Java

Java入门

CMD相关

打开CMD

1. win+R
2. 输入CMD
3. 按下回车键

常见CMD命令

操作指令	执行结果
盘符名称+冒号（E：）	盘符切换
dir	查看目录
cd 目录	进入单级目录
cd ..	回退到上一级目录
cd 目录1\目录2...	进入多级目录
cd \	回退到盘符目录
cls	清屏
exit	退出命令提示符窗口

Path环境变量

为什么要配置Path环境变量?

想要在任意的目录下都可以打开指定的软件，就可以把软件的路径配置到Path环境变量中

配置Path环境变量步骤?

此电脑→右键点击空白处→属性→高级系统设置→环境变量→选中系统变量中的path→点击编辑→新建（粘贴待添加的路径）→点击确定

Java概述和学习方法

JDK的安装目录

目录名称	内容
bin	存放了各种工具命令，比较重要的有javac、java
conf	存放了相关配置文件
include	存放了一些平台特定的头文件
jmods	存放了各种模块
legal	存放了各模块的授权文档
lib	存放了一些工具的补充JAR包

记事本编写代码的方式

1. 记事本编写代码，随后修改后缀名为.java
2. 打开cmd，找到记事本.java文件所在的路径（切换盘符与路径）
3. 在该路径下输入javac xx.java（javac是JDK提供的编译工具，可以通过该工具将.java文件编译成.class文件）
4. 无任意报错的情况下，输入java xx（java是运行工具，会运行当前路径下的xx.class文件，不过运行时候不输入后缀名）

配置Panth环境变量

java自动配置的环境仅仅包含四个工具，如果想要使用其他工具就需要重新配置，步骤如下：

1. 先配置JAVA_HOME。（路径不带bin）
2. 在配置Path。（%JAVA_HOME%\bin）

如果重启之后环境变量失效的话，那么可以：

1. JAVA_HOME不动
2. 在Path当中不要引用JAVA_HOME，直接写完整路径

Java版本

- Java SE：属于标准版，用于桌面应用的开发
- Java ME：Java语言的小型版，用于嵌入式电子设备或者小型移动设备
- **Java EE：Java语言的企业版，用于wed方向的网站开发**

Java的跨平台原理

- Java语言的跨平台是用过虚拟机实现的
- Java说并不是直接运行在操作系统里面的，而是运行在虚拟机中
- 针对不同的操作系统，安装不同的虚拟机就可以了

JRE和JDK

- JDK：
  - JVM：java程序运行的地方
  - 核心类库：java已经具有的库
  - 开发工具：javac编译工具、java运行工具、jdb调试工具、jhat内存分析工具……
- JRE：
  - JRE是java运行环境
  - JVM、核心类库、运行工具

- JDK包含JRE、JRE包含JVM

Java基础概念

基础概念

注释

- 单行注释：//
- 多行注释：/* */
- 文档注释：/** */

```
/**
文档注释的示例
*/
public class HelloWorld{
    //叫做main方法，表示程序的主入口
    public static void main(String[] args){
        /*叫做输出语句（打印语句）
        会把小括号里面的内容进行输出打印*/
        System.out.println("你好！");
    }
}
```

关键字

- 被java赋予了特定涵义的英文单词叫做关键字
    - class（关键字之一）：用于创建/定义一个类，class后跟随类名

字面量

字面量类型	说明	举例
整数类型	不带小数点的数字	-88、666
小数类型	带小数点的数字	12.3、-6.23
字符串类型	用双引号括起来的内容	“你好”
字符类型	用单引号括起来的内容，只能有一个字符	‘0’
布尔类型	布尔值，表示真假	true、false
空类型	一个特殊的值、空值	null

特殊字符

- 例如：\t，制表符，用来在打印的时候将前面字符串的长度补到8或8的整数倍

变量

- 当某个数据经常发生改变时，就可以用变量存储，当数据变量时，只要修改变量里面记录的值即可
- 变量的定义格式：
    - 数据类型 变量名 = 数据值
- Java使用前变量要赋初值

计算机的存储规则

- 不同进制在代码中的表现形式
  - 二进制：代码中以0b开头
  - 十进制：前面不加任何前缀
  - 八进制：代码中以0开头
  - 十六进制：代码中以0x开头
- 计算机存储规则小结
  - 文本数据：数字转二进制、字母和汉字查询码表
  - 图片数据：通过每一个像素点中的RGB三原色来存储
  - 声音数据：对声音的波形图进行采样再存储

数据类型

- 基本数据类型
  - 整数
    - byte（-128~127）
    - short
    - int
    - **long**（定义long类型的数据，**添加L做后缀**，否则会报错）
  - 浮点数
    - **float**（定义float数据时，**添加F做后缀**，否则会报错）
    - double
  - 字符：char（特殊的是，与C相比，**Java中的char占用2字节内存**，从0~65535，这样可以直接用char输出宽字符）
  - 布尔：boolean
- 引用数据类型（目前不涉及）

标识符

- 由数字、字母、下划线_和美元符$组成、不能以数字开头、不能是关键字、区分大小写
- 标识符建议
  - 小驼峰命名法：方法、变量
    - 标识符是一个单词的时候全部小写
    - 标识符由多个单词组成的时候，第一个单词首字母小写，其他单词首字母大写
  - 大驼峰命名法：类名
    - 标识符不管几个单词组成，每个单词首字母都大写

键盘录入

- Java中有一个类叫Scanner，这个类可以接收键盘输入的数字
- 使用步骤如下：
  1. 导包：`import  java.util.Scanner;`
  2. 创建对象：`Scanner sc = new Scanner(System.in);`
     - 上述中只有sc属于变量名可以变
  3. 接收数据：`int i = sc.nextInt();`
     - 上述中只有i是变量名可以变

IDEA

IDEA项目结构介绍

- 项目
  - 模块
    - 包
      - 类

运算符

运算符基础概念

运算符与表达式

- 运算符：对字面量或者变量进行操作的符号
- 表达式：用运算符把字面量或者变量连接起来符合java语法的式子就称为表达式，不同运算符连接的表达式体现的是不同类型的表达式

算术运算符

- 算数运算符
  - ：+ - * / %；（与c类似，/也是整数与整数结果只能是整数）
- 算数运算符的高级用法
  - 取值范围：byte<short<int<long<float<double
  - 当数据类型不一样时，不能进行计算，需要转成一样的才可以进行计算（隐式转换）
  - 取值范围小的和取值范围大的进行计算时，会将小的还提升为大的，再计算（隐式转换）
  - byte、short、char都会先转成int再计算，即使只有byte与byte，执行结果都会是int，除非强制转换过
- 强制转换
  - 目标数据类型 变量名 = (目标数据类型)被强转的数据;
- 字符串相加
  - 当"+"这个操作中出现字符串时，这个“+”是字符串连接符，而不是算术运算符了，**会将前后的数据直接进行拼接**，并产生一个新的字符串
    - 注意：如“123”+123则执行的结果是“123123”
  - 当出现连续“+”操作时，则从左到右逐个执行
    - 例如：1+99+“年”则执行结果是“100年”
  - 字符+字符或者字符+数字时，其中的字符都会根据ASCII码表查询对应的数字再进行计算
  - 只要有字符串出现那就是拼接！！！

自增自减运算符

- ++与--，与c一样

赋值运算符

- 与c一样

关系运算符

- 与c一样，不同之处在于其返回的结果时布尔类型的数据，true或false，其比较结果可以用布尔类型的变量接收

逻辑运算符

符号	作用	说明
&	按位与	同真才真，一假为假
|	按位或	同假才假，一真为真
^	按位异或	相同则真，不同则假
!	非	为真则假，为假则真

短路逻辑运算符

符号	作用	说明
&&	短路与	同真才真，一假为假
||	短路或	同假才假，一真为真

- 这就与C里面的一样，逻辑或运算中只要检测到一个真就不会执行后面的语句，同理，逻辑与中只要有一个为假则就跳过后面的语句，可以提升效率

三元运算符

- 与C里面的条件运算符相同，需要注意的是，其返回结果一定要使用，不管是打印还是用值接收，否则会报错，如下：

```
int number1 = 10;
int number2 = 20;
number1 > number2 ? number1 : number2;
```

- 正确格式应该如下：

```
int max = number1 > number2 ? number1 : number2;
```

```
System.out.println(number1 > number2 ? number1 : number2);
```

- 三元运算符也可以嵌套

优先级

- 与c类似，还是：非算关与或短条赋（没有逗，短表示短路逻辑）
- 但是新增了几个运算符：>>>、instanceof、普通逻辑运算优先级高与短路逻辑运算

原码反码补码

- 原码
  - 十进制数据的二进制表现形式，最左边是符号位，0为正，1为负。利用原码对正数进行计算是不会有问题的。
    但是如果是负数计算，结果就出错，实际运算的结果，跟我们预期的结果是相反的
- 反码：
  - 正数的反码不变，负数的反码是**符号位不变**，其余位全部取反。但是反码存在的问题是存在+0与-0两个零，所以当结果从负数加到正数时，会经过两个零，这就导致会得到的结果差了1
- 补码：
  - 正数的补码不变，负数的补码是负数的反码加1，这就可以解决反码差1的问题

- 计算机中比较特殊的数字是-128，因为在反码中存在两个0，补码中负数位往前统一补了一位，所以原码为-127的位置其补码也前进一位，所以对应的补码就空余了一位，即补码为1000_0000的位置，因为该位不存在原码与反码，所以计算机特殊定义补码为1000_0000的数字的值是-128
- 计算机的存储和计算都是以补码的形式进行的

其他的运算符

运算符	含义	运算规则
<<	左移	向左移动，低位补0
>>	右移	向右移动，高位补0或1
>>>	无符号右移	向右移动，高位补0

流程控制语句

if语句

if语句格式

- 与c一样
  - if
  - if...else...
  - if...else if...else

switch语句

switch语句格式

- 与c一样
  - **但是存在一些不同**，表达式中将要匹配的值可以是byte、short、int、char、枚举（JDK5新增）、String（而c中只能使用整型，JDK8中新增）
  - case后面跟的值只能是字面量，不能是变量
- JDK12的新特性
  -
  - 利用箭头+大括号可以省略break，需要注意的是，如果只有一行的话则可以去掉大括号

```
switch(number)
{
    case 1-> {
        System.out.println(1);
    }
    case 2-> {
        System.out.println(2);
    }
    default -> {
        System.out.println("other");
    }
}
```

  - JDK中还支持一个case中有多个判断，不论是：还是->都支持（**但是不清楚这是那一版本的新特性**）

```
case 1,2,3,4:
{
    System.out.println(1);
    break;
}
```

- 如果是对于范围性的判断，一般使用if的第三种格式，如果是对于有限个结果，则一般使用switch

循环结构

for循环

- 可以在循环初使化语句中申请变量，但是需要注意的是，在初始化语句中申请的变量只能在循环体中使用，相当于局部变量

```
for(int i=1; i<=5; i++) //这儿是没有问题的
{
    System.out.println(i);
}
System.out.println(i);  //这儿会报错
```

- 输入for循环的快捷方式：x.fori（x是可以自己定义的值）或者fori也可以

while循环

- for循环与while循环本质上没有什么区别，最大的区别在于如果for循环是在循环初使化语句中申请变量的，那么该循环变量属于循环体中的局部变量，但是while循环只能在循环体外申请，其范围比循环体要大

do...while循环

- 与c一样

跳转控制语句

- continue与break也与c一样

数组

数组介绍

- 数组是一种容器，可以用来存储同种数据类型的多个值
- 使用数组时，最好将容器的类型和存储的数据类型保持一致

数组的定义与静态初始化

数组的定义

- Java中数组的定义有两种方式
    - 数组类型[] 数组名（这种用的多）
    - 数组类型 数组名[]（这种和C的一样）

数组的静态初始化

- 初始化：在内存中为数组开辟空间并肩数据存进数组的过程

- 完整格式：
  - 数组类型[] 数组名 = new 数据类型[]{元素1,元素2...};
  - `int[] array = new int[]{11,22,33};`
- 简化格式：
  - 数组类型[] 数组名 = {元素1,元素2...};
  - `int[] array = {11,22,33};`

数组的地址值和数据访问

数组的地址值

- 数组名**表示的是数组的地址**
- 地址值的含义：
  - `[I776ec5645`：其中，`[`表示是数组、`I`表示是整型、后面的`776ec5645`是地址

数组的数据访问

- 与c一样

数组的遍历

数组的遍历

- `数组名.length`可以获取数组的长度，与for循环结合可以实现遍历
- IDEA中可以使用`数组名.fori`实现快速生成数组遍历
- 数组遍历的一个建议：一个遍历只做一件事情

数组动态初始化

动态初始化

- 动态初始化：初始化时只指定数组长度，由系统为数组分配初始值
  - 格式：数据类型[] 数组名 = new 数据类型[数组长度];
  - 范例：`int[] arr = new int[3];`
  - 创建的时候可以主动指定数组长度，虚拟机自动给出默认的初始值
    - 整数类型的默认初始化值是`0`
    - 浮点数类型的默认初始化值是`0.0`
    - 字符类型的默认初始化值是`'\u0000'`,输出表现为空格
    - 布尔类型的默认初始化值是`false`
    - 引用数据类型的默认初始值为`null`

数组内存图

Java内存分配

- **栈**：方法运行时使用的内存，比如main方法运行，进入方法栈中执行
- **堆**：存储对象或者数组，new来创建的，都存储在堆内存
  - new出来的东西会在堆中占据空间
- 方法区：存储可以运行的class文件
- 本地方法栈：JVM在使用操作系统功能的时候使用，和使用者开发无关

- 寄存器：CPU使用，与使用者开发无关

数组的内存图

- 数组的变量名存储在栈中，即实际的存储地址值
- 数组中的元素存储在堆中，因为元素是new出来的，即变量名指向的地址对应的值，如果没有new而是直接将之间new出来的空间的地址赋值的话，那么不会开辟新空间

两个数组指向同一个空间的内存图

- 如果代码如下

```
int[] arr1 = {11, 22};
int[] arr2 = arr1;
```

那么此时arr2与arr1指向的空间相同，其实就类似于指针赋值

数组的注意事项

- java中的数组名与C存在不同，java中的数组名是可以被赋值的，而C语言中数组名不能被改变，java中数组名只是表示引用到了堆中的一个地址，这个指向是可以改变的

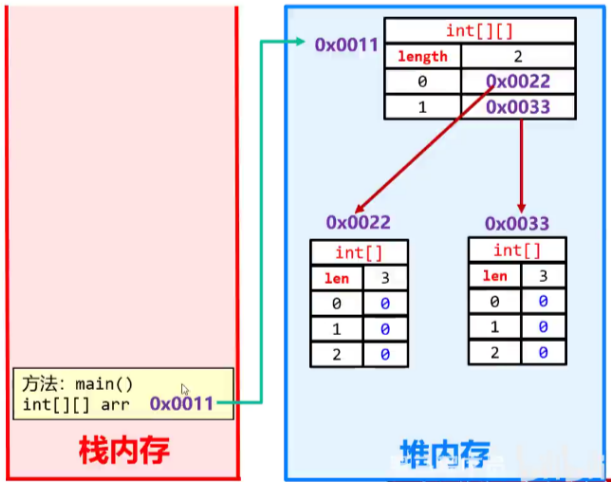
二维数组

二维数组

- 静态初始化
  - 格式：`数据类型[][]  数组名 = new 数据类型[][] {{元素1,元素2},{元素3,元素4}};`
  - 简化格式：`数据类型[][]  数组名 = {{元素1,元素2},{元素3,元素4}};`
  - 举例1：`int[][] arr = new int[][]{{1,2}, {3,4,5}};`
  - 举例2：`int[][] arr = {{1,2}, {3,4,5}};`
- 元素访问
  - `数组名[索引][索引]`
- 双重循环遍历二维数组
- 动态初始化
- 格式：`数据类型[][]  数组名 = new 数据类型[][];`
- 二维数组的内存图

```
public static void main(String[] args) {
    int[][] arr = new int[2][3];

}
```



二维数组的特殊情况

- 特殊情况1：

```
int[][] arr = new int[2][];
int[] arr1 = {11, 22};
int[] arr2 = {44, 55, 66};
arr[0] = arr1;
arr[1] = arr2;
```

  - 上述是允许的，因为在java中，arr中存放的其实是一位数组的地址，所以只需要指定引用了这个地址就行，随后将引用的地址赋值就可以使用了，与C不一样
- 特殊情况2：

```
int[][] arr = new int[2][3];
int[] arr1 = {11, 22};
int[] arr2 = {44, 55, 66};
arr[0] = arr1;
arr[1] = arr2;
```

  - 上述也是允许的，二维数组也是可以改变引用对象的

方法

方法

方法的基础

- 什么是方法
  - 方法是程序中最小的执行单元（有点类似于C中的函数）
- 方法的使用场景
  - 对于重复的代码或者具有独立功能的代码可以封装到方法中
- 方法的好处
  - 使用方法可以提高代码的复用性以及可维护性

方法的格式

最简单的方法定义格式

- 定义格式

```
public static void 方法名()
{
    方法体（即打包好的代码）;
}
```

- 调用

`方法名();`

带参数的方法定义格式

- 定义格式

```
public static void 方法名(参数1,参数2...)
{
    方法体（即打包好的代码）;
}
```

- 调用格式

  `方法名(实参);`

- 形参与实参
  - 形参：形式参数，方法定义中的参数
  - 实参：实际参数，方法调用中的参数

带返回值的方法定义格式（完整格式）

- 定义格式

```
public static 返回值类型 方法名(参数1,参数2...)
{
    方法体（即打包好的代码）;
    return 返回值;
}
```

- 调用格式
  - 直接调用：`方法名(实参);`
  - 赋值调用：`返回值类型 变量名 = 方法名(实参);`
  - 输出调用：`System.out.println(方法名(实参));`

方法的注意事项

- 方法不调用就不执行
- 方法与方法之间是平级关系，不能互相嵌套定义，但是可以相互调用
- 返回值类型为void的方法中可以有return，但是不饿能携带任何数值，相当于方法的结束
- 方法有返回值的时候，必须return，否则会报错

方法的重载

方法的重载

- 在同一个类中，定义了多个同名的方法，这些同名的方法具有同种的功能
- 每个方法具有不同的参数类型或者参数个数，这些同名的方法，就构成了重载关系
  - 参数不同包括**个数不同、类型不同、顺序不同**
- 可以简单理解为：**同一个类中**具有**相同的方法名**，但是**参数却不同**，与返回值是什么类型无关

方法的内存

方法调用的基本内存原理

- 调用时方法入栈，调用结束方法出栈（与C函数的执行类似）
- 疑问?
  - 在被调用方法中申请的变量空间，在方法执行结束后会释放吗？**经过验证，数组返回之后没有释放**

    - 猜测：这是因为在被调用方法中申请的数组名，虽然执行完毕后数组名被释放了，但是在堆里申请的数组存放的数据还在，而执行完毕之后是由主调用方法所引用，因此被调方法中数组名释放与否不会影响主调函数中的指向

方法传递基本数据类型的内存原理

- 基本数据类型
    - 数据存储在自己的空间中
    - 当复制给其他变量时，赋值的也是真实的值
- 引用数据类型
    - 数据值存储在其他空间中，自己空间中存储的是地址值
    - 赋值给其他变量，赋值的是地址值

方法传递引用数据类型的内存原理

- 传递基本数据类型时，传递的是真实的数据，形参的改变不影响实参的值（值传递）
- 传递引用数据类型时，传递的是地址值，形参的改变会影响实参的值（地址传递）

面向对象

设计对象并使用

类和对象

- 类：是对象共同特征的描述
- 对象：是根据类创建的真实存在的具体东西
- 必须先有类再有对象

如何定义类

```
public class 类名{
    1、成员变量（代表属性,一般是名词）
    2、成员方法（代表行为,一般是动词）
    3、构造器(后面学习)
    4、代码块(后面学习)
    5、内部类(后面学习)
}
```

```
public class Phone {

    //属性
    String brand;
    double price;

    //方法
    public void call(){
        System.out.println("打电话");
    }

    public void play(){
        System.out.println("玩游戏");
    }

}
```

类的几个补充

- 用来描述一类事物的类，专业叫做:Javabean类。在Javabean类中，是不写main方法的
- 在以前，编写main方法的类，叫做测试类。我们可以在测试类中创建javabean类的对象并进行赋值调用
- 类名首字母建议大写，需要见名知意，大驼峰模式。
- 一个java文件中可以定义多个class类，且只能一个类是public修饰，而且public修饰的类名必须成为代码文件名。**实际开发中建议一个文件定义一个类**
- 成员变量的完整定义格式是:`修饰符 数据类型 变量名称=初始化值;`一般无需指定初始化值，会自动赋默认值，即基本类型的默认为0或0.0，引用类型默认null

封装

面向对象三大特征

- 封装
- 继承
- 多态

封装

- 对象代表什么，就得封装对应的数据，并提供数据对应的行为
- private关键字
  - 是一个权限修饰符
  - 可以修饰成员（成员变量和成员方法）
  - 被private修饰的成员只能在本类中才能访问
  - 一般使用时是通过public成员方法去操作private成员变量

成员变量与局部变量

- 如果变量放在成员方法之外就是成员变量
- 如果变量放在成员方法之中就是成员变量

this关键字

- 当成员变量与局部变量重名时，那么谁离调用处近就使用谁（**就近原则**），如果非要使用成员变量，那么就可以通过`this.成员变量`来使用

```
//成员变量
private int age;
//成员方法
public void method(String age){
    //局部变量
    int age = 10;
    //此处采用就近原则输出，故输出10
    System.out.println(age);
    //此处会输出成员位置的变量，故输出0
    System.out.println(this.age);
}
```

- this的作用
  - 可以区别成员变量和局部变量

构造方法

构造方法概述

- 构造方法也叫作构造器、构造函数
- 作用：在创建对象的时候由**虚拟机自动调用**给成员变量进行**初始化**

构造方法的格式

```
public class Student{
    修饰符 类名{
        方法体;
    }
}
```

- 特点
    - 方法名与类名相同，大小写也要一致
    - 没有返回值类型，连void都没有
    - 没有具体的返回值（不能由retrun带回结果数据）
- 执行时机
    - 创建对象的时候由虚拟机调用，不能手动调用构造方法
    - 每创建一次对象，就会调用一次构造方法
- 注意事项
    - 构造方法的定义
        - 如果**没有定义**构造方法，系统将给出一个**默认的无参数构造**方法
        - 如果定义了构造方法，系统将不再提供默认的构造方法
    - 构造方法的重载
        - 带参构造方法，和无参数构造方法，两者方法名相同，但是参数不同，这叫做构造方法的重载
    - 推荐的使用方式
        - 无论是否使用，都组好手动书写无参构造方法和带全部参数的构造方法

构造举例

```
//无参构造，假设类名为Studnet
public Student(){
    //初始要执行的语句
}
//有参构造，假设类名为Studnet
public Student(String name, int age){
    //初始赋值
    this.name = name;
    this.age = age;
    //要执行的其他初始语句
    ;
}
```

构造理解

- 大概相当于一个初始化变量的函数，在创建此类的对象时在这个函数中的语句都会自动由虚拟机调用执行

标准的JavaBean类

执行原则

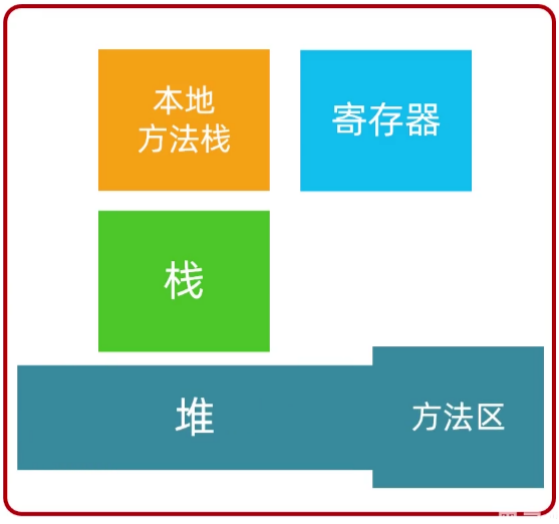
- 类名需要见名知意
- 成员变量使用private修饰提供至少两个构造方法
  成员变量使用专用修饰
- 提供至少两个构造方法
  - 无参构造方法
  - 带全部参数的构造方法
- 成员方法
  - 提供每一个成员变量对应的setXxx()/getXxx()
  - 如果还有其他行为，也需要写上
- 举例

```
public class User {
    //属性
    private String username;
    //无参构造
    public User(){
    }
    //有参构造
    public User(String username, String password, String email, String gender,
int age){
        this.username = username;
    }
    //getXxx方法
    public String getUsername() {
        return username;
    }
    //setXxx方法
    public void setUsername(String username) {
        this.username = username;
    }
}
```

对象内存图

java内存分配之方法区

- JDK7以前，堆与方法区连在一起



- JDK8开始，取消方法区，新增元空间。把原来方法区的多种功能进行拆分,有的功能放到了堆中，有的功能放到了元空间中

单个对象的内存图

- 执行`Student s = new Student();`之后：
    1. 加载class文件
    2. 申明局部变量
    3. 在堆内存中开辟一个空间
    4. 默认初始值
    5. 显示初始值
    6. 构造方法初始化
    7. 将堆内存中的地址值赋值给左边的局部变量

两个引用指向同一个对象

- 在堆里申请的对象一旦没有引用，那么该对象就会无法使用

基本数据类型与引用数据类型

基本数据类型

- 数据值存储在自己的空间中
- 去赋值则赋值的是真实值

引用数据类型

- 数据之存储在其他空间中，自己空间中存储的时地址值
- 去赋值则赋值的是地址值

this的内存原理

this的内存原理

- this的本质
    - 代表方法调用者的地址值

成员变量与局部变量的区别

区别	成员变量	局部变量
类中位置不同	类中方法外	方法内、方法申明上（形参）
初始化值不同	有默认初始化值	没有默认初始化值。使用之前需要赋值
内存位置不同	堆内存	栈内存
生命周期不同	随着对象的创建而存在，随着对象的消失而消失	随着方法的调用而存在，随着方法的运行结束而消失
作用域	整个类中有效	当前方法中有效

scanner接收

- `nextInt(), next()`等从键盘输入时，遇见回车、空格、制表符就停止接收
- `nextline()`从键盘输入时，遇见回车才停止接收
- 键盘录入的两套体系不能混用，例如先用nextInt，再用nextLine的话，nextInt处的回车就会被nextLine接收，导致nextLine无法生效

API

API

API

API

- API：应用程序编程接口
- 简单的说：API就是别人已经写好的东西，我们不需要自己编写，直接使用即可
- Java API：指的是JDK中封装的各种功能的Java类，我们不需要关心其具体实现，只要学习这些类时如何使用的就行

API帮助文档

- 帮助开发人员你更好的使用API和查询API的一个工具

字符串

字符串

String概述

- java.lang.String代表字符串，Java程序中的所有字符串文字都为此类的对象
- 字符串的内容是不会发生改变的，它的对象在创建后值不能被更改
- String是定义好的一个类，定义在java.lang包（属于Java的核心包）中，所以使用的时候不需要导包，Java程序中的所有字符串文字都被实为此类的对象

创建String对象的两种方式

1. 直接赋值
    - `String s1 = "abc";`
2. new

- `String s2 = new String();//获取空白字符串对象`
- `String s3 = new String("abc");//传递字符串，没太大意义`
- ```
  //传递一个字符数组，根据字符数组的内容再创建一个新的字符串对象
  //当需要修改字符串的内容时，就可以通过修改字符数组中的值来达到目的
  char[] chs = {'a', 'b', 'c'};
  String s4 = new String(chs);
  ```
- ```
  //传递字节数组，根据字节数组的内容创建新的字符串对象
  //其输出结果为abc，即会根据bytes中的元素进行ASCII码表的转换存储
  byte[] bytes = {97, 98, 99};
  String s5 = new String(bytes);
  ```

String串池

- 在Java内存中存在一个区域：“串池”
  - 在JDK7以前存在方法区中
  - JDK8以后存在堆区中（但是要注意，串池到了堆中，但是串池与堆不同）

1. 直接赋值的String内存
   - 当使用双引号直接赋值时，系统会检查该字符串在串池中是否存在
   - 如果不存在则创建新的，否则会直接引用

```
String s1 = "abc";
String s2 = "abc";
//假设s1赋值时串池中没有“abc”，那么就会在串池新创建并将地址返回给s1（设为0x0011），而对于s2来说，此时已经存在“abc”了，则将该地址返回给s2，即s2也是0x0011
```

2. new出来的String内存
   - new不同之处在于，每new一次，都会在内存中开辟空间

```
char[] chs = {'a', 'b', 'c'};
String s1 = new String(chs);
String s2 = new String(chs);
//此时s1根据chs新创建了对象放在了堆中而不是串池中，而s2同理也是根据chs新创建了对象放在了堆中，因此，s1与s2有着不同的引用地址对象
```

String类中的常见方法

- `==`号比较的是什么？
  - 基本数据类型：比较的是其数据值
  - 引用数据类型：比较的是地址值
- 字符串比较
  - 如果都是直接赋值的方式，那么可以直接`==`，但是不建议，尽量别使用
  - `boolean equals`方法（要比较的字符串）
  - `boolean equalsIgnoreCase`方法（要比较的字符串并且忽略大小写）
- 字符串遍历
  - `public char charAt(int index):根据索引返回字符`
  - `public int length(): 返回此字符串的长度`
  - 数组的长度：`数组名.length`

    - 字符串的长度:`字符串对象.length()`
        - 需要注意的是：Java中字符串不支持str[i]，即不能以数组的方式输出，因此如果需要遍历，只要通过下面这种方式

```
for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    System.out.println(ch);
}
```

    - 字符串拼接可以使用`+`，这是很关键的一点，别忘记
- 字符串截取
    - `String substring(int beginIndex, int endIndex)`
        - 包头不包尾，包左不包右
        - 只有返回值才是截取的小串
    - `String substring(int beginIndex)`
        - 截取到末尾
- 字符串替换
    - `String replace(旧值，新值)`
        - 只有返回值才是替换之后的结果

StringBuilder

StringBuilder概述

- StringBuilder可以看成一个容器，创建之后里面的内容是可变的
    - 作用：提高字符串的操作效率

StringBuilder构造方法

- `public StringBuilder()`
    - 创建一个空白可变字符串对象，不含有任何内容
- `public StringBuilder(String str)`
    - 根据字符串的内容来创建一个可变字符串对象

StringBuilder常用成员方法

- append方法：添加数据，并返回数据本身
- reverse方法：反转容器中的内容
- length：返回长度
- toString：将StringBuilder抓换为String

StringBuilder应用场景

- StringBuilder的一个场景就是为String服务，当追加数据结束之后就可以转换为String，在String格式在开始进行操作，可以认为StringBuilder是String的工具
- 字符串拼接时使用
- 字符串反转时使用

链式编程

- 当调用方法的时候，不需要用遍历接收其他的结果，可以继续调用其他方法

- `sb.append("aaa").append(1).append(2.3);`
- 其实就是把方法的返回值继续使用

StringJoiner

StringJoiner概述

- StringJoiner跟StringBuilder一样，也可以看成是一个容器，创建之后里面的内容是可变的。
- 作用：提高字符串的操作效率，而且代码编写特别简洁，但是目前市场上很少有人用。
- JDK8出现的

StringJoiner的构造方法

- `public StringJoiner(间隔符号)`
  - 创建一个StringJoiner的对象，指定拼接时的间隔符号
- `public StringJoiner(间隔符号，开始符号，结束符号)`
  - 创建一个StringJoiner的对象，指定拼接时的间隔符号、开始符号、结束符号

StringJoiner的成员方法

- add方法：添加数据并返回对象本身（貌似只能添加字符串数据，那么可以在待添加的非字符串数据后`+`上空字符串即可）
- length方法：返回长度
- toString：返回一个字符串

字符串拼接扩展

字符串拼接的底层原理

- JDK8之前的原理
  - 拼接的时候没有变量，都是字符串，那么就会触发字符串的优化机制，在编译的时候就已经是最终结果，相当于直接赋值的过程，会复用
  - 在拼接的时候，如果有变量参与，系统底层会自动创建一个StringBuilder对象，然后再调用其append方法完成拼接。拼接后，再调用其toString方法转换为String类型，而toString方法的底层是直接new了一个字符串对象，不会复用，如下：

```
String s1 = "a";         //s1的字符串是放在串池中
String s2 = s1 + "b";    //s2会放在堆里面，b放在串池中
//上面第二句在JDK7中等价于下句
Strnig s2 = new String Builder().append(s1).sppend("b").toString();
```

- JDK8之后的原理
  - 系统会预估要字符串拼接之后的总大小，把要拼接的内容都放在数组中，此时也是产生一个新的字符串
- 总结：
  - 如果字符串中有很多变量参与，那就不要直接`+`，会浪费和性能，使用StringBuilder或StringJoiner即可

StringBuilder提高效率原理

- StringBuilder是一个容器，所以即使追加数据也不会产生新的空间，节约内存

StringBuilder源码分析

- 默认创建长度为16的字节数组
- 添加的内容长度小于16则直接存
- 添加的内容大于16会扩容（原来的容量*2+2）
- 如果扩容之后还不够则以实际长度为准

集合

集合基础

集合与数组的区别

- 长度
  - 数组的长度固定，集合的长度可变
- 存储类型
  - 数组可以存基本数据类型和引用数据类型，但是**集合只能存引用数据类型**，如果要存储基本数据类型的话**需要将其变成对应的包装类**

泛型

- 泛型用来限定集合存储数据的类型，在API帮助手册中来表示可以使用泛型

java.util
类 ArrayList<E>

集合的简单定义

```
ArrayList<String> list = new ArrayList<String>();  //JDK8之前
ArrayList<String> list = new ArrayList<>();         //JDK8之后
/*
<string>表示限定列表中的数据只能是字符串，不能直接限定为基本数据类型
要使用基本数据类型必须需要将其变成对应的包装类*/
```

ArrayList成员方法

方法名	说明
boolean add(E e)	添加元素，返回值表示是否添加成功
boolean remove(E e)	删除指定元素,返回值表示是否删除成功
E remove(int index)	删除指定索引的元素,返回被删除元素
E set(int index,E e)	修改指定索引下的元素,返回原来的元素，原元素会被覆盖
E get(int index)	获取指定索引的元素
int size()	集合的长度，也就是集合中元素的个数

ArrayList限定基本数据类型

- 要在ArrayList中使用基本数据类型，首先需要转换为对应的包装类：

```
byte ---> Byte          short   ---> Short
char ---> Character     double  ---> Double
int  ---> Integer       boolean ---> Boolean
long ---> Long          float   ---> Float
```

- 其使用方法如下

```
ArrayList<Byte> list = new ArrayList<>();
//创建之后就可以往集合中添加Byte类型的数据
```

面向对象进阶

static与继承

static

- `static`表示静态，是`Java`中的一个修饰符，可以修饰成员方法和成员变量
  - 被`static`修饰的成员变量叫做静态变量
    - 特点：被该类所有对象共享，是不属于对象而属于类，随着类的加载而加载，优先于对象存在
    - 调用方式：类名调用（推荐）、对象名调用
  - 被`static`修饰的成员方法叫做静态方法（后续仍有学习）
    - 特点：多用在测试类和工具类中，`Javabean`中很少会用
    - 调用方式：类名调用（推荐）、对象名调用

static内存

- `static`修饰的成员变量放在堆中的静态区（JDK8以后，JDK7以前放在方法区）
- 在还未创建对象时，只要赋值了`static`修饰的成员变量，那么该成员变量就会优先于对象的创建而创建在静态区中，即该成员变量不依赖对象的创建

工具类

- `Javabean`类：用来描述一类事物的类，例如`Student`、`Dog`等
- 测试类：用来检查其他类是否书写正确，带有`main`方法的类，是程序的入口
- 工具类：不是用来描述一类事物的，而是帮我们做一些事情的类（Util）
  - 类名需要见名知意
  - 需要私有化构造方法（因为工具类是解决问题的，创建对象是没有意义的）
  - 方法定义为静态的（方便调用）

static的注意事项

- 静态方法只能访问静态变量和静态方法
- 非静态方法可以访问静态变量或者静态方法，也可以访问非静态的成员变量和非静态的成员方法
- 静态方法中是没有`this`关键字
- 总的来说就是：
  - **静态方法中只能访问静态，非静态方法可以访问所有，静态方法中没有`this`关键字**

- 可以站在成员方法和静态方法的角度去分析，静态方法相当于共享方法，经常来表示一个解决问题的工具，工具不需要知道使用它的对象是谁，因此不需要`this`关键字来传递使用者即对象的地址，因此，静态方法也没有办法通过`this`关键字去访问成员中的信息，自然无法使用成员方法和成员变量。但是因为静态方法属于工具，因此非静态方法自然可以使用这个工具，不管是以对象个人的名义（即对象名调用）还是类名的名义（即类名调用）都可以实现，但是因为总的来说，静态方法不是属于成员，而是属于类的，因为及建议通过类去调用
- 静态是随着类的加载而加载，非静态是跟对象有关的

重新认识main方法

main方法

- `punlic`：被JVM调用，访问权限足够大
- `static`：被JVM调用，不用创建对象，直接类名访问，因为main方法是静态的，所以测试类中其他方法也需要是静态的
- `void`：被JVM调用，不需要给JVM返回值
- `main`：一个通用的名称，虽然不是关键字，但是被JVM识别
- `String[] args`：以前用于接收键盘录入数据的，现在没用，现在使用Scanner即可实现功能

继承

继承

- Java中提供一个关键字extends，用这个关键字，我们可以让一个类和另一个类建立起继承关系

```
public class Student extends Person {}
//Student称为子类（派生类），Person称为父类（基类或超类）
```

- 使用继承的好处
  - 可以把多个子类中重复的代码抽取到父类中，提高代码的复用性
  - 子类可以在父类的基础上增加其他的功能，使子类更强大

继承小结

- 什么时候用继承
  - 当类与类之间，存在相同（共性）的内容，并满足子类是父类中的一种，就可以考虑使用继承，来优化代码
- 继承的格式

```
public class 子类 extends 父类 {}
```

- 继承后子类的特点
  - 子类可以得到父类的属性和行为，子类可以使用
  - 子类可以在父类的基础上新增其他功能，子类更强大

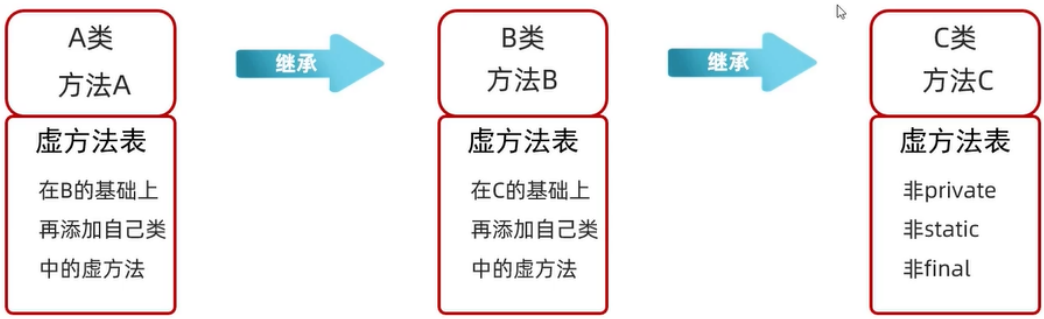
继承的特点

- Java只支持单继承，不支持多继承，但支持多层继承
  - 单继承表明只能继承一个父类，而不能继承多个父类
  - 但是可以多层基层，即A可以继承B，B可以继承C
- Java中如果有父类则继承自父类，如果不存在父类则继承于Object类

- 在继承过程中，如果父类的方法前有private修饰符，那么子类中就无法调用该父类方法了，即**子类只能访问父类中非私有的成员**

子类对于父类的继承

- 父类中拥有三种内容：
  - 构造方法
  - 成员变量
  - 成员方法
- 构造方法是否可以被子类继承
  - 无论父类的构造方法是私有还是非私有，父类的构造都**不能**被子类继承，因为构造方法都是类中独有的，因此父类不可能把自己的构造方法传承给子类，子类想要用构造方法，则子类需要自己写
- 成员变量是否可以被子类继承
  - **无论父类的成员变量是私有还有非私有，父类的成员变量都能被子类继承**，但是继承的时候仍然保留了private和public修饰符，有因为私有的变量是无法直接使用的，所以，**子类无法直接使用继承下来的私有成员变量**，但是可以通过set和get方法去操作
- 成员方法是否可以被继承
  - 当父类的成员方法被加载到虚方法表中可以被继承，**否则无法被继承**
  - 在继承成员方法时会建立一个虚方法表：



  - 即子类在继承父类得方法时，会直接将所有父类的虚方法都存储进虚方法中，并随着字节码文件加载到方法区中，因此总的来说，只有当方法被加载到虚方法表中的话就可以继承，否则不行

继承中的访问特点

- 继承中的成员变量的访问特点
  - 就近原则：先在局部位置找，然后本类成员位置找，随后父类成员位置找，逐级往上
  - 直接调用以及`this`和`super`都遵循这个原则，例如`this`，是直接从本类成员位置找，找不到会直接去父类位置找

```
public class Fu{
    String name = 'Fu';
}

public class Zi extends Fu{
    String name = 'Zi';

    public void ziShow(){
        String name = "ziShow";
        sout(name);          //没有指定则默认就近原则输出Zi
        sout(this.name);     //this指定自身调用则输出zishow
        sout(super.name);    //super指定父类调用则输出Fu
```

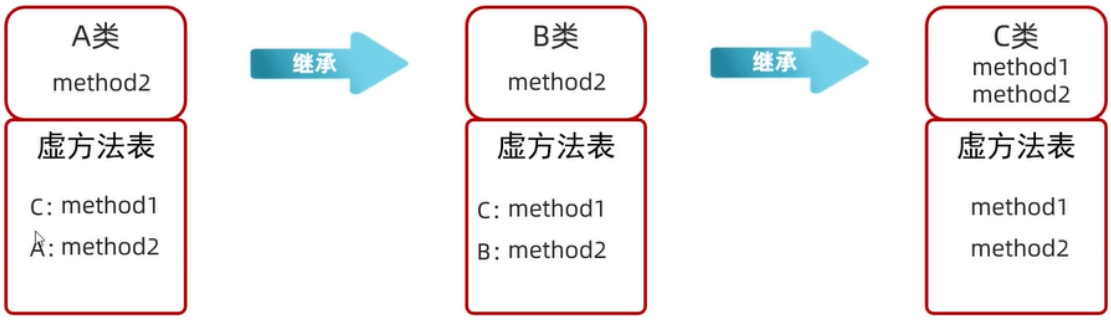
```
    }
}
```

- 继承中成员方法的访问特点
  - 直接调用或者`this`调用都满足就近原则，逐级往上
  - `super`调用会直接调用父类
  - 如果父类的方法不能满足子类的需求时，就需要对方法进行**重写**
- 继承中构造方法的访问特点
  - 父类中的构造方法不会被子类继承
  - 子类中所有的构造方法默认先访问父类中的父类构造，再执行自己
    - 因为子类再初始化的时候有可能会使用到父类中的数据，因此需要先让父类完成初始化
  - 子类是如何调用父类构造方法的?
    - 子类构造方法的第一行语句默认都是`super()`，不写也存在，且必须在第一行
    - 如果想调用父类有参构造则必须手动写`super`进行调用，并将待传入的值放进去，如`super(参数);`
  - 继承中构造方法的访问特点是什么?
    - 子类不能继承父类的构造方法，但是可以通过`super`调用
    - 子类构造方法的第一行，有一个默认的`super();`
    - 默认先访问父类中无参的构造方法，再执行自己
    - 如果想要方法文父类有参构造，必须手动书写

重写

重写

- 在继承体系中，子类出现了和父类中一摸一样的方法声明，则就称子类的这个方法是重写的方法
  - @Override重写注解
    - @Override是放在重写后的方法上，检验子类重写时语法是否正确
    - 加上注解后如果有红色波浪线则表示语法错误
    - 建议重写方法都加@Override注解，代码会更加安全
- 方法重写的本质
  - 如果发生了重写则会覆盖



- 方法重写注意事项和要求
  - 重写方法的名称、形参列表必须与父类中的一致。
  - 子类重写父类方法时，访问权限子类**必须**大于等于父类（暂时了解：空着不写<protected < public)
  - 子类重写父类方法时，**返回值类型子类必须小于等于父类**
  - 建议：重写的方法尽量和父类**保持一致。**
  - 只有被添加到**虚方法表**中的方法才能被重写

this、super使用总结

this、super使用总结

- this：可以理解为一个变量（且是局部变量），当方法被调用的时候this会有值，表示当前方法调用者的地址值
- super：代表父类存储空间

关键字	访问成员变量	访问成员方法	访问构造方法
`this`	`this.成员变量`（访问本类成员变量）	`this.成员方法(...)`（访问本类成员方法）	`this(...)`（访问本类构造方法）
`super`	`super.成员变量`（访问父类成员变量）	`super.成员方法(...)`（访问父类成员方法）	`super(...)`（访问父类构造方法）

this(...)使用场景

```
public Student(){
    //表示调用本类其他构造方法，在本例中相当于调用下面的带参构造，可以用来给某个变量添加默认值
    //同时：虚拟机就不会再默认添加super();
    //因为再调用本类其他构造方法时会默认添加super();
    this(null, 0, "xx学校")
    //此句相当于执行了Student(null, 0, "xx学校")
}
public Student(String name, int age, String school){
    //此处会默认添加super();
    this.name = name;
    this.age = age;
    this.school = school;
}
```

多态

多态

多态

- 什么是多态?
  - 同类型的对象表现出不同的形态就叫多态，例如动物类为父类，其子类有猫类和狗类，如果不知道接下来具体是要猫类还是狗类创建对象，就可以使用多态
- 多态的表现形式
  - `父类类型 对象名称 = 子类对象;`
- 多态的前提
  - 有继承关系
  - 有父类引用指向子类对象（`Fu f = new Zi();`）
  - 有方法重写
- 多态的好处
  - 使用**父类型作为参数**，**可以接受所有子类对象**，体现多态的扩展性与便利

多态调用成员的特点

- 变量调用：编译看左边，运行也看左边
  - 在用javac编译代码的时候，会看**父类**方法中有没有这个变量，如果有则编译成功，否则失败
  - **同理**，在java代码运行的时候实际获取的就是左边**父类**中成员变量的值
- 方法调用：编译看左边，运行看右边
  - 在用javac编译代码的时候，会看**父类**方法中有没有这个方法，如果有则编译成功，否则失败
  - **但是**，在java代码运行的时候实际获取的就是右边**子类**中的方法
- 解释：
  - 在调用变量的时候，因为是根据父类申请的变量，而实际创建的对象是子类，这就导致在调用变量的时候，会直接调用父类的变量，但是因为子类中方法实现了重写，其中的虚方法表会将父类的调用覆盖，因此调用方法时调用的是子类的方法

多态的优势

- 在多态形式下，右边对象可以实现解耦合，便于扩展和维护

```
//假设有个工作是学生在做，但是现在要求老师去做
//则只需要将第一行代码替换为Person p = new Teacher()即可
//其他的逻辑都不需要改变
Person p = new Student();
p.work();
```

- 定义方法的时候，使用父类型作为参数，可以接收**所有子类对象**，实现多态的扩展性

多态的弊端

- 多态无法调用子类的特有功能
- 解决方案：
  - 将变量强制转换为子类类型就可以了（细节：转换的时候不能乱转）

```
//假设a是：Animal a = new Dog();
//则通过下述可以实现强转
Dog a = (Dog) a;
```

- 完整的强转

```
//instanceof是关键字，相当于“是不是”
if(a instanceof Dog){
    Dog d = (Dog) a;
    d.lookHome();
}
else if(a instanceof cat){
    Cat c = (Cat) a;
    d.catchMouse();
}
else
    sout("不存在该类型，无法转换");
```

- 强转的新特性（JDK14及其之后）

```
//判断a是不是Dog，是则强转为Dog并转换之后变量名为d，Cat同理
if(a instanceof Dog d){
    d.lookHome();
}else if(a instanceof Cat c){
    d.catchMouse();
}
```

- 强制转换能解决什么问题
  - 可以转换成真正的子类类型，从而调用子类独有功能
  - 转换类型与真实对象类型不一致会报错
  - 转换的时候用instanceof关键字进行判断

引用数据类型的类型转换

- 自动类型转换

  `Person p = new Student();`

- 强制类型转换

  `Student s = (Student) p;`

包和final

包

- 什么是包?
  - 包就是文件夹，用来管理各种不同功能的Java类，方便后期代码维护
- 包名的规则：
  - 公司域名反写+包的作用，需要全部英文小写，见名知意
- 使用其他类的规则
  - 使用其他类时，需要使用全类名（即包名+类名）
    - 使用用一个包中的类时，不需要导包
    - 使用Java.lang包中的类时，不需要导包
    - 其他情况都需要导包（导包需要使用`import`关键字）
    - 如果同时使用两个包的同名类，则需要使用全类名（因为两个不同包的类名一样的话，Java就不知道该根据那个类申请对象，因此可以导入一个包，另外一个申请时使用全类名）

final

final

- `final`是一个修饰符
  - 修饰方法：表明该方法是最终方法，不能被重写
  - 修饰类：表明该类是最终类，不能被重写
  - 修饰变量：表明该量是常量，只能被赋值一次
- final修饰变量的细节
  - `final`修饰的变量是基本类型：变量存储的数据值不能发生改变
  - `final`修饰的变量是引用类型：变量存储的**地址值**不能发生改变，但是**对象内部**的属性值等等可以改变

        - 例如申请数组`ARR`，则数组的地址值就不能改变了（就和C中的数组名一样了），但是数组中的值可以改变
- 常量
    - 实际开发中，常量一般作为系统的配置信息，方便维护，提高可读性
    - 命名规则：
        - 单个单词：全部大写
        - 多个单词：全部大写，单词之间用下划线隔开

权限修饰符

权限修饰符

- 权限修饰符:是用来控制一个成员能够被访问的范围的
- 可以修饰成员变量，方法，构造方法，内部类

权限修饰符的分类

- 权限修饰符有四种，作用范围由小到大（`private<空着不写<protected<public`）
    - `private`：只能自己使用
    - 默认：只能本包中使用
    - `protected`：本包以及其他包下的子类可以使用
    - `public`：本包以及其他包都可以调用

修饰符	同一个类中	同一个包中的其他类	不同包下的子类	不同包下的无关类
`private`	√			
空着不写	√	√		
`protected`	√	√	√	
`public`	√	√	√	√

权限修饰符的使用规则

- 实际开发中，一般只用`private`和`public`
    - 成员变量私有
    - 方法公开
- 如果方法中的代码是抽取其他方法中共性代码，这个方法一般也私有，即仅限本包中的方法调用

代码块

代码块

- 局部代码块
    - 在方法中写的代码块，目的是提前结束变量的生命周期

```
public static void main(String[] args){
    {
        int a = 10;
    }
    //此处a就会被释放
}
```

- 构造代码块（现在已经被淘汰了）
  - 写在成员位置的代码块
  - 作用：可以把多个构造方法中重复的代码抽取出来
  - 执行时机：在创建本类对象的时候会先执行构造代码再执行构造方法

```
public class Studnet(){
    private String name;
    //这就是构造代码块，在创建对象时构造方法还先执行一下
    {
        sout("开始创建对象");
    }
    public Studnet(){}
    //...
}
```

- 静态代码块
  - 格式：`static{}`
  - 特点：需要通过static关键字修饰，随着类的加载而加载，并且自动触发，只执行一次
  - 使用场景：在类加载的时候，做一些数据初始化的时候使用，例如从文件读取初始数据的话，那么最好放在静态代码块中。放在方法中初始化是存在弊端的，因为即使是mian方法都可以被调用

```
public class Studnet(){
    private String name;
    //这就是静态代码块，在加载类时会执行一次
    static {
        sout("开始创建对象");
    }
    public Studnet(){}
    //...
}
```

抽象类

抽象类和抽象方法

- 抽象方法：
  - 将共性的行为（方法）抽取到父类之后。由于每一个子类执行的内容是不一样,所以在父类中不能确定具体的方法体，该方法就可以定义为抽象方法（即不在父类的抽象方法中添加方法体就可）
- 抽象类：
  - 如果一个类中存在抽象方法，那么该类就必须声明为抽象类

抽象类和抽象方法的定义格式

- 抽象方法的定义格式：

    - `public abstract 返回值类型 方法名(参数列表);`
- 抽象类的定义格式：
    - `public abstract class 类名{}`

抽象类和抽象方法的注意事项

- 抽象类不能实例化（例如`Person`类中的`work()`方法设为了抽象方法，那么该类就成为了抽象类，则就不能根据`Person`类申请对象了，因为`Person`类中的`work`方法不存在方法体）
- 抽象类中不一定有抽象方法，有抽象方法的类一定是抽象类
- 虽然不能创建对象，但是可以有构造方法（当创建子类对象时，可以给属性进行赋值）
- 抽象类的子类
    - 要么重写抽象类中的所有抽象方法
    - 要么是抽象类

接口

为什么有接口

- 接口就是一种规则，是对行为的抽象

接口的定义和使用

- 接口用关键字interface来定义

  `public interface 接口名{}`
- 接口不能实例化
- 接口和类之间时实现关系，用过`implements`关键字表示

  `public class 类名 implements 接口名{}`
- 接口的子类（实现类）
    - 要么重写接口中的所有抽象方法
    - 要么是抽象类

接口的注意事项

- 接口和类的实现关系，可以单实现，也可以多实现

  `public class 类名 implements 接口名1，接口名2{}`
- 实现类还可以在继承一个类的同时实现多个接口
  `public class 类名 extends 父类 implements 接口名1，接口名2{}`

接口的示例

```
public class Dog extends Animal implements Swim{

    public Dog() {}

    public Dog(String name, int age)
    {        super(name, age);
    }

    //这个方法是对抽象父类中的抽象方法的重写
    @Override
    public void work()
```

```
    {
        System.out.println("狗在看家");
    }
    //这个方法是对接口中的行为的重写
    @Override
    public void swim()
    {
        System.out.println("狗子在游泳");
    }
}
```

接口中成员的特点

- 成员变量
  - 只能是常量
  - 默认修饰符：`public static final`
- 构造方法
  - 没有
- 成员方法
  - 只能是抽象方法
  - 默认修饰符：`public abstract`
- JDK14以前：接口中只能定义抽象方法
- 接口和类之间的关系
  - 类和类的关系
    - 继承关系，只能单继承，不能多继承，但是可以多层继承
  - 类和接口的关系
    - 实现关系，可以单实现，也可以多实现，还可以在继承一个类的同时实现多实现。
    - 如果多个接口中有重名的方法，那么就只会其中留一个方法（如果多个接口的方法重名但是返回值不同，那么就会报错）
  - 接口和接口的关系
    - 继承关系，可以单继承，也可以多继承
    - 如果子接口继承了另外一个接口，那么在类中实现子接口的时候就要重写子类与所继承的类中的所有方法

JDK8开始接口中新增的方法

- JDK7以前：接口中只能定义抽象方法
- JDK8的新特性：接口中可以定义有方法体的方法（默认、静态）
  - 允许在接口中定义默认方法，需要使用关键字default修饰
    - 作用：解决接口升级的问题
  - 接口中默认方法的定义格式：
    - 格式：`public default 返回值类型 方法名(参数列表) {}`
    - 范例：`public default void show(){}`
  - 接口中默认方法的注意事项
    - 默认方法不是抽象方法，所以不强制被重写。但是如果被重写，重写的时候去掉default关键字
    - public可以省略，default不能省略
    - 如果实现了多个接口，多个接口中存在相同名字的默认方法，子类就必须对该方法进行重写

    - 允许在接口中定义静态方法，需要用static修饰
    - 接口中静态方法的定义格式:
        - 格式: `public static 返回值类型 方法名 (参数列表){}`
        - 范例: `public static void show(){}`
    - 接口中静态方法的注意事项:
        - 静态方法只能通过接口名**调用**，不能通过实现类名或者对象名调用，也不能被重写
        - public可以省略，static不能省略
- JDK9的新特性：接口中可以定义私有方法
    - 接口中定义私有方法的作用
        - 将接口中重复的代码可以封装进方法中，在JDK8以前允许抽取封装为`public`形式，但是对于外界来说调用该方法无意义，因此在JDK9之后允许将其定义为`private`形式
    - 接口中私有方法的定义格式
        - 格式1：`private 返回值类型 方法名(参数列表){}`
        - 范例1：`private void show(){}`
        - 格式2：`private static 返回值类型 方法名(参数列表){}`
        - 范例2：`private static void show(){}`
    - 接口中定义私有方法分类
        - 普通的私有方法，为接口中的默认方法服务
        - 静态的私有方法，为接口中的静态方法服务

接口的应用

- 接口代表规则，是行为的抽象。想要让哪个类拥有一个行为，就让这个类实现对应的接口就可以了
- 当一个方法的参数是接口时，可以传递接口所有实现类的对象，这种方式称之为接口多态

适配器设计模式

- 设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性、程序的重用性。
    - 设计模式就是解决各种问题的套路
- 适配器设计模式：解决接口与接口实现类之间的矛盾问题
- 适配器设计模式的书写步骤
    1. 编写中间类XXXAdapter，实现对应的接口，让接口中的抽象方法实现空实现
    2. 让真正的实现类继承中间类，并重写需要用的方法
    3. 为了避免其他类创建适配器类的对象，中间的适配器类用abstract进行修饰

内部类

内部类

- 类的五大成员
    - 属性、方法、构造方法、代码块、内部类
- 什么是内部类
    - 在一个类的里面再定义一个类就是内部类
- 什么时候用到内部类
    - B类表示的事物是A类的一部分，且B单独存在没有意义
    - 例如：汽车的发动机，人的心脏等等，脱离主体没有意义

内部类的访问特点

- 内部类可以直接访问外部类的成员，包括私有
- 外部类要访问内部类的成员必须创建对象

内部类的分类

- 成员内部类
  - 写在成员位置的，属于外部类的成员之一（相当于外部类的成员）

```
public class Car{
    String carName;
    int carAge;
    //Engine类就是成员内部类，其与上面的成员变量等级相等
    class Engine{
        String engineName;
        int engineAge;
    }
}
```

  - 成员内部类可以被一些修饰符所修饰，比如：`private`，默认，`protected`，`public`，`static`
  - 在成员内部类中，JDK16之前不能定义静态变量，JDK16开始才可以定义静态变量
  - 如何创建内部类对象
    - 方式1：`外部类名.内部类名 对象名 = new 外部类名().内部类名();`
    - 范例1：`Car.Engine e1 = new Car().new Engine();`
    - 方式2：`外部类编写方法，对外提供内部类对象`（假如内部类设为私有，则就可以通过这种方式来获取内部类对象）
    - 范例2：

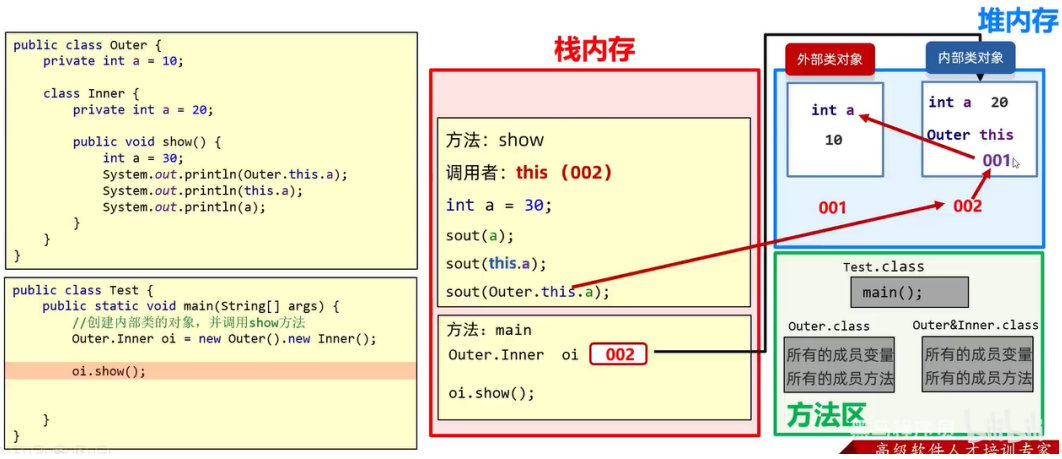
```
//在外部类中定义下述方法
public Engine getInstance(){
    return new
}
```

  - 成员内部类如何获取外部类的成员变量

```
//这是外部类
public class Outer {
    //外部类成员变量
    private int a = 10;
    //内部类（也属于外部类的成员变量之一）
    class Inner {
        //内部类成员变量
        private int a = 20;
        //内部类成员方法
        public void show(){
            int a = 30;
            //本行的a因为没有前缀，故采取就近原则，会输出30
            System.out.println(a);
            //本行的a前缀为this，即Inner对象，故会在Inner本类中的成员变量找，输出20
```

```
                System.out.println(this.a);
                //Outer.this是定义的前缀，意思为根据外部类的对象的地址，故会输出10
                System.out.println(Outer.this.a);
            }
        }
    }
```

- `Outer.this`变量
  - 内部类定义的对象中隐藏着一个`Outer.this`变量，该变量保存的是外部类对象的地址
  - 必须先有外部类对象，再有内部类对象，但是同时必须把内部类对象与外部类对象能够相关联，因此出现了`Outer.this`来保存外部类对象的地址
- 成员内部类的内存图



- 静态内部类
  - 属于成员内部类的一种特殊情况，当内部类用`static`修饰的时候就创建了静态内部类
  - 静态内部类只能访问外部类中的静态变量和静态方法，如果想要访问非静态的需要创建对象（这是因为只有静态方法和静态变量是没有创建对象的时候就存在的，也就是说，如果内部类对象已经创建成功，而外部对象还没有存在（**这是因为静态内部类对象创建不需要以外部对象创建为前提**），那么静态内部类能够调用的也只能是静态的相关数据，而非静态的数据依赖于对象的创建，因此，如果想要输出非静态数据，就必须先创建对象才能调用）
  - 创建静态内部类对象的格式：
    - `外部类名.内部类名 对象名 = new 外部类名.内部类名();`
    - 因为是静态的，所以就不需要先创建外部类对象，静态内部类有点独立于外部类
  - 调用非静态方法的格式：
    - 先创建对象，再对象调用
  - 调用静态方法的格式
    - `外部类名.内部类名.方法名();`
- 局部内部类
  - 将内部类定义在方法里面就叫做局部内部类，类似于方法里面的局部变量（与局部变量等价）
  - 外界是无法直接使用局部内部类，需要再方法内部创建对象并使用
  - 该类可以直接访问外部类的成员，也可以访问方法内的局部变量
- 匿名内部类
  - 匿名内部类本质上就是隐藏了名字的内部类，可以写在成员位置，也可以写在局部位置
  - 格式

```
new 类名或者接口名(){
    重写方法;
};//别忘记这儿的分号
```

- 匿名内部的本质（整体就是一个类的子类对象或者接口的实现类对象）
  - 继承类或者实现接口
  - 方法进行重写
  - 创建对象
- 匿名内部类可以理解为没有对象名字的接口的实现类对象或者其他类的继承类对象
- 使用场景
  - 当方法的参数是接口或者类时，以接口为例，可以传递这个接口的实现类对象，如果实现类只要使用一次，就可以用匿名内部类简化代码