用
    }

    @Override
    public String say() {
        //return "姓名 : " + getName() + ", 年龄 : " + getAge() + ", 性
别 : " + getGender() + ", 学校 : " + school;
        //super.的作用就是限定从父类继承来的成员
        return super.say() + ", 学校 : " + school; // 使用连环调用好处在于
灵活性和可维护性动态性.
    }

}
```

每日一考_day12

1. 什么是继承? 为什么要继承? 如何继承?

   子类对于父类扩展

   父类不能满足需要时

   子类 extends 父类

2. 子类能继承父类的私有成员吗? 如何处理?

   能继承,

   子类继承父类的所有成员.

   继承的本质 : 子类是完全的且更强大的父类

   子类虽然能继承, 但是不能直接访问. 通常要在父类中再提供公共方法, 用于间接访问.

3. 为什么父类又叫基类或超类?

   子类以父类为基础进行扩展

   在子类中使用super关键字限定从父类继承的成员

4. 什么是方法覆盖? 方法覆盖有什么条件?

   子类根据需要重写父类的方法.

   1. 方法签名一致, 返回值类型 方法名 参数列表
   2. 访问控制修饰符 子类 大于等于 父类.

最好使用注解 @Override 修饰方法

public 全局

protected 本类 本包 其他包的子类

default 本类 本包

private 本类

5. 如果A类被B类继承,B类又被C类继承, 在所有类中都有一个方法test(),
   创建C类对象,调用test()方法,执行的是哪个类的方法? 在C类中有几个test()方法?

   执行C类中的方法

   从测试类角度看 C类中有1个方法

   在C类的内部看, C类中有2个方法, this.test(), 另一个是super.test()

   从继承的概念来看, C类中有3个方法,

多态

```
package com.atguigu.javase.polymorphism;

/**
 * 多态 :
 *       从右向左 : 右面是子类对象  左面是多种父类引用
 *       从左向右 : 左面是父类类型的引用, 实际可以指向多种不同子类对象
 * 本态 :
 *       子类对象的子类形态
```

```
 *
 * 多态引用 : 子类对象赋值于父类类型的引用变量
 *
 * 虚拟方法 : 在父类中应该具有的行为, 但是父类类型的模糊性导致 此行为无法具体清晰化.
 *
 * 若编译时类型和运行时类型不一致, 就出现多态 (Polymorphism)
 *
 * 成员方法具备多态性 : 父子类中的同名方法是覆盖关系
 * 成员变量不具备多态 : 父子类中的同名属性是共存关系, 成员变量的访问不会动态绑定, 而是只看引用的类型是什么.
 *
 */
public class Person extends Object {

    private String name;
    private int age;
    private String gender;

    public Person() {}

    public Person(String name, int age, String gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
```

```
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public String say() {
        return "姓名 : " + name + ", 年龄 : " + age + ", 性别 : " +
gender;
    }

    public void sayHello() { // 虚拟方法，虚拟在只有一个作用，就是骗过编译器
        System.out.println("打个招呼");
    }

}



package com.atguigu.javase.polymorphism;

public class PersonTest {

    /**
     * 多态参数方法
     * 此方法可以接收任意的本类或子类对象，兼容性很好
     *
```

```
     * 带来了模糊性和不确定性.
     * @param p
     */
    public static void test(Person p) {
        p.sayHello(); // 虚拟方法调用, 它是体现子类个性的最好的方式.
        //p.spring(); // 多态副作用.
        // 把p引用指向的对象 切换为 Chinese类型的视角.
        // instanceof 左面的引用指向的 "对象" 是否真的是Chinese对象)
        // 对象的类型判断的顺序必须从最子类到最父类.
        if (p instanceof Peking) {
            ((Peking)p).playBird();
        } else if (p instanceof Chinese) {
            Chinese ch = (Chinese) p; // 造型, ClassCast, 造型有风险, 必
须加上判断
            ch.spring();
        } else if (p instanceof American) {
            ((American)p).shoot();
        } else {
            System.out.println("普通人一个");
        }
    }

    public static void main(String[] args) {
        Chinese ch = new Chinese("张三", 33, "男", "马");
        American am = new American("Paul", 22, "male", true);
        Person p = new Person();
        Peking pk = new Peking("李四", 50, "男", "虎");
        test(ch);
        test(am);
        test(p);
        test(pk);
    }

    public static void main2(String[] args) {
        // 能用多态, 绝不本态
        Chinese ch = new Chinese("张三", 32, "男", "牛");
```

```java
        // 多态数组，用于统一处理不同类型的对象!!!
        Person[] arr = new Person[5];
        arr[0] = ch;
        arr[1] = new American("Rose", 20, "female", false);
        arr[2] = new Person("某人", 30, "未知");
        arr[3] = new American("Jack", 40, "male", true);
        arr[4] = new Chinese("李四", 34, "女", "蛇");

        // 遍历
        for (Person person : arr) {
            System.out.println(person.say()); // 虚拟方法调用
        }

        System.out.println("************************************");
        // 排序
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 0; j < arr.length - 1 - i; j++) {
                if (arr[j].getAge() > arr[j + 1].getAge()) {
                    Person tmp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = tmp;
                }
            }
        }

        // 遍历
        for (Person person : arr) {
            System.out.println(person.say()); // 虚拟方法调用
        }
    }

    public static void main1(String[] args) {
        // 把子类对象 "当作是" 父类类型的对象.
        Person p = new Chinese("张三", 32, "男", "牛");
        System.out.println(p.getName());
```

```
        //p.spring(); // 子类特有成员无法再访问. 多态副作用. 多态引用p具有模糊性, 所以不能冒然调用子类的个性方法.
        // 编译器关注的是引用的类型, 运行时动态绑定具体子类类型
        // 编译时看父类, 运行时看子类.
        p.sayHello(); // 虚拟方法调用 virtual method invoke. :通过多态引用调用覆盖. 实际执行子类中的方法

        p = new American("John", 38, "male", true);
        //p.shoot();
        p.sayHello();
    }
}
```

Object类

```
package com.atguigu.javase.object;

/**
 *  public boolean equals(Object obj) : 判断当前对象this和obj的内容是否相等
 *  public int hashCode() : 获取当前对象this的哈希码.
 *  public String toString() : 获取当前对象的详细信息字符串, 所有属性值的拼接串.
 *
 *  散列码
 *  特征码
 *  2个对象的equals为true时, 说明2个对象的内容相等, 内容相等它们2个对象的哈希码必须相同, 表达特征性
 *  2个对象的equals为false时, 说明2个对象的内容不等, 内容不等它们2个对象的哈希码必须不同, 表达散列性
 */
public class Point  {

    private int x;
```

```
    private int y;

    public Point() {}

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    public String say() {
        return "x : " + x + ", y : " + y;
    }

    /** 这个父类Object中的equals方法. 它根本没有比较对象的内容, 不可能完成内容比较, 这个方法烂的很
     * 子类必须重写, 才能达标
     * public boolean equals(Object obj) {
     *     return (this == obj);
     *}
     */
```

```
    @Override
    public boolean equals(Object obj) {
        // 在这里真的比较this对象的内容和obj对象的内容
        if (obj instanceof Point) {
            Point pp = (Point) obj;
            if (this.x == pp.x && this.y == pp.y) {
                return true;
            }
        }
        return false;
    }
    // 此方法是父类中的方法，它的作用是根据 对象的物理地址经过绝对散列算法．并没有体现特征性
    // 此方法也必须要经过重写才能达标
    //public native int hashCode();

    @Override
    public int hashCode() {
        // 所有属性参与就是体现特征性
        // 再有一些小变化就可以体现散列性
        return x * 7 + y * 13;
    }

    /* 此方法不好，不能表现对象的详细信息
    public String toString() {
        // 类名和哈希码16进制拼接的串
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }*/

    @Override
    public String toString() {
        return "x : " + x + ", y : " + y;
    }
}
```

```java
package com.atguigu.javase.object;

public class PointTest {

    public static void main(String[] args) {
        Object p1 = new Point(20, 90);
        Point p2 = new Point(50, 60);

        System.out.println(p1 == p2); // 比较2个对象的引用 就是比较2个引用中的地址值. 没有意义
        System.out.println(p1.equals(p2)); // equlas方法有对称性
        System.out.println(p2.equals(p1));

        //System.out.println(p1.equals("aaaaa"));
        System.out.println(p1.hashCode());
        System.out.println(p2.hashCode());

        //p1 = null;
        System.out.println(p1);
        System.out.println(p1.toString());

        // 字符串和对象拼接时 也会自动调用它的toString
        String s = "adfs";
        s += p1;
        s += 200;

        System.out.println(s);

        //System.out.println("hello" == new java.util.Date());

    }
}
```

每日一考_day13

1. 什么是多态? 如何在java中体现多态,如何使用?

   子类对象的多种父类形态

   父类引用指向多种不同子类对象

   编译型类型(引用的类型) 和运行时类型(对象的类型) 不一致

   多态引用 : 用父类引用指向子类对象

2. 什么是多态副作用? 解释原因.

   多态引用时, 不能使用子类的特有成员

   多态引用的不确定性, 导致不可以冒然使用子类特有成员.

   造型 : 把子类对象的视角切换一下.

3. 什么是虚方法调用? 有什么特点?

   多态引用调用重写的方法.

   编译时要检查引用所属的类型中是否有些行为.

   运行时会动态绑定引用指向的对象的实体的类的子类中的方法

4. instanceof 操作的符的作用是什么?

   引用 instanceof 类名

   判断左面的引用指向的对象的实体 是否是 右面的类名的一个实例

5. 描述equals方法和hashCode方法的含义和作用.

   equals方法，a.equals(b) 判断此方法的调用者对象this和参数中的对象的内容(属性值) 是否相等

   hashCode方法, 获取某个对象的哈希码.

2个对象的equals为true, 说明2个对象的内容相等, 2个对象的哈希码必须一样, 体现特征性

2个对象的equals为false, 说明2个对象的内容不等, 2个对象的哈希码必须不一样, 体现散列性.

第9章 面向对象3

static关键字

```
package com.atguigu.javase.statictest;
```

```
/**
 * static : 静态 : 它的存在是确定的,唯一的.  类就是静态的
 * 非static : 非静态(动态), 它的存在是不确定的. 对象就是非静态的.
 *
 * 属性隶属于类或者说是要被所有对象共享的数据就声明为static
 * 方法如果和调用者对象无关时, 就可以使用static修饰.
 *
 * 在Java类中, 可用static修饰属性、方法、代码块、内部类
 * 被修饰后的成员具备以下特点:
     * 随着类的加载而加载
     * 优先于对象存在
     * 修饰的成员, 被所有对象所共享
     * 访问权限允许时, 可不创建对象, 直接被类调用
 *
 * 在非静态环境中可以直接访问静态成员
 * 在静态环境中不可以直接访问非静态成员, 原因是静态方法中没有this的存在!!! super也不行
 * 非静态成员必须要依赖对象的存在而存在.
 * 所以在静态方法中通过新建对象, 再通过对象来访问对象属性和对象方法.
 *
 * this和super都是非静态的概念, 和对象相关的.
 *
 * 编译器会把所有static块和所有静态属性的显式赋值全部整合成一个方法, 按顺序整合
 * public static void <clinit>() {
 *     System.out.println("static {} 0..... 0");
 *     no = 10001;
 *
 *     company = "atguigu";
 *     no = 1;
 *     emp = new Employee();
 *
 *     System.out.println("static {} 1 ....  1");
 *     no = 101;
 *
 *     System.out.println("static {} 2. .... 2");
 *     no = 1001;
```

```
 * }
 *
 * 所有的非静态块和属性的显式赋值按顺序整合后，再和所有重载的构造器整合成多个重载的
<init>(...) {
 *
 * }
 */
class Base {

    static int no = -100;
}

public class Employee extends Base {

    // 全局常量
    final static public String DEFAULT_COMPANY;

    static {
        System.out.println("static {} 0..... 0");
        no = 10001;
        //System.out.println(no); // 访问no会出问题，编译器担心这个是不是要
访问父类中的no
        System.out.println(Base.no);
        System.out.println(Employee.no);
        DEFAULT_COMPANY = "atguigu";

    }

    // 类属性，数据只隶属于类型本身，属于类模板，会被所有对象所共享
    private static String company = DEFAULT_COMPANY;
    private static int no = 1;
    // 因为emp是静态属性，隶属于类模板，所以它永远不会消失，它指向的对象将会是一
个不死的对象.
    //private static Employee emp = new Employee();

    // 类方法，通过类来调用，不需要对象，所以称为工具方法.
```

```
    public static void test() {
        System.out.println("test()..." + company);
    }

    public static void test2() {
        /*
        this.name = "赵六";
        this.age = 45;
        this.salary = 9000;
        System.out.println(super.toString());
        */
        System.out.println("**********************************");
        //Employee emp = new Employee(); // 要想访问对象属性，必须要有对象.
        //emp.name = "赵六";
        //emp.age = 45;
        //emp.salary = 9000;
        //System.out.println(emp.toString());
    }

    static { // 静态语句块，静态初始化块，在类加载时执行仅有一次，第一次要使用某个类时,会加载这个类. 类初始化器
        // 这是一个方法 : public static void <clinit>() {}
        System.out.println("static {} 1 ....  1");
        //no = 101;
        System.out.println(no);
    }

    static {
        System.out.println("static {} 2. .... 2");
        //no = 1001;
    }

    {
        System.out.println("非静态块 0 ...");
        age = 10;
        System.out.println(this.age);
```

```
    }

    // 对象属性，这个数据将来要隶属于对象，被对象所独享
    private final int id; // 员工号，blank final 空final很危险，它必须要尽快完成一次仅有的赋值.
    private String name;
    private int age = 20;
    private double salary;

    {
        this.id = no++;
    }

    public Employee() {
        System.out.println("Employee()...");
        age = 30;
    }

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

    public Employee(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
        System.out.println("Employee.Employee(有参)...");
    }

    /*
    public void <init>() {
        System.out.println("非静态块 0 ...");
        age = 10;
        System.out.println(this.age);

        age = 20;
```

```
        System.out.println("非静态块 1 ...");
        age = 40;

        System.out.println("非静态块 2 ...");
        age = 50;

        System.out.println("Employee()...");
        age = 30;
    }

    public void <init>(String name, int age, double salary) {
        System.out.println("非静态块 0 ...");
        age = 10;
        System.out.println(this.age);

        age = 20;

        System.out.println("非静态块 1 ...");
        age = 40;

        System.out.println("非静态块 2 ...");
        age = 50;

        this.id = no++;
        this.name = name;
        this.age = age;
        this.salary = salary;
        System.out.println("Employee.Employee(有参)...");
    }
    */

    {
        // 非静态块的作用是为对象进行初始化工作的. 会在创建对象时执行仅有一次.
        // 无论是调用了哪个构造器, 非静态块都要执行仅有一次
        System.out.println("非静态块 1 ...");
```

```
        age = 40;
    }

    {
        System.out.println("非静态块 2 ...");
        age = 50;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }
```

```
    // 返回对象的详细信息

    @Override
    public String toString() {
        return "Employee{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                ", salary=" + salary +
                '}';
    }
}
```

final关键字

```
package com.atguigu.javase.statictest;

/**
 * final 可以修饰类，方法，变量
 * final修饰类表明类是终极类
 * final修饰方法表明方法是终极完美方法，不能被重写.
 * final修饰变量表明这个量是最终量，只能必须赋值仅有一次.
 *
 * public static final : 全局常量
 */

final class A {}

// class B extends A {}

class B {
    final void test() {}
}
```

```
class C extends B {

    // @Override public void test() {}
}

public class FinalTest {
}
```

抽象类

```
package com.atguigu.javase.abstracttest;

/**
 * 具体类 : 某种事物的描述
 * 抽象类 : 某大类中不同种事物的描述
 *
 * 抽象类可以包含抽象方法
 * 抽象方法 : 只有方法声明, 没有方法体, 绝对不能执行.
 */
public abstract class Pet {

    private String name;
    private int age;
    private double weight;

    public Pet() {}

    public Pet(String name, int age, double weight) {
        this.name = name;
        this.age = age;
        this.weight = weight;
    }
```

```
    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public double getWeight() {
        return weight;
    }

    public void setWeight(double weight) {
        this.weight = weight;
    }

    @Override
    public String toString() {
        return "Pet{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", weight=" + weight +
                '}';
    }

    public abstract void speak();

    public abstract void eat();
```

```
}
```

每日一考_day14

1. 被static修饰的成员有什么特点?
    1. 和类相关, 随着类的加载而加载
    2. 优先于对象存在而存在
    3. 为所有对象共享
    4. 通过类直接使用

2. 什么成员应该声明成静态的, 什么成员应该声明为非静态的?

    属性数据不再隶属于每个对象, 而是直接隶属于类型本身.

    方法的调用如果和调用者对象无关时, 就可以使用static修饰. 只需要通过类调用, 静态方法也称为工具方法

    属性数据必须要被每个对象独享的, 就是非静态属性

    方法的调用必须通过调用者对象来支持时, 就是非静态方法.

3. 静态环境中可以直接访问非静态成员吗? 非静态环境中可以访问静态成员吗? 各自的原因是什么?

   不可以, 在静态环境中, 不能直接访问非静态成员, 没有this的存在. 可以再通过new一个新的对象, 再通过对象来访问.

   可以, 因为在非静态环境中, 已经有也this的存在,

4. 写出饿汉式单例代码

```
class Singleton {
    private Singleton(){

    }
    private static Singleton instance = new Singleton();
    public static Singleton getInstance() {
        return instance;
    }
}
```

5. final可以修饰什么? 被修饰的元素具有什么特点?

final可以修饰类, 方法, 变量

final修饰类, 说明这个类是完美的, 不允许被扩展

final修饰方法, 说明这个方法是完美的, 不允许子类重写.

final修饰变量, 说明这个量是最终量, 必须赋值仅有一次.

抽象类2

```
package com.atguigu.javase.abstracttest;

/**
 * 具体类 : 某种事物描述
 * 抽象类 : 某类不同种事物描述
 *
 * 抽象类可以包含抽象方法
 * 抽象方法 : 只有方法声明，没有方法体，绝不能执行.
 *
 * 具体类不能包含抽象方法
 * 如果一个类中有抽象方法，这个类必须是抽象类
 *
 * 抽象类专门用于被具体子类继承
 *
 * 抽象方法会把压力给到子类，具有强制性.
 */
public abstract class Pet {

    private String name;
    private int age;
    private double weight;

    public Pet() {}

    public Pet(String name, int age, double weight) {
        this.name = name;
        this.age = age;
        this.weight = weight;
    }

    public String getName() {
        return name;
```

```
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public double getWeight() {
        return weight;
    }

    public void setWeight(double weight) {
        this.weight = weight;
    }

    @Override
    public String toString() {
        return "Pet{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", weight=" + weight +
                '}';
    }

    public abstract void speak();

    public abstract void eat();

}
```

```java
package com.atguigu.javase.abstracttest;

/**
 * 具体类不能包含抽象方法
 */
public class Dog extends Pet {

    private String type;

    public Dog() {}

    public Dog(String name, int age, double weight, String type) {
        super(name, age, weight);
        this.type = type;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    @Override
    public String toString() { // 锦上添花
        return super.toString() + ", 品种 : " + type;
    }

    @Override
    public void speak() { // 彻底反转 特别称为 实现(implement) : 有了方法体.
        System.out.println("汪汪汪....");
    }
```

```
    @Override
    public void eat() {
        System.out.println("狗狗吃骨头");
    }

}


package com.atguigu.javase.abstracttest;

public class PetTest {

    public static void main(String[] args) {
        // Pet pet = new Pet(); // 抽象类中可能有抽象方法,
        // 抽象类类型的引用 100%是 多态引用
        Pet pet = new Dog("小汪", 2, 8, "哈士奇");
        System.out.println(pet); // 虚拟方法调用!!!!!!!!!
        // 多态引用调用抽象方法时 100% 是 虚拟方法调用!!!
        pet.speak(); // 虚拟方法调用, 编译时看引用的类型, 运行时看引用指向的对象的类型.
        pet.eat();
    }
}
```

接口

```
package com.atguigu.javase.interfacetest;

/**
 * 具体类 : 某种事物的描述
 * 抽象类 : 某类不同种事物的描述
 * 接口 : 不同类不同种事物的共同行为的描述
```

```
 *
 * 接口中的所有方法都是公共的抽象方法，非公共方法是禁止的
 * 所以接口中的方法不再需要添加修饰符.
 *
 * 接口中只有2个成员 : 全局常量，公共抽象方法.
 *
 * 接口只能由子类实现(implements)，并要实现所有的抽象方法.
 *
 * 接口可以继承另外的接口，并且可以多重继承
 *
 * 接口通常用于表达某种能力. 接口名称通常是形容词，表达某种能力.
 * 接口也用于表达某种标准规范. 所有成员都一定是公共的，有越多的子类实现，接口越有价值.
 *
 * 面向接口 : 把所有子类对象都多态为接口(标准)类型的对象，所有的对象的个性全部抹杀.
 */
public interface Flyer {

    //int no = 1;
    void takeOff();

    void fly();

    void land();

}


package com.atguigu.javase.interfacetest;

import java.io.Serializable;

/**
 * 类可以继承另一个类的同时再实现接口
 * 类继承父类表达子类就是父类
```

```
 * 类实现接口表达的是类又额外具备了一些其他的东西.
 */
public class Bird extends Pet implements Flyer , Serializable,
Runnable, Comparable {

    private int flySpeed;

    public Bird() {}

    public Bird(String name, int age, double weight, int flySpeed) {
        super(name, age, weight);
        this.flySpeed = flySpeed;
    }

    public int getFlySpeed() {
        return flySpeed;
    }

    public void setFlySpeed(int flySpeed) {
        this.flySpeed = flySpeed;
    }

    @Override
    public String toString() {
        return super.toString() + " Bird{" +
                "flySpeed=" + flySpeed +
                '}';
    }

    @Override
    public void speak() {
        System.out.println("吱吱吱 ...");
    }

    @Override
    public void eat() {
```

```
        System.out.println("小鸟吃小虫子");
    }

    @Override
    public void takeOff() {
        System.out.println("翅膀一动就飞了");
    }

    @Override
    public void fly() {
        System.out.println("天高任我飞");
    }

    @Override
    public void land() {
        System.out.println("我爱电线杆");
    }

    @Override
    public int compareTo(Object o) {
        return 0;
    }

    @Override
    public void run() {

    }
}



package com.atguigu.javase.interfacetest;

public class FlyerTest {

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

```
        Pet pet = new Bird("小飞", 2, 0.05, 40);
        pet.speak();
        pet.eat();

        // 多态的本质 : 把子类"当作是"某个类型的对象. 造型的本质 : 切换对象的视角
        // 把pet引用指向的对象 重新"当作" 是 Flyer类型的一个对象.
        if (pet instanceof Flyer) {
            Flyer f = (Flyer) pet;
            f.takeOff();
            f.fly();
            f.land();
        }

    }

    public static void main1(String[] args) {
        //Flyer f = new Flyer();
        //System.out.println("kljsdf");
        // 把子类对象"当作"是父类(接口)类型的对象.
        // 接口类型的引用 100% 是多态引用
        Flyer f = new Plane();
        f.takeOff();
        f.fly();
        f.land(); // 100%是虚拟方法调用!!!
    }
}
```

每日一考_day15

1. 抽象类是什么? 和具体类的区别是什么?

   具体类 : 某种事物的描述, 不能包含抽象方法

抽象类 : 某类不同种事物的描述, 可以包含抽象方法

接口 : 对行为的描述, 全部都是抽象方法

2. 判断 :
    1. 抽象类中必须包含抽象方法x
    2. 抽象类中不能包含普通方法x
    3. 抽象类中可以包含抽象方法v
    4. 抽象类不能创建对象, 所以可以省略构造器x
    5. 具体类最多允许包含1个抽象方法x
    6. 抽象类主要用于被具体子类继承,具体子类可以不必理会抽象父类中的抽象方法x
    7. 具体类中如果包含抽象方法, 编译出错v
    8. 一个类中如果包含抽象方法, 这个类必须是抽象类v
    9. 抽象类中必须包含属性,构造器和普通方法和抽象方法x.
    10. 抽象类中可以不包含抽象方法,具体子类必须实现全部的父类的抽象方法.v
3. 判断:

    1 类可以继承接口,x
    2 接口也可以继承类F,
    3 接口可以继承接口, 只能单继承x
    4 接口可以继承接口, 并且可以多继承v
    5 类可以实现接口, 并对其中的抽象方法不予处理x
    6 类可以实现接口, 但是只能实现一个接口x
    7 具体类可以实现接口, 并对其中的抽象方法不予处理x
    8 类可以实现接口, 并可以实现多个接口v
    9 具体类可以实现接口, 并要实现其接口中的方法v
    10具体类可以实现接口, 并可以实现多个接口, 并要实现所有接口中的所有方法.v
4. 抽象类和接口的比较

    比较项目 抽象类 接口
    定义 同类事物特征的抽象 不同事物相同行为的抽象
    组成 比具体类多抽象方法 全局常量和公共的抽象方法

常见设计模式	模版设计模式	代理模式和工厂模式
创建对象	通过多态来实例化对象	
局限性	没法进行多继承	无前面所述局限性
实际应用	模板	标准规范, 能力
选择	能选接口就选接口	

5. 描述什么是代理设计模式. 在什么样的场景中使用

   把代理对象当作被代理使用.

   因为它们都实现了共同的接口.

   1. 无法获取到被代理对象
   2. 需要对被代理对象的方法进行升级, 无法修改被代理类.

内部类

```
package com.atguigu.javase.inner;

/**
 * 内部类 ： 在一个类中再声明另外的类，外面的就是外部类，里面的就是内部类
 *
 * 成员内部类 ： 声明在类中方法外的内部类
 *       普通内部类 ： 没有static修饰的，这个类隶属于外部类对象.  夸张的对象关联，因为在内部类中可以随意使用外部类对象的任何成员.
 *                普通的对象关联，只能使用到关联的对象的公共成员.
 *
 *       嵌套类 ： 有static修饰的 ： 这个类隶属于外部类
 *
 * 局部内部类 ： 声明在方法中的内部类
 *       普通局部内部类
 *       匿名内部类 ： 没有类名. 是最重要的!!!!!
 *            因为它没有类名，所以必须在声明的同时就创建唯一对象，并且此唯一对象也只能被多态引用.
```

```
 *          父类类型或接口 引用 = new 父类类名|接口() {
 *              类体部分就是new后面的类名或接口的子类部分.
 *          };
 */
class Outer {

    private String name = "Outer";
    private Object obj;

    public void test1() {
        System.out.println("Outer.test1");
    }

    // 普通内部类, 隶属于外部类对象
    protected class Inner1 {
        // 除了静态final属性外, 其他static属性不允许!!
        final static int no = 1;

        private String name = "Inner1";
        private Object obj; // 关联的普通对象.

        public void test2() {
            // 此方法的调用者对象 this 就是test3方法中的i1
            // 此方法还有另外一个外部类的对象 this 就是test3方法中的o1
            System.out.println("inner name : " + Inner1.this.name); //
其实就是i1
            System.out.println("outer name : " + Outer.this.name); //
外部类的this对象, 就是o1
            System.out.println("Inner1.test2");
            test1();
        }
    }

    /**
     * 嵌套类, 隶属外部类
     */
```

```
    static class Inner2 {

        static String name = "Inner2";

        int id;

        public static void test4() {
            System.out.println("Inner2.test4");
        }

        public void test5() {
            System.out.println("Inner2.test5");
        }
    }

    public void test3() {
        // 此方法的调用者对象 this 就是main中的那个o1
        // 使用内部类, 必须要创建对象
        Inner1 i1 = this.new Inner1(); // 内部类的隶属的外部类对象就是this,
就是o1
        i1.test2(); // 此方法调用时会传入2个this, 一个是o1, 另一个是i1
        System.out.println("Outer.test3");
    }


}

interface ITest {
    void hello(String str);
}

public class InnerTest {

    public static void main(String[] args) {
        Object obj1 = new Object();
        System.out.println(obj1);
```

```
        Object obj2 = new Object() {}; // Object的匿名子类对象
        System.out.println(obj2);
        /*
        class NoName implements ITest {
            @Override
            public void hello(String str) {
                System.out.println("你好 " + str);
            }
        }
        ITest it = new NoName();
        */
        ITest it = new ITest() {
            // 匿名类的类体部分就是外面的接口的实现子类.
            @Override
            public void hello(String str) {
                System.out.println("你好 " + str);
            }
        };
        it.hello("李焕英");
    }

    public static void main3(String[] args) {
        // 普通局部内部类, 此类只能在本方法中使用!!!
        class Inner3 {
            private int age;
            private String name;

            public Inner3(int age, String name) {
                this.age = age;
                this.name = name;
            }

            public int getAge() {
                return age;
            }
```

```
            public void setAge(int age) {
                this.age = age;
            }

            public String getName() {
                return name;
            }

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

        Inner3 i3;
        i3 = new Inner3(22, "aa");
    }

    public static void test33() {
        //Inner3 i3;
    }

    public static void main2(String[] args) {
        Outer.Inner2.test4(); // 静态成员
        Outer.Inner2 oi2 = new Outer.Inner2(); // 嵌套类创建对象更简单

    }

    public static void main1(String[] args) {
        Outer o1 = new Outer();
        //1.test3();
        // 外部类对象.new 内部类();
        Outer.Inner1 oi1 = o1.new Inner1();
        oi1.test2();

        //Outer.Inner1 oi2 = new Outer().new Inner1();
    }
```

```
}
```

变量复习

内存中的一块被命名的且有数据类型约束的空间, 此空间中可以保存一个数据类型范围内的数据, 此空间中的数据可以在数据类型范围内随意变化.

变量声明

数据类型 变量名;

数据类型的作用 :

1) 决定空间的大小及数据范围

2) 决定空间中可以保存什么数据

3) 决定空间中的数据可以做什么

变量按照数据类型来分

1) 基本数据类型(primitive) : 内存空间中保存数据本身

    1) 数值型(可以运算, 比较大小)

        1) 整数 : byte, short, int, long, char

2) 浮点数 : float, double

2) 布尔型(只能逻辑运算)

2) 引用数据类型(reference) : 内存空间中保存对象(只要是对象一定是Object的)地址. 通常占用固定的8字节(64位JDK), 地址 : 内存中的每个字节都有的唯一编号. 编号0表示null

1) 数组 []

2) 类类型 class

3) 接口 interface

4) 枚举 enum

5) 注解 @interface

变量按照声明语句的位置来分

1) 局部变量(local) : 声明在方法中的变量, 范围小(最大的就是方法体), 寿命短(最长也就和方法的一次调用)

方法的参数, 方法体中声明的变量, 隶属于方法的某次调用的栈桢, 在栈中, 是临时的没有自动初始化.

2) 成员变量(member) : 声明在类中方法外的变量, 范围大(理论是全局的, 必须要使用访问控制修饰符控制), 寿命长.

类变量, 类属性 : 被static修饰的, 隶属于类模板 ,保存在永久区, 有自动初始化值0

实例变量, 对象属性 : 没有被static修饰的, 隶属于对象, 保存在GC区, 有自动初始值0.

面向对象3条主线

1) 类和类的成员研究

    1) 属性(field) : 描述事物的特征

    2) 方法(method) : 描述事物行为

    3) 构造器(constrctor) : 创建对象时为对象进行一次初始化工作.

    4) 语句块

    5) 内部类

2) 三大特征

    1) 封装 encapsurlation :

        成员私有化, 保护内部成员

        该我做事情我做, 不该我做的事情不做.

    2) 继承 inheritance :

        方法覆盖

    3) 多态 polymorphism : 2个角度, 从右向左, 从左向右

        虚拟方法调用

3) 其他关键字

    this, super, package, import, extends, implements, static, final, instanceof,

interface, abstract, native.

设计模式

单例

```
package com.atguigu.javase.statictest;

import java.io.IOException;

/**
 * 单例 ： 只允许有一个对象的类的设计模式 ， 这个对象无论有多少引用， 它们都是==的
 *
 * 饿汉式单例 ： 在类加载时创建唯一对象， 如果创建对象很快， 优选饿汉
 *      1）不能在外部随意new对象， 私有化构造器， 封死new
 *      2）在类中new唯一对象， 并把唯一对象的地址保存在私有的类属性引用变量中.
 *      3）在类中提供一个公共的静态方法， 用于获取到唯一对象的地址.
 *
 * 懒汉式单例 ： 类加载时不创建对象， 别人第一次要获取唯一对象时才真的创建对象， 有线程安全问题， 如果创建对象很耗时,优先懒汉
 **     1）不能在外部随意new对象， 私有化构造器， 封死new
 *  *  2）用私有的类属性引用变量中.
 *  *  3）在类中提供一个公共的静态方法， 用于获取到唯一对象的地址. 第1次调用静态方法时才创建唯一对象
 */
class Singleton {

    public static final Singleton only = new Singleton();
```

```
    public static Singleton getInstance() {
        return only;
    }

    private Singleton() {}

}

class Singleton2 {

    // 私有的静态的属性，引用型变量，用于保存唯一对象的地址
    private static Singleton2 only = null;

    // 公共静态方法，用于获取唯一对象的地址
    public static Singleton2 getInstance() {
        if (only == null) { // 只有第一次调用此方法时才创建唯一对象
            only = new Singleton2();
        }
        return only;
    }

    // 封装构造器
    private Singleton2() {}
}

class Test {

    @Override
    public void finalize() {
        System.out.println(this + "要死了..... 由GC调用");
    }

}
public class SingletonTest {

    public static void main(String[] args) throws IOException {
```

```
        //new Runtime();
        Object rt1 = Runtime.getRuntime();
        Object rt2 = Runtime.getRuntime();
        Runtime rt3 = Runtime.getRuntime();
        System.out.println(rt1 == rt2);
        System.out.println(rt3 == rt2);

        System.out.println("rt3.availableProcessors() = " +
rt3.availableProcessors()); // 获取CPU线程数
        System.out.println("rt3.freeMemory() = " + rt3.freeMemory());
// 空余内存
        System.out.println("rt3.totalMemory() = " +
rt3.totalMemory());

        Test t = new Test();
        t = null;

        rt3.gc(); // 建议GC清理垃圾.

        //rt3.exit(0); // JVM无条件退出. 最恐怖的方法.
        System.out.println("alkjsdfasdf");
        rt3.exec("explorer http://www.baidu.com"); // 执行OS命令
    }

    public static void main2(String[] args) {
        Singleton2 s1 = Singleton2.getInstance();
        Singleton2 s2 = Singleton2.getInstance();
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s1 == s2);
    }

    public static void main1(String[] args) {
        //Singleton s1 = new Singleton();
        //Singleton s2 = new Singleton();
        //System.out.println(s1 == s2);
```

```
        /*
        Singleton s1 = Singleton.only;
        Singleton.only = null; // 此操作很危险.
        Singleton s2 = Singleton.only;
        System.out.println(s1 == s2);
        */

        Singleton s1 = Singleton.getInstance();
        s1 = null;
        Singleton s2 = Singleton.getInstance();
        System.out.println(s1 == s2);

    }
}
```

模板

```
package com.atguigu.javase.abstracttest;

/**
 * 模板方法设计模式 ： 抽象类本身就体现模板方法
 * 模板 ： 一部分功能确定(具体方法), 一部分功能不确定(抽象方法).
 *
 */
abstract class Template {

    /**
     * 统计code方法消耗的时间
     */
    public final void getTime() {
        long time1 = System.currentTimeMillis();
        code();
        long time2 = System.currentTimeMillis();
```

```
        System.out.println("code方法消耗的时间 : " + (time2 - time1) +
"毫秒");
    }

    public abstract void code();

}

class SubTemplate extends Template {

    @Override
    public void code() {
        int n = 1;
        for (int i = 0; i < 20_0000_0000; i++) {
            n = n-- + n++;
        }
    }
}

public class TemplateTest {

    public static void main(String[] args) {
        Template tmp = new SubTemplate();
        tmp.getTime();
    }
}
```

代理

```
package com.atguigu.javase.interfacetest;

/**
 * 代理模式 : 把代理对象当作被代理对象来使用
```

```
 *
 *      场景 :
 *          1) 用户无法直接创建被代理对象
 *          2) 需要对被代理的方法进行升级，但是又不能直接修改被代理类.
 */
interface HouseRent {
    void rent();
}
class FangDong implements HouseRent {
    @Override
    public void rent() {
        System.out.println("我有房子要出租，是婚房，请爱护 ，账号 :
234234");
    }
}
class FangDong2 implements HouseRent {
    @Override
    public void rent() {
        System.out.println("我也有房子要出租，刚死了2个人，晚上要小心 账号 :
2232334234");
    }
}
class LianJia implements HouseRent {

    private HouseRent hr = new FangDong2();

    @Override
    public void rent() {
        System.out.println("请交中介费20000");
        hr.rent(); // 被代理对象方法调用!! // AOP 面向切面编程
        System.out.println("及时交房租，不然赶走.. 支持微信和支付宝...");
    }
}

public class ProxyTest {
```

```
    public static void main(String[] args) {
        HouseRent hr = new LianJia();
        hr.rent();
    }
}
```

工厂

```
package com.atguigu.javase.interfacetest;

/**
 * 工厂方法 : 通过方法调用的方式获取到对象, 适用于不知道子类类名或者创建对象太过复杂
 * 通常就是本类就是工厂, 如果本类并没有工厂方法, 再找工厂类
 *
 * 得到对象 :
     *  1) new
     *  2) 工厂方法(主流)
     *  3) 反序列化
     *  4) 反射
 *      5) Unsafe
 */
class Teacher {

    public static Teacher getInstance() {
        return new Teacher();
    }

    public void work() {
        System.out.println("老师在工作");
    }
}
```

```
class TeacherBuilder {

    public static Teacher getTeacher() {
        return new Teacher();
    }
}

public class FactoryTest {

    public static void main(String[] args) {
        //new Teacher(.........);
        Teacher t1 = Teacher.getInstance();
        t1.work();

        Teacher t2 = TeacherBuilder.getTeacher();
        t2.work();
    }
}
```

数据结构之链表

```
package com.atguigu.javase.ds;

class Node {
    // 值域
    Object value;
    // 指针域
    Node next;

    public Node(Object value, Node next) {
```

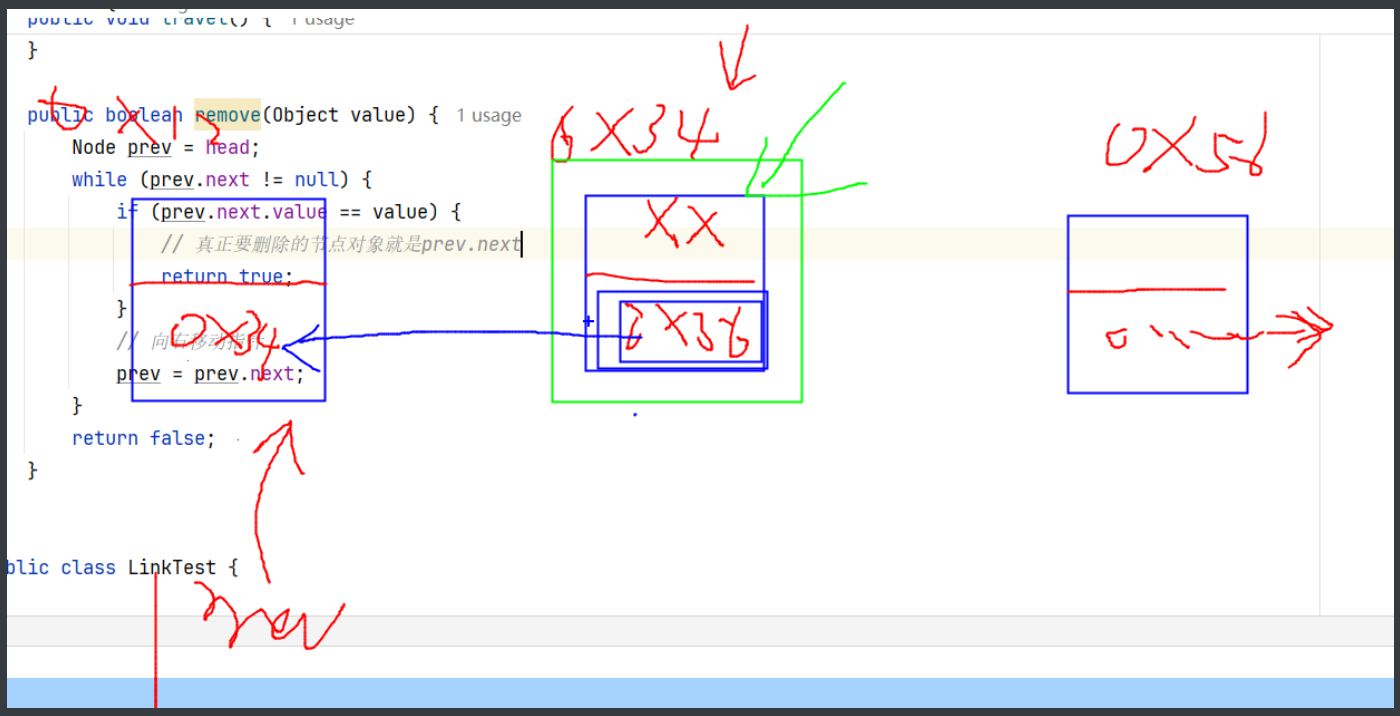
```
            this.value = value;
            this.next = next;
        }
    }

    class Link {

        // 维护2个指针，头和尾
        private Node head;
        private Node tail;
        private int size;

        /**
         * 添加新元素，可以添加任意对象
         * @param value
         */
        public void add(Object value) {
            //linkHead(value);
            linkTail(value);
            size++;
        }

        /**
         * 头部链入
         * @param value
         */
        public void linkHead(Object value) {
            head = new Node(value, head);
            if (head.next == null) {
                tail = head;
            }
        }

        /**
         * 尾部链入
         * @param value
```

```
     */
    public void linkTail(Object value) {
        Node newNode = new Node(value, null);
        if (head == null) { // 链表为空
            head = tail = newNode;
        } else { // 链表非空
            tail.next = newNode;
            tail = newNode;
        }
    }

    public void travel() {
        Node tmp = head;
        while (tmp != null) {
            System.out.print(tmp.value + "   ");
            tmp = tmp.next;
        }
        System.out.println();
    }

    /**
     * 删除链表中的第1个满足value的元素
     * @param value
     * @return 删除成功或失败
     */
    public boolean remove(Object value) {
        if (head == null) return false;
        if (head.value.equals(value)) {
            // 删除头
            head = head.next;
            --size;
            return true;
        }
        Node prev = head;
        while (prev.next != null) {
```

```
            if (prev.next.value.equals(value)) { // 对象的比较必须使用
equals方法调用, 不能使用 ==
                // 真正要删除的节点对象就是prev.next
                if (prev.next == tail) { // 如果删除尾
                    tail = prev;
                }
                // 用目标的next属性给prev.next属性赋值
                prev.next = prev.next.next;
                --size;
                return true;
            }
            // 向右移动指针
            prev = prev.next;
        }
        return false;
    }

    /**
     * 获取元素个数
     * @return
     */
    public int size() {
        /*
        int size = 0;
        Node tmp = head;
        while (tmp != null) {
            ++size;
            tmp = tmp.next;
        }*/
        return size;
    }

    /**
     * 遍历以tmpHead为头的链表
     * @param tmpHead
     */
```

```
    private void view(Node tmpHead) {
        if (tmpHead == null) {
            System.out.println();
            return;
        }
        System.out.print(tmpHead.value + " == ");
        view(tmpHead.next);
    }

    public void travel2() {
        view(head);
    }
}

public class LinkTest {

    public static void main(String[] args) {
        Link link = new Link();
        link.add("cc");
        link.add("aa");
        link.add("bb");
        link.add("qq");
        link.add("xx");
        link.add("zz");
        link.add("pp");
        link.add("tt");
        link.add("xx");
        link.add("qq");
        link.add("oo");
        link.add("xx");
        link.add("yy");

        link.travel2();
        System.out.println("link.size() = " + link.size());

        while (link.remove("xx"));
```

```
        link.add("999");

        link.travel2();
        System.out.println("link.size() = " + link.size());
    }
}
```



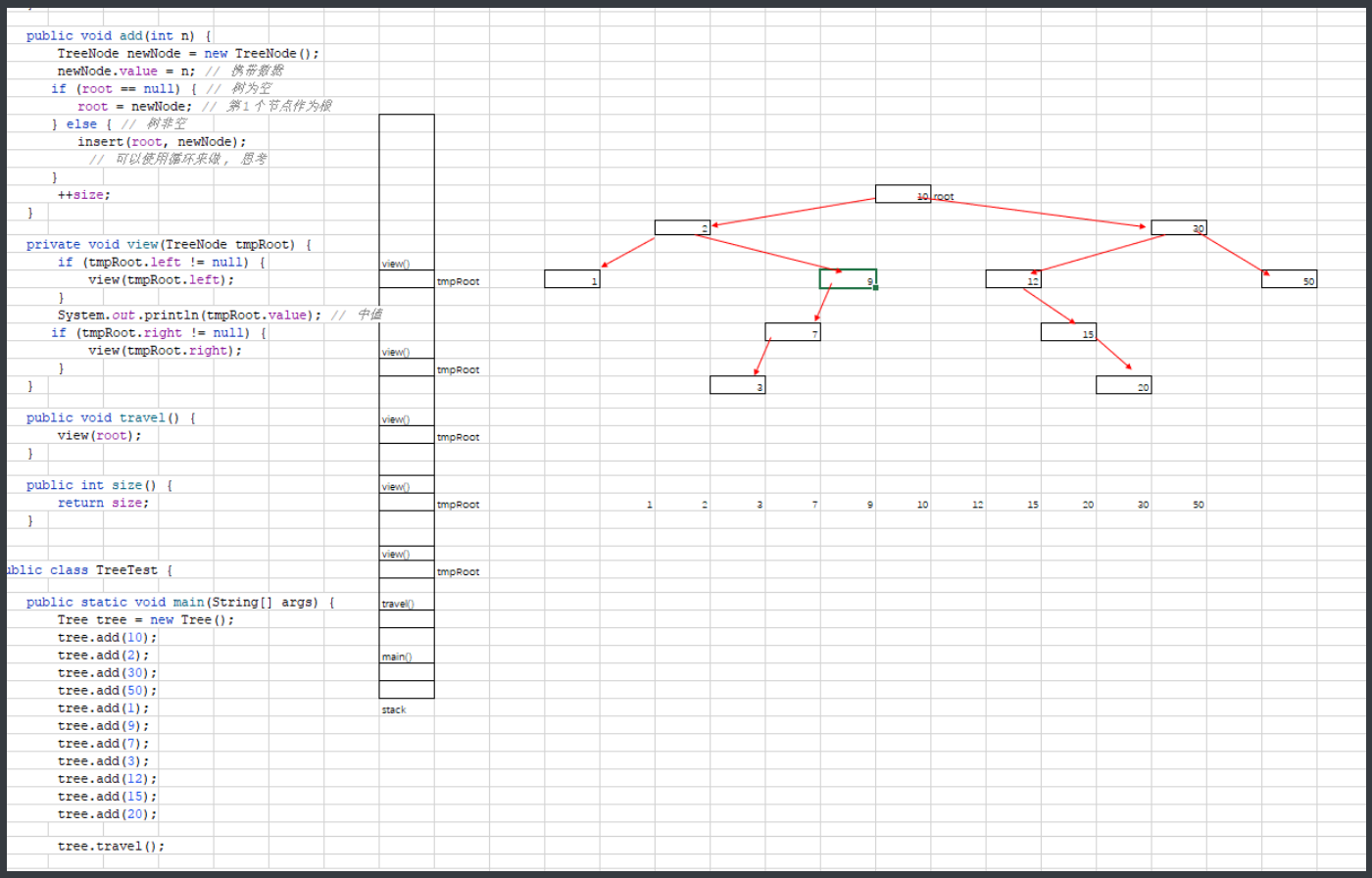
数据结构之二叉树

```
package com.atguigu.javase.ds;
```

```
class Tree {

    /**
     * 每个节点都有2个指针，指向的节点分别称为左子和右子
     * 左子的数据要小于父节点中的数据，右子的数据大于父节点中的数据
     */
    private class TreeNode {
        // 值域
        int value;
        // 指针域
        TreeNode left;
        TreeNode right;
    }


    // 维护根引用
    TreeNode root;
    int size;

    /**
     * 递归式插入元素newNode到以tmpRoot为根的树中
     * @param tmpRoot 树的根节点
     * @param newNode 新节点
     */
    private void insert(TreeNode tmpRoot, TreeNode newNode) {
        if (newNode.value < tmpRoot.value) { // 向左走
            if (tmpRoot.left == null) { // 根的左子为空，这是最好的，直接插
入
                tmpRoot.left = newNode;
            } else { // 左子非空，把左子当作新的子树的根，递归处理
                // 有条件递归，不用再写递归基
                insert(tmpRoot.left, newNode);
            }
        } else { // 向右走
            if (tmpRoot.right == null) { // 根的右子为空，这是最好的，直接
插入
```

```
                tmpRoot.right = newNode;
            } else { // 右子非空，把右子当作新的子树的根，递归处理
                insert(tmpRoot.right, newNode);
            }
        }
    }

    public void add(int n) {
        TreeNode newNode = new TreeNode();
        newNode.value = n; // 携带数据
        if (root == null) { // 树为空
            root = newNode; // 第1个节点作为根
        } else { // 树非空
            insert(root, newNode);
            // 可以使用循环来做，思考
        }
        ++size;
    }

    private void view(TreeNode tmpRoot) {
        if (tmpRoot.left != null) {
            view(tmpRoot.left);
        }
        System.out.println(tmpRoot.value); // 中值
        if (tmpRoot.right != null) {
            view(tmpRoot.right);
        }
    }

    public void travel() {
        view(root);
    }

    public int size() {
        return size;
    }
```

```
    public boolean remove(int n) {
        // 定位到目标节点
        // 把它从父节点中断开
        // 目标节点的左子树插入到当前树中
        // 目标节点的右子树插入到当前树中
        return false;
    }
}

public class TreeTest {

    public static void main(String[] args) {
        Tree tree = new Tree();
        tree.add(10);
        tree.add(2);
        tree.add(30);
        tree.add(50);
        tree.add(1);
        tree.add(9);
        tree.add(7);
        tree.add(3);
        tree.add(12);
        tree.add(15);
        tree.add(20);

        tree.travel();

        System.out.println("tree.size = " + tree.size());
    }
}
```



二叉树升级版

```
package com.atguigu.javase.homework;

class Tree {

    private class TreeNode {
        // 值域
        Comparable value;
        // 指针域
        TreeNode left;
        TreeNode right;
        public TreeNode(Comparable value, TreeNode left, TreeNode
right) {
            this.value = value;
            this.left = left;
            this.right = right;
        }
    }
```

```
    private TreeNode root;
    private int size;

    private void insert(TreeNode newNode) {
        if (newNode == null) {
            return;
        }
        TreeNode parent = root;
        while (true) {
            int n = newNode.value.compareTo(parent.value);
            if (n < 0) { // 向左走
                if (parent.left == null) {
                    parent.left = newNode;
                    break;
                }
                parent = parent.left;
            } else { // 向右走
                if (parent.right == null) {
                    parent.right = newNode;
                    break;
                }
                parent = parent.right;
            }
        }
    }

    public void add(Comparable value) {
        TreeNode newNode = new TreeNode(value, null, null);
        if (root == null) {
            root = newNode;
        } else {
            insert(newNode);
        }
        ++size;
    }
```

```
    private void view(TreeNode tmpRoot) {
        if (tmpRoot == null) {
            return;
        }
        view(tmpRoot.left);
        System.out.print(tmpRoot.value + " ");
        view(tmpRoot.right);
    }

    public void travel() {
        view(root);
        System.out.println();
    }

    public boolean remove(Comparable value) {
        if (root.value.equals(value)) {
            // 删除根
            TreeNode right = root.right;
            root = root.left;
            insert(right);
            --size;
            return true;
        }
        TreeNode parent = root;
        int n = 0;
        while (parent != null) {
            n = value.compareTo(parent.value);
            if (n < 0) {
                if (value.compareTo(parent.left.value) == 0) {
                    break;
                }
                parent = parent.left; // 移动parent指针 向左下
            } else {
                if (value.compareTo(parent.right.value) == 0) {
                    break;
```

```
                }
                parent = parent.right; // 移动parent指向 向右下
            }
        }
        if (parent != null) {
            TreeNode target = parent.left;
            if (n < 0) {
                parent.left = null; // 把父的左指针 断开
            } else {
                target = parent.right;
                parent.right = null; // 把父的右指针 断开
            }
            insert(target.left); // 目标左子插入
            insert(target.right); // 目标右子插入
            --size;
            return true;
        }
        return false;
    }

    public int size() {
        return size;
    }

}
public class TreeTest {

    public static void main(String[] args) {
        Tree tree = new Tree();
        tree.add("虽国");
        tree.add("汉字");
        tree.add("ccc");
        tree.add("sxxx");
        tree.add("234234");
        tree.add("afasdf");
        tree.add("asdf");
```

```
        tree.add("ASDFASDF");
        tree.add("SDFAS");
        tree.add("32423");
        tree.add("0000");

        tree.travel();

        tree.remove("sxxx");

        tree.travel();
    }

}
```

第10章 枚举

```
package com.atguigu.javase.enumtest;

import java.lang.annotation.Target;

/**
 * 枚举 : 对象可以列举的特殊类型.
 */
enum TrafficSignal {
    /*
    public static final TrafficSignal GO = new TrafficSignal("GO");
    public static final TrafficSignal STOP = new
TrafficSignal("STOP");
```

```
    public static final TrafficSignal CAUTION = new
TrafficSignal("CAUTION");
    */
    GO, STOP, CAUTION
}


public class EnumTest {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]); // 把命令行参数的第1个字符串转换
为int值
        // 获取枚举对象
        //new TrafficSignal();
        TrafficSignal ts = TrafficSignal.GO; // 直接获取对象引用
        //TrafficSignal.GO = null;
        System.out.println(ts);

        ts = TrafficSignal.valueOf("STOP"); // 根据常量对象名获取到常量对象
引用
        System.out.println(ts);

        System.out.println("***************");
        TrafficSignal[] arr = TrafficSignal.values(); // 获取保存了所有常
量对象地址的引用数组
        ts = arr[n];
        System.out.println(ts); // ts引用变量究竟指向了哪个对象???
        // 对一个变量中的可能的值进行穷举或列表
        /*
        switch (变量) { 变量的数据类型 : 非long整除, 字符串, 枚举
            case 常量1 :
                语句;
                break;
            case 常量2 :
                语句;
                break;
            case 常量3 :
```

```
                语句;
                break;
            default:
                语句;
                break;
        }*/
        switch (ts) {
            case GO:
                System.out.println("绿灯行");
                break;
            case STOP:
                System.out.println("红灯停");
                break;
            case CAUTION:
                System.out.println("黄灯慢");
                break;
        }
    }
}
```

第11章 异常

```
package com.atguigu.javase.exception;

import java.io.IOException;
import java.sql.SQLClientInfoException;
import java.sql.SQLException;

/**
 * 异常 : 程序在运行时出现的非正常状况, 会导致程序崩溃.
```

```
 *
 * 异常分类 :
 *     1) 按照程度
 *          1) Error
 *          2) Exception
 *     2) 按照处理方式
 *          1) 受检异常(checked) : 在程序中必须要处理的异常, 如果不处理, 编译出错, 也称为编译时异常
 *              Exception及其子类(RuntimeException除外)
 *          2) 非受检异常(unchecked) : 在程序中不是必须要处理的异常, 如果不处理, 编译不出错, 也称为运行时异常
 *              Error及其子类 : 极其严重
 *              RuntimeException及其子类 : 极其常见.
 *
 * 所有异常都会导致程序崩溃
 *
 * 异常处理的方式
 *      1) 捕获
 *          try {
 *              可能出现异常的语句
 *          } catch (可能的异常类型 引用) {
 *              // 提醒或补救
 *          } catch (可能异常类型2 引用) {
 *
 *          } .... {
 *
 *          } finally {
 *              释放资源, 保证资源一定释放!
 *          }
 *
 *          组合 :
 *              try catch
 *              try catch finally
 *              try finally
 *
```

```
 *       2）抛出 ： 在方法声明中添加throws 异常类型列表，作用是警告调用者，此方法有的风险
 *            在方法中使用throw 异常对象，就可以抛出异常给此方法调用者，throw一旦执行和return一样，会导致方法结束
 *            return是和平结束 ，throw是让调用很受伤的结束
 *
 *            如果方法中有throw 并且是受检异常，必须要有throws和它配合.
 *            如果方法声明中有throws ，不是必须要在方法中有throw配合.
 *
 *            通常的做法就是throw和throws配合.
 *
 *       3）先捕获再抛出 ： 可以把所有不相干的异常全部统一为一个特定的异常，便于维护. 把异常的类型进行转换.
 *            选择受检还是非受检异常的标准就是此方法是否要引起编译器重视.
 *            先捕获异常a，再把它包装到异常b中，实际再抛出的是b.
 *
 * 该如何选择??
 *       功能方法通常要抛出，提醒调用者此方法出了问题，请采集必须措施
 *       入口方法通常要捕获，因为它如果抛出异常，程序崩溃.
 *
 *       如果方法抛出异常会直接或间接导致栈的崩溃，必须捕获.
 *       如果方法抛出异常不会直接或间接导致栈的崩溃，必须抛出.
 *
 * 方法覆盖 ： 子类根据需要重写从父类继承的方法.
 * 覆盖条件 ：
 *       1）重写方法和被重写方法的签名完全一致，返回值类型，方法名，参数列表(类型，个数，顺序).
 *            返回值类型如果是引用类型时，子类重写时的方法的返回值类型可以是父类方法的返回值类型的子类类型
 *       2）访问控制修饰符要求子类重写方法要大于等于父类被重写方法
 *       3）父类方法不可以被一些关键字修饰 ： static，final，private
 *       4）子类重写方法抛出的受检异常要小于等于父类方法抛出的受检异常
 *
 */
// 自定义异常 1）继承Exception，2）提供若干构造器
class DividedByZeroException extends Exception {
```

```
    public DividedByZeroException(String message) {
        super(message);
    }
    // 此构造器用于对象关联，自定义异常对象关联参数中的异常对象
    // 参数的作用就是此异常的原因异常.
    public DividedByZeroException(Throwable cause) {
        super(cause);
    }
}

public class ExceptionTest {

    public static int divide(int x, int y) throws
DividedByZeroException {
        try {
            int z = x / y;
            return z;
        } catch (ArithmeticException e) {
            //throw e;
            DividedByZeroException e2 = new DividedByZeroException(e);
            throw e2;
        }
    }

    public static void main(String[] args) {
        System.out.println("main begin");

        try {
            System.out.println("divide(10, 2) = " + divide(10, 2));
            System.out.println("divide(10, 0) = " + divide(10, 0));

        } catch (DividedByZeroException e) {
            e.printStackTrace();
        }

        System.out.println("main end");
```

```
    }

    public static int divide3(int x,int y) throws
DividedByZeroException { // throws的作用是警告
        if (y == 0) {
            throw new DividedByZeroException("除数为0错误!!!!!!");
        }
        int z = x / y;
        return z;
    }

    //public static void main3(String[] args) throws
DividedByZeroException { // main方法中抛出异常, 非常不好
    public static void main3(String[] args) {
        System.out.println("main begin");
        try {
            System.out.println(divide3(10, 2));
            System.out.println(divide3(10, 0));
        } catch (DividedByZeroException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("main end");
    }

    public static int divide2(int x, int y) {
        if (y == 0) {
            throw new RuntimeException("除数不可以为0!!!!!");
        }
        int z = x / y;
        return z;
    }

    public static void main2(String[] args) {
        System.out.println("main begin");

            System.out.println("divide2(20, 2) = " + divide2(20, 2));
```

```
            System.out.println("divide2(20, 0) = " + divide2(20, 0));

        System.out.println("main end"); // 核心
    }

    public static void main1(String[] args) {
        System.out.println("main begin");


        try {
            if (true) {
                //return;
                //Runtime.getRuntime().exit(0);
            }
            int n = Integer.parseInt(args[0]);
//ArrayIndexOutOfBoundsException
            System.out.println("n = " + n);
            int[] arr = null;

            try {
                //System.out.println(arr.length);
            } catch (Exception e) {
                e.printStackTrace();
            }

            System.out.println("我也想被保护");
        } catch (ArrayIndexOutOfBoundsException e) {
            e.printStackTrace(); // 打印栈踪迹
        } catch (NumberFormatException e) {
            System.out.println(e.getMessage()); // 打印异常对象消息
        } catch (Exception e) { // 超级无敌万能夹
           System.out.println(e);
        } finally { // 最终地，末了地，无论前面try catch中 发生什么，我都要执行. return也拦不住我
            // 在这里要释放不在GC区中的资源，通常是向OS申请的资源
            System.out.println("finally");
```

```
        }

        System.out.println("main end"); // 核心代码
    }
}
```

方法覆盖

```
package com.atguigu.javase.exception;

import java.io.EOFException;
import java.io.IOException;

class Base {
    int test() {return 0;}
    Object test1() {return null; }
    void test2() {}
    private void test3() {}
    public void test4() throws IOException {}
    public void test5() {}
}

class Sub extends Base {
    //@Override short test() {return 1;}
    @Override String test1() {return  null;}
    //@Override private void test2() {} 访问控制修饰符要大}
    @Override public void test2() {}
    //@Override public void test3() {}
    @Override public void test4() throws EOFException {}
    @Override public void test5() throws Error {}
}

public class OverrideTest {
```

```
    public static void main(String[] args) {
        Base base = new Sub();
        base.test();

        try {
            base.test4();
        } catch (IOException e) {
            e.printStackTrace();
        }

        base.test5();
    }
}
```

第12章 常用类

包装类

```
package com.atguigu.javase.common;

import org.junit.Test;

/**
 * 包装类 : 把基本数据变成对象
 *
 * byte         Byte
 * short        Short
 * int          Integer
 * char         Character
```

```
 * long         Long
 * float        Float
 * double       Double
 * boolean      Boolean
 *
 * 包装类之间不再兼容
 *
 * 装箱 (boxing) : 基本数据 => 对象
 *      Integer obj1 = new Integer(300);
 *      Xxx obj = new Xxx(xxx);
 *      Xxx obj = xxx;
 *      Xxx obj = Xxx.valueOf(xxx);
 *
 *      Xxx obj = new Xxx("xxx");
 *      Xxx obj = Xxx.valueOf("xxx");
 *
 * 拆箱(unboxing) : 对象 => 基本数据
 *      xxx = obj.xxxValue();
 *      xxx = obj;
 *
 * String => 基本数据
 *      String str = "xxx";
 *      xxx = Xxx.parseXxx(str);
 *      xxx = new Xxx(str);
 *      xxx = Xxx.valueOf(str);
 *
 * 基本数据
 *      String s = "" + xxx;
 */
public class WrapperTest {

    @Test
    public void test3() {
        Double obj1 = new Double(3.22);
        Double obj2 = 2349.234;
        //Double obj3 = 34; // 右面是Integer, 左面是Double
```

```
        //Double obj3 = 324.234f;
        Float obj3 = 324.234f;
        Long obj4 = 324L;
        Boolean obj5 = new Boolean(true);
        Boolean obj6 = false;

        Double obj7 = new Double("324.234");
        Boolean obj8 = new Boolean("false");

        double v1 = obj1.doubleValue();
        double v2 = obj2;
        float v3 = obj3;
        boolean b1 = obj5.booleanValue();

        // String => double
        String str = "324.234";
        double v4 = Double.parseDouble(str);

        System.out.println("v4 = " + v4);

        String str2 = "TRUE";
        boolean b2 = Boolean.parseBoolean(str2);
        System.out.println("b2 = " + b2);



    }

    // 练习 : 声明2个字符串, 内容是2个整数, 把第1个串转换为int后再装箱, 把第2个串直接变成Integer对象
    // 手工拆箱求和, 自动拆箱求积, 并打印输出.
    @Test
    public void exer() {
        String s1 = "324";
        String s2 = "82";
        int i1 = Integer.parseInt(s1);
```

```
        Integer obj1 = new Integer(i1);
        //Integer obj2 = new Integer(s2);
        //Integer obj2 = Integer.decode(s2); // 把字符串解码为Integer对象
        //Integer obj2 = Integer.getInteger(s2); // 不靠谱，它只是获取系
统属性的
        Integer obj2 = Integer.valueOf(s2); // 好用!!!


        int sum = obj1.intValue() + obj2.intValue();
        System.out.println("sum = " + sum);
        int multi = obj1 * obj2;
        System.out.println("multi = " + multi);
    }

    @Test
    public void test1() {
        int n1 = 203;
        Integer obj1 = new Integer(n1); // 手工装箱
        Integer obj2 = 3423; // 自动装箱. Integer.valueOf(3423)

        int i1 = obj1.intValue(); // 手工拆箱
        int i2 = obj2; // 自动拆箱，obj2.intValue()

        Object obj3 = 234;

        boolean b1 = obj1 == obj2; // 不会拆箱，比较2个对象的引用地址值
        System.out.println(b1);

        boolean b2 = obj1 > obj2; // 自动拆箱.
        System.out.println(b2);

        // String => int
        String s1 = "234288";
        //int i3 = s1;
        int i3 = Integer.parseInt(s1);
        // int => String
```

```
        int i4 = 89234;
        //String s2 = i4;
        String s2 = "" + i4;



    }
}
```

字符串 String对象存储

```
package com.atguigu.javase.common;

import org.junit.Test;

/**
 * 字符串 :
 *      String : 内容不可改变的Unicode字符序列(下标), 它实现的CharSequence接口, 内部就是使用了一个char[]来保存所有字符.
 *
 * 字符串的内容的任何改变, 都将导致 产生新的对象
 *
 * 所有的字符串字面量, 也称为字符串常量对象, 它们都保存在内存的常量区中.
 */
public class StringTest {

    @Test
    public void test2() {
        final String s1 = "atguigu";
        String s2 = "java";
        String s4 = "java";
        String s3 = new String("java");
        System.out.println(s2 == s3); // false, s3指向的对象在gc区
```

```
        System.out.println(s2 == s4); // true, 这是特例, 不能推广
        System.out.println(s2.equals(s3)); // true, 字符串对象的比较最靠谱的做法.

        System.out.println("**************************");
        String s5 = "atguigujava";
        //String s6 = s1 + s2; // 字符串拼接, 如果有变量参与, 就会拼接的结果在gc区
        //String s6 = "atguigu" + "java"; // 字符串拼接, 全是常量拼接, 拼接的结果在常量区
        //String s6 = s1 + "java"; // 字符串拼接, 全是常量拼接, 拼接的结果在常量区
        //String s6 = s1 + s2; // gc
        String s6 = (s1 + s2).intern(); // 强行把字符中对象拘留到常量区, 如果常量区已经有了此对象, 直接返回对象地址, 没有再塞入.

        System.out.println(s5 == s6);
        System.out.println(s5.equals(s6));


    }

    @Test
    public void test1() {
        String s1 = "";
        String s2 = null;

        System.out.println(s1);
        System.out.println(s2);

        String s3 = "alsdkjflkajsdf"; // 右面的称为字符串常量对象
        String s4 = new String();

        System.out.println("s1.equals(s4) = " + s1.equals(s4));
        System.out.println("s1 == s4 : " + s1 == s4);
```

```
        String s5 = new String(s3);
        System.out.println("s3.equals(s5) = " + s3.equals(s5));

        char[] arr = {'1', '2', 'a', 'x', '在', '小', '你'};
        String s6 = new String(arr); // 把char[]中的内容全部取到构建一个
String对象
        System.out.println("s6 = " + s6);

        // 最实用，它把char[]的一部分
        String s7 = new String(arr, 3, 4);// 第2个参数是开始索引，第3个参
数是取的字符数
        System.out.println("s7 = " + s7);


    }
}
```

String中的方法

```
package com.atguigu.javase.common;

import org.junit.Test;

/**
 * String : 内容不能改变的Unicode字符序列(实现了CharSequence).
 * 内部使用了一个char[] 来保存所有字符，所以它也是有下标索引的概念.
 *
 * 如果改变了字符串的内容，它将会产生一个新的字符串对象.
 *
 *                 2       10       16        23        30    36 38
 * String str = "  abc zzQ 我喜欢你，你喜欢我吗？我不喜欢你！ zz99 XX  ";
 * ### public int length() : 获取字符串的长度(字符个数)
 *      str.length() => 39
```

```
 *
 * ### public char charAt(int index) : 获取指定下标index位置处的字符
 *      str.charAt(18) => '欢',  str.charAt(27) => '你'
 *
 * # public char[] toCharArray() : 获取字符串对应的char[]对象, 是内部数组的一个副本
 * public char[] toCharArray() {
 *         char result[] = new char[value.length];
 *         // 第1个参数是源数组对象, 第2个参数是源数组的开始下标, 第3个参数是目标数组对象, 第4个参数是目标数组的开始下标,
 *         // 第5个参数是要复制的元素个数
 *         System.arraycopy(value, 0, result, 0, value.length);
 *         return result;
 *  }
 *
 * ##### public boolean equals(Object anObject) : 比较字符串内容
 * public boolean equalsIgnoreCase(String s2) : 比较字符串内容, 忽略大小写
 * public int compareTo(String anotherString) : 实现了Comparable, 完成对象的比较大小
 * public int compareToIgnoreCase(String s2) : 忽略大小写比较大小
 *
 * public boolean contains(CharSequence s2) : 判断当前串中是否包含子串.
 *
 *                2       10       16        23        30    36 38
 * String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99 XX  ";
 * ##### public int indexOf(String s) : 获取参数中的子串s在当前字符串中的首次出现下标索引
 *          str.indexOf("喜欢") => 11
 *
 * public int indexOf(String s ,int startpoint) : 获取参数中的子串s在当前字符串中的首次出现下标索引, 从startPoint下标开始检索
 *          str.indexOf("喜欢", 0) => 11
 *          str.indexOf("喜欢", 12) => 17
 *          str.indexOf("喜欢", 18) => 25
 *          str.indexOf("喜欢", 26) => -1
 *
```

```
 * public int lastIndexOf(String s) : 从右向左检索参数中的子串在当前串中下标
 *          str.lastIndexOf("喜欢") => 25
 *
 * public int lastIndexOf(String s ,int startpoint) : 从右向左检索参数中的子串在当前串中下标, 从startPoint下标开始
 *          str.lastIndexOf("喜欢", str.length() - 1) => 25
 *          str.lastIndexOf("喜欢", 24) => 17
 *          str.lastIndexOf("喜欢", 16) => 11
 *          str.lastIndexOf("喜欢", 10) => -1
 *
 * #public boolean startsWith(String prefix) : 判断当前串是否以参数中的子串prefix为开始
 *          str.startsWith("  abc") => true
 * #public boolean endsWith(String suffix) : 判断当前串是否以参数中的子串suffix为结束
 *          str.endsWith("XX") => false

 *                2       10       16        23        30    36 38
 * String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99 XX  ";
 *
 * ##### public String substring(int start,int end) : 获取当前串中的子串, 从start下标开始(包含), 到end下标结束(不包含)
 *          str.substring(16, 21) => "你喜欢我吗" // 结束索引-开始索引 = 子串长度
 *          str.substring(16, str.length()) => "你喜欢我吗? 我不喜欢你! zz99 XX  "; // 取到最后了
 *
 * ##### public String substring(int startpoint) : 获取当前串中的子串, 从start下标开始(包含), 取到末尾
 *          str.substring(16) => "你喜欢我吗? 我不喜欢你! zz99 XX  "; // 取到最后了
 *
 * ##public String replace(char oldChar,char newChar) : 替换字符串中的所有的oldChar为newChar
 * ##public String replace(CharSequence target, CharSequence replacement) : 替换字符串中的所有target为replacement
```

```
 *
 * public String replaceAll(String old,String new) : 支持正则表达式的替换
 *
 * ## public String trim() : 修剪字符串首尾的空白字符，码值小于等于32的所有字符
 *
 * public String concat(String str)
 * #public String toUpperCase() : 变大写
 * #public String toLowerCase() : 变小写
 *
 * public String[] split(String regex) : 把字符串以参数中的子串为切割器，切割成多个子串，切割器被丢弃
 * 根据给定正则表达式的匹配拆分此字符
 *
 * public static String valueOf(...) : 把任意数据转换为相应的字符串
 *
 */
public class StringTest {

    @Test
    public void test7() {
        double d = 234.234234;
        String s1 = "" + d;
        String s2 = String.valueOf(d);

        System.out.println(s1.equals(s2));
    }

    @Test
    public void test6() {
        String s1 = "abcYYY";
        String s2 = "abcqqq";

        int i = s1.compareTo(s2);
        System.out.println("i = " + i);
        int i2 = s1.compareToIgnoreCase(s2);
```

```
        System.out.println("i2 = " + i2);

        String s3 = "abCDEfg";
        String s4 = "ABcdefG";
        System.out.println(s3.equals(s4));
        System.out.println(s3.equalsIgnoreCase(s4));


 System.out.println(s3.toUpperCase().equalsIgnoreCase(s4.toUpperCase()
));
    }

    @Test
    public void test5() {
        String s = "abc,外侧膝状体,234234,234驾照,423423花飘万家雪,lkj lkj
l, lkj LK J , LK LKJ";
        String[] arr = s.split(",");
        for (String string : arr) {
            System.out.println(string);
        }
        System.out.println("**********************");
        String path = "D:\\Atguigu\\02_Program\\jdk-
17.0.10\\bin;D:\\Atguigu\\02_Program\\02_JDK8_64\\bin;C:\\Windows\\sys
tem32;C:\\Windows;C:\\Windows\\System32\\Wbem;C:\\Windows\\System32\\W
indowsPowerShell\\v1.0\\;C:\\Windows\\System32\\OpenSSH\\;C:\\MyProgra
m\\_MyBin;C:\\Program Files\\Git\\cmd;";
        String[] split = path.split(";");
        for (String string : split) {
            System.out.println(string);
        }

    }

    @Test
    public void test4() {
```

```
        //*                  2       10       16        23        30
36 38
        // * String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99
XX  ";
        String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99 XX
";
        String str2 = str.replace(' ', '#');
        System.out.println(str2);
        String str3 = str.replace("喜欢", "讨厌");
        System.out.println(str3);
        // 消除字符串中的所有空格
        String str4 = str.replace(" ", "");
        System.out.println(str4);

        String str5 = str.replace("", "$");
        System.out.println(str5);

        String str6 = str.replaceAll("喜欢", "爱");
        System.out.println(str6);


        String trim = str.trim();
        System.out.println(trim);

        String s2 = "       \r\n   \b a aasdflj lkajsd flkja sdf \r\n
\b \r\n     \t";
        System.out.println(s2);
        String s3 = s2.trim();
        System.out.println(s3);


        String s4 = s2.replaceAll("\\s", "");
        System.out.println(s4);

        String upperCase = str.toUpperCase();
        String lowerCase = str.toLowerCase();
```

```
        System.out.println(upperCase);
        System.out.println(lowerCase);
    }

    // 将一个字符串进行反转。将字符串中指定部分进行反转。比如将“abcdefg”反转为”abfedcg”
    @Test
    public void exer32() {
        String str = "abcdefghijklmn";
        int begin = 2;
        int end = 9;
        char[] arr = str.toCharArray();
        for (int i = 0; i < (end - begin) / 2; i++) {
            // 左侧下标 begin + i, 右侧是 end - 1 - i
            char tmp = arr[begin + i];
            arr[begin + i] = arr[end - 1 - i];
            arr[end - 1 - i] = tmp;
        }
        String result = new String(arr);
        System.out.println("result = " + result);
    }

    @Test
    public void exer3() {
        String str = "abcdefghijklmn";
        int begin = 2;
        int end = 9;
        // 切成3段, 只反转中间部分
        String s1 = str.substring(0, begin);
        String s2 = str.substring(begin, end);
        String s3 = str.substring(end);

        String s4 = "";
        for (int i = 0; i < s2.length(); i++) {
            char ch = s2.charAt(i);
            s4 = ch + s4;
```

```
        }
        // 再拼接回去
        String result = s1 + s4 + s3;
        System.out.println("result = " + result);
    }

    @Test
    public void test3() {
        //*                 2       10       16        23        30
36 38
        // * String str = "  abc zzQ 我喜欢你，你喜欢我吗？我不喜欢你！zz99
XX  ";
        String str = "  abc zzQ 我喜欢你，你喜欢我吗？我不喜欢你！zz99 XX
";
        String substring = str.substring(16, 21);
        System.out.println("substring = " + substring);
        String substring2 = str.substring(16, str.length()); // 取到末
尾
        System.out.println("substring2 = " + substring2);
        String substring3 = str.substring(16);
        System.out.println("substring3 = " + substring3);
        System.out.println("substring2.equals(substring3) = " +
substring2.equals(substring3));
    }

    //获取一个字符串在另一个字符串中出现的次数。比如：获取"ab"在
"abkkcadkabkebfkabkskab" 中出现的次数
    @Test
    public void exer2() {
        String s1 = "*****"; // 4
        String s2 = ""; // 2
        int count = 0;
        int begin = 0;
        while (true) {
            // 返回值index一定在begin位置或它的右侧
            int index = s1.indexOf(s2, begin);
```

```
            if (index == -1 || index < begin) {
                // 完事
                break;
            }
            // 计数
            count++;
            // 继续向右检索
            begin = index + 1;
        }
        System.out.println("count = " + count);
    }

    @Test
    public void test2() {
        //*                   2       10       16        23        30
36 38
        // * String str = "  abc zzQ 我喜欢你，你喜欢我吗？我不喜欢你！zz99
XX  ";
        String str = "  abc zzQ 我喜欢你，你喜欢我吗？我不喜欢你！zz99 XX
";
        String ss = "喜欢";
        int index1 = str.indexOf(ss);
        System.out.println("index1 = " + index1);
        int index2 = str.indexOf(ss, index1 + 1);
        System.out.println("index2 = " + index2);
        int index3 = str.indexOf(ss, index2 + 1);
        System.out.println("index3 = " + index3);
        int index4 = str.indexOf(ss, index3 + 1);
        System.out.println("index4 = " + index4);
        int index5 = str.indexOf(ss, index4 + 1);
        System.out.println("index5 = " + index5);

        int index6 = str.lastIndexOf(ss);
        System.out.println("index6 = " + index6);
        int index7 = str.lastIndexOf(ss, index6 - 1);
        System.out.println("index7 = " + index7);
```

```
        int index8 = str.lastIndexOf(ss, index7 - 1);
        System.out.println("index8 = " + index8);
        int index9 = str.lastIndexOf(ss, index8 - 1);
        System.out.println("index9 = " + index9);

        System.out.println("str.startsWith(\"  abc\") = " +
str.startsWith("  abc"));
        System.out.println("str.endsWith(\"XX\") = " +
str.endsWith("XX"));
    }

    // 练习 : 反转一个字符串
    @Test
    public void exer14() {
        //*                  2       10       16       23       30
36 38
        // * String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99
XX  ";
        String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99 XX
";
        char[] arr = str.toCharArray();
        for (int i = 0; i < arr.length / 2; i++) {
            // i和length-1-i位置
            char tmp = arr[i];
            arr[i] = arr[arr.length - 1 - i];
            arr[arr.length - 1 - i] = tmp;
        }
        String s2 = new String(arr);
        System.out.println(str);
        System.out.println(s2);
    }

    @Test
    public void exer13() {
        //*                  2       10       16       23       30
36 38
```

```
        // * String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99
XX  ";
        String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99 XX
";
        String s2 = "";
        for (int i = 0; i < str.length(); i++) {
            char ch = str.charAt(i);
            s2 = ch + s2;
        }
        System.out.println(s2);
    }
    @Test
    public void exer12() {
        //*                 2       10       16       23       30
36 38
        // * String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99
XX  ";
        String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99 XX
";
        String s2 = "";
        for (int i = 0; i < str.length(); i++) {
            char ch = str.charAt(str.length() - 1 - i);
            s2 += ch;
        }
        System.out.println(s2);
    }

    @Test
    public void exer1() {
        //*                 2       10       16       23       30
36 38
        // * String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99
XX  ";
        String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99 XX
";
        String s2 = "";
```

```
        // 反向遍历字符串
        for (int i = str.length() - 1; i >= 0; i--) {
            char ch = str.charAt(i);
            s2 += ch;
        }
        System.out.println(str);
        // 把字符串到s2中
        System.out.println(s2);
    }

    @Test
    public void test1() {
        //*                 2       10       16        23        30
36 38
        // * String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99
XX  ";
        String str = "  abc zzQ 我喜欢你, 你喜欢我吗? 我不喜欢你! zz99 XX
";
        System.out.println("str.length() = " + str.length());
        System.out.println("str.charAt(12) = " + str.charAt(12));
        System.out.println("str.charAt(20) = " + str.charAt(20));

        System.out.println("*****************************");
        // 字符串遍历
        for (int i = 0; i < str.length(); i++) {
            char ch = str.charAt(i);
            System.out.println(ch);
        }
        System.out.println("**********************");
        // 转换成char[]
        char[] arr = str.toCharArray();
        for (char c : arr) {
            System.out.println(c);
        }
    }
}
```

StringBuffer

```
package com.atguigu.javase.common;

import org.junit.Test;

/**
 * StringBuffer : 内容可以改变的Unicode字符序列.
 * 内部也是使用一个char[]来保存所有字符，但是它是伸缩性，可以内部自动的扩容或缩减!!
 *
 * 任何的修改都不会产生新对象
 * 内部的扩容机制 : 新容量 = 老容量 * 2 + 2; // +2的原因就是防止数组在初始的时候长度为0.
 *
 * public StringBuffer append(...) 在当前串中末尾追加任意数据新内容
 * public StringBuffer insert(int index, ...) : 向目标下标位置处插入新内容
 * public StringBuffer delete(int beginIndex, int endIndex) : 删除一个区间
 *
 * public void setCharAt(int index, char newChar) : 替换字符
 *
 * StringBuffer 是古老的StringBuilder
 * 老的速度慢，线程安全.
 * 新的速度快，线程不安全.
 */
public class StringBufferTest {

    @Test
    public void test4() {
        String text = "";
        long startTime = 0L;
        long endTime = 0L;
```

```
        StringBuffer buffer = new StringBuffer("");
        StringBuilder builder = new StringBuilder("");
        startTime = System.currentTimeMillis();
        for (int i = 0; i < 200000; i++) {
            buffer.append(String.valueOf(i));
        }
        endTime = System.currentTimeMillis();
        System.out.println("StringBuffer的执行时间: " + (endTime -
startTime));
        startTime = System.currentTimeMillis();
        for (int i = 0; i < 200000; i++) {
            builder.append(String.valueOf(i));
        }
        endTime = System.currentTimeMillis();
        System.out.println("StringBuilder的执行时间: " + (endTime -
startTime));
        startTime = System.currentTimeMillis();
        for (int i = 0; i < 200000; i++) {
            text = text + i;
        }
        endTime = System.currentTimeMillis();
        System.out.println("String的执行时间: " + (endTime -
startTime));

    }

    // 练习 : 声明4个字符串, 以第1个串为内容创建StringBuilder对象, 把第2个串串
到末尾, 把第3个串插入到开头, 再把第4个串插入到中间某个位置, 再删除一个长度为5的区
间.
    // 最后打印. 使用链式调用
    @Test
    public void exer() {
        String s1 = "lakjsdfasdf";
        String s2 = "234234234234";
        String s3 = "汉籽去哪里虽加在";
        String s4 = "ADSFASDFDFASDF";
```

```
        StringBuilder sb = new StringBuilder(s1); // 汉籽去哪里ADSDFASDF
虽加在lakjsdfasdf234234234234
        sb.append(s2).insert(0, s3).insert(5, s4).delete(8, 13);
        System.out.println(sb);
    }


    @Test
    public void test3() {
        StringBuilder sb = new StringBuilder();
        // 链式调用

 sb.append('大').append('a').append(234).append("hello").append(false)
.append('你').append('我').insert(5, "我是汉字").delete(8, 15);

        System.out.println(sb);// 大a234我是汉alse你我
    }

    @Test
    public void test2() {
        StringBuilder sb = new StringBuilder(0);
        sb.append('大');
        sb.append('a');
        sb.append(234);
        sb.append("hello");
        sb.append(false);
        sb.append('你');
        sb.append('我');
        System.out.println(sb);
        // 大a234hellofalse你我
        sb.insert(5, "我是汉字");
        // 大a234我是汉字hellofalse你我
        System.out.println(sb);
        // 大a234我是汉字hellofalse你我
        sb.delete(8, 15);
```

```
        System.out.println(sb);
        //大a234我是汉alse你我
        sb.setCharAt(5, '他');
        System.out.println(sb);
        System.out.println(sb.length());
    }

    @Test
    public void test1() {
        StringBuilder sb1 = new StringBuilder(); // 内部char[]的容量为16
        StringBuilder sb2 = new StringBuilder(1024); // 直接以指定容量为长度创建内部数组
        StringBuilder sb3 = new StringBuilder("abcd"); // 以参数中字符串为初始内容 创建缓冲区


    }
}
```

日期相关

```
package com.atguigu.javase.common;

import org.junit.Test;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.time.LocalDate;
import java.util.Calendar;
import java.util.Date;
```

```
/**
 * Letter  Date or Time Component  Presentation  Examples
 * G  Era designator  Text  AD
 * y  Year  Year  1996; 96
 * Y  Week year  Year  2009; 09
 * M  Month in year (context sensitive)  Month  July; Jul; 07
 * L  Month in year (standalone form)  Month  July; Jul; 07
 * w  Week in year  Number  27
 * W  Week in month  Number  2
 * D  Day in year  Number  189
 * d  Day in month  Number  10
 * F  Day of week in month  Number  2
 * E  Day name in week  Text  Tuesday; Tue
 * u  Day number of week (1 = Monday, ..., 7 = Sunday)  Number  1
 * a  Am/pm marker  Text  PM
 * H  Hour in day (0-23)  Number  0
 * k  Hour in day (1-24)  Number  24
 * K  Hour in am/pm (0-11)  Number  0
 * h  Hour in am/pm (1-12)  Number  12
 * m  Minute in hour  Number  30
 * s  Second in minute  Number  55
 * S  Millisecond  Number  978
 * z  Time zone  General time zone  Pacific Standard Time; PST; GMT-
08:00
 * Z  Time zone  RFC 822 time zone  -0800
 * X  Time zone  ISO 8601 time zone  -08; -0800; -08:00
 *
 * Date :
 *      toString不友好
 *      year有1900问题
 *      month是从0开始
 *
 * Calendar
 *      toString不友好
 *      month是从0开始
 *      内部使用了一个int[]来保存所有属性值, 操作极其不便.
```

```
 *      内容可以改变，所有历史 数据都找不回来.
 *
 * LocalDate : 本地日期
 *      private final int year; // 属性不可以改变
 *      private final short month;
 *      private final short day;
 *
 * LocalTime : 本地时间
 * LocalDateTime : 本地日期时间
 */
public class DateTest {

    @Test
    public void test6() {
        //LocalDate date = new LocalDate();
        LocalDate date = LocalDate.now();
        System.out.println(date);

        LocalDate date2 =
date.withYear(2008).withMonth(8).withDayOfMonth(8).plusYears(20);
        System.out.println(date2);
    }

    @Test
    public void test5() {
        //LocalDate date = new LocalDate();
        LocalDate date = LocalDate.now();
        System.out.println(date);

        int year = date.getYear();
        int month = date.getMonthValue();
        int day = date.getDayOfMonth();

        System.out.println("year = " + year);;
        System.out.println("month = " + month);
        System.out.println("day = " + day);
```

```
        //date.setYear(2008); 不能修改对象的属性数据
        LocalDate date2 = date.withYear(2008); // 在当前对象基础上伴随新数
据产生新对象
        LocalDate date3 = date2.withMonth(8);
        LocalDate date4 = date3.withDayOfMonth(8);
        System.out.println(date4);

        // 日期累积
        //date4.add
        LocalDate date5 = date4.plusYears(20);
        System.out.println(date5);
        LocalDate date6 = date5.minusMonths(-300);
        System.out.println(date6);
        LocalDate date7 = date6.plusDays(-100000);
        System.out.println(date7);
    }

    // 练习 : 获取一个Calendar对象, 把它修改为你的生日, 获取你的百岁.
    @Test
    public void exer2() {
        Calendar cal = Calendar.getInstance();
        // 1978, 6, 9
        cal.set(Calendar.YEAR, 1978);
        cal.set(Calendar.MONTH, 5);
        cal.set(Calendar.DAY_OF_MONTH, 9);

        cal.add(Calendar.DAY_OF_MONTH, 100);

        System.out.println(cal.getTime());
    }

    @Test
    public void test4() {
        // Calendar cal = new Calendar();
        Calendar cal = Calendar.getInstance();
```

```
        System.out.println(cal);

        // 获取属性
        // cal.getYear();
        int year = cal.get(Calendar.YEAR);// 通用的获取某个属性的方法，必须提供属性对应的整数，记不住需要常量辅助
        System.out.println("year = " + year);
        int month = cal.get(Calendar.MONTH); // 内部月从0开始的.
        System.out.println("month = " + month);
        int day = cal.get(Calendar.DAY_OF_MONTH);
        System.out.println("day = " + day);

        // 设置属性
        // 奥运会 2008, 8, 8
        // cal.setYear(2008);
        cal.set(Calendar.YEAR, 2008); // 通用的设置某个属性的方法，必须提供属性的对应的整数
        cal.set(Calendar.MONTH, 7);
        cal.set(Calendar.DAY_OF_MONTH, 8);

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        System.out.println(sdf.format(cal.getTime())); // 把Calendar转换为Date对象

        // 累积
        cal.add(Calendar.YEAR, 20); // 奥运会20年后
        System.out.println(sdf.format(cal.getTime()));
        cal.add(Calendar.MONTH, -500); // 奥运会20年后再500月之前
        System.out.println(sdf.format(cal.getTime()));
        cal.add(Calendar.DAY_OF_MONTH, 1000);  // 奥运会20年后再500月之前1000天后
        System.out.println(sdf.format(cal.getTime()));
    }

    @Test
```

```
    public void test3() {
        Date date = new Date();
        System.out.println(date);
        Date date2 = new Date(2008 - 1900, 8 - 1, 8);
        System.out.println(date2);
    }

    public static void main(String[] args) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        // 写一个小时钟
        boolean flag = true;
        while (flag) {
            Date date = new Date();
            String format = sdf.format(date);
            System.out.print("\r" + format);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    @Test
    public void test2() throws ParseException {
        long time = System.currentTimeMillis();
        System.out.println("time = " + time);
        Date date = new Date();
        System.out.println(date);
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        String format = sdf.format(date);
        System.out.println(format);

        String format1 = sdf.format(time); // 也可以格式化一个毫秒数
```

```
        System.out.println(format1);

        String str = "2012-05-08 11:20:30"; // 内容必须符合格式要求
        Date parse = sdf.parse(str);
        System.out.println(parse);
    }

    @Test
    public void test1() {
        long time = System.currentTimeMillis();
        System.out.println("time = " + time);

        int[] arr1 = {1, 2, 9, 8, 0, 5, 4};
        int[] arr2 = new int[4];
        //System.arraycopy(源数组, 开始下标, 目标数组, 目标下标, 长度);
        System.arraycopy(arr1, 2, arr2, 0, arr2.length);

        for (int i : arr2) {
            System.out.println(i);
        }
        System.out.println("***************");

        String str = "abcdefg";
        System.out.println(str.hashCode());
        int code = System.identityHashCode(str); // 还原对象最原生的那个
hashCode.
        System.out.println(code);

        System.gc();
        System.exit(0);
    }
}
```

每日一考_day17

1. 完成代码

```
interface IA {
    Double toDouble(String str);
}
class TestIA {
    main() {
        String s="12.1";
        IA ia=new IA(){
            @Override
            public Double toDouble(String str){
                Double d1=new Double(str);
                return d1;
            }
        };
        Double d2=ia.toDouble(s);
        // 使用匿名内部类创建接口对象并调用其抽象方法
    }
}
```

2. 异常按照处理方式分为几种? 各包含哪些类?

   受检异常, 编译时异常 它是必须要处理的异常，如果不处理编译出错.

   Throwable 和 Exception及其子类（除RuntimeException)

   非受检异常: 运行时异常, 它不是必须要处理的异常.

   Error. 太严重

   RuntimeException及其子类

3. 判断

   1) 非受检异常就是必须不要处理的异常F
   2) 受检异常就是可以处理的异常F
   3) 非受检异常是不必须处理的异常T
   4) 受检异常可以对其忽略F
   5) 无论是什么异常,都必须对其进行处理F

6) 只有受检异常会引起程序中断F
7) 受检异常是必须对其进行处理的异常T
8) 只有非受检异常会引起程序中断F
9) 异常处理只适用于受检异常F
10) 异常处理适用于所有异常, 包括Error T

4. 异常的处理有几种方式, 各是什么, 如何处理?

   捕获：try catch [finally]

   抛出：使用throws

   先捕获再抛出：

5. 包装类的作用是什么? 包装类有哪些? 如何装箱和拆箱

   把基本数据变成对象的特殊类型, 除了int,char以外, 都是首字母大写, int : Integer, char : Character

   装箱 :

   ```
   XXX obj=new XXX(xxx)
   XXX obj=XXX.valueOf(xxx)
   XXX obj=xxx;
   Xxx obj = new Xxx("xxx");
   Xxx obj = Xxx.valueOf("xxx");
   ```

   拆箱 :

   ```
   xxx=obj;
   xxx=obj.xxxValue();
   ```

第13章 集合

Collection集合

```java
package com.atguigu.javase.collection;

import org.junit.Test;

import java.util.*;

/**
 * 集合 : 解决批量的对象的存储问题
 * 可以简单理解集合就是一个可变长Object[].
 *
 * Collection : 一堆一群, 保存一个一个的对象.
 *      无序可重复, 无序 : 不按用户的添加顺序保存元素. 可重复 : 重复的元素可以多次放入. 很难区分相同的元素的分布.
 *
 *      public boolean add(Object obj) : 向集合中添加一个任意对象, 返回值表示添加成功或失败
 *      public boolean contains(Object obj) : 判断当前集合中是否包含了参数中指定的对象元素
 *      public boolean remove(Object obj) : 从集合中删除指定的对象元素
 *      public int size() : 获取集合中的元素个数
 *
 *      Set : 一套一组, 保存一个一个的对象
 *          无序不可重复 : 不按添加顺序保存元素, 不可重复.
 *
 *          HashSet : 使用了哈希算法实现的Set集合
 *              认定2个对象重复的标准 : 2个对象equals为true, 并且这2个对象的hashCode也一样.
 *
 *          TreeSet : 基于二叉搜索树实现的Set集合
 *              认定2个对象重复的标准 : 2个对象的compare为0
 *
 *              支持2种排序 :
 *                  自然排序(对象之间自己比较大小)
 *                  定制排序(通过关联的比较器来完成对象之间比较)
 *
 *              优先定制排序, 原因是定制排序更好, 因为它是非侵入式.
```

```
 *
 *      List : 列表,清单, RandomAccess
 *          有序可重复, 会按照用户添加用户保存元素, 有序意味着有下标索引的概念
 *
 *          public void add(int index, Object obj) : 向集合中指定下标
index位置处插入新元素obj, 添加元素或插入元素永不失败.
 *          public Object get(int index) : 获取集合中指定下标Index位置的元
素, 类似于数组的[]和String的charAt
 *          public Object set(int index, Object obj) : 替换集合中指定下标
index位置的元素为新的obj, 并返回此位置处的老元素
 *          public Object remove(int index) : 删除集合中指定下标index位置
的元素, 并返回被删除的元素对象
 *
 *
 *          ArrayList : 基于数组实现的List集合
 *          Vector : 古老的ArrayList
 *          LinkedList : 基于双链表实现的List集合
 *
 * 泛型 generic : 在集合中应用时, 用于解决集合的数据类型安全问题
 *      集合类<泛型类型> 引用 = new 集合类<泛型类型>();
 *
 *
 *
 * Map : 放的是一对一对的对象
 */
class Student implements Comparable {

    private int id;
    private String name;
    private int grade;
    private double score;

    public Student() {}

    public Student(int id, String name, int grade, double score) {
        this.id = id;
```

```
        this.name = name;
        this.grade = grade;
        this.score = score;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

    public int getGrade() {
        return grade;
    }

    public void setGrade(int grade) {
        this.grade = grade;
    }

    public double getScore() {
        return score;
    }

    public void setScore(double score) {
        this.score = score;
    }
```

```
    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", grade=" + grade +
                ", score=" + score +
                '}';
    }

    /*
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Student &&
            this.id == ((Student)obj).id &&
            this.name.equals(((Student)obj).name) &&
            this.grade == ((Student)obj).grade &&
            this.score == ((Student)obj).score) {
            return true;
        }
        return false;
    }*/

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return id == student.id && grade == student.grade &&
Double.compare(score, student.score) == 0 && Objects.equals(name,
student.name);
    }

    @Override
    public int hashCode() {
```

```
            return Objects.hash(id, name, grade, score);
        }

        //@Override
        public int compareTo(Object o) {
            if (!(o instanceof Student)) {
                throw new RuntimeException("对象不可比!!!");
            }
            int n = this.grade - ((Student)o).grade;
            if (n == 0) { // 为了防止去重，永远不要返回0
                n = 1;
            }
            return n;
        }

    }

    // 自定义比较器
    class MyStudentComparator implements Comparator {
        @Override
        public int compare(Object o1, Object o2) {
            if (!(o1 instanceof Student) || !(o2 instanceof Student)) {
                throw new RuntimeException("对象不可比");
            }
            return -(int)(((Student)o1).getScore() * 100 -
    ((Student)o2).getScore() * 100);
        }
    }

    public class CollectionTest {

        @Test
        public void exer6() {
            Person p1 = new Person("张三", 178, 20, 80);
            Person p2 = new Person("李四", 168, 30, 70);
            Person p3 = new Person("王五", 180, 40, 60);
```

```
        Person p4 = new Person("赵六", 156, 50, 30);
        Person p5 = new Person("马七", 172, 10, 100);

        Set<Person> set = new TreeSet<Person>(new Comparator() {
                                                    @Override
                                                    public int
compare(Object o1, Object o2) {
                                                        return
((Person) o1).getHeight() - ((Person) o2).getHeight();
                                                    }
                                                });

        set.add(p1);
        set.add(p2);
        set.add(p3);
        set.add(p4);
        set.add(p5);

        for (Object o : set) {
            System.out.println(o);
        }
    }

    @Test
    public void test8() {
        MyStudentComparator comp = new MyStudentComparator();
        Set<Student> set = new TreeSet<Student>(comp); // 对象关联，二叉
树对象中拥有比较器对象.
        Student s1 = new Student(1, "小明", 5, 70);
        Student s2 = new Student(2, "小丽", 1, 90);
        Student s3 = new Student(3, "小花", 4, 100);
        Student s4 = new Student(4, "小刚", 6, 80.9);
        Student s5 = new Student(5, "小杰", 1, 80.2);
        set.add(s1);
        set.add(s2);
        set.add(s3);
```

```
        set.add(s4);
        set.add(s5);
        for (Object o : set) {
            System.out.println(o);
        }

    }

    @Test
    public void exer5() {
        Person p1 = new Person("张三", 178, 20, 80);
        Person p2 = new Person("李四", 168, 30, 70);
        Person p3 = new Person("王五", 180, 40, 60);
        Person p4 = new Person("赵六", 156, 50, 30);
        Person p5 = new Person("马七", 172, 10, 100);

        Set<Person> set = new TreeSet<Person>();

        set.add(p1);
        set.add(p2);
        set.add(p3);
        set.add(p4);
        set.add(p5);

        for (Object o : set) {
            System.out.println(o);
        }
    }

    @Test
    public void test7() {
        Set<Student> set = new TreeSet<Student>();
        Student s1 = new Student(1, "小明", 5, 70);
        Student s2 = new Student(2, "小丽", 1, 90);
        Student s3 = new Student(3, "小花", 4, 100);
        Student s4 = new Student(4, "小刚", 6, 80);
```

```
        Student s5 = new Student(5, "小杰", 1, 50);
        //set.add("abc");
        set.add(s1);
        set.add(s2);
        set.add(s3);
        set.add(s4);
        set.add(s5);
        for (Object o : set) {
            System.out.println(o);
        }

    }

    @Test
    public void test6() {
        Set<Integer> set = new TreeSet<Integer>();

        set.add(500);
        set.add(20);
        set.add(30);
        set.add(90);
        set.add(800);
        set.add(5);
        //set.add("999");

        System.out.println(set);
    }

    @Test
    public void test5() {
        Set<String> set = new TreeSet<String>(); // 二叉搜索树实现的Set集合，无序不可重复，内部实现了从小到大的序
        set.add("ooo");
        set.add("lkjlkj");
        set.add("234234");
        set.add("中尖是蝇");
```

```
        set.add("WERWER");
        set.add("aaa");
        set.add("bbb");
        set.add("ccc");
        //set.add(500);  //会抛出异常，因为这个500要和已有元素比较大小，

        for (Object o : set) {
            System.out.println(o);
        }
    }

    // 练习 :写一个类Person，name，height，age，weight
    // 在测试方法中创建几个Person对象，有相同的属性的多个对象放入HashSet中，是否去重. 如何做?
    @Test
    public void exer4() {
        Person p1 = new Person("张三", 178, 20, 80);
        Person p2 = new Person("李四", 168, 30, 70);
        Person p3 = new Person("张三", 178, 20, 80);

        Set set = new HashSet();

        set.add(p1);
        set.add(p2);
        set.add(p3);

        for (Object o : set) {
            System.out.println(o);
        }
    }

    @Test
    public void test3() {
        Set set = new HashSet(); // 使用哈希算法实现的Set集合，无序不可重复
        Student s1 = new Student(1, "小明", 5, 70);
        Student s2 = new Student(2, "小丽", 1, 90);
```

```
        Student s3 = new Student(1, "小明", 5, 70);

        System.out.println("s1.equals(s3) = " + s1.equals(s3));
        System.out.println(s1.hashCode());
        System.out.println(s3.hashCode());

        set.add(s1);
        set.add(s2);
        set.add(s3); // ?
        set.add(s2); // 不能添加

        //set.add(30);
        //set.add(30);

        for (Object o : set) {
            System.out.println(o);
        }
    }

    // 综合练习 : 创建Set集合，保存10个随机的50以内的整数，保证10个，并把所有数据转存到另一个List集合中
    // 求所有数的平均值，并基于List集合排序.
    @Test
    public void exer3() {
        // 集合一旦使用了泛型，它的里面的元素必须是泛型类型.
        Set<Integer> set = new HashSet<Integer>();
        while (set.size() != 10) {
            set.add((int)(Math.random() * 50));
        }
        // set.add("abc"); // 类型安全，非泛型类型的对象禁止放入
        System.out.println(set);

        List<Integer> list = new ArrayList<Integer>();
        //list.addAll(set); // 直接把参数中的集合的所有元素全部添加到当前List集合中
        // 遍历Set并转存
```

```
        for (Integer tmp : set) {
            list.add(tmp);
        }

        System.out.println(list);
        int sum = 0;
        for (int i = 0; i < list.size(); i++) {
            sum += list.get(i);
        }
        int avg = sum / list.size();
        System.out.println("avg = " + avg);
        // 冒泡
        for (int i = 0; i < list.size() - 1; i++) {
            for (int j = 0; j < list.size() - 1 - i; j++) {
                if (list.get(j) > list.get(j + 1)) {
                    Integer tmp = list.get(j);
                    list.set(j, list.get(j + 1));
                    list.set(j + 1, tmp);
                }
            }
        }
        System.out.println(list);

        System.out.println("最大值 : " + list.get(list.size() - 1));
        System.out.println("最小值 : " + list.get(0));
    }

    // 练习 : 创建一个List集合, 保存10个20以内的随机整数
    @Test
    public void exer2() {
        List list = new LinkedList();
        for (int i = 0; i < 10; i++) {
            int rand = (int)(Math.random() * 20);
            list.add(rand);
        }
        for (Object o : list) {
```

```
            System.out.print(o + " ");
        }
        System.out.println();
    }
    // 练习 : 创建一个List集合, 保存10个20以内的随机整数, 不要重复.
    @Test
    public void exer22() {
        List list = new LinkedList();
        for (int i = 0; i < 10; i++) {
            int rand = (int)(Math.random() * 20);
            if (list.contains(rand)) {
                i--;
                continue;
            }
            list.add(rand);
        }
        for (Object o : list) {
            System.out.print(o + " ");
        }
        System.out.println();
    }

    @Test
    public void exer23() {
        List list = new LinkedList();
        for (int i = 0; i < 10; ) {
            int rand = (int)(Math.random() * 20);
            if (list.indexOf(rand) == -1) {
                list.add(rand);
                i++;
            }
        }
        for (Object o : list) {
            System.out.print(o + " ");
        }
        System.out.println();
```

```
    }

    @Test
    public void test2() {
        //new List();
        List<Object> list = new ArrayList<Object>(); // 有序可重复
        list.add("DDDD"); // 尾部插入
        list.add(new Integer(300));
        list.add(300); // list.add(Integer.valueOf(300))
        list.add("yyyy");
        list.add(2);
        list.add(new Student(2, "小花", 1, 80));
        list.add("DDDD");

        System.out.println(list);
        list.add(2, "我是汉字"); // 中间插入
        System.out.println(list);

        Object o = list.get(6); // 随机获取某元素
        System.out.println(o);

        list.set(0, "大家好"); // 替换

        // list.remove(2); // 优先当作下标处理了
        list.remove(Integer.valueOf(2)); // 要想当作对象删除，必须真的传入
对象

        System.out.println(list);
        // 经典for
        for (int i = 0; i < list.size(); i++) {
            Object tmp = list.get(i);
            System.out.print(tmp + " ");
        }
        System.out.println();

    }
```

```java
// 练习 : 创建一个Set集合, 保存10个20以内的随机整数
@Test
public void exer1() {
    Set set = new HashSet();
    for (int i = 0; i < 10; i++) {
        int rand = (int)(Math.random() * 20);
        set.add(rand);
    }
    // 遍历
    for (Object tmp : set) {
        System.out.print(tmp + " ");
    }
    System.out.println();
}

// 练习 : 创建一个Set集合, 保存10个20以内的随机整数, 真的10个.
@Test
public void exer12() {
    Set set = new HashSet();
    while (set.size() != 10) {
        int rand = (int)(Math.random() * 20);
        set.add(rand);
    }
    // 遍历
    for (Object tmp : set) {
        System.out.print(tmp + " ");
    }
    System.out.println();
}

@Test
public void exer13() {
    Set<Integer> set = new HashSet<Integer>();
    for (int i = 0; i < 10; i++) {
        int rand = (int)(Math.random() * 20);
```

```
            boolean flag = set.add(rand);
            if (!flag) {
                i--; // 当次循环作作废
            }
        }

        // 遍历
        for (Object tmp : set) {
            System.out.print(tmp + " ");
        }
        System.out.println();
    }

    @Test
    public void exer14() {
        Set<Integer> set = new HashSet<Integer>();
        for (int i = 0; i < 10; ) {
            int rand = (int)(Math.random() * 20);
            if (set.add(rand)) {
                i++;
            }
        }

        // 遍历
        for (Object tmp : set) {
            System.out.print(tmp + " ");
        }
        System.out.println();
    }

    @Test
    public void test1() {
        //new Set();
        Set<Object> set = new HashSet<Object>(); // Set:无序不可重复
        boolean b1 = set.add("ccc");
        boolean b2 = set.add("QQQQQ");
```

```
        boolean b3 = set.add(new Integer(500));
        boolean b4 = set.add(20); // 自动装箱.
set.add(Integer.valueOf(20));
        boolean b5 = set.add(new Student(1, "小明", 5, 70));
        boolean b6 = set.add("ccc");
        boolean b7 = set.add(500); // 重复了, 内容相等的对象就是重复的. 不仅
仅是同一个对象

        System.out.println("b1 = " + b1);
        System.out.println("b2 = " + b2);
        System.out.println("b3 = " + b3);
        System.out.println("b4 = " + b4);
        System.out.println("b5 = " + b5);
        System.out.println("b6 = " + b6);
        System.out.println("b7 = " + b7);
        System.out.println("set.size() = " + set.size()); // 获取元素个
数
        System.out.println(set);
        System.out.println("set.contains(30) = " + set.contains(30));
// 判断是否包含
        System.out.println("set.contains(20) = " + set.contains(20));
        System.out.println("set.remove(\"ccc\") = " +
set.remove("ccc")); // 删除元素
        // 遍历集合, 使用增强型for
        /*
        for (元素数据类型 临时变量 : 集合名) {
            访问临时变量
        }*/
        for (Object tmp : set) {
            System.out.println(tmp);
        }

    }

}
```

迭代器和Map集合

```
package com.atguigu.javase.collection;

import org.junit.Test;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.*;

/**
 * Collection : 一个一个的对象
 *      无序可重复
 *
 *      Set : 无序不可重复
 *      List : 有序可重复
 *
 * Map : 一对一对的对象
 *      保存的是具有单向一对一映射的 Key -> Value键值对对象
 *      内部保存所有键对象的集合是一个Set, 无序不可重复
 *      保存所有值对象的集合是一个普通的Collection.
 *
 *      Map就是一个普通的Set, 里面保存的都是Entry条目对象, 每个条目都包含了键和值对象.
 *
 *      Object put(Object key, Object value) : 放入键值对对象, 写入一个词条, 如果键重复, 则会返回老值对象.
 *      Object get(Object key) : 获取到参数指定的键对象所映射的值对象, 查词典
 *      Object remove(Object key) : 删除键对象所映射的值的条目对象, 返回被删除的值对象
 *      int size() : 获取集合中的条目个数
 *      Set keySet() : 获取Map中保存了所有键对象的Set子集合
 *      Set entrySet() : 获取Map对应的Set集合, 里面全是条目对象.
```

```
 *
 *      HashMap : 使用哈希算法实现的Map集合
 *      Hashtable : 古老的HashMap
 *          Properties : 专门处理属性的配置的信息的Map
 *      TreeMap : 基于二叉搜索树实现的Map集合.
 */
public class CollectionTest {

    @Test
    public void test4() {
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            list.add((int)(Math.random() * 20));
        }
        System.out.println(list);
        Collections.sort(list); // 自然排序
        System.out.println(list);
        Collections.shuffle(list); // 乱序
        System.out.println(list);
        Collections.sort(list, new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return -(o1 - o2);
            }
        }); // 定制排序
        System.out.println(list);
        Collections.reverse(list); // 反转
        System.out.println(list);
        System.out.println("Collections.max(list) = " +
Collections.max(list));
        System.out.println("Collections.min(list) = " +
Collections.min(list));
    }

    @Test
    public void test3() throws IOException {
```

```
        // name = 张三, age = 40, gender = 男
        Properties properties = new Properties();
        // 一次性读取配置文件中的所有条目
        properties.load(new FileInputStream("test.properties"));

        String url = properties.getProperty("url");
        System.out.println(url);
        String port = properties.getProperty("port");
        System.out.println(port);

    }

    @Test
    public void exer() {
        Map<Integer, Integer> map = new HashMap<Integer, Integer>();
        for (int i = 1; i < 51; i++) {
            int area = (int)(Math.PI * i * i);
            map.put(i, area);
        }
        Set<Integer> keys = map.keySet();
        Iterator<Integer> iterator = keys.iterator();
        while (iterator.hasNext()) {
            Integer key = iterator.next();
            Integer value = map.get(key);
            System.out.println(key + " ************ " + value);
        }
    }

    @Test
    public void test2() {
        Map<Integer, String> map = new HashMap<Integer, String>();
        map.put(9, "nine"); // 写入词条
        map.put(2, "two");
        //map.put("aa", 30);
        map.put(0, "zero");
        map.put(5, "five");
```

```
        map.put(10, "ten");
        map.put(2, "TWO"); // 重复的键对象会导致新值覆盖老值

        System.out.println("map.size() = " + map.size());
        String val = map.get(10); // 查词典，如果没有键对象，返回null
        System.out.println("val = " + val);
        String val2 = map.get(20);
        System.out.println("val2 = " + val2);
        System.out.println(map);

        Set<Integer> keySet = map.keySet();// 获取所有键对象
        Iterator<Integer> iterator = keySet.iterator();
        while (iterator.hasNext()) {
            Integer key = iterator.next();
            String value = map.get(key); // 动态地根据键对象获取相应的值对
象
            System.out.println(key + " >>>>>>>>>>>>>>>>>>>>>>> " +
value);
        }
        System.out.println("**************************");



        Set<Map.Entry<Integer, String>> entries = map.entrySet();
        for (Map.Entry<Integer, String> entry : entries) {
            System.out.println(entry.getKey() + "
::::::::::::::::::: " + entry.getValue());
        }
    }

    @Test
    public void test1() {
        Set<Integer> set = new HashSet<Integer>();
        while (set.size() != 10) {
            set.add((int)(Math.random() * 20));
        }
```

```
        System.out.println(set);
        int sum = 0;
        for (Integer tmp : set) {
            sum += tmp;
        }
        System.out.println("sum = " + sum);
        int avg = sum / set.size();
        System.out.println("avg = " + avg);

        List<Integer> list = new ArrayList<>();
        list.addAll(set);
        list.add(11);
        System.out.println(list);
        // 迭代器 : 是集合的一个观察者对象，通过使用迭代器，就可以间接地遍历到集合中的元素
        // 1) 不可以new，而是向集合对象要
        Iterator<Integer> iterator = list.iterator();
        // 2) 内部游标刚开始时在第1个元素之前
        // 注意点1 迭代器拿到后,应该马上使用，不要再修改集合，如果修改集合会导致迭代器被污染
        // list.add(500);
        // 3) 循环不断询问当前游标后面是否还有下一个元素
        while (iterator.hasNext()) { // hasNext仅用于检测
            // 4) 如果有，真的取到下一个元素，游标要向后移动一下
            Integer next = iterator.next(); // 注意2 : next方法在循环中必须只能调用1次
            System.out.println(next);
        }
        // 注意3 : 迭代器用完作废.
    }
}
```

每日一考_day19

1. 写出代码, 获取当前的年月日, 并生成或修改一个对象, 使对象代表的时间是2012-12-25日, 再获取此日期的500天前的日期

```
public static void main(String[] args) {
    // LocalDate是内容不可改变的对象
  LocalDate now = LocalDate.now () ;
    LocalDate d = now.withYear(2012).withMonth(12).withDayOfMonth(25);
    LocalDate d2 = d.minusDays(500);

    // Calendar是内容可以改变的对象
    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.YEAR, 2012);
    cal.set(Calendar.MONTH, 11);
    cal.set(Calendar.DAY_OF_MONTH, 25);
    cal.add(Calendar.DAY_OF_MONTH, -500);

}
```

2. Collection接口表示的集合, 存储数据有什么特点? 它的两个子接口分别是什么? 分别的存储数据的特点又是什么?

```
Collection 无序的可重复的对象集合;
```

Set： 无序不可重复的对象集合，可以存null;

List： 有序可重复的对象的集合

3. ArrayList和LinkedList有区别吗? 区别是什么? 为什么能用链表实现List集合?

有区别, 底层实现不同，ArrayList是数组实现的. LinkedList是双链表实现的.

每个链表节点都有next指针, 元素之间可以按照顺序链接,

Set集合为什么无序？

内部有特定的序, 且这个序和元素自身相关. 一定和使用者无关.

4. HashSet集合中是如何判断两个对象重复的? 要想让两个自定义对象重复, 如何才能做到?

   2个对象的equals为true, 并且这2个对象的hashCode也一样.

   在自定义类中必须重写这2个方法.

5. 泛型的作用是什么? 如何使用?

   泛型表示的是某种类型, 可以用于集合, 用于约束集合中的元素的类型, 使得集合的使用更安全和方便.

   List list = new ArrayList();

第14章 泛型

```
package com.atguigu.javase.collection;

import org.junit.Test;

import java.util.*;
```

```java
/**
 * 泛型 :解决类型安全问题
 * 用在集合中可以让集合更安全和方便
 *
 * 泛型类
 */
class Person<C> { // C在这里就是代表泛型的类型参数，C代表某种类型，具体是什么类型 ，不知道. C类型虽然不知道是什么
    // 但是并不影响我们使用它作为类型. 数据类型可以声明变量，作为方法的参数的类型，作为方法返回值类型.
    // C是可以理解为类的成员. C类型会在此类被使用时，由使用者在类名<实际类型>决定.
    // 泛型类型主要用于替换Object类型，因为Object类型太过模糊.
    // 泛型类本质上隶属于当前类的对象. 不可以在静态环境中使用泛型类型

    //private static C info2;

    private String name;
    private C info;

    public Person() {}

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

    public String getName() {
        return name;
    }

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

```
    public C getInfo() {
        return info;
    }

    public void setInfo(C info) {
        this.info = info;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", info=" + info +
                '}';
    }
}
 /*
 * 泛型方法 : 局部泛型
 */
class GenericMethod<Q> {

    // 在方法的返回值类型左面使用<泛型类型参数>
     // 泛型方法100%要有参数，并且是泛型类型的参数 只要这样，方法在调用时才能通过实参的对象类型来反推泛型类型
     // 泛型类型和方法的一次调用相关.
    public static <D> D test(D d) {
        return null;
    }

    // 非静态泛型方法所属的类如果也是泛型化了的，必须在使用时把类也泛型化才能使用
    public <F> F test2(F f) {
        return f;
    }

    // public static void test2(D d){}
```

```
 }

/*
 * 泛型和继承的关系
 */
class Base<X> {

    private X x;

    public X getX() {
        return x;
    }
    public void setX(X x) {
        this.x = x;
    }
}

class Sub1 extends Base {} // 子类继承泛型父类时，选择忽略泛型

class Sub2 extends Base<String> {} // 子类继承泛型父类时，把泛型写死
class Sub3 extends Base<Double> {}

class Sub4<Y> extends Base<Y> {} // 子类继续保持泛型

/*
 * 泛型和多态的关系 ：
 * 泛型不支持直接的多态
 */
public class GenericTest {

    /**
     * 终极版的找最大值算法
     * @param col，任意集合，任意可比较的对象
     * @return 最大值
     */
    public Comparable max(Collection<? extends Comparable> col) {
```

```
        Iterator<? extends Comparable> iterator = col.iterator();
        Comparable max = iterator.next();
        while (iterator.hasNext()) {
            Comparable next = iterator.next();
            if (next.compareTo(max) > 0) {
                max = next;
            }
        }
        return max;
    }

    @Test
    public void test11() {
        List<Integer> list1 = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            list1.add((int) (Math.random() * 20));
        }
        view(list1);
        System.out.println("max(list1) = " + max(list1));
        List<Float> list2 = new ArrayList<Float>();
        for (int i = 0; i < 10; i++) {
            list2.add((int)(Math.random() * 20 * 1000) / 1000.0f);
        }
        view(list2);
        System.out.println("max(list2) = " + max(list2));

        Set<Integer> set = new HashSet<>();
        for (int i = 0; i < 10; i++) {
            set.add((int)(Math.random() * 50));
        }
        System.out.println(set);
        System.out.println("max(set) = " + max(set));

        Set<String> set2 = new HashSet<>();
        set2.add("22虽是夺");
        set2.add("lkajsdflkjadsf");
```

```
        set2.add("ASDFASDF");
        set2.add("234324234kjadsf");
        set2.add("汉字孙不jadsf");
        set2.add("为为为不中lkjadsf");
        set2.add("基材asddfasdf");

        System.out.println("max(set2) = " + max(set2));
    }

    /*
    public Number max(List<? extends Number> list) {
        Number max = list.get(0);
        for (int i = 0; i < list.size(); i++) {
            Number number = list.get(i);
            if (number.doubleValue() > max.doubleValue()) {
                max = number;
            }
        }
        return max;
    }

    @Test
    public void test11() {
        List<Integer> list1 = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            list1.add((int) (Math.random() * 20));
        }
        view(list1);
        System.out.println("max(list1) = " + max(list1));
        List<Float> list2 = new ArrayList<Float>();
        for (int i = 0; i < 10; i++) {
            list2.add((int)(Math.random() * 20 * 1000) / 1000.0f);
        }
        view(list2);
        System.out.println("max(list2) = " + max(list2));
    }
```

```
     */

    /**
     * 求任意泛型类型为Number子类类型的集合的平均值
     * ? extends Number的好处是此集合是安全的，且使用也是方便的，兼容性也很好.
     * @param list
     * @return
     */
    public double avg(List<? extends Number> list) {
        double sum = 0;
        for (int i = 0; i < list.size(); i++) {
            Number number = list.get(i);
            sum += number.doubleValue();
        }
        double avg = sum / list.size();
        return avg;
    }

    @Test
    public void test10() {
        List<Integer> list1 = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            list1.add((int) (Math.random() * 20));
        }
        view(list1);
        System.out.println("avg(list1) = " + avg(list1));
        List<Double> list2 = new ArrayList<Double>();
        for (int i = 0; i < 10; i++) {
            list2.add((int)(Math.random() * 20 * 1000) / 1000.0);
        }
        view(list2);
        System.out.println("avg(list2) = " + avg(list2));
    }

    @Test
    public void test9() {
```

```
        // list1中保存的全是未知类型的对象
        List<?> list1 = null;

        // list2中保存了Number及其未知父类类型的对象
        // list2集合适合于添加元素，但是不适合获取元素
        List<? super Number> list2 = new ArrayList<Number>();
        list2.add(300); // 添加Nubmer对象时，可以被集合兼容
        list2.add(3.22);
        //list2.add(new Object()); // 添加已知父类对象时，被禁止.
        Object o = list2.get(0);

        // list3集合中保存了Number及其未知子类类型的对象
        // list3不适合添加元素，只要添加一定是子，并且子是已知的. 适合获取元素，
可以获取到Number类型的对象.
        List<? extends Number> list3 = new ArrayList<Number>();
        //list3.add(500);
        //list3.add(3.222);
        Number number = list3.get(0);
    }

    /**
     * 不会给集合乱加元素，是一个只读式的使用.
     * 在方法中是禁止乱加元素的. 更安全更兼容
     * @param list
     */
    public void view(List<?> list) {
        // list.add("aaa");
        // list.add(500);
        // list.add(3.22);
        for (int i = 0; i < list.size(); i++) {
            Object obj = list.get(i);
            System.out.print(obj + " ");
        }
        System.out.println();
    }
```

```java
    @Test
    public void test8() {
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            list.add((int) (Math.random() * 20));
        }
        view(list);

        List<Double> list2 = new ArrayList<Double>();
        for (int i = 0; i < 10; i++) {
            list2.add((Math.random() * 20));
        }
        view(list2);
    }

    @Test
    public void test7() {
        // 左面的泛型和右面的泛型必须是一样的，如果有差异性就会导致 编译时理论上可保存的元素类型和 实际运行时集合可以保存的元素类型之间矛盾了.
        //List<Number> list = new ArrayList<Integer>();
        // 如果这样可以，会出现编译时和运行时冲突!!!!
        //list.add(3.22);

        List<Integer> list = new ArrayList<Integer>();
        //List list2 = list; // 这样写 ，虽然不报错，但是逻辑上有问题了.
        //list2.add("asdf");

        // ?在这里的作用是表示未知，此list3集合中保存的都是未知类型的对象.
        // 此list3集合不能添加元素，获取元素也只能获取到类型最模糊的Object类型的对象

        List<?> list3 = list;
        //list3.add(40); // 因为添加40时，40是一个已知类型的对象.
        //list3.add("aa");
        //list3.add(new Object());
        list3.add(null); // null表示空，什么也没有，对象也是未知类型的状况.
```

```
        //Integer obj = null;
        //list3.add(obj);
        Object o = list3.get(0);

    }

    @Test
    public void test6() {
        Sub1 sub1 = new Sub1();
        Object x = sub1.getX();

        Sub2 sub2 = new Sub2();
        String x1 = sub2.getX();
        String x2 = new Sub2().getX();

        Double x3 = new Sub3().getX();
        Double x4 = new Sub3().getX();

        Sub4<Float> floatSub4 = new Sub4<>();
        Float x5 = floatSub4.getX();
        Sub4<Long> longSub4 = new Sub4<>();
        Long x6 = longSub4.getX();
        Sub4 sub4 = new Sub4();
        Object x7 = sub4.getX();
    }

    @Test
    public void test5() {
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            list.add((int) (Math.random() * 20));
        }
        //Object[] arr1 = list.toArray();
        Integer[] arr2 = list.toArray(new Integer[0]);// 参数中的数组对象
的作用仅用作提醒方法
        for (Integer i : arr2) {
```

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

    @Test
    public void test4() {
        //Object test = GenericMethod.test();
        Integer test = GenericMethod.test(200);
        String test1 = GenericMethod.test("agcc");
        Object test2 = GenericMethod.test(null); // 调用泛型方法时，不要
传入null
        Double obj = null;
        Double test3 = GenericMethod.test(obj);

        Float v = new GenericMethod<String>().test2(30.22f);


    }

    @Test
    public void test3() {
        Person<Integer> p1 = new Person<Integer>("张三", 30);
        Integer info1 = p1.getInfo();
        p1.setInfo(40);

        Person<Boolean> p2 = new Person<Boolean>("李四", true);
        Boolean info2 = p2.getInfo();

        Person p3 = new Person("王五", false);
        Object info3 = p3.getInfo();
    }

    @Test
    public void test2() {
        Person p1 = new Person("张三", 30);// Integer
        p1.setInfo(false);
```

```
        Object info1 = p1.getInfo();

        Person p2 = new Person("李四", 3.22);
        Object info2 = p2.getInfo();
    }

    @Test
    public void test1() {
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            list.add((int)(Math.random() * 20));
        }
        // list.add("aaa"); 类型安全
        for (int i = 0; i < list.size(); i++) {
            Integer tmp = list.get(i); // 获取元素方便
            System.out.println(tmp);
        }
    }
}
```

源码研究

ArrayList

```
package com.atguigu.javase.collection;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/**
 * 集合 : 可以看作是一个可变长的Object数组
```

```
 * 泛型 : 解决类型安全问题.
 */
public class ArrayListTest {

    public static void main(String[] args) {
        // 左面是编译时泛型，右面是运行时泛型. 左右的泛型必须是一致的. 否则出现矛盾.
        List<String> list = new ArrayList<>(); // 初始状态时，内部数组是空数组
        list.add("yyy"); // 第1次添加元素时一定会扩容
        list.add("aa");
        //list.add(300);
        list.add("bbb");
        list.add("zz");
        list.add("xx");
        list.add("qq");
        list.add("cc");
        list.add("ppp");
        list.add("hello");
        list.add("汉字");
        list.add("大家好");


        System.out.println("list.size() = " + list.size());

        // 迭代器
        Iterator<String> iterator = list.iterator();

        list.remove(4); // 会导致迭代器被污染，内部的修改次数++

        // 迭代器必须要使用新鲜的，热气的.
        while (iterator.hasNext()) {
            String next = iterator.next();
            System.out.println(next);
        }
    }
```

```
}
```

```
/**
 * 基于数组实现的List集合，有序可重复
 * 内部就是一个Object[] 来保存所有元素
 *
 * 可以随机访问，直接访问某个元素
 * @param <E>
 */
public class ArrayList<E> extends AbstractList<E> implements List<E>,
RandomAccess, Cloneable, java.io.Serializable {

    /**
     * Default initial capacity.
     * 缺省的容量 ： 数组的长度. 默认为10
     */
    private static final int DEFAULT_CAPACITY = 10;

    /**
     * Shared empty array instance used for empty instances.
     */
    private static final Object[] EMPTY_ELEMENTDATA = {};

    /**
     * Shared empty array instance used for default sized empty
instances. We
     * distinguish this from EMPTY_ELEMENTDATA to know how much to
inflate when
     * first element is added.
     * 缺省的空数组对象
     */
    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA =
{};

    /**
```

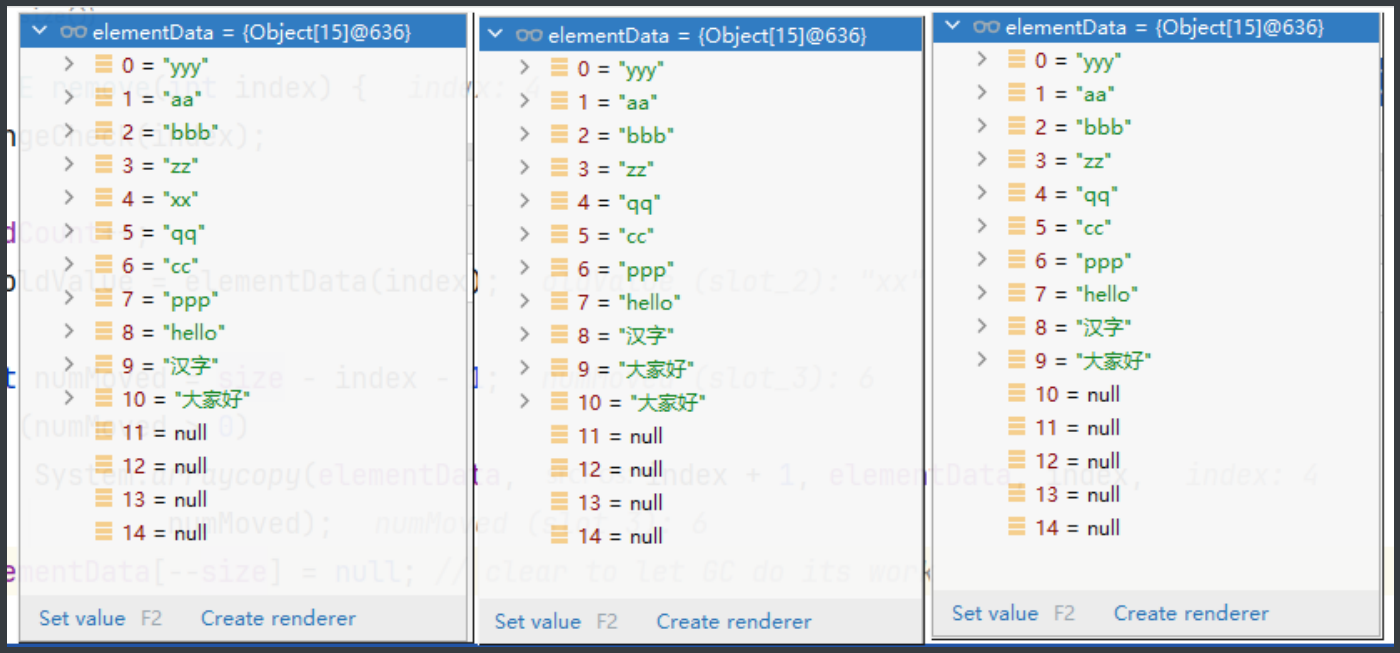
```
     * The array buffer into which the elements of the ArrayList are
stored.
     * The capacity of the ArrayList is the length of this array
buffer. Any
     * empty ArrayList with elementData ==
DEFAULTCAPACITY_EMPTY_ELEMENTDATA
     * will be expanded to DEFAULT_CAPACITY when the first element is
added.
     * 内部数组，始终是结构良好的状态.
     */
    transient Object[] elementData; // non-private to simplify nested
class access

    /**
     * The size of the ArrayList (the number of elements it contains).
     * 计数器，是算法的灵魂，控制数组的尾部插入的下标指示器.
     *
     * @serial
     */
    private int size;

    // 修改次数计数器，如果要修改集合的内容都会导致这个modCount++，添加元素，删
除，插入
    // 用于控制迭代器同步的一个关键变量.
    protected transient int modCount = 0;

    // 此时内部数组是一个空数组，不可以保存对象.
    public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }

    public boolean add(E e) {
        // 确保容量的内部方法. 如果容量不够就要扩容
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        // 标准的尾部插入操作
        elementData[size++] = e;
```

```
        // List集合的插入永不失败!!
        return true;
    }

    private void ensureCapacityInternal(int minCapacity) { // 第1次调用
时参数为1
        // 先计算容量 (内部数组, 最小容量:1), 第1次添加元素时会返回10
        // 确保明确的容量(10)
        ensureExplicitCapacity(calculateCapacity(elementData,
minCapacity));
    }

    private void ensureExplicitCapacity(int minCapacity) { // 10
        // 添加元素一定要成功, 修改次数++
        modCount++;

        // overflow-conscious code
        // 需要的最小容量 比 数组实际的容量 要大
        if (minCapacity - elementData.length > 0) { // 10 - 0
            // 第1次扩容时就直接扩容到10
            grow(minCapacity);
        }
    }

    private void grow(int minCapacity) { // 第1次是10
        // overflow-conscious code
        // 老容量 : 第1次是0
        int oldCapacity = elementData.length;
        // 新容量 : 扩容1.5倍, 位运算效率最高. 第1次时还是0
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        if (newCapacity - minCapacity < 0) { // 第1次还会进入, 0 - 10
            // 新容量 = 10
            newCapacity = minCapacity;
        }
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
```

```
        // minCapacity is usually close to size, so this is a win:
        // 真正扩容(内部数组, 新容量10);
        elementData = Arrays.copyOf(elementData, newCapacity);
    }

    public static <T,U> T[] copyOf(U[] original, int newLength,
Class<? extends T[]> newType) {
        T[] copy = ((Object)newType == (Object)Object[].class)
                ? (T[]) new Object[newLength]
                : (T[]) Array.newInstance(newType.getComponentType(),
newLength);
        // 数组复制 : 第1个参数是源数组对象, 第2个参数是源数组开始下标, 第3个参
数是目标数组对象, 第4个参数是目标数组开始下标
        // 第5个参数是要复制的元素个数.
        System.arraycopy(original, 0, copy, 0,
Math.min(original.length, newLength)); // 第1次扩容时什么也没复制
        return copy;
    }

    private static int calculateCapacity(Object[] elementData, int
minCapacity) { // 1
        if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) { // 第1
次添加元素时一定会进入if
            return Math.max(DEFAULT_CAPACITY, minCapacity); // max(10,
1)
        }
        return minCapacity;
    }

    public E remove(int index) { // 4
        // 检查参数的正确性
        rangeCheck(index);

        // 修改次数++, 11=>12
        modCount++;
        // 要删除的老值
```

```
        E oldValue = elementData(index);
        // 移动后面的所有有效元素的个数 : size:11 - index:4 - 1 == 6
        int numMoved = size - index - 1;
        if (numMoved > 0) {
            // 把后面的所有有效元素向前移动一位
            System.arraycopy(elementData, index + 1, elementData,
index, numMoved);
        }
        // 把之前的最后有效元素置空，并调整了计数器
        elementData[--size] = null; // clear to let GC do its work
        // 返回被删除的老值
        return oldValue;
    }


    public Iterator<E> iterator() {
        return new Itr();
    }

    /**
     * An optimized version of AbstractList.Itr
     */
    private class Itr implements Iterator<E> {
        int cursor;       // index of next element to return
        int lastRet = -1; // index of last element returned; -1 if no
such
        // 迭代器对象中的 期望中的修改次数 : 11
        int expectedModCount = modCount;

        Itr() {}

        public boolean hasNext() {
            return cursor != size;
        }

        @SuppressWarnings("unchecked")
```

```
        public E next() {
            checkForComodification();
        }

        final void checkForComodification() {
            // 内部期望的修改次数 和 当前集合的真实的修改次数不一样，说明出问题了，直接抛出异常.
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }
    }


}
```



LinkedList

```
/**
 * 双链表实现的List集合
```

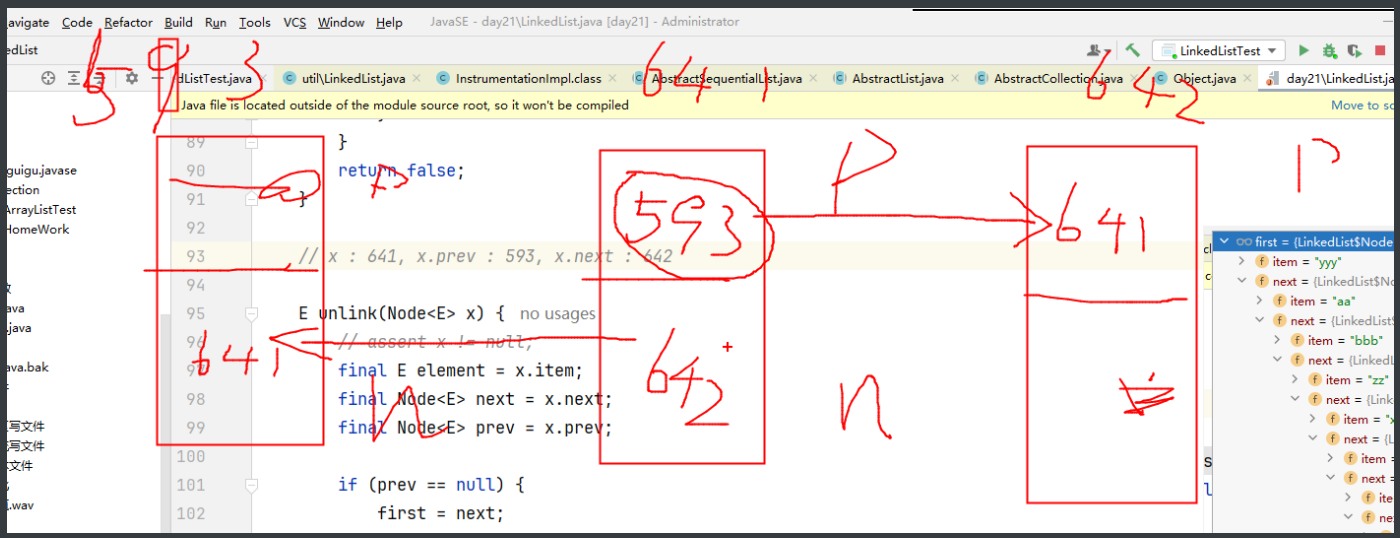
```
 * @param <E>
 */
public class LinkedList<E> extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, java.io.Serializable {

    // 节点类
    private static class Node<E> {
        E item; // 值域
        Node<E> next; // 下一个指针
        Node<E> prev; // 上一个指针
        // 上一个指针，数据，下一个指针
        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }

    // 计数器
    transient int size = 0;

    /**
     * Pointer to first node.
     * Invariant: (first == null && last == null) ||
     * (first.prev == null && first.item != null)
     */
    transient Node<E> first;

    /**
     * Pointer to last node.
     * Invariant: (first == null && last == null) ||
     * (last.next == null && last.item != null)
     */
    transient Node<E> last;

    // 控制迭代器同步性的修改次数.
```

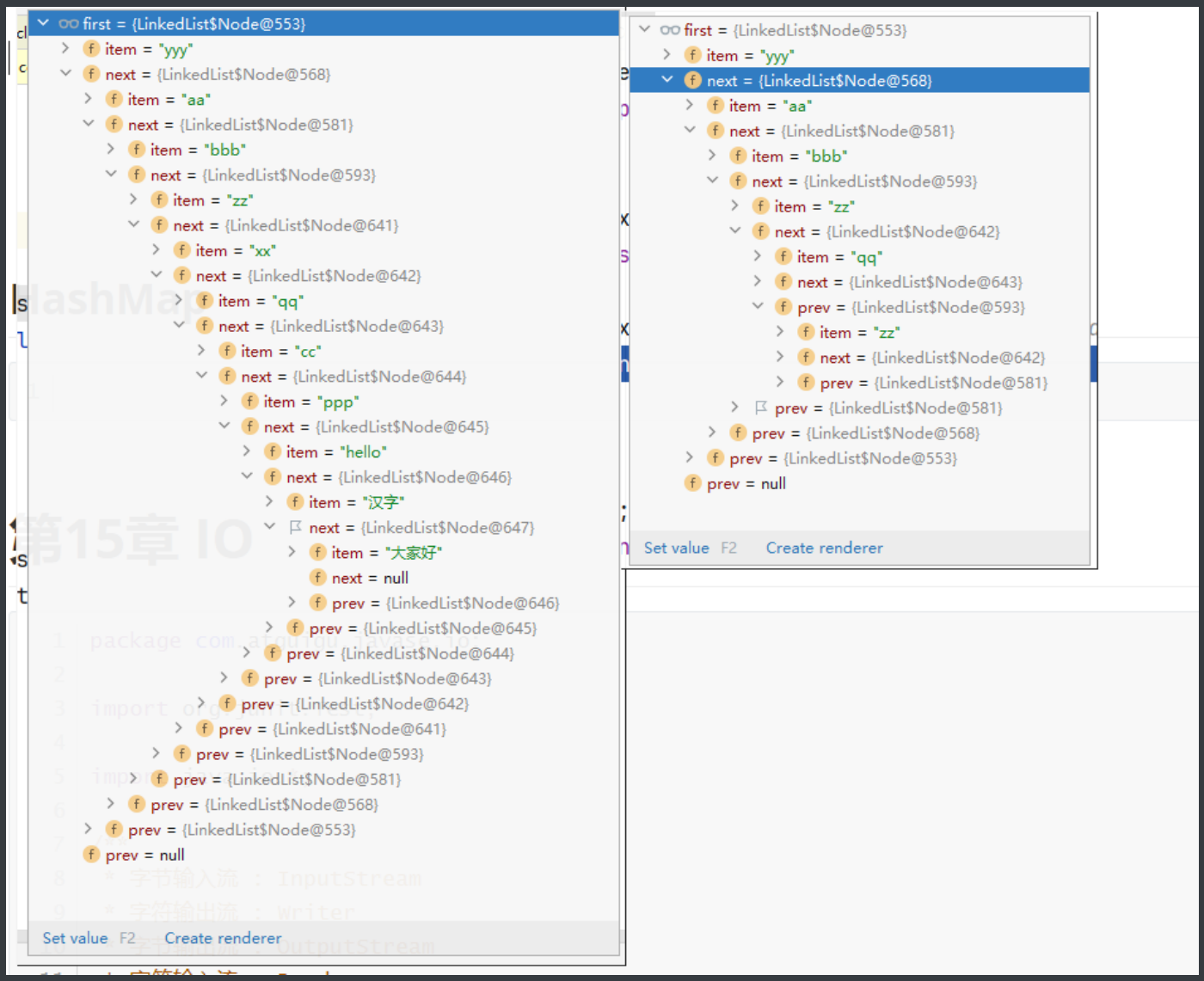
```
    protected transient int modCount = 0;

    /**
     * Constructs an empty list.
     * 头指针和尾指针都是null
     */
    public LinkedList() {
    }

    public boolean add(E e) {
        // 默认就是尾部链入
        linkLast(e);
        // 永不失败
        return true;
    }

    void linkLast(E e) {
        // l就是当前尾指针
        final Node<E> l = last;
        // 创建新的节点对象，携带数据e，新节点的上一个指针直接就是当前尾，下一个指针是null
        final Node<E> newNode = new Node<>(l, e, null);
        // 刷新尾指针，指向最新节点
        last = newNode;
        if (l == null) { // 如果老尾为null，链表为空
            // 第1个新节点直接作为头
            first = newNode;
        } else { // 链表非空
            // 尾部链入，老尾的next指针 指向 新节点
            l.next = newNode;
        }

        // 调整计数器和修改次数
        size++;
        modCount++;
    }
```

```java
public boolean remove(Object o) { // "xx"
    if (o == null) {
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null) {
                unlink(x);
                return true;
            }
        }
    } else {
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item)) {
                unlink(x);
                return true;
            }
        }
    }
    return false;
}

// x : 641, x.prev : 593, x.next : 642
// x.prev.next = x.next => 593.next = 642
// x.next.prev = x.prev => 642.prev = 593
E unlink(Node<E> x) {
    // assert x != null;
    // 删除的节点的值
    final E element = x.item;
    // 下一个指针 : 642
    final Node<E> next = x.next;
    // 上一个指针 : 593
    final Node<E> prev = x.prev;

    if (prev == null) { // 如果要删除头节点
        first = next; // 刷新头指针, 直接指向了目标的下一个.
    } else { // 不是删除头
```

```
            prev.next = next; // 上一个节点对象的next指针 指向 目标的下一个节点
            x.prev = null; // 内部的prev指针置空
        }

        if (next == null) { // 如果要删除尾节点
            last = prev; // 刷新尾指针, 直接指向了目标的上一个
        } else { // 不是删除尾
            next.prev = prev; // 下一个节点对象的prev指针 指向 目标的上一个节点
            x.next = null; // 内部的next指针置空
        }

        x.item = null; // 目标的内部的值域指针也置空.
        // 调整计数器和修改次数
        size--;
        modCount++;
        return element;
    }


    public Iterator<E> iterator() {
        return listIterator();
    }

    private class ListItr implements ListIterator<E> {
        private Node<E> lastReturned;
        private Node<E> next;
        private int nextIndex;
        // 内部期望的修改次数 : 11
        private int expectedModCount = modCount;

        ListItr(int index) {
            // assert isPositionIndex(index);
            next = (index == size) ? null : node(index);
            nextIndex = index;
```

```
        }

        public boolean hasNext() {
            return nextIndex < size;
        }

        public E next() {
            checkForComodification();
        }

        final void checkForComodification() {
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }
    }
}
```





考试：

写出ArrayList, LinkedList, HashSet, TreeSet 各自的优点和缺点及应用场景

ArrayList：纯粹的数组

优点：尾部操作最快, 遍历速度尚可.

缺点：对内存要求高, 非末端操作最慢, 检索速度很慢

适用：存档数据, 偶尔检索

LinkedList : 纯粹的双向链表

优点 : 对内存要求低, 修改操作是非常快.

缺点 : 遍历最慢, 检索最慢

适用 : 资源管理类应用

TreeSet : 纯粹的红黑树(自平衡二叉树)

优点 : 对内存要求低, 检索速度极快

缺点 : 插入和删除操作慢, 因为要有旋转操作

适用 : 频繁检索, 不是太频繁插入或删除

HashSet : 主体是数组, 内部可能分化出链表和红黑树.

优点 : 全是优点, 都是最快

底层就是一个HashMap, 内部就是一个数组, 数组的元素类型全是条目(有键和值双对象)

put(键对象, 值对象) : 以条目对象的形式存入内部数组, 下标由键对象的hashCode()和数组的长度取余后得到最终的目标下标.

数组中会有一些空洞, 保证性能, 数组的使用率 : 加载因子.

加载因子大 : 空间使用上来了, 性能很差

加载因子小：空间使用差，性能很好.

HashMap

```
package com.atguigu.javase.map;

import java.util.HashMap;
import java.util.Map;
```

```
public class HashMapTest {

    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>(); // 此时内部数组是
null
        map.put(9, "nine"); // 第1次添加元素时一定会创建数组.
        map.put(3, "three");
        map.put(100, "hundred");
        map.put(19, "nineteen");
        map.put(50, "fifty");
        map.put(35, "thrtyfive");
        map.put(3, "THREE"); // 键重复, 刷新值
        map.put(51, "fiftyone");
        map.put(200, "200");
        map.put(300, "300");
        map.put(400, "400");
        map.put(16, "16");
        map.put(20, "20");
        map.put(30, "thirty");

        System.out.println(map.size());
        System.out.println(map);
    }
}


/**
 * 散列表, 保存键值对对象, 实现了Map
 * @param <K> K是键对象的泛型类型
 * @param <V> V是值对象的泛型类型
 */
public class HashMap<K,V> extends AbstractMap<K,V> implements
Map<K,V>, Cloneable, Serializable {

    // 缺省的初始化容量(数组的长度) : 16
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

```
    /**
     * The maximum capacity, used if a higher value is implicitly
specified
     * by either of the constructors with arguments.
     * MUST be a power of two <= 1<<30.
     *
     * 最大容量，是一个2^n的值
     */
    static final int MAXIMUM_CAPACITY = 1 << 30;

    /**
     * The load factor used when none specified in constructor.
     * 缺省的加载因子 : 数组的使用率，缺省的是 3 / 4
     */
    static final float DEFAULT_LOAD_FACTOR = 0.75f;

    /**
     * The bin count threshold for using a tree rather than list for a
     * bin.  Bins are converted to trees when adding an element to a
     * bin with at least this many nodes. The value must be greater
     * than 2 and should be at least 8 to mesh with assumptions in
     * tree removal about conversion back to plain bins upon
     * shrinkage.
     * 链表变成红黑树的节点数门槛. 为了提升 检索性能
     */
    static final int TREEIFY_THRESHOLD = 8;

    /**
     * The bin count threshold for untreeifying a (split) bin during a
     * resize operation. Should be less than TREEIFY_THRESHOLD, and at
     * most 6 to mesh with shrinkage detection under removal.
     * 红黑树变链表的节点数门槛，为了简化 处理的.
     */
    static final int UNTREEIFY_THRESHOLD = 6;
```

```
    /**
     * The smallest table capacity for which bins may be treeified.
     * (Otherwise the table is resized if too many nodes in a bin.)
     * Should be at least 4 * TREEIFY_THRESHOLD to avoid conflicts
     * between resizing and treeification thresholds.
     * 最小变红黑树的容量(数组的长度) , 数组长度只有到达此值时才可能变红黑树, 否则只是扩容处理.
     */
    static final int MIN_TREEIFY_CAPACITY = 64;

    // 散列表数组的元素类型, 条目, 内部拥有双对象
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash; // 键对象的原始哈希值, 防止key对象内部属性变化导致hashCode变化的.
        final K key; // 键对象
        V value; // 值对象
        Node<K, V> next; // 用于构建链表的下一个指针

        // 原生哈希, 键对象, 值对象, 下一个指针
        Node(int hash, K key, V value, Node<K, V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

    // 对象属性
    // 内部的散列表数组, 元素是条目类型
    transient Node<K,V>[] table;


    // 条目个数, 也是计数器, 纯粹的计数
    transient int size;

    /**
```

```
     * The number of times this HashMap has been structurally modified
     * Structural modifications are those that change the number of
mappings in
     * the HashMap or otherwise modify its internal structure (e.g.,
     * rehash).  This field is used to make iterators on Collection-
views of
     * the HashMap fail-fast.  (See ConcurrentModificationException).
     * 控制迭代器同步性的修改次数
     */
    transient int modCount;

    /**
     * The next size value at which to resize (capacity * load
factor).
     *
     * @serial
     */
    // (The javadoc description is true upon serialization.
    // Additionally, if the table array has not been allocated, this
    // field holds the initial array capacity, or zero signifying
    // DEFAULT_INITIAL_CAPACITY.)
    // 门槛 : 数组要扩容的门槛，元素到达此值后就要数组扩容
    int threshold;

    /**
     * The load factor for the hash table.
     * 当前散列表的加载因子
     * @serial
     */
    final float loadFactor;

    public HashMap() {
        // 只初始化了一个属性，加载因子 : 缺省的值:3/4
        // 其他的所有属性都是缺省的，内部散列表数组的引用table 是空指针，计数器
和门槛是0
```

```
        this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields
defaulted
    }

    static final int hash(Object key) { // 9
        int h;
        // AAAAAAAA BBBBBBBB CCCCCCCC DDDDDDDD ^
        // 00000000 00000000 AAAAAAAA BBBBBBBB =
        // AAAAAAAA BBBBBBBB EEEEEEEE EEEEEEEE =>  让键对象的哈希码值的低
16位更具散列性.
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }

    // 放入键和值对象
    public V put(K key, V value) { // 9, "nine"
        // 先通过键对象获取到它的原生哈希值.
        return putVal(hash(key), key, value, false, true);
    }

    // 最重要的放入键值对对象方法. 参数 : 原生哈希9, 键对象9, 值对象"nine"
    final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
boolean evict) {
        // 局部的tab要引用到内部散列表对象table
        Node<K,V>[] tab;
        Node<K,V> p; // 条目的对象的引用
        // n就是散列表数组的长度
        int n;
        int i;

        tab = table;
        n = tab.length; // 数组的长度
        if (tab == null || n == 0) { // 第1次添加元素时,一定会进入, 因为tab
和table此时都是null
            tab = resize(); // 指向新数组
            n = tab.length; // n是数组长度 : 第1次是16
        }
```

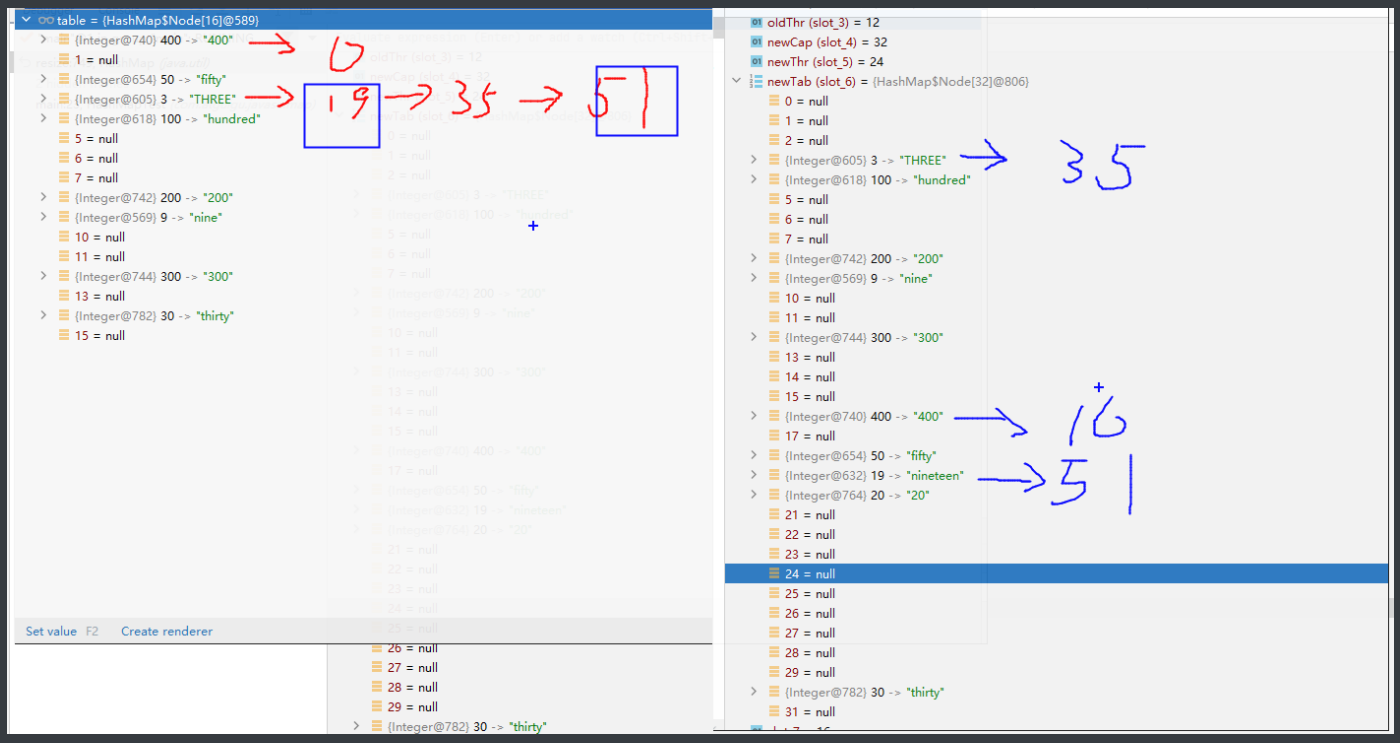
```
        // i = (16 - 1) & 9  = 9 相当于 9 % 16, 获取到目标下标
        // 0001 0000 - 1 => 0000 1111 & 只要有0就是0, 除非全是1才是1
                         // 1111 1001 =
                         // 0000 1001 => 9
        // i就是键对象最终的在数组中的目标下标
        i = (n - 1) & hash;
        // p就是目标下标位置的老元素
        p = tab[i];
        if (p == null) { // 如果老元素为null, 这是最好的情况, 因为目标下标位置虚位以待.
            // 请插入目标下标位置
            tab[i] = newNode(hash, key, value, null);
        }
        else { // 出现了下标冲突!!!!
            Node<K,V> e; // 条目
            // 键对象类型引用
            K k; // 老键对象
            // 判断新键对象key和老键对象k是否重复.
            if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k)))) {
                e = p;
            }
            else if (p instanceof TreeNode) { // 判断目标下标位置处理是否已经是红黑树了.
                // 以红黑树方式插入新的条目.
                e = ((TreeNode<K, V>) p).putTreeVal(this, tab, hash, key, value);
            }
            else { // 目标位置处要处理成链表
                // binCount是链表中的节点数的计数器
                for (int binCount = 0; ; ++binCount) {
                    // e是p的下一个节点的对象的引用.
                    e = p.next;
                    if (e == null) { // 如果p.next为null, 说明p就是尾节点了
                        // 把新条目链接到p的后面.
```

```
                        p.next = newNode(hash, key, value, null);
                        // 如果链表中的节点数到达了变树门槛
                        if (binCount >= TREEIFY_THRESHOLD - 1) {// -1
for 1st
                            // 把链表真的变成红黑树
                            treeifyBin(tab, hash);
                        }
                        // 尾部链入后结束 for
                        break;
                    }
                    // 判断p的下一个条目的键对象是否和新键对象key是否重复!!
                    if (e.hash == hash && ((k = e.key) == key || (key
!= null && key.equals(k)))) {
                        // 如果键重复, 不再链表式处理, 直接结束.
                        break;
                    }
                    // 如果没有重复.
                    p = e;
                }
            }
            if (e != null) { // 键重复了, existing mapping for key
                // 获取老值对象
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null) {
                    // 老条目的值刷新为新值
                    e.value = value;
                }
                afterNodeAccess(e);
                // 返回老值
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
```

```
        return null;
    }

    /**
     * 调整容量大小
     * @return
     */
    final Node<K,V>[] resize() {
        // 老表oldTab指向内部散列表数组对象，第1次进入时值为null，第2次不是
null
        Node<K,V>[] oldTab = table;
        // 老容量 = 老表长度，第1次进入值为0，第2次是16
        int oldCap = (oldTab == null) ? 0 : oldTab.length;
        // 老上限，第1次进入时值为0，第2次是12
        int oldThr = threshold;
        // 新容量
        int newCap = 0;
        // 新上限
        int newThr = 0;

        if (oldCap > 0) { // 第1次不进入，第2次进入
            if (oldCap >= MAXIMUM_CAPACITY) {
                threshold = Integer.MAX_VALUE;
                return oldTab;
            }
            // 第2次进入，新容量 = 老容量 * 2 => 32，容量总是2的n次方，为了让
n-1&hash效果等同于取余
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
oldCap >= DEFAULT_INITIAL_CAPACITY) {
                // 新上限 = 老上限 * 2 -> 24
                newThr = oldThr << 1; // double threshold
            }
        }
        else if (oldThr > 0) { // 第1次也不进入
            newCap = oldThr;
        }
```

```
        else { // 第1次时会进入
            // zero initial threshold signifies using defaults
            // 新容量 = 16
            newCap = DEFAULT_INITIAL_CAPACITY;
            // 新上限 = 16 * 3 / 4 = 12
            newThr = (int)(DEFAULT_LOAD_FACTOR *
DEFAULT_INITIAL_CAPACITY);
        }
        if (newThr == 0) {
            float ft = (float)newCap * loadFactor;
            newThr = (newCap < MAXIMUM_CAPACITY && ft <
(float)MAXIMUM_CAPACITY ?
                      (int)ft : Integer.MAX_VALUE);
        }
        // 当前散列表中的门槛值定为12
        threshold = newThr;
        @SuppressWarnings({"rawtypes","unchecked"})
        // 最重要的语句，真的创建散列表数组对象，长度是新容量，第1次时是16
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        // 内部散列表数组引用也指向新表
        table = newTab;
        if (oldTab != null) { // 第1次不进入，第2次进入时，会把老表中的所有
条目重新散列到新表中.
            for (int j = 0; j < oldCap; ++j) {
                Node<K,V> e;
                if ((e = oldTab[j]) != null) {
                    oldTab[j] = null;
                    if (e.next == null)
                        newTab[e.hash & (newCap - 1)] = e;
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j,
oldCap);
                    else { // preserve order
                        Node<K,V> loHead = null, loTail = null;
                        Node<K,V> hiHead = null, hiTail = null;
                        Node<K,V> next;
```

```
                        do {
                            next = e.next;
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)
                                    loHead = e;
                                else
                                    loTail.next = e;
                                loTail = e;
                            }
                            else {
                                if (hiTail == null)
                                    hiHead = e;
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        if (loTail != null) {
                            loTail.next = null;
                            newTab[j] = loHead;
                        }
                        if (hiTail != null) {
                            hiTail.next = null;
                            newTab[j + oldCap] = hiHead;
                        }
                    }
                }
            }
        }
        // 返回新的散列表数组对象的引用
        return newTab;
    }
}
```



第15章 IO

```
package com.atguigu.javase.io;

import org.junit.Test;

import java.io.*;

/**
 * 字节输入流 : InputStream
 * 字符输出流 : Writer
 * 字节输出流 : OutputStream
 * 字符输入流 : Reader
 *
 * FileXxx : 文件流
 * FileInputStream : 读二进制文件
 * FileWriter : 写文本文件
```

```
 * FileOutputStream : 写二进制文件
 * FileReader : 读文本文件
 *
 * BufferedXxx : 缓冲流
 *  BufferedReader
 *  BufferedWriter
 *
 *
 * 二进制文件 :
 *      内存中如何保存数据, 在文件中也是如何保存数据
 *
 * UTF8编码格式的字符串, 一个汉字3个字节
 * 61 62 63 e6 88 91 e5 92 8c e4 bd a0 79 79 79
 *
 * 0110 0001
 *
 * e6 88 91 => 拆解完 : 62 11
 * 1110 0110 1000 1000 1001 0001
 *
 * e5 92 8c
 * 1110 0101 1001 0010 1000 1100
 *      0101   01 0010   00 1100
 * 0101 0100 1000 1100 => 0x548C
 *
 *
 * utf8编码规则 : 从第1个字节开始数一下, 有几个连续的1, 说明有几个字节来编码, 可变长字符编码, 上面的数据是1110说明有3个字节
 * 汉字排序不友好
 * 后面的所有字节都必须要以10为开头
 * 1110 0110 1000 1000 1001 0001 , 去掉长度信息1110, 每个字节的10头也去掉
 *      0110   00 1000   01 0001
 * 0110 0010 0001 0001 => 0x6211
 *
 * GBK编码 : 码值是国标码, 每个汉字占2个字节, 数据本身就是码值. 汉字排序友好
 * CE D2 => '我' : 52946,
 */
```

```java
class Student implements Serializable {

    private int id;
    private String name;
    private int grade;
    // transient修饰符修饰的属性不会被序列化.
    private transient double score;

    public Student() {}

    public Student(int id, String name, int grade, double score) {
        this.id = id;
        this.name = name;
        this.grade = grade;
        this.score = score;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

    public int getGrade() {
        return grade;
    }
```

```
    public void setGrade(int grade) {
        this.grade = grade;
    }

    public double getScore() {
        return score;
    }

    public void setScore(double score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", grade=" + grade +
                ", score=" + score +
                '}';
    }
}
public class IOTest {

    @Test
    public void test11() {
        FileInputStream fis = null;
        ObjectInputStream ois = null;
        try {
            fis = new FileInputStream("对象序列化");
            ois = new ObjectInputStream(fis);

            Object o1 = ois.readObject();
            Object o2 = ois.readObject();
            Object o3 = ois.readObject();
```

```
            System.out.println(o1);
            System.out.println(o2);
            System.out.println(o3);
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            if (ois != null) {
                try {
                    ois.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    @Test
    public void test10() {
        // 对象序列化，把对象持久化到文件中，把对象在GC区中的属性数据写入文件
        FileOutputStream fos = null;
        ObjectOutputStream oos = null;
        try {
            fos = new FileOutputStream("对象序列化");
            oos = new ObjectOutputStream(fos);

            Student s1 = new Student(1, "小明", 5, 80);
            Student s2 = new Student(2, "小丽", 1, 90);
            Student s3 = new Student(3, "小刚", 3, 100);

            oos.writeObject(s1);
            oos.writeObject(s2);
            oos.writeObject(s3);

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
```

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

    @Test
    public void test9() throws UnsupportedEncodingException {
        int n1 = 0x6211;
        System.out.println("n1 = " + n1);
        System.out.println((char)n1);
        int n2 = 0x548C;
        System.out.println("(char)n2 = " + (char)n2);

        int n3 = 0xCED2;
        System.out.println("n3 = " + n3);
        System.out.println("(char)n3 = " + (char)n3);

        String str = "abc我和你yyy";
        // String => byte[] : 编码, 需要把字符串写入文件或通过网络传输时.
        byte[] arr1 = str.getBytes("utf8"); // 字符串以指定的编码方式utf8
来 编码
        for (byte b : arr1) {
            System.out.print(Integer.toHexString(b) + " ");
        }
        System.out.println();
        byte[] arr2 = str.getBytes("gbk"); // 以gbk编码方式编码的byte[]
        for (byte b : arr2) {
            System.out.print(Integer.toHexString(b) + " ");
        }
        System.out.println();
        // 解码 : byte[] => String , 读文本文件或从网络接收到了字符串数据.
```

```
        String str2 = new String(arr1, "utf8"); // 以指定的编码方式对byte[]进行解码
        System.out.println("str2 = " + str2);

        String str3 = new String(arr2, "gbk"); // 以指定的编码gbk方式，对arr2进行解码
        System.out.println("str3 = " + str3);


    }

    // 30, 45

    // 练习 ： 写一个二进制文件，写入20个[30~80)之间的随机int整数. 文件大小 ： 86字节
    @Test
    public void exer1() {
        FileOutputStream fos = null;
        BufferedOutputStream bos = null;
        ObjectOutputStream oos = null;
        try {
            fos = new FileOutputStream("20个随机数");
            bos = new BufferedOutputStream(fos);
            oos = new ObjectOutputStream(bos);

            for (int i = 0; i < 20; i++) {
                int rand = (int)(Math.random() * 50 + 30);
                oos.writeInt(rand);
            }

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (oos != null) {
                try {
                    oos.close();
```

```
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
// 用notepad++打开，尝试解决第1个和最后一个随机数是多少
// 再写程序，用对象流读取这些随机数并打印输出
@Test
public void exer2() {
    FileInputStream fis = null;
    BufferedInputStream bis = null;
    ObjectInputStream ois = null;
    try {
        fis = new FileInputStream("20个随机数");
        bis = new BufferedInputStream(fis);
        ois = new ObjectInputStream(bis);

        for (int i = 0; i < 20; i++) {
            int n = ois.readInt();
            System.out.println("n = " + n);
        }

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

```java
@Test
public void test8() {
    FileInputStream fis = null;
    BufferedInputStream bis = null;
    ObjectInputStream ois = null;
    try {
        fis = new FileInputStream("二进制文件");
        bis = new BufferedInputStream(fis);
        ois = new ObjectInputStream(bis);

        int n = ois.readInt();
        System.out.println("n = " + n);
        boolean b1 = ois.readBoolean();
        boolean b2 = ois.readBoolean();
        System.out.println("b1 = " + b1);
        System.out.println("b2 = " + b2);
        long l = ois.readLong();
        System.out.println("l = " + l);
        double v = ois.readDouble();
        System.out.println("v = " + v);

        String str = ois.readUTF();
        System.out.println("str = " + str);

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

```java
@Test
public void test7() {
    FileOutputStream fos = null;
    BufferedOutputStream bos = null;
    ObjectOutputStream oos = null;
    try {
        fos = new FileOutputStream("二进制文件");
        bos = new BufferedOutputStream(fos);
        oos = new ObjectOutputStream(bos);

        oos.writeInt(20); // 4
        oos.writeBoolean(true);
        oos.writeBoolean(false);
        oos.writeLong(30); // 8
        oos.writeDouble(3.14); // 8

        String str = "abc我和你yyy";
        oos.writeUTF(str); // 编码
        oos.writeChars(str); // 字符串在内存中如何 ,写到文件中也如何

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (oos != null) {
            try {
                oos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

}
```

```
    @Test
    public void test6() {
        FileWriter fileWriter = null;
        BufferedWriter bufferedWriter = null;
        try {
            fileWriter = new FileWriter("使用缓冲流写文件");
            bufferedWriter = new BufferedWriter(fileWriter);
            String[] content = {
                                "我是一些内容, 请我写到文件中吧!!!1",
                                "我是一些内容, 请我写到文件中吧!!!2",
                                "我是一些内容, 请我写到文件中吧!!!3",
                                "我是一些内容, 请我写到文件中吧!!!4",
                                "我是一些内容, 请我写到文件中吧!!!5",
                                "我是一些内容, 请我写到文件中吧!!!6",
                                "我是一些内容, 请我写到文件中吧!!!7",
                                "我是一些内容, 请我写到文件中吧!!!8",
                                "我是一些内容, 请我写到文件中吧!!!9",
                                "我是一些内容, 请我写到文件中吧!!!10",
                                "我是一些内容, 请我写到文件中吧!!!11",

 "lkjasdflkjalksdjflaksjdflaksjdflkasjdflkasjdflkjasdlkfjasdf",

 "llkJlkj234098720398472308942309842039842390840"};
            for (String s : content) {
                bufferedWriter.write(s); // 直接写字符串
                bufferedWriter.newLine(); // 写入跨平台的换行符. 最有价值方
法.
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (bufferedWriter != null) {
                try {
                    bufferedWriter.close();
                } catch (IOException e) {
                    e.printStackTrace();
```

```
                }
            }
        }
    }

    // 使用缓冲流读文本文件
    @Test
    public void test5() {
        FileReader fileReader = null;
        BufferedReader bufferedReader = null;
        try {
            fileReader = new FileReader("ArrayList.java");

            // 所谓包装就是对象关联，高级流对象关联了低级流对象
            bufferedReader = new BufferedReader(fileReader);

            String line;
            // readLine()方法读到的字符串行中没有了换行符
            while ((line = bufferedReader.readLine()) != null) { // 最有价值方法.
                // 处理已经读到的数据
                System.out.println(line);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 只需要关闭高级流
            if (bufferedReader != null) {
                try {
                    bufferedReader.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

```
    @Test
    public void test4() {
        FileWriter fileWriter = null;
        try {
            fileWriter = new FileWriter("使用缓冲区写文件");
            String[] content = {
                    "我是一些内容，请我写到文件中吧!!!1",
                    "我是一些内容，请我写到文件中吧!!!2",
                    "我是一些内容，请我写到文件中吧!!!3",
                    "我是一些内容，请我写到文件中吧!!!4",
                    "我是一些内容，请我写到文件中吧!!!5",
                    "我是一些内容，请我写到文件中吧!!!6",
                    "我是一些内容，请我写到文件中吧!!!7",
                    "我是一些内容，请我写到文件中吧!!!8",
                    "我是一些内容，请我写到文件中吧!!!9",
                    "我是一些内容，请我写到文件中吧!!!10",
                    "我是一些内容，请我写到文件中吧!!!11",

 "lkjasdflkjalksdjflaksjdflaksjdflkasjdflkasjdflkjasdlkfjasdf",
                    "llkJlkj234098720398472308942309842039842390840"};
            // 练习 : 创建一个字符串数组，内容是一些姓名，写一个文件name.txt，
把姓名中的名的部分写入文件 "张三丰"，每个名占一行.
            for (String s : content) {
                char[] arr = s.toCharArray();
                //fileWriter.write(arr); // 一次性批量的把数组中的所有字符
写入到文件中.

                // 第2个参数是数组的开始下标，第3个参数是要写的字符数
                fileWriter.write(arr, 1, arr.length - 1); // 最重点方法.

                fileWriter.write(13);
                fileWriter.write(10);
            }
        } catch (Exception e) {
            e.printStackTrace();
```

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

    @Test
    public void test3() {
        FileReader fileReader = null;
        try {
            fileReader = new FileReader("ArrayList.java");
            // 提前创建一个缓冲区数组对象
            char[] buf = new char[1000];
            // 可以读批量的字符到buf数组中去，实际读了的个数由返回值说明，返回个数为-1时表明文件读完了.
            int count;
            while ((count = fileReader.read(buf)) != -1) {
                // 处理已经读的数据，实际处理count个字符，因为数组中有可能有脏数据.
                for (int i = 0; i < count; i++) {
                    System.out.print(buf[i]);
                }
            }

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

```
                    e.printStackTrace();
                }
            }
        }
    }

    @Test
    public void test2() {
        // 写一个文本文件
        FileWriter fileWriter = null;
        try {
            fileWriter = new FileWriter("写一个文本文件"); // 文件不存在会
自动创建
            fileWriter.write('你');
            fileWriter.write('好');
            fileWriter.write('吧');
            fileWriter.write('我');
            fileWriter.write('不');
            fileWriter.write('好');
            fileWriter.write(13); // 回车
            fileWriter.write(10); // 换行

            fileWriter.write('1');
            fileWriter.write('2');
            fileWriter.write('9');
            fileWriter.write('3');
            fileWriter.write('4');
            fileWriter.write('\r');
            fileWriter.write('\n');

            fileWriter.write('a');
            fileWriter.write('x');
            fileWriter.write('c');
            fileWriter.write('u');
            fileWriter.write('z');
            fileWriter.write('q');
```

```
            fileWriter.write('p');
            fileWriter.write(13);
            fileWriter.write('\n');

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (fileWriter != null) {
                try {
                    fileWriter.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    @Test
    public void test1() {
        // 1) 声明引用，赋值为null
        FileReader fileReader = null;
        // 2) try catch finally
        try {
            // 5) 在try中创建流对象，并赋值给引用
            fileReader = new FileReader("ArrayList.java");
            // 6) 使用流对象处理数据
            int ch;
            while ((ch = fileReader.read()) != -1) {
                System.out.print((char)ch);
            }
        } catch (Exception e) {
            // 4) 在catch中打印异常消息
            e.printStackTrace();
        } finally {
            // 3) 在finally中关闭流
            if (fileReader != null) {
```

```
                try {
                    fileReader.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

转换流

```
package com.atguigu.javase.io;

import org.junit.Test;

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

/**
 * 字符输入流 : Reader
 * 字节输入流 : InputStream
 * 字符输出流 : Writer
 * 字节输出流 : OutputStream
 *
 * 文件流 : FileXxx
 *
 * 缓冲流 : BufferedXxx
 *
 * 对象流 : ObjectOutputStream 序列化, ObjectInputStream 反序列化
 *
 * 编码 : 把字符串转换为相应的byte[]过程, 写文件或通过网络传输数据
 *      String str = ....;
```

```
 *      byte[] arr = str.getBytes("编码名称");
 *      写文件
 *
 * 解码 : byte[]还原成String过程, 读文件或网络接收到数据
 *      byte[] buf = ...;
 *      String str = new String(buf, "编码名称");
 *      读文件
 *
 * FileReader是垃圾, 因为它只能处理项目默认编码的文本文件.
 * FileWriter是垃圾
 */
public class IOTest {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()) {
            if (scanner.hasNextInt()) {
                int n = scanner.nextInt();
                System.out.println("n = " + n);
            } else if (scanner.hasNextDouble()) {
                double v = scanner.nextDouble();
                System.out.println("v = " + v);
            } else {
                String next = scanner.next();
                System.out.println("next = " + next);
                if (next.equals("exit")) {
                    break;
                }
            }
        }
        scanner.close();
    }

    // 练习 : 从键盘获取若干字符串, 把它们以追加的方式写入文件
keyboard_gbk.txt. 输入内容, 再观察文件, 直到输入exit或quit为止.
    public static void main3(String[] args) {
```

```
        InputStream is = System.in;
        InputStreamReader isr = null;
        BufferedReader br = null;

        FileOutputStream fos = null;
        OutputStreamWriter osw = null;
        BufferedWriter bw = null;

        try {
            isr = new InputStreamReader(is);
            br = new BufferedReader(isr);

            fos = new FileOutputStream("keyboard_gbk.txt", true);
            osw = new OutputStreamWriter(fos, "gbk");
            bw = new BufferedWriter(osw);

            String line;
            while ((line = br.readLine()) != null) {
                if (line.equals("exit") || line.equals("quit")) {
                    break;
                }
                bw.write(line);
                bw.newLine();
                bw.flush();
            }

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

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

    public static void main2(String[] args) {
        InputStream in = System.in; // 键盘输入流
        InputStreamReader isr = null;
        BufferedReader br = null;
        try {
            isr = new InputStreamReader(in);
            br = new BufferedReader(isr);

            String line;
            while ((line = br.readLine()) != null) {
                if (line.equals("exit")) {
                    break;
                }
                System.out.println(line);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
```

```
    }

    public static void main1(String[] args) {
        //System.in;
        //System.out; 是一个打印流，并且有自动冲刷效果
        System.out.println("汉字");
        //System.out.flush();
        //System.out.close();

        System.out.println("herjlwejr");
        // err是错误流，它用另一个线程打印
        System.err.println("lajksdflkjasdf");
    }

    @Test
    public void test3() {
        // 字符缓冲流
        FileOutputStream fos = null;
        OutputStreamWriter osw = null;
        BufferedWriter bw = null;
        try {
            // 写文件时会以追加的方式写文件，如果没有第2个参数默认使用清空方式
            fos = new FileOutputStream("一个文件gbk", true);
            osw = new OutputStreamWriter(fos, "gbk");
            bw = new BufferedWriter(osw);

            bw.write("我是汉字");
            bw.newLine();
            bw.write("abcdefg");
            bw.newLine();
            bw.write("123");
            bw.newLine();

            bw.flush(); // 强行冲刷缓冲区，把数据刷入硬盘.

        } catch (Exception e) {
```

```
            e.printStackTrace();
        } finally {
            if (bw != null) {
                try {
                    bw.close(); // 在关闭输出流时会自动flush
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }

    @Test
    public void test2() {
        // 字符缓冲流,
        FileInputStream fis = null;
        InputStreamReader isr = null;
        BufferedReader br = null;
        try {
            fis = new FileInputStream("LinkedList_gbk.java");
            // 处理不同编码的文本文件时, 必须使用转换流.
            isr = new InputStreamReader(fis, "gbk");
            br = new BufferedReader(isr);

            String line;
            while ( (line = br.readLine()) != null) {
                System.out.println(line);
            }

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

```
                e.printStackTrace();
            }
        }
    }

}

@Test
public void test1() {
    // 字符缓冲流
    FileReader fr = null;
    BufferedReader br = null;
    try {
        fr = new FileReader("LinkedList.java");
        br = new BufferedReader(fr);

        String line;
        while ( (line = br.readLine()) != null) {
            System.out.println(line);
        }

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (br != null) {
            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    // 字符缓冲流
    FileWriter fw = null;
    BufferedWriter bw = null;
```

```
        try {
            fw = new FileWriter("一个文件");
            bw = new BufferedWriter(fw);

            bw.write("我是汉字");
            bw.newLine();
            bw.write("abcdefg");
            bw.newLine();
            bw.write("123");
            bw.newLine();

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (bw != null) {
                try {
                    bw.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }


}
```

第16章 多线程

```java
package com.atguigu.javase.thread;

import org.junit.Test;

/**
 * 程序(program) : 一段静态代码
 * 进程(process) : 内存中正在执行中的程序(有生命周期)
 * 线程(thread) : 进程中的子任务(有生命周期), 线程直接由CPU调度执行, 效率很高.
 *
 * 多个线程隶属于同一个进程, 可以方便地共享内存, 共享数据
 *
 * 创建并启动线程方式(4种)
 * 1) 实现的方式
 *      1) 写一个具体类, 实现Runnable接口, 并实现run方法, 这个run方法就是线程体
 *      2) 创建这个具体类对象, 再以它为实参创建Thread对象, Thread对象就是线程对象
 *      3) 调用Thread对象的方法start
 *
 * 2) 继承的方式
 *      1) 写一个子类, 继承Thread类, 必须重写父类中的run方法, 因为父类中的run只是调用了它关联的target对象的run,
 *          如果父类并没有关联到target对象, 就会导致 父类的run什么也没有.
 *      2) 创建这个子类对象, 相当于创建了Thread对象
 *      3) 调用对象的start
 *
 * 实现的好处
 * 1) 避免了单继承的局限性
 * 2) 多个线程可以共享同一个接口实现类的对象, 非常适合多个相同线程来处理同一份资源
 *
 * public void start():  启动线程, 并执行对象的run()方法, 激活子栈
 * public void run():  线程在被调度时执行的操作
 * public String getName():  返回线程的名称
 * public void setName(String name):设置该线程名称
 * public static Thread currentThread(): 返回当前线程, 此方法所在的栈的线程, 和调用者对象无关.
```

```
 * public int getPriority() ：返回线程优先值
 * public void setPriority(int newPriority) ：改变线程的优先级，线程创建时
继承父线程的优先级
 * public void stop(): 强制线程生命期结束，绝不调用
 * boolean isAlive(): 返回boolean，判断线程是否还活着
 * public static void sleep(long millis) : 此方法会导致它进入的栈的睡眠 ，
进入睡眠的线程不再抢夺CPU.
 *      解除sleep阻塞，1) 自然醒
 */
class MyThread extends Thread {

    @Override
    public void run() {
        for (int i = 0; i < 500; i++) {
            System.out.println(Thread.currentThread().getName() + " :
" + i);
        }
    }
}

public class ThreadTest {

    public static void main(String[] args) {
        Thread thread = new MyThread();
        thread.setName("子线程");
        thread.start();

        try {
            //thread.join(); // 子线程要继续，主线程会阻塞，直到子线程执行完.
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            thread.sleep(10); // 主线程睡眠 ，永远和调用者对象无关.
        } catch (InterruptedException e) {
```

```
                throw new RuntimeException(e);
            }

            for (int i = 0; i < 500; i++) {
                System.out.println(Thread.currentThread().getName() + " :
" + i);
            }

        }
}



package com.atguigu.javase.thread;

/**
 * 程序 : 静态代码
 * 进程 : 运行中的程序
 * 线程 : 进程中的子任务
 *
 * 创建并启动线程方式
 * 1) 实现的方式
 *      1) 写一个具体类, 实现Runnable接口, 实现run方法
 *      2) 创建具体类对象, 并把它关联给new的Thread对象, 内部属性名为target
 *      3) 调用Thread对象的start.
 *
 * 2) 继承的方式
 *
 * static Thread currentThread() : 返回此方法被压入的栈的线程对象, 和调用者无
关.
 * static void sleep(long millis) : 让此方法被压入的栈的线程进入睡眠状态, 不
再参与cpu抢夺.
 *      解除sleep阻塞 : 1) 自然醒, 2) 被唤醒.
 *
 * name属性描述名称
 * priority 描述优先级
```

```
 *
 * 生命周期
 *                                                                BLOCKED(阻塞)
 *                                          | 时间到,被interrupt()
        | sleep(...)
 *                                          | 拿到同步锁
      | synchronized(等待同步锁)
 *                                          | 被notify()
        | wait() 纯粹的等待
 *                                          | resume()
      | suspend
 *                                          +
       ^
 *  new Thread(...)                                          获取到cpu执行权
 *      NEW(新建) -----------------> RUNNABLE(可运行)--------------------
-> RUNNING(正运行) --------------------> TERMINATED(终止)
 *                     start()                          <------------------
--                     run()结束
 *                                                          丢失了cpu执行权
 * 守护线程 : 为用户线程服务的线程.
 * 如果进程中没有了用户线程, 所有的守护线程都会退出. 必须在线程start()之前把线程
设置为守护线程.
 *
 *
 * 线程同步 : 2个同
 *     多个线程 "同" 时 修改  "同" 一数据
 *
 * */
public class ThreadTest {

    public static void main(String[] args) {
        try {
            Thread.sleep(300);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
    }
}
```

线程同步

```
package com.atguigu.javase.thread;

/**
 * 编写一个线程程序（Counter类），声明实例变量counter，初值为200。在run方法中循环50次，每次对counter做减2操作（用-=实现），睡眠10毫秒，并打印counter值，然后继续下一次循环；
 * 在main方法中创建Counter实例，并用它创建和运行一个线程，观察counter值每次是否减去了2；
 * 用刚才的Counter实例再创建第二个线程，两个线程同时运行，观察counter值每次是否减去了2 。
 */
public class Counter implements Runnable {

    private int counter = 200;
    //private static Object lock = new Object();

    @Override
    //public synchronized void run() { // 这样写最不好了，因为整个方法全锁死!!!
    public void run() {
        //synchronized (this) { // 锁的粒度太大了
            for (int i = 0; i < 50; i++) {
                // 只锁上对于共享数据修改的部分，其他的不要锁.
                // 锁对象选取原则 ：一定要选择全局唯一对象.
                synchronized ("") { // (互斥锁对象) { // 一旦某个线程进入此块中时，其他的所有线程如果也到这里,就进不来了，必须等待
                    counter -= 2;
                    try {
```

```
                    Thread.sleep(10);
                } catch (Exception e) {
                    e.printStackTrace();
                }

 System.out.println(Thread.currentThread().getName() + " : " +
counter);
            } // 块结束后有一个自动隐式释放锁操作
        }
    //}
}

}
```

线程通信

```
package com.atguigu.javase.homework;

public class Withdraw implements Runnable {

    private Account account;

    public Withdraw(Account account) {
        this.account = account;
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            synchronized ("") {
                int money = 1000;
                if (account.getBalance() < money) {
                    // 不够消费, 进入等待.....
```

```
                    System.out.println("要取的钱是[" + money + "], 实际账户 : " + account + " 钱不够, 进入等待");
                    try {
                        "".wait(); // 当前线程进入阻塞, 必需释放锁. 让生产者有机会生产, 此方法必须由锁对象来调用!!!
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                account.setBalance(account.getBalance() - money);
                System.out.println(Thread.currentThread().getName() + " 取钱[" + money + "] 后 : " + account);
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}


package com.atguigu.javase.homework;

//* 写一个普通 类Deposit实现Runnable, 在run方法中存钱
// * 有两个柜台(线程)分别同时向同一个账户存3000元, 每次存1000, 存3次。每次存完打印账户余额。睡眠10毫秒
public class Deposit implements Runnable {

    private Account account;

    public Deposit(Account account) {
        this.account = account;
    }
```

```
    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            synchronized ("") {
                int money = 1000;
                account.setBalance(account.getBalance() + money);
                System.out.println(Thread.currentThread().getName() +
" 存钱[" + money + "]后 : " + account);
                "".notifyAll(); // 通知所有等待的线程，可以结束等待了. 如果
我这里没有释放锁，消费者也不能执行.
            }
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

第17章 反射

```
package com.atguigu.javase.reflect;

import org.junit.Test;
```

```java
import java.io.IOException;
import java.io.InputStream;
import java.io.Serializable;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Properties;

/**
 * Class对象可以描述java中的所有数据类型
 * 数据类型的对象化
 *
 * 数据类型分类 :
 *      1) 基本数据类型 : 保存数据自身
 *          8种
 *          void
 *      2) 引用数据类型 : 保存对象地址
 *          类类型, 用class关键字声明的
 *          接口类型, 用interface关键字
 *          数组
 *          枚举
 *          注解
 */
class Base extends ArrayList {

    private void method1() {
        System.out.println("Base.method1");
    }
}
class Teacher extends Base implements Serializable, Runnable,
Comparable {

    public static void test1() {
```

```
        System.out.println("Teacher.test1");
    }

    private String name;
    private int age;
    private String gender;

    public Teacher() {}

    public Teacher(String name, int age, String gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
```

```
        this.gender = gender;
    }

    @Override
    public String toString() {
        return "Teacher{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                '}';
    }

    public void eat(String some) {
        System.out.println(name + "老师在吃" + some);
    }

    private void lesson(String content, int room, int time) {
        System.out.println(name + "老师在教室[" + room + "]上[" +
content + "]课, 共上了[" + time + "]小时");
        //throw new RuntimeException("无端端的异常");
    }

    @Override
    public int compareTo(Object o) {
        return 0;
    }

    @Override
    public void run() {

    }
}

public class ReflectTest {

    @Test
```

```
    public void test8() throws ClassNotFoundException,
NoSuchMethodException, InstantiationException, IllegalAccessException,
InvocationTargetException {
        Class<?> clazz =
Class.forName("com.atguigu.javase.reflect.Teacher");
        Object obj = clazz.newInstance();

        System.out.println("clazz.getClassLoader() = " +
clazz.getClassLoader());
        Class<?> superclass = clazz.getSuperclass();
        System.out.println("superclass = " + superclass);
        Class<?> superclass2 = superclass.getSuperclass();
        System.out.println("superclass2 = " + superclass2);
        Class<?> superclass3 = superclass2.getSuperclass();
        System.out.println("superclass3 = " + superclass3);

        Class<?>[] interfaces = clazz.getInterfaces();  // 获取到实现的
所有接口.
        for (Class<?> anInterface : interfaces) {
            System.out.println(anInterface);
        }
        System.out.println("********************");
        Field[] declaredFields = clazz.getDeclaredFields(); // 获取所有
本类声明的属性
        for (Field declaredField : declaredFields) {
            System.out.println(declaredField);
        }
        System.out.println("*********所有公共方法************");
        Method[] methods = clazz.getMethods();
        for (Method method : methods) {
            System.out.println(method);
        }
        // 获取父类的私有方法
        Method method1 =
clazz.getSuperclass().getDeclaredMethod("method1");
        System.out.println(method1);
```

```
        method1.setAccessible(true);

        method1.invoke(obj); // 子类对象调用父类私有方法
    }

    @Test
    public void test7() {
        try {
            Class clazz =
Class.forName("com.atguigu.javase.reflect.Teacher");
            //Object obj = clazz.newInstance();
            //public Teacher(String name, int age, String gender) {
            // 反射式通过全参构造器创建对象. 向类模板要全参构造器
            // 必须要提供我们要获取的构造器方法的形参的数据类型的列表. 若干个
Class对象, 分别是形参类型的Class对象
            Constructor constructor1 =
clazz.getConstructor(String.class, int.class, String.class);
            //Teacher obj = new Teacher("大海", 28, "男");
            // 使用时必须再提供方法调用需要的实参值列表
            Object obj1 = constructor1.newInstance("大海", 28, "男");
            System.out.println(obj1);

            Constructor constructor2 = clazz.getConstructor(); // 获取
无参构造器
            Object obj2 = constructor2.newInstance();
            System.out.println(obj2);

            //public void lesson(String content, int room, int time) {
            //((Teacher)obj1).lesson("mysql", 302, 4);
            // 反射式调用方法. 先从类模板中获取到方法
            // 提供方法名, 和方法的形参的数据类型列表
            // getMethod可以获取到本类及从父类继承的任意的public方法.
            //Method lessonMethod = clazz.getMethod("lesson",
String.class, int.class, int.class);
            Method lessonMethod = clazz.getDeclaredMethod("lesson",
String.class, int.class, int.class);
```

```
            System.out.println(lessonMethod);
            lessonMethod.setAccessible(true); // 暴力反射

            // 调用方法时，必须提供此方法的调用者对象this和 方法实际执行时需要的实参值列表，实参和形参也是完全匹配.
            Object retValue = lessonMethod.invoke(obj1, "mysql", 302, (short)2);
            System.out.println("retValue = " + retValue);

            Method hashCode = clazz.getMethod("hashCode");
            System.out.println("hashCode.invoke(obj1) = " + hashCode.invoke(obj1));

            Method test1 = clazz.getMethod("test1");
            // 静态方法调用时不需要this对象
            test1.invoke(null); // 不会出现空指针异常.
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) { // 访问了不能访问的成员.
            e.printStackTrace();
        } catch (NoSuchMethodException e) { // 没有找到方法，方法名错误，参数错误!!
            e.printStackTrace();
        } catch (InvocationTargetException e) { // 调用的目标方法中出现异常时
            e.printStackTrace();
        }
    }

    @Test
    public void test6() throws ClassNotFoundException, IOException {
        // 类加载器有3个层次
        // 系统类加载器(应用程序类加载器) 加载我们自己写的类
        ClassLoader cl1 = ClassLoader.getSystemClassLoader();
```

```
        System.out.println(cl1);
        // 扩展类加载器, 加载jre/lib/ext目录下面的.jar
        ClassLoader cl2 = cl1.getParent();
        System.out.println(cl2);
        // 引导类加载器 加载jre中最核心的jar文件
        ClassLoader cl3 = cl2.getParent();
        System.out.println(cl3);

        // 双亲委派机制.
        // 加载类时, 系统类加载器要先委派父类加载器(Ext)
        // 扩展类加载器继续向上委派给引导类加载器, 这个类是否是核心类, 如果不是,
驳回委派
        // 扩展类加载器检查也不是我需要加载的. 驳回, 系统类加载器才有资格加载类.
        Class clazz1 =
cl1.loadClass("com.atguigu.javase.reflect.Teacher");
        System.out.println("clazz1.getClassLoader() = " +
clazz1.getClassLoader());

        Class clazz2 = cl1.loadClass("java.util.HashMap");
        System.out.println("clazz2.getClassLoader() = " +
clazz2.getClassLoader());

        // 类加载器可以在src目录下及其他classpath路径下, 比FileInputStream更
好用!!!
        //InputStream inputStream =
cl1.getResourceAsStream("test.properties");
        InputStream inputStream =
cl1.getResourceAsStream("com/sun/corba/se/impl/logging/LogStrings.prop
erties");
        Properties properties = new Properties();
        properties.load(inputStream);
        System.out.println(properties);
    }

    @Test
```

```
    public void test4() throws ClassNotFoundException,
InstantiationException, IllegalAccessException {
        // 获取类模板对象的4种方法
        // 1) 已知类，通过类的静态属性.class获取，性能最好，效率最高，也最安
全，是硬编码
        Class clazz1 = Teacher.class;

        // 2) 已经得到了对象，再通过调用对象的getClass()来获取它对应的类模板对
象
        Object obj = new Teacher("张三", 30, "男");
        Class clazz2 = obj.getClass();
        System.out.println(clazz1 == clazz2);

        // 3) 已知类的全限定名称，通过Class的工厂方法forName获取到
        Class clazz3 =
Class.forName("com.atguigu.javase.reflect.Teacher");

        System.out.println(clazz1.hashCode());
        System.out.println(clazz3.hashCode());
        System.out.println(clazz1 == clazz3);

        // 4) 通过类加载器对象，调用方法loadClass("类的全限定名称");
        // 获取当前类的类加载器对象
        ClassLoader classLoader1 = this.getClass().getClassLoader();
        ClassLoader classLoader2 = ReflectTest.class.getClassLoader();
        System.out.println(classLoader1 == classLoader2);
        Class clazz4 =
classLoader1.loadClass("com.atguigu.javase.reflect.Teacher");

        System.out.println(clazz3 == clazz4);
    }

    public void testClass(Class clazz) {
        System.out.println(clazz);
        System.out.println("clazz.isPrimitive() = " +
clazz.isPrimitive());
```

```
        System.out.println("clazz.isEnum() = " + clazz.isEnum());
        System.out.println("clazz.isArray() = " + clazz.isArray());
        System.out.println("clazz.isInterface() = " +
clazz.isInterface());
        System.out.println("clazz.isAnnotation() = " +
clazz.isAnnotation());

        System.out.println("**********************");
    }

    @Test
    public void test5() {
        Class clazz1 = Integer.class;
        Class clazz2 = int.class;
        System.out.println(clazz2 == clazz1);
        Class clazz3 = double.class;
        System.out.println(clazz3.isPrimitive());
        Class clazz4 = int[].class;
        Class clazz5 = int[][].class;
        Class clazz6 = void.class;
        Class clazz7 = Void.class;
        Class clazz8 = Thread.State.class; // 枚举
        Class clazz9 = Runnable.class; // 接口
        Class clazz10 = Override.class; // 注解使用@interface关键字声明.

        testClass(clazz1);
        testClass(clazz2);
        testClass(clazz3);
        testClass(clazz4);
        testClass(clazz5);
        testClass(clazz6);
        testClass(clazz7);
        testClass(clazz8);
        testClass(clazz9);
        testClass(clazz10);
```

```
    }

    @Test
    public void test3() {
        try {
            Class clazz =
Class.forName("com.atguigu.javase.reflect.Teacher");// 必须提供全限定类名
的字符串
            Object obj = clazz.newInstance();
            System.out.println(obj);

            // getField可以获取到本类的和从父类继承的所有公共属性
            //Field ageField = clazz.getField("age");
            // getDeclaredField可以获取本类中声明的任意属性
            Field ageField = clazz.getDeclaredField("age");
            System.out.println(ageField); // private int
com.atguigu.javase.reflect.Teacher.age
            ageField.setAccessible(true); // 强制让这个属性可以被访问，暴力
反射. 不推荐使用
            ageField.set(obj, 40);
            System.out.println(ageField.get(obj));

            Field genderField = clazz.getDeclaredField("gender");
            genderField.setAccessible(true);
            genderField.set(obj, "男");
            System.out.println(genderField.get(obj));

            Field nameField = clazz.getDeclaredField("name");
            nameField.setAccessible(true);
            nameField.set(obj, "佟刚"); // t1.name = "佟刚"
            System.out.println(nameField.get(obj));

            System.out.println(obj);
        } catch (ClassNotFoundException e) { // 类真的没有找到，类名写错
了.
            e.printStackTrace();
```

```
        } catch (InstantiationException e) { // 创建对象时出问题了，没有相应构造器.
            e.printStackTrace();
        } catch (IllegalAccessException e) { // 访问了不能访问的成员时会出的异常
            e.printStackTrace();
        } catch (NoSuchFieldException e) { // 在类中并没有指定的属性，或属性名写错了.
            e.printStackTrace();
        }
    }

    @Test
    public void test2() {
        // 反射是软编码，对于类是完全不依赖.
        try {
            // 手工干预类的加载
            Class clazz = Class.forName("com.atguigu.javase.reflect.Teacher");// 必须提供全限定类名的字符串
            Object obj = clazz.newInstance(); // 创建类模板对象所代表的类的对象，默认使用无参构造器.
            System.out.println(obj);
            //通过类模板获取到属性的定义对象
            Field ageField = clazz.getField("age");// 根据 属性名 获取到相应的属性定义对象
            //obj.age = 40;
            ageField.set(obj, 40); // 相当于 obj.age = 40; 设置obj对象的age属性值为40
            //System.out.println(obj.age);

 System.out.println(ageField.get(obj));//System.out.println(obj.age); // 获取obj对象的age属性值

            //obj.gender = "男";
            Field genderField = clazz.getField("gender");
```

```
            genderField.set(obj, "男");
            System.out.println(genderField.get(obj));

            Field nameField = clazz.getField("name");
            nameField.set(obj, "佟刚"); // t1.name = "佟刚"
            System.out.println(nameField.get(obj));

            System.out.println(obj);
        } catch (ClassNotFoundException e) { // 类真的没有找到，类名写错了.
            e.printStackTrace();
        } catch (InstantiationException e) { // 创建对象时出问题了，没有相应构造器.
            e.printStackTrace();
        } catch (IllegalAccessException e) { // 访问了不能访问的成员时会出的异常
            e.printStackTrace();
        } catch (NoSuchFieldException e) { // 在类中并没有指定的属性，或属性名写错了.
            e.printStackTrace();
        }
    }

    @Test
    public void test1() {
        // 硬编码，对于类是强依赖

        //Teacher t1 = new Teacher();
        //System.out.println(t1);

//        t1.name = "佟刚";
//        t1.age = 40;
//        t1.gender = "男";
//        System.out.println(t1.name);
//        System.out.println(t1.age);
//        System.out.println(t1.gender);
```

```
        //System.out.println(t1);
        //t1.lesson();


        Teacher t2 = new Teacher("许姐", 30, "女");
        System.out.println(t2);

    }
}
```

第18章 网络

```
package com.atguigu.javase.net;

import org.junit.Test;

import java.io.*;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.UnknownHostException;

/**
 * IP : 用于在网络中定位某个特定的主机, 像电话号码
 * 端口号 : 用于标识主机上的某个应用程序. 使用命令netstat -a查看
 *
 * IP + 端口 => 套接字 socket
 *
 * 通信协议 :
 *      TCP : 传输控制协议 : 稳定可靠 建立通道
 *          服务器端Server : 被动地等待客户端的连接.
```

```
 *          客户端Client : 主动发起连接请求.
 *
 *      UDP : 用户数据报协议 : 高效但是容易丢数据
 *
 */
public class NetTest {

    @Test
    public void server2() {
        // 被动等待
        ServerSocket server = null;
        Socket socket1 = null;
        BufferedReader br = null;
        try {
            server = new ServerSocket(9999); // 在本机注册并绑定端口9999.
            socket1 = server.accept();// 此方法会造成阻塞，要等待，只要有客户端真的连接了此服务器，此方法就不再阻塞
            System.out.println(socket1);
            // 服务器接收字符串
            InputStream is = socket1.getInputStream();
            InputStreamReader isr = new InputStreamReader(is);
            br = new BufferedReader(isr);
            String line = br.readLine();
            System.out.println(line);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (socket1 != null) {
                try {
```

```
                socket1.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        if (server != null) {
            try {
                server.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

}

@Test
public void client2() {
    // 主动连接服务器
    Socket socket2 = null;
    BufferedWriter bw = null;
    try {
        socket2 = new Socket("127.0.0.1", 9999);
        System.out.println(socket2);

        // 让客户端给服务器发送一个字符串
        OutputStream os = socket2.getOutputStream();
        OutputStreamWriter osw = new OutputStreamWriter(os);
        bw = new BufferedWriter(osw);

        bw.write("你好服务器, 我是客户端, 今天天气不错!!!!");
        bw.newLine();
        bw.flush();
    } catch (Exception e) {
        e.printStackTrace();
```

```
        } finally {
            // 释放资源的顺序是逆序
            if (bw != null) {
                try {
                    bw.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (socket2 != null) {
                try {
                    socket2.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

    }

    @Test
    public void server() throws IOException {
        // 被动等待
        ServerSocket server = new ServerSocket(9999); // 在本机注册并绑定端口9999.
        Socket socket1 = server.accept();// 此方法会造成阻塞，要等待，只要有客户端真的连接了此服务器，此方法就不再阻塞
        System.out.println(socket1);
        // 服务器接收字符串
        InputStream is = socket1.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String line = br.readLine();
        System.out.println(line);
        br.close();
```

```
        socket1.close();
        server.close();
    }
    @Test
    public void client() throws IOException {
        // 主动连接服务器
        Socket socket2 = new Socket("127.0.0.1", 9999);
        System.out.println(socket2);
        // 让客户端给服务器发送一个字符串
        OutputStream os = socket2.getOutputStream();
        OutputStreamWriter osw = new OutputStreamWriter(os);
        BufferedWriter bw = new BufferedWriter(osw);
        bw.write("你好服务器，我是客户端，今天天气不错!!!!");
        bw.newLine();
        bw.flush();
        // 释放资源的顺序是逆序
        bw.close();
        socket2.close();
    }

    @Test
    public void test1() throws UnknownHostException {
        InetAddress byName = InetAddress.getByName("192.168.36.48");
        System.out.println(byName);
        //System.out.println("byName.getHostName() = " +
byName.getHostName());

        InetAddress byName1 = InetAddress.getByName("atguigu.com");
        System.out.println(byName1);
    }
}
```

服务器

```java
package com.atguigu.javase.homework;

import org.junit.Test;

import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.time.LocalDateTime;

public class NetTest {

    @Test
    public void server() {
        ServerSocket server = null;
        try {
            server = new ServerSocket(7777);
        } catch (IOException e) {
            e.printStackTrace();
        }
        boolean flag = true;
        while (flag) {
            System.out.println("服务器在端口7777监听中......");
            final Socket socket1;
            try {
                socket1 = server.accept(); // 服务器端的套接字
                // 每个客户端都使用一个单独的线程处理
                Runnable runner = new Runnable() {
                    @Override
                    public void run() {
                        BufferedWriter bw = null;
                        try {
                            // 服务器发送字符串给客户端
                            bw = new BufferedWriter(new
OutputStreamWriter(socket1.getOutputStream()));
```

```
                            bw.write("你好，客户端，我是服务器，现在时间 :
" + LocalDateTime.now());
                            bw.newLine();
                            bw.flush();
                            Thread.sleep(1 * 1000);
                        } catch (Exception e) {
                            e.printStackTrace();
                        } finally {
                            if (bw != null) {
                                try {
                                    bw.close();
                                } catch (IOException e) {
                                    e.printStackTrace();
                                }
                            }
                            if (socket1 != null) {
                                try {
                                    socket1.close();
                                } catch (IOException e) {
                                    e.printStackTrace();
                                }
                            }
                        }
                    }
                };
                new Thread(runner).start();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    @Test
    public void client() {
        Socket socket2 = null;
        BufferedReader br = null;
```

```
        try {
            socket2 = new Socket("127.0.0.1", 7777); // 客户端套接字
            // 客户端接收字符串
            br = new BufferedReader(new
InputStreamReader(socket2.getInputStream()));

            String line = br.readLine();
            System.out.println(line);

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }

            }
            if (socket2 != null) {
                try {
                    socket2.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

    }
}
```

第19章 Java8

lambda表达式

```
package com.atguigu.javase.java8;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

/**
 * lambda主要用于替代匿名内部类对象的写法的
 * 依据左面来推断右面的东西
 *
 * 省略了 new 接口() {}
 * 也省略了方法的返回值,修饰符及方法名.
 *       参数列表(输入) -> 方法体(输出);
 *
 * 无参无返回 : 无输入无输出
 * 有参无返回 : 有输入无输出
 * 无参有返回 : 无输入有输出
 * 有参有返回 : 有输入有输出
 *
 * lambda只适用于只有一个抽象方法的接口.
 * 这样的接口称为函数式接口, 使用注解 @FunctionalInterface
 *
 * 最终目标就是让我们关注在函数的本质上
 *       输入 -> 输出
 *
 */

@FunctionalInterface
interface ITest {
```

```
    int test(String str);
    default void test2() {}
}

//@FunctionalInterface
interface ITest2 {
    void test1();
    void test3();
}

public class LambdaTest {

    @Test
    public void test3() {
        Comparator<Integer> comparator1 = new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o1 - o2;
            }
        };
        int n1 = comparator1.compare(30, 50);
        System.out.println("n1 = " + n1);

        //Comparator<String> comparator2 = (String s1, String s2) ->
{return s1.length() - s2.length();};
        Comparator<String> comparator2 = (s1, s2) -> s1.length() -
s2.length(); // 面向函数编程思想.
        int n2 = comparator2.compare("abcd", "yyyy");
        System.out.println("n2 = " + n2);
    }

    @Test
    public void test2() {
        ITest iTest1 = new ITest() {
            @Override
            public int test(String str) {
```

```
                return Integer.parseInt(str);
            }

            @Override
            public void test2() {

            }
        };
        int v1 = iTest1.test("999");
        System.out.println("v1 = " + v1);

        // lambda只保留了参数列表和方法体
        // 如果有return语句，也省略它
        // 因为在接口中，方法的参数类型也是明确的，所以也可以推断，也可省略参数类
型
        // 只有一个参数时，()省略
        ITest iTest2 = str -> Integer.parseInt(str);
        int v2 = iTest2.test("3242");
        System.out.println("v2 = " + v2);
    }

    @Test
    public void test1() {
        // 推断机制，根据一侧来断定 另一侧的东西
        List<String> list = new ArrayList<>();
        //var a = 200;
        //var b = "abc";
        Runnable runner1 = new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName());
            }
        };
        new Thread(runner1).start();

        // lambda表达式只保留下2个东西，方法的参数列表 -> 方法体;
```

```
        //Runnable runner2 = () ->
{System.out.println(Thread.currentThread().getName());};
        // 如果lambda体中只有一个语句时，可以省略方法体的一对{}
        Runnable runner2 = () ->
System.out.println(Thread.currentThread().getName());
        new Thread(runner2).start();
    }
}
```

函数式接口

```
package com.atguigu.javase.java8;

import com.atguigu.javase.javabean.Student;
import org.junit.Test;

import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.Supplier;

/**
 * Consumer<T> :
 *      消费器，作用是消费一个T类型的对象，并没有任何结果
 *      void accept(T t) : 有参无返回，有输入没有输出
 *
 * Supplier<T>
 *      供给器，作用是供给一个T类型的对象，并不需要输入数据
 *      T get() : 无参有返回，无输入有输出
 *
   Function<T, R>
        转换器，作用是输入一个T类型的对象，并转换为一个R类型的对象返回
        R apply(T t) : 有参有返回，有输入有输出
```

```
 * Predicate<T>
 *      判定器，作用是对输入的T类型的对象进行某种判定，如果满足判定返回true，不满足返回false
 *      boolean test(T t) : 有参有固定返回，有输入有固定输出(真或假)
 *
 * BiFunction<T, U, R>
 *      双转换器，把输入的2个对象(T类型和U类型) 转换成一个对象(R类型)
 *      R apply(T t, U u)
 *
 * UnaryOperator<T>
 *      一元运算符
 *      T apply(T t) : 有输入有输出
 * BinaryOperator<T>
 *      T apply(T t1, T t2) : 有2个输入有输出
 *
 * 方法引用 : lambda体中的调用的别的方法正好是接口描述的方法模式时，就可以使用方法引用.
 *
 */
public class FunctionalInterfaceTest {

    // 练习 : 写一个判定器，判定某个Student是否及格.
    @Test
    public void exer4() {
        Student s1 = new Student(9, "小伟", 5, 30);
        Predicate<Student> predicate1 = new Predicate<Student>() {
            @Override
            public boolean test(Student student) {
                return student.getScore() > 59;
            }
        };
        boolean b1 = predicate1.test(s1);
        System.out.println("b1 = " + b1);

        Supplier<Student> supplier = () -> new Student(11, "小芳", 2, 90);
```

```
        Predicate<Student> predicate2 = t -> t.getScore() >= 60;
        boolean b2 = predicate2.test(supplier.get());
        System.out.println("b2 = " + b2);
    }

    @Test
    public void test4() {
        Predicate<String> predicate1 = new Predicate<String>() {
            @Override
            public boolean test(String s) {
                return s.endsWith("aaa");
            }
        };
        boolean b1 = predicate1.test("lakjsdflkajsaaa");
        System.out.println("b1 = " + b1);

        Predicate<Integer> predicate2 = t -> t % 2 == 0;
        boolean b2 = predicate2.test(99);
        System.out.println("b2 = " + b2);
    }

    // 练习 : 写转换器, 把Student对象转换成Double对象
    @Test
    public void exer3() {
        Function<Student, Double> function1 = new Function<Student,
Double>() {
            @Override
            public Double apply(Student student) {
                return student.getScore();
            }
        };
        Student s1 = new Student(9, "小花", 5, 100);
        Double apply1 = function1.apply(s1);
        System.out.println("apply1 = " + apply1);

        Function<Student, Double> function2 = t -> t.getScore();
```

```
        Supplier<Student> supplier = () -> new Student(10, "小李", 2,
99);
        Student s2 = supplier.get();
        Double apply2 = function2.apply(s2);
        System.out.println("apply2 = " + apply2);
    }

    @Test
    public void test3() {
        Function<Double, Integer> function1 = new Function<Double,
Integer>() {
            @Override
            public Integer apply(Double aDouble) {
                return (int)Math.round(aDouble);
            }
        };
        Integer apply1 = function1.apply(324.90999);
        System.out.println("apply1 = " + apply1);

        //Function<String, Float> function2 = t -> Float.valueOf(t);
        Function<String, Float> function2 = Float::valueOf;
        Float apply2 = function2.apply("234234.234234");
        System.out.println("apply2 = " + apply2);
    }

    // 练习 : 写一个供给器, 让它供给一个Student对象.
    @Test
    public void exer2() {
        Supplier<Student> supplier1 = new Supplier<Student>() {
            @Override
            public Student get() {
                return new Student();
            }
        };
        Student s1 = supplier1.get();
        System.out.println(s1);
```

```
        //Supplier<Student> supplier2 = () -> new Student();
        Supplier<Student> supplier2 = Student::new; // 构造器引用

        Student s2 = supplier2.get();
        Consumer<Student> consumer = t -> System.out.println(s2);
        consumer.accept(s2);
    }

    @Test
    public void test2() {
        Supplier<String> supplier1 = new Supplier<String>() {
            @Override
            public String get() {
                return "abc";
            }
        };
        String s1 = supplier1.get();
        System.out.println("s1 = " + s1);

        //Supplier<Double> supplier2 = () -> Math.random();
        Supplier<Double> supplier2 = Math::random;
        Double v1 = supplier2.get();
        System.out.println("v1 = " + v1);
    }

    // 练习，写消费器，消费一个Student对象.
    @Test
    public void exer1() {
        Student s = new Student(9, "小刚", 3, 90);

        Consumer<Student> consumer1 = new Consumer<Student>() {
            @Override
            public void accept(Student student) {
                System.out.println(student);
            }
```

```
        };
        consumer1.accept(s);

        Consumer<Student> consumer2 = t -> System.out.println(t);
        consumer2.accept(s);
    }


    @Test
    public void test1() {
        // 一个消费字符串对象的消费器
        Consumer<String> consumer1 = new Consumer<String>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }
        };
        consumer1.accept("lkjasdflkjasd");
        // 一个消费Double对象的消费器
        //Consumer<Double> consumer2 = t -> System.out.println(t);
        Consumer<Double> consumer2 = System.out::println;
        consumer2.accept(98324.234234);

    }
}
```

Stream流

```
package com.atguigu.javase.java8;

import com.atguigu.javase.javabean.Student;
import com.atguigu.javase.javabean.StudentData;
import org.junit.Test;
```

```java
import java.util.Arrays;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.Stream;

/**
 * Stream ： 是一个抽象概念，就是一个流，流中全是对象
 *
 * Stream关注是流中的对象的处理，而不关注对象如何存储
 * 集合关注的是数据如何存储，Stream关注的是数据的计算和处理.
 *
 * 1）Stream不是集合，对于数据的处理并不影响数据源. 相反，会得到一个持有新结果的新的Stream
 * 2）Stream只能"消费"一次，用完就废，要想继续处理，使用它产生新的Stream继续处理
 * 3）Stream的所有中间操作都是延迟执行的，只有执行了终止操作后，所有中间操作才真的执行(一系列中间操作称为流水线)
 *      终止操作是必须的，中间操作是可选的.
 * 4）象一个高级迭代器，单向，不可往复. 只用一次.
 *
 * 流的操作的3个步骤 ：
 *      1）创建流
 *          1）基于集合
 *              集合对象.stream();
 *          2）基于数组
 *              Arrays.stream(数组对象);
 *          3）使用工厂方法
 *              Stream.of(...) ： 散数据
 *              Stream.generate(供给器) ： 通过不断地使用供给器来形成Stream，会产生无限流
 *              Stream.iterate(种子，一元运算);
 *
 *      2）中间操作(可以有多个，形成流水线）可选的
```

```
 *          filter(判定器) : 把流中的所有对象都经过判定器判定, 结果为true的留下并放入新流, 结果为false的丢弃
 *          distinct() : 去重, 认定对象重复的标准和hashset一样.
 *          limit(long maxSize) : 截断流, 使其元素不超过参数个. 通常用于处理无限流
 *          skip(long n) : 跳过前n个元素
 *          map(转换器) : 把流中所有A类型对象通过转换器处理,得到的都是B类型对象, 新流中放的都是B类型对象
 *          sorted() : 自然排序
 *          sorted(比较器) : 定制排序
 *
 *      3) 终止操作(必须有的, 只有一个)
 *          forEach(消费器) : 把流中的所有对象都扔给消费器消费.
 *          allMatch(判定器) : 必须所有的都满足判定器, 才是true
 *          anyMatch(判定器) :
 *          noneMath(判定器)
 *          findFirst() : 返回第1个
 *          findAny() : 返回任意个
 *          reduce(二元运算符) : 前2个处理成一个新结果, 再让它和第3个处理出新结果, 再继续和后面的所有元素都两两处理, 最终有一个结果
 *          collect(采集器) : 可以通过工具类的工厂方法得到现成的采集器 Collectors.
 *
 *  Optional是一个指针的容器, 因为在java中太容易出空指针异常. 最大化减少空指针异常
 *      get() 有可能还是会有异常
 *      orElse(非空指针) : 永远不会得到空指针 .
 */
public class StreamTest {

    // 练习 : 找出3年级以上的同学中最牛的同学. 使用归约
    @Test
    public void test14() {
        List<Student> list = StudentData.getList();
        List<Student> list1 = list.stream().distinct().filter(t -> t.getGrade() == 3).collect(Collectors.toList());
```

```
        list1.forEach(System.out::println);
    }
    @Test
    public void test13() {
        List<Student> list = StudentData.getList();
        // 求全校总分
        Optional<Double> reduce = list.stream().distinct().map(t ->
t.getScore()).reduce((t1, t2) -> t1 + t2);
        Double v = reduce.orElse(-1.0);
        System.out.println("v = " + v);
        // 求平均分
        long count = list.stream().distinct().count();
        double avg = v / count;
        System.out.println("avg = " + avg);
    }
    @Test
    public void test12() {
        List<Student> list = StudentData.getList();
        Optional<Student> reduce =
list.stream().distinct().reduce((t1, t2) -> t1.getScore() <
t2.getScore() ? t1 : t2);
        Student student = reduce.orElse(new Student());
        System.out.println(student);
    }

    @Test
    public void test11() {
        List<Student> list = StudentData.getList();
        Optional<Student> any = list.stream().distinct().filter(t ->
t.getScore() > 95).findAny();
        Student student = any.orElse(new Student());
        System.out.println(student);
    }

    // 找出所有3年级没有及格的同学，倒序显示前2位.
    @Test
```

```
    public void exer() {
        StudentData.getList()
                .stream().distinct()
                .filter(t -> t.getGrade() == 3)
                .filter(t -> t.getScore() < 60)
                .sorted((t1, t2) -> -(int)(t1.getScore() * 100 -
t2.getScore() * 100))
                .limit(2).forEach(System.out::println);
    }

    @Test
    public void test10() {
        List<Student> list = StudentData.getList();
        list.stream().distinct().sorted((t1, t2) -> -(t1.getGrade() -
t2.getGrade())).forEach(System.out::println);
    }

    @Test
    public void test9() {
        List<Student> list = StudentData.getList();
        Stream<Integer> stream = list.stream().distinct().map(t ->
t.getGrade());
        stream.forEach(System.out::println);
    }

    @Test
    public void test8() {
        List<Student> list = StudentData.getList();

 list.stream().distinct().skip(10).limit(5).forEach(System.out::printl
n);
    }

    // 练习，找出所有姓张的同学且没有及格的.
    @Test
    public void test7() {
```

```
        List<Student> list = StudentData.getList();
        list.stream().filter(t ->
t.getName().startsWith("张")).filter(t -> t.getScore() <
60).forEach(System.out::println);
    }

    @Test
    public void test6() {
        List<Student> list = StudentData.getList();
        list.stream().filter(t -> t.getGrade() == 2).filter(t ->
t.getScore() > 80).forEach(System.out::println);
    }


    @Test
    public void test5() {
        List<Student> list = StudentData.getList();
        Stream<Student> stream = list.stream();
        Stream<Student> stream2 = stream.filter(t -> t.getGrade() ==
2);
        Stream<Student> stream3 = stream2.filter(t -> t.getScore() >
80);
        stream3.forEach(System.out::println);

    }

    @Test
    public void test4() {
        Stream<Integer> iterate = Stream.iterate(1, t -> t + 2); // 无
限流
        iterate.forEach(System.out::println);
    }
    @Test
    public void test3() {
        Stream<? extends Number> stream = Stream.of(3.2, 5.8, 9.0, 10,
20, 30);
```

```
        stream.forEach(System.out::println);
        Stream<Integer> generate = Stream.generate(() -> 200); // 无限
流
        generate.forEach(System.out::println);
    }

    @Test
    public void test2() {
        Integer[] arr = {9, 2, 4, 1, 0, 8, 99};
        Stream<Integer> stream = Arrays.stream(arr);
        stream.forEach(System.out::println);
    }
    @Test
    public void test1() {
        List<Student> list = StudentData.getList();
        Stream<Student> stream = list.stream();
        stream.forEach(System.out::println);
    }
}
```