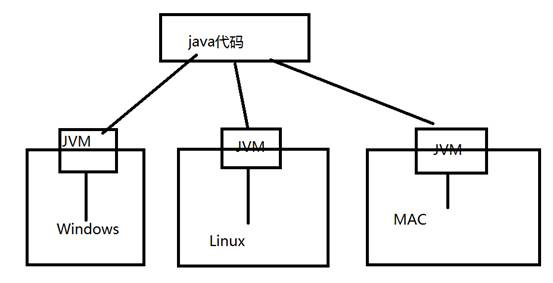
day01

**1.跨平台**

JVM（Java Virtual Machine）---Java虚拟机---将Java代码转化为对应的操作系统能够理解的指令。---不同的操作系统有不同的虚拟机与之对应，同一段代码交给虚拟机之后，虚拟机再转化给操作系统。---JAVA语言能够跨平台的前提是JVM---JVM不是跨平台的。



**2.JDK、JRE、JVM**

JVM---Java Virtual Machine---是Java语言能够跨平台的条件

JRE---Java Runtime Environment --- Java运行时环境 --- JVM+核心类库

JDK---Java Development Kit --- Java开发工具包 --- 开发工具+JRE

3.关键字:

在Java中一共有53个关键字---其中有2个关键字目前没有使用：goto，const---保留字

用于定义数据类型的关键字:8大+class+interface+void+enum

用于定义数据类型值的关键字:true+false+null

用于定义流程控制的关键字:

if+else+switch+case+default+while+do+for+break+continue+return

用于定义访问权限修饰符的关键字:private+protectde+public

用于定义类、函数、变量修饰符的关键字:abstract+final+static+synchronized同步的

用于定义类与类之间关系的关键字:extends+implements

用于定义建立实例、判断实例的关键字:new +this+super+instanceof

instanceof用来在运行时指出对象是否是特定类的一个实例。

用于异常处理的关键字:try+catch+finally+throw+throws

用于包的关键字:package+import

其他修饰符关键字:native+strictfp+transient+volatile+assert(断言)

transient关键字标记的成员变量不参与序列化过程

strictfp应用于类、接口或方法,使浮点运算更加精确,而且不会因为不同的硬件平台所执行的结果不一致

volatile是一个类型[修饰符](https://baike.baidu.com/item/%E4%BF%AE%E9%A5%B0%E7%AC%A6),简单地说就是防止编译器对代码进行优化.

2+12+3+11+3+4+2+4+5+2+5=53个

4.标识符的命名规则

由字母、数字、\_、$组成,但是不建议使用$

Java支持中文命名，但是不建议

数字不能开头

不能使用关键字

尽量见名知意

大小写敏感

day02

1.**计算机常量**

本身不可改变的量

整数常量：所有的整数  3，4，100

小数常量：所有的小数  1.02,5.36，8.7209

字符常量：将一个符号用 ‘’ 标志起来就构成一个字符常量 ‘a’ ‘+’ ‘1’ ‘ ’

字符串常量：将多个字符用 “”标志起来就构成了一个字符串常量 “abc” “12w” “a” “”

布尔常量：只有两个值---true/false---用于表示逻辑值

空常量：null

2.**进制**



3.**数据类型**

**基本数据类型 8大**

**引用数据类型**

数组[]   类class  接口interface

4.**运算符**

**算术运算符:**+ - \* / % ++ --

**赋值运算符:**= += -= \*= /= %= &= |= ^= <<= >>= >>>=

**关系运算符:**==等于 !=不等 > < >= <=

**逻辑运算符:**&与---And |或---Or !非---Not ^异或---Xor &&--短路与 ||--短路或

**位运算符:**&与 |或 ^异或 <<左移 >>右移 >>>无符号右移 ~取反

针对的整数的补码

**三元运算符**:逻辑值? 表达式1:表达式2

运算符的优先级:() ++ -- ~ ！ 算术 << >> >>> 关系 逻辑 & | ^ 三元 赋值

1.day03

**1.流程控制**

**顺序控制**:是指代码是从上倒下从左到右依次运行的

**条件控制:if语句和switch语句**

**循环控制:while do...while for**

day04

**1.数组**

定义格式:

数据类型[] 数组名 = new 数据类型[数组的大小/长度]; --- 动态初始化

数据类型[] 数组名 = new 数据类型[]{元素1, 元素2, ... , 元素n}; --- 静态初始化

2.**内存**

Java将内存分为了5块：栈内存、堆内存、方法区、本地方法栈、寄存器

栈内存：存储变量的。栈内存不会自动给变量赋值。而且变量用完之后会从栈内存中立即移除以释放内存空间

堆内存：存储的是数组或者对象。碰到new关键字的时候会到堆内存中开辟空间。堆内存会对存入其中的元素自动的赋予一个默认值---byte/short/int-0, long-0L, float-0.0f, double-0.0, char- ‘\u0000’, boolean-false, 除了这八种基本类型以外的所有的类型的默认值都是null。 堆内存中的数据用完之后会在不定的某个时刻被回收。

boolean[] bs = new boolean[3];  bs[0] --->false

String[] ss = new String[5]; ss[0] ---> null

3.**数组的应用**

遍历数组

数组排序:

冒泡排序:相邻两个数进行比较，第一波比较后，最大的数在最后。（每比较完之后，后面的数就减少一个比较 ）

选择排序:让数组中的每一个数，依次与后面的数进行比较，如果前面的数大于后面的数，就进行位置的交换。这种说法或许有些人看不明白。换个说法，选择排序：第一个数依次与后面的数比较，第一次比较完之后最小的数在最前面 。

day05

1.数组的反转

2.数组元素的折半查找

3.数组的复制

System.arraycopy(要复制的数组,要复制的起始下标,要存放的数组,要存放的起始下标,要复制的元素的个数);

数组 = Arrays.copyOf(要扩容的数组, 扩容之后的长度);

数组的扩容/缩容---本质上就是数组的复制，所以扩容之后地址必然发生了改变

4.二维数组

数据类型[][] 数组名 = new 数据类型[二维数组的长度][包含的一维数组的长度];

数据类型[][] 数组名 = new 数据类型[二维数组的长度][];

定义了包含3个一维的整型数组，没有规定每一个一维数组的大小

数据类型[][] 数组名 = {{一维数组1},{一维数组2},{一维数组3}, ... ,{一维数组n}};

5.方法:

将一段代码/逻辑提取出来进行包装，这种包装形式就称之为方法/函数。---为了方便重复利用这段代码

修饰符 返回值类型 方法名(参数列表){

代码块;

return 返回值; }

6.方法的递归-在方法中调用了自己本身

当方法执行的时候需要在栈中占用一部分空间，当递归的次数比较多的时候，会耗费大量的内存。如果栈内存不够用了就会出现栈溢出错误

循环的效率要高于递归---所以能使用循环的而时候减少使用递归

day06

**1.方法的重载**

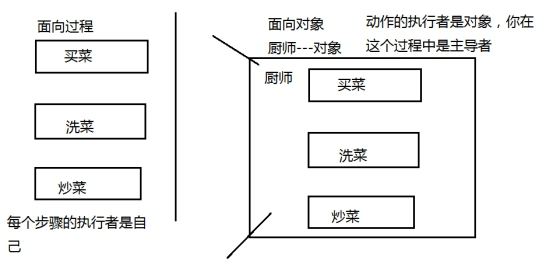
**2.方法的传值**

在方法传参的时候，基本类型传递的是实际的数据，所以新方法中数据发生改变不影响原方法中的数据；数组传递的是地址，所以在新方法中如果不改变地址，则会影响原来的方法中的数组；如果在新方法中改变了地址，则不会影响原来方法中的数组

**3.面向对象的思想**

面向过程：注重的过程，强调的动作

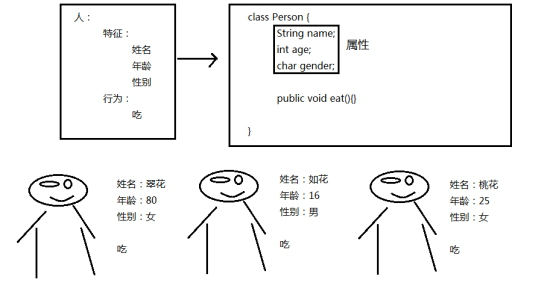
面向对象：所关注的是对象。只要有了对应的对象，就能拥有这个对象所具有的一切功能



面向对象是基于面向过程的

面向对象一定优于面向过程吗？---不一定---当处理一些相对简单的事务的时候，建议使用面向过程；如果处理的事务比较复杂，建议使用面向对象。

**4.类和对象的关系**



根据一类对象的共有的特征和行为进行抽取，将这类事物用类来表示，将特征抽取成了属性，将行为抽取成方法---类是对象的抽取/概括

利用new关键字来创建对应的类的对象/实例---对象是类的具体化

day07

**1.成员变量（member）和局部变量(local variable)**

1. 定义位置：成员变量（属性）是指定义在**类内方法外**的变量；定义在了方法或者语句内的变量就是局部变量。

2. 作用范围：成员变量是作用在**整个类**内；局部变量只能作用于对应的方法或者语句中。

3. 内存位置：成员变量是存储在**堆内存中，并且自动的赋予了一个默认值**；局部变量存在栈**内存中，不会自动给默认值**。

4. 生命周期：成员变量随着**对象的创建而存储到堆内存中，随着对象的回收而被销毁**；**局部变量在方法或者语句执行完成之后立即回收**。

**2.构造方法**

构造方法作用就是对类进行初始化。

在类中与类同名而没有返回值类型的方法---构造方法---创建对象

在类中，如果没有手动添加构造方法，那么Java会在编译的时候添加一个默认的无参构造，如果手动添加了构造方法，Java就不再添加默认的构造方法了。

构造方法可以重载。

new一个对象的时候要用到构造函数,有参数的构造方法就最好加一个无参数的构造方法.

**3.this.**

1.代表当前在活动的对象的引用---可以认为是一个虚拟对象，用于在类内调用本类中的方法或者属性.

引用类的隐藏数据域；函数中的局部变量和成员变量重名时，成员变量会被屏蔽掉，这时我们可以通过"this.成员变量名"来引用成员变量，没有重名的时候也是可以用哒

2.this语句是用于构造函数间简化。

用于构造函数之间进行互相调用。让构造方法调用同一个类的另一个构造方法；用法是"this(参数列表)"

注意：Java要求在构造方法中，语句this(参数列表)应该在其他语句之前出现。

**4.super关键字:**

在子类中表示父类对象的引用---代替了父类对象---虚拟对象---在子类调用父类中的方法或者属性。

在Java中，要创建子类对象必然先创建一个父类对象---父类对象是通过super语句来创建的。---如果子类的构造函数中没有手动提供super语句，那么默认调用父类的无参构造---super语句表示调用父类中对应形式的构造函数---如果父类只提供了含参构造，那么子类的构造函数中就得手动提供对应形式的super语句---super语句必须放在子类构造函数的第一行

调用父类的构造方法；用法是"super(),或者super(参数列表)"，还有该条语句必须出现在子类构造方法中的第一行，这是显式调用父类构造方法的唯一方法

调用父类的方法；用法是"super.方法名（参数）；"通过这样的方式访问父类的方法

**5.构造代码块**

在类内方法外用{}括起来的一段代码---构造代码块/初始化代码块---在创建对象的时候先于构造方法执行,可以给所有对象进行初始化的。创建几个对象就调用几次，是给对象进行通用性的初始化。

**6.局部代码块**

在方法或者语句内部用{}括起来的一段代码---局部代码块---限制变量的使用范围和生命周期，以提高内存的利用率

*静态代码块、构造代码块、构造方法的执行顺序？*

*静态-》构造代码块-》构造方法*

*静态代码块：只执行一次*

*构造方法：每次调用构造方法都执行*

7.**面向对象的特征**

封装、继承、多态（抽象）

**封装**

体现形式：方法，属性的私有化---将类中的属性用private修饰，限制属性的使用范围，对外提供操作和获取这个的属性的公共方法，在这些方法中可以根据场景来限制属性的值，使之更符合场景要求；内部类

优点：提高代码的复用性，保证数据的合法性

**继承**

根据一些类中的共有特征和共有的行为进行抽取，这些行为和特征抽取到一个新的类中，让原来的类通过extends关键字与新的类产生关系，这种关系称之为继承。新抽取出来的类是父类（超类/基类），原来的类就是子类（派生类）。

子类通过继承父类，就可以使用父类中的一部分方法和属性

在Java中，类和类之间支持的是单继承---一个子类只能有一个父类，但是一个父类可以有多个子类。支持多层继承。---继承结构树

单继承一定比多继承好吗？---不一定

多继承在代码的复用性上要优于单继承

**多态**

编译时多态：方法的重载

运行时多态：方法的重写，向上造型---运行时多态是基于继承的,父类声明，子类创建---父类定义行为的集合，子类实现行为的细节

注意：对于使用向上造型创建的对象而言，能干什么看的是父类；怎么干看的是子类。---对于使用向上造型创建的对象，不能去调用子类中独有的方法

向上造型可以看做多态的一种体现。即允许一种类型有不同的实现（可以理解为事物存在的多种体现形态）。

向上造型：向上造型可以大大提高程序的可扩展性。利于代码维护。

1.**父类引用能指向子类对象**，子类引用不能指向父类对象；

2.向上造型：父类引用指向子类对象，例如：Father f1 = new Son();

3.在不确定父类引用是否指向子类对象时，可以用instanceof来判断

8.**权限修饰符**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 本类中 | 子类中 | 同包类中 | 其他类中 |
| public | 可以 | 可以 | 可以 | 可以 |
| protected | 可以 | 可以 | 可以 | 不可以 |
| 默认 | 可以 | 同包子类可以 | 可以 | 不可以 |
| private | 可以 | 不可以 | 不可以 | 不可以 |

范围：public > protected > 默认 > private

9.**重写的理解**

1. 子类重写的方法的权限修饰符的范围要大于等于父类方法的权限修饰符的范围.子大

2. 子类重写的方法的返回值类型是父类方法返回值类型的子类或者是其本身。子小

3.异常,子小

10.**静态变量**

用static修饰的变量称之为静态变量/类变量。在类加载的时候加载到了方法区中的静态区，并在静态区中自动的赋予了默认值。静态变量是先于对象而存在的，所以它可以不通过对象而是通过类名来调用，也可以通过对象来调用，但是习惯上建议使用类名调用。这个类的每一个对象存储的是静态变量在静态区中的地址，所以这些对象是共享这个静态变量的。也因此，只要有一个对象对这个静态变量的值做了改变，其他对象获取的值也会改变。---如果一些对象共享了某个特征，可以设置为静态的；如果每一个对象的某个属性各不一样，那么使用非静态的。

类加载到了方法区中的静态常量池（存储类信息），类只加载一次，不再移除。除了核心类库以外，其他的类在第一次使用的时候才加载

静态变量只能定义在类内，不能定义到方法或者代码块中

11.重写与重载的区别是什么?

重载：在同一个类中存在方法名一致而参数列表不同的方法，构成了重载。重载只和方法签名有关，和修饰符、返回值类型以及异常没有关系。重载是一种编译时多态。

重写：在父子类中存在了方法签名完全一致的非静态方法，构成了重写。在方法重写的时候，子类方法的权限修饰符的范围要比父类权限修饰符的范围大，至少保持一致。如果父类方法的返回值是基本类型/void/最终类，那么子类的返回值类型要保持一致。如果父类方法的返回值类型是引用类型，子类方法的返回值类型要是父类方法返回值类型的子类。最后，子类重写的方法的异常范围不能超过父类方法的异常的范围。重写是一种运行时多态。

day08

1.**静态方法**

static修饰的方法就叫静态方法/类方法。随着类的加载而加载到方法区中的静态区，只存储在了静态区没有执行。在方法被调用的时候，到栈内存中执行。静态方法先于对象出现的，意味着可以通过类名来调用，也可以通过对象来调用的

静态方法可以重载吗？---可以

静态方法可以被继承吗？---可以

静态方法可以被重写吗？---不能

在父子类中可以存在方法签名一致的静态方法吗？---可以---如果父子类中存在了方法签名一致的方法，要么都是静态方法，要么都是非静态方法

2.**静态代码块**

用static修饰的代码块---只在类加载的时候执行一次---在类第一次真正使用（创建对象或者调用方法）的时候执行一次

执行顺序：先加载父类---父类静态，然后加载子类---子类静态，其次创建父类对象---父类的构造，最后创建子类---子类的构造

3.**final**

修饰数据、方法、类

final修饰的**数据**---常量---值不可改变---对于基本类型而言指的是实际值不可变，对于引用类型而言，指的是地址不可改变，而属性值可以改变

final修饰的**方法**---最终方法---不能够被重写---可以被重载，能被继承

final修饰的**类**---最终类---不能被继承---System

父类的所有的数据和方法子类都可以继承

在子类继承父类的时候，父类的什么东西对子类不可见？---被隐藏

private、代码块、构造函数

4.**abstract**

如果子类对父类的某一个或者多个方法做了重写，父类中对应的方法没有定义方法体的必要，那么就可以省略这个方法的方法体，用abstract关键字把这个方法定义为抽象方法---抽象方法所在的类一定是抽象类，但是抽象类中不一定有抽象方法

抽象类不允许实例化---不能用Java代码实例化，底层是用C语言来构建的这个对象

子类继承抽象类之后必须重写其中的抽象方法，除非这个子类本身也是抽象类

抽象方法可以进行重载吗？---可以

能够用static/final/private修饰抽象方法吗？--- 不能

抽象方法可以是默认的权限修饰符吗？---可以，要求父子类同包

抽象方法所在的类一定是抽象类，一定不是最终类

**5.接口**

用interface来定义接口，用implements关键字让类和接口之间差生联系，这种联系称之为实现。

接口不能实例化，也不能定义构造方法

Java中支持单继承，多实现---类和类之间是单继承，类与接口之间是多实现的---一个类可以实现多个接口

多实现不能提高代码的复用性，反而导致实现的时候出现方法的混乱---接口不是为了提高复用性，而是为了统一方法结构，方便同一的调用

接口可以继承接口，而且是多继承

在JDK1.7及其以前，接口中只能定义抽象方法

接口中的方法默认用public abstract修饰

接口中的属性默认用public static final修饰

**6.接口与抽象类的区别**

抽象类---不能被实例化，但是有构造方法，抽象类中可以定义一切属性和方法；子类在继承了抽象类之后必须重写抽象类中的所有抽象方法，除非这个子类也是抽象类。抽象类一定不是最终类

4用implements关键字与类产生实现关系---多实现。接口可以继承多个接口---类实现了接口之后必须重写接口中的所有的抽象方法。接口不能定义构造方法。---接口中所有的方法都是抽象方法，默认用public abstract修饰。接口中定义的属性默认是用public static final修饰

**接口:不同类事物的共同的行为特征的抽象**

day09

**1.方法内部类**

定义在方法中的类.只能在定义它的方法内使用。方法内部类保证了一个或者多个逻辑只被指定的方法重复使用

方法内部类中不能定义静态变量和静态方法，但是可以定义静态常量。

方法内部类中可以使用外部类的成员属性和成员方法，也可以使用外部类的静态属性和静态方法

// 方法内部类不能使用当前方法中的变量，但是可以使用当前方法中的常量

// 只能用final或者是abstract修饰??

2.**成员内部类**

定义在类内方法外的类

Outer2.Inner2 oi2 = new Outer2().new Inner2();

成员内部类可以继承任何一个能被继承的类，也可以实现任何指定的接口。

不能定义静态变量和静态方法，只能定义静态常量

可以实现外部类的一切属性和方法。

调用外部类中的同名方法：Outer2.this.m();

在main方法内创建成员内部类对象：Outer2.Inner2 O2=new Outer2().new Inner2;

InnerDemo2

成员内部类中可以定义非静态方法和非静态属性，不能定义静态变量和静态方法，但是可以定义静态常量。要求能够 看懂类外创建内部类对象的方式

**3.静态内部类**（JDK底层中运用，掌握）

静态内部类中可以定义一切的实体方法和属性，包括静态方法和静态属性。

静态内部类只能使用外部类中的静态属性和静态方法。

可以在main方法中（类外）调用静态内部类中的静态方法。非静态方法只能通过对象来调用

表示调用外部类Outer3中静态内部类Inner3中的静态方法m

Outer3.Inner3.m2();

Outer3.Inner3 O3=new Outer3.Inner3;

InnerDemo3

A.B.m();---如果A是对象，就意味着B是A中的属性,那么m就是一个普通方法；如果A是类，B也是类，那么B就是A的静态内部类，m就是B中的静态方法；如果A是类，而B是对象，那么m就是B对象身上的普通方法

用static修饰的内部类

Outer3.Inner3 oi3 = new Outer3.Inner3();

**4.匿名内部类**---（掌握）

没有名字的内部类

任何一个可以被继承的类以及任何一个接口都可以存在匿名内部类的形式---匿名内部类本质上是继承了对应的类或者是实现了对应的接口

A a=new A(){};

匿名内部类本质上就是继承了对应的类，或者是实现了对应的接口。

匿名内部类中必须重写父类中的抽象方法。

只要一个类/接口可以被继承/实现，那么就可以存在匿名内部类的形式。除了最终类（final类）。

如果匿名内部类定义在了方法中，本质上就是一个方法内部类。

如果匿名内部类定义在了类中方法外，本质上就是一个成员内部类。

5.**包**

6.**垃圾分代回收机制**

回收针对的是堆内存。

将堆内存划分为了2块：新生代和老生代。新生代有划分为伊甸园区和幸存区。对象刚创建的时候是放在新生代的伊甸园区，在伊甸园区经过一次回收之后如果这个对象依然存在，则会挪到幸存区。如果在幸存区经过多次回收，会挪到老生代。老生代的扫描频率是远低于新生代。老生代的对象一旦回收，往往会导致程序的卡顿甚至于崩溃。

发生在新生代的回收---初代回收(minor gc --- garbage collector)

发生在老生代的回收---完全回收(full gc)

如果一个对象在新生代存放不开，会试图直接放入老生代---OutOfMemoryError

超过70%触动回收机制

Young Generation：Old Generation=9：1

Eden:Survivor=8:1

**7.类加载机制**

加载（将class文件转换为字节码放入内存）---校验（检查这段字节码是否合法）---准备（为所有的类变量分配空间并默认初始化）---解析（引用替换过程）---初始化（类变量赋值，静态代码块执行）

类加载器---ClassLoader

启动/引导类加载器 --- Bootstrap ClassLoader---加载Java的原生类

扩展类加载器 --- Extension ClassLoader --- 加载扩展类

应用类加载器 --- Application ClassLoader --- 加载自定义类

在Java中，加载类的时候双亲委派机制

day10

1.**Object**

Object是Java中的顶级父类---也是Java中唯一一个没有父类的类,,构造方法都是public修饰

2.**重要方法**

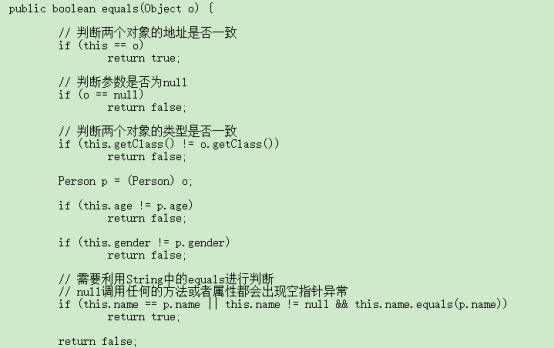
clone()---克隆出一个和原有对象的属性值一样的新对象---如果一个类产生的对象想要被克隆，那么这个类必须一个接口Cloneable---这个接口仅仅标志这个类的对象可以被克隆

finalize()---这个方法通知GC回收垃圾---但是GC不一定运行---System.gc()

getClass()---获取对象的实际类型而不是声明类型

toString()---将对象转化为字符串，默认输出的是地址---当打印一个对象的时候实际上是默认调用了这个对象的toString方法

equals(Object o)---比较两个对象是否相等---默认比较的两个对象的地址是否一致，但是实际开发过程中，往往会手动重写equals方法---地址、参数、类型、属性值---要求掌握两到三个属性的判断



3.**String**

 代表字符串的类，最终类---字符串是一个常量，定义好之后不能改变；因为字符串是常量，所以可以被共享。字符串在Java底层是以数组形式存储的。final（方法区中

**StringBuilder**是一个线程不安全的类，StringBuffer是一个线程安全的类

1. 在这方面运行速度快慢为：StringBuilder > StringBuffer > String

　　String最慢的原因：

　　String为字符串常量，而StringBuilder和StringBuffer均为字符串变量，即String对象一旦创建之后该对象是不可更改的，但后两者的对象是变量，是可以更改的。

在线程安全上，StringBuilder是线程不安全的，而StringBuffer是线程安全的

String：适用于少量的字符串操作的情况

　　StringBuilder：适用于单线程下在字符缓冲区进行大量操作的情况

　　StringBuffer：适用多线程下在字符缓冲区进行大量操作的情况

1.String str = “a”; str += “b”;5

-> str = new StringBuilder(str).append(“b”).toString();

2.String str = “a”;1

->String str = new String(“ab”);2

day12

1.String API

charAt:返回指定索引处的 char 值。

compareTo // compatrToIgnoreCase:按字典顺序比较两个字符串。//忽略病例差异.(大小写)

concat():将指定的字符串连接到该字符串的末尾。底层是基于字符数组的复制。

contains:当且仅当此字符串包含指定的char值序列时才返回true。

startsWith() // endsWith()测试此字符串是否以指定的前缀开始/后缀结尾。底层倒叙遍历

equals(),将此字符串与指定对象进行比较。(已经重写，底层实际上是按位比较)

qualsIgnoreCase()将此 String与其他 String比较，忽略案例注意事项

getBytes()使用平台的默认字符集将此 String编码为字节序列，将结果存储到新的字节数组

中。

hashCode--哈希码--根据哈希散列算法算出来的整数值。

[intern](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/String.html" \l "intern())()   返回字符串对象的规范化表示形式。//直接将常量池的地址返回。

[indexOf](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/String.html" \l "indexOf(int))(int ch)  返回指定字符在此字符串中第一次出现处的索引。

[isEmpty](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/String.html" \l "isEmpty())()   当且仅当 [length()](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/String.html" \l "length()) 为 0 时返回 true

[replace](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/String.html" \l "replace(char, char))(char oldChar, char newChar) 返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有oldChar 得到的。

[split](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/String.html" \l "split(java.lang.String))([String](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/String.html) regex)   根据给定[正则表达式](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/regex/Pattern.html" \l "sum)的匹配拆分此字符串

[substring](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/String.html" \l "substring(int))(int beginIndex)   返回一个新的字符串，它是此字符串的一个子字符串。

[toCharArray](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/String.html" \l "toCharArray())()    将此字符串转换为一个新的字符数组

[trim](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/String.html" \l "trim())()    返回字符串的副本，忽略前导空白和尾部空白。

2.正则表达式:

matches("");

捕获组 [\\n](\\\\n) 从捕获组左半边（出现的位置开始计算） 判断叠字

str.replaceAll(“(.)\\1+”, “$1”);

**[split](" \l "split-java.lang.String-)([String](mk:@MSITStore:D:\\jdk 1.7_X64\\jdk+api+1.8_中文版.CHM::/java/lang/../../java/lang/String.html) regex)**:将此字符串分割为给定的 [regular expression的](" \l "sum)匹配。

day13

1.包装类:8大+Void

Void本身是一个最终类，不能创建对象 必须返回,并且只能是null.

将基本类型转化为对应的引用类型的对象---封箱

将基本类型直接赋值给引用类型---自动封箱，底层实际是调用了对应类身上的valueOf方法

将引用类型直接赋值给基本类型---自动拆箱，底层实际上调用了对应对象身上的\*\*\*Value方法

[parseInt](" \l "parseInt-java.lang.String-)([String](mk:@MSITStore:D:\\jdk 1.7_X64\\jdk+api+1.8_中文版.CHM::/java/lang/../../java/lang/String.html) s) --将字符串参数解析为带符号的十进制整数

2.isNan:和任何数据都不相等，包括自己本身。

3.数学类

Math-最终类-构造函数私有化

Math.E--->2.7182818...

Math.ceil(3.2)-->向上取整4

Math.floor(3.9)-->向下取整3

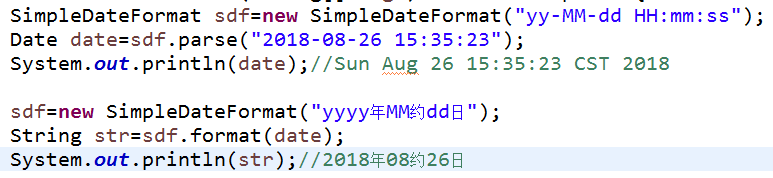
Math.round(3.6)-->四舍五入4

Math.pow(2,3)-->次方8

Math.random()-->[0,1]的随机小数

BigDecimal-用于精确运算,要求参数必须是字符串,存储比long类型还大的

4.日期类



Date--

 SimpleDateFormat

parse-将字符串转化为日期

 format-将日期转化为字符串

Calendar--

5.JDK8的部分内容

从JDK1.8开始，允许在接口中定义的普通的实体方法，但是必须用default来修饰。

也允许在接口中定义静态方法（不能用default）

**Lambda表达式**(参数) -> {代码块};

重写接口中的抽象方法---要求这个接口只能有一个抽象方法。参数类型可以省略，Java会根据程序的上文自动推导。任意一个Lambda表达式都不能用Object来接住。

如果接口中只有一个抽象方法，这个接口称之为函数式接口

非显式常量：在JDK1.8中，方法内部类++++++++使用当前方法中的数据的时候，这个数据可以不用显式声明为常量

**6.时间包**

对时间体系重新划分，划分出来一个全新的时间包---java.time包

**Java -time-**

**LocalDate**只含日期而不含时间

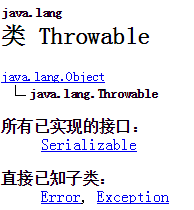
**LocalTime**只含时间而不含日期,**最小单位到纳秒**

[now](" \l "now--)()  从默认时区的系统时钟获取当前日期。

day14

**1.异常概述**

异常:是Java中一套用于问题的反馈和处理的机制。



编译时异常必须在出现之后才能捕获，但是运行时可以随意抛出随意捕获

**2.throw与throws的比较**

throw是语句抛出一个异常，一般是在代码块的内部，当程序出现某种逻辑错误时由程序员主动抛出某种特定类型的异常

throws是方法可能抛出异常的声明。(用在声明方法时，表示该方法可能要抛出异常)

1、throws出现在方法函数头；而throw出现在函数体。

2、throws表示出现异常的一种可能性，并不一定会发生这些异常；throw则是抛出了异常，执行throw则一定抛出了某种异常对象。

3、两者都是消极处理异常的方式（这里的消极并不是说这种方式不好），只是抛出或者可能抛出异常，但是不会由函数去处理异常，真正的处理异常由函数的上层调用处理。

**3.运行时异常和编译时异常**

**运行时异常,**这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。运行时异常的特点是Java编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用try-catch语句捕获它，也没有用throws子句声明抛出它，也会编译通过。

**编译时异常,**是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。

**异常处理机制为：抛出异常，捕捉异常。**

**抛出异常**：当一个方法出现错误引发异常时，方法创建异常对象并交付运行时系统，异常对象中包含了异常类型和异常出现时的程序状态等异常信息。运行时系统负责寻找处置异常的代码并执行。

**捕获异常**：在方法抛出异常之后，运行时系统将转为寻找合适的异常处理器（exception handler）。潜在的异常处理器是异常发生时依次存留在调用栈中的方法的集合。当异常处理器所能处理的异常类型与方法抛出的异常类型相符时，即为合适 的异常处理器。运行时系统从发生异常的方法开始，依次回查调用栈中的方法，直至找到含有合适异常处理器的方法并执行。当运行时系统遍历调用栈而未找到合适 的异常处理器，则运行时系统终止。同时，意味着Java程序的终止。

**4.断言:**

根据已有条件对结果做准确性的判断---断言

在Java中，断言没有默认开启

assert 断言条件 : 提示信息;-ea--enable assertion

在文本中，javac AssertDemo.java---------java -ea AssertDemo

**5.内部接口**

类中可以定义类，类中可以定义接口，接口中可以定义接口，接口中可以定义类

类中定义的接口以及接口中定义的类和接口都是静态的。

day15

**1.Collection<E>:**

存储多个数据的容器---大小不固定

Collection<E>,Java中集合的顶级接口。<E>---泛型---用于定义存储的元素类型。---由于泛型的限制，要求集合中只能存储引用类型的数据。---Collection<Integer>

Collection<String> c;  --- c的数据类型是Collection, c的元素类型是String

**2.List-列表**

保证元素的存储顺序---有序；有下标--存储重复的元素---List是**有序可重复**的集合

**3.ArrayList**

底层用数组存储数据。内存连续的。默认初始容量为10，每次默认在上一次的长度的基础上增加一半，底层是基于了右移运算来实现的。 10 -> 15 -> 22。是一个线程不安全的集合。便于查询，不便于增删

右移几位就是除以2的几次方。

作业:用数组实现一个山寨版的ArrayList

**4.LinkedList**

底层是基于链表（节点---Node）来实现的。内存是不连续的。没有初始容量，不需要考虑扩容问题。是一个线程不安全的集合。便于增删元素，不便于查询元素

扩展：用Node节点，然后利用node节点实现山寨版的LinkedList

**5.Vector**

向量---基于数组来实现的。内存空间连续的。默认初始容量是10，默认每次容量翻倍。基于三元运算来实现。是一个线程**安全**的集合。

**6.Stack**

栈。是Vector的子类。遵循先进后出的原则。向栈中放入元素---入栈/压栈，从栈中取出元素---出栈/弹栈。最先放入栈中的元素---栈底元素，最后放入栈中的元素---栈顶元素

底层用的数组，表层用的Enumeration枚举

用数组/Vector实现最小栈---尽量使时间复杂度最小

day16

**1.Set---散列集合**

元素不重复。

HashSet---线程不安全

存储机制：底层是用的一个数组来存储的数据，这个数组的默认大小是16。数组的每一个位置称为一个桶(bucket)，每一个桶中维系了一个链式栈结构。存放元素的时候会先计算元素的哈希码，然后将哈希码进行二次运算，根据运算的结果放入对应的桶中。如果某个元素要存放的桶中已经有了元素，会先拿这个元素和桶中的元素依次比较，如果每一个比较都不相等，则会插到这个链表的首位。根据加载因子的值确定桶的使用数量，加载因子的默认值是0.75f，当使用的桶的数量超过了（总数量\*加载因子)的值的时候，则会发生扩容操作，每次默认增加一倍。扩容之后，原来的所有的元素都要进行重新分布的操作（rehash）。

如果初始容量比较小，则在初期会频繁发生扩容和rehash操作；如果初始容量过大会导致内存的浪费。

如果加载因子较小，会导致扩容和rehash操作频繁发生，同时也会导致内存的大量的浪费。如果加载因子过大，会导致每个桶中的链表的长度过长，而链表长度过长会导致查询以及增删操作的效率都会降低。如果加载因子为1，则意味着不会扩容。

**2.Queue---队列**

先进先出(FIFO)。

[element](" \l "element--)() 检索，但不删除，这个队列的头。若队列为空，则返回NoSuchElementException

[peek](" \l "peek--)()  检索但不删除此队列的头，如果此队列为空，则返回 null 。

**3.Iterator**

迭代器---本质上是通过指针的挪动来获取每一个元素。在遍历的时候，会先复制原集合，然后一次给每一个元素一个标记，如果这个元素被移除，则标记为false，反之标记为true，在遍历完成之后，会去比对原集合，如果标记为false，则将该元素从原集合中移除---可以it.remove移除当前在迭代的元素,在迭代过程中，不允许直接增删原集合中的元素

Iterable---允许实现这个接口的类的对象能够被增强for循环进行遍历。---增强for循环的遍历过程实际上是迭代遍历的过程。---增强for循环是JDK1.5的特性之一

**4.Collections**

是操作集合的工具类。---提供了大量的操作集合的静态方法

**5.Comparator**

比较器---需要重写compare方法，在compare方法中指定比较规则，然后根据返回值的正负来确定两个参数的大小。如果返回了一个正数，说明第一个参数比较大，应该排在后边；如果返回一个负数，说明第一个参数比较小，应该排在前边

在排序的时候如果没有指定比较规则，要求集合中的元素实现Comparable。

**6.泛型**

参数化类型---ParameterizedType---JDK1.5的特性之一

List list = new ArrayList();

将泛型替换为具体类型的过程---泛型的擦除--发生在编译期

能看懂泛型的通配符以及对应的泛型的继承

**6.Comparable**

该接口对实现它的每个类的对象强加一个整体排序。 这个排序被称为类的*自然排序*，类的 compareTo方法被称为其*自然比较方法* 。

day18

**1.Map<K,V>---映射**

两组变量---键，值。每一个键必然对应一个值，键不可重复---一个映射实际上是由多个键值对组成的。

键值对所对应的接口---Entry，它是Map中的内部接口---每一个键值对都是一个Entry对象，那么实际上Map是由多个Entry对象来组成的

Map是集合吗？---不是，是集合框架的一员。

集合框架：和集合、数组、映射相关的接口和类都是集合框架的成员 ---Map/Collection/Collections/Iterator/Comparator/Comparable/Arrays

Iterable不是

如何遍历一个映射？

1. 先获取到映射中的所有的键，然后遍历这些键，再根据键来获取值

2. 将映射中所有的键值对放入到一个集合中

方法一 通过Map.entrySet遍历key和value 在for-each循环中使用entries来遍历,这是最常见的并且在大多数情况下也是最可取的遍历方式。在键值都需要时使用。

for (Map.Entry<Integer, Integer> entry : **map.entrySet())**{

    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());

}

注意：for-each循环在java 5中被引入所以该方法只能应用于java 5或更高的版本中。如果你遍历的是一个空的map对象，for-each循环将抛出NullPointerException，因此在遍历前你总是应该检查空引用。

 方法三 通过Map.entrySet使用iterator遍历key和value 使用Iterator遍历

Iterator<Map.Entry<Integer, Integer>> entries = map.entrySet().**iterator**();

while (entries.hasNext()) {

    Map.Entry<Integer, Integer> entry = entries.next();

    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());

}

你也可以在keySet和values上应用同样的方法。该种方式看起来冗余却有其优点所在。首先，在老版本java中这是惟一遍历map的方式。另一个好处是，你可以在遍历时调用iterator.remove()来删除entries，另两个方法则不能。根据javadoc的说明，如果在for-each遍历中尝试使用此方法，结果是不可预测的。

从性能方面看，该方法类同于for-each遍历（即方法二）的性能。

方法二 通过Map.values()遍历value,无法遍历key;通过Map.keySet()key,无法遍历value

在for-each循环中遍历keys或values。

如果只需要map中的键或者值，你可以通过keySet或values来实现遍历，而不是用entrySet。

//遍历map中的键

for (Integer key : **map.keySet()**) {

    System.out.println("Key = " + key);   }

//遍历map中的值

for (Integer value : **map.values())** {

    System.out.println("Value = " + value);   }

该方法比entrySet遍历在性能上稍好（快了10%），而且代码更加干净。

方法四、通过Map.KeySet遍历key和value 通过键找值遍历（效率低）

for (Integer key : **map.keySet()**) {

String value=map.get(key);

}

作为方法一的替代，这个代码看上去更加干净；但实际上它相当慢且无效率。因为从键取值是耗时的操作（与方法一相比，在不同的Map实现中该方法慢了20%~200%）。如果你安装了FindBugs，它会做出检查并警告你关于哪些是低效率的遍历。所以尽量避免使用。

总结

如果仅需要键(keys)或值(values)使用方法二。如果你使用的语言版本低于java 5，或是打算在遍历时删除entries，必须使用方法三。否则使用方法一(键值都要)。

**HashMap与Hashtable的区别**

HashMap:允许键或者值为null。默认初始容量是16，加载因子是0.75f，默认增加一倍。是一个异步式线程不安全的映射考虑效率用，考虑安全也不用Hashtable,而是ConcurrentHashMap--异步式线程安全的映射。

Hashtable:不允许键或者值为null。默认初始容量是11，加载因子是0.75f。是一个同步式线程安全的映射---仅仅在笔试中出现

ConcurrentHashMap---异步式线程安全的映射

HashSet是基于HashMap

**2.File**

boolean createNewFile();创建文件

boolean mkdir() // mkdirs() 创建目录

boolean isFile()判断是文件

boolean isDirectory()判断是目录

fs[] listFiles()获取子文件和子目录

day19

**1.IO流**

根据数据的传输方向：输入流、输出流

根据数据的传输形式：字符流、字节流

四个基本流都是抽象类

练习:复制TXT文件

**2.缓冲流**

在构建BufferedReader对象的时候需要传入一个Reader对象，底层读取数据靠的是传入的Reader对象，BufferedReader提供了缓冲区。---这种方式称之为装饰设计模式???

注意：字符流只能读取纯字符类的文件  .txt  .java .html

注意：字符流不能读取Office组件---POI能够读取

**3.字节流**

复制一个压缩文件

**4.合并流**

SequenceInputStream --- 将多个输入流合并为一个输入流进行操作---只有输入没有输出---合并流的构建需要一个Enumeration（本质上是一个迭代器，Java中最古老的迭代器）对象，需要利用Vector来获取Enumeration对象---elements()

hashtable是第一个可以用的映射。

day20

**1.转换流**

转换输出流OutputStreamWriter：将字符流转化为字节流。

转换输入流InputStreamReader：将字节流转化为字符流

都是以字节形式来读取或者写出数据，表现形式都是以字符形式来展现---转换流都是字符流

**2.系统流/标准流**

系统流都是字节流.本质上是静态对象---一旦关闭就无法使用

System.in 标准输入流

System.out 标准输出流

System.err 标准错误流

**3.打印流**

PrintStream/PrintWriter--- 能够便捷的向外输出数据

**4.序列化/反序列化流**

将对象进行存储的过程就称之为序列化 -将对象保存到硬盘上就叫持久化 -将对象还原回来的过程就称之为反序列化

1. 一个对象如果想要被序列化，那么要求这个对象对应的类必须实现一个接口---Serailizable --- 仅仅用于标志实现这个接口的类的对象允许被序列化

2. 被**static/transient**修饰的属性不能被序列化

3. 需要在类中手动指定一个属性---**serialVersionUID**---版本号。当一个对象序列化出去的时候会携带这个类的版本号，在反序列化的时候会检查对象携带的版本号和类中定义的版本号是否一致，如果一致则允许被反序列化。如果一个类中没有手动指定版本号，那么Java在编译的时候会根据当前类中的属性和方法自动计算产生一个版本号。也就意味着类中的属性或者方法产生变动，版本号就会改变。---因此需要手动指定，一旦手动指定就不再变化了。---版本号默认用private static final long 修饰限定

4. 在Java中数组和集合中的元素以及大部分的映射不能随着集合或者映射一起序列化

序列化和反序列化的加密机制---盒子---锁---钥匙---生产---了解即可

创建了一个序列化流对象:

ObjectOutputStream oos=new ObjectOutputStream(new FileOutputStream("D:\\p.data"));

**5.properties**

继承了Hashtable,是一个可以持久化的映射。键或者值只能是字符串。必须要存储到.properties文件中---Properties文件的默认编码是西欧编码，因此不能存储中文---一旦放入中文就会出现乱码或者是转化为对应的Unicode编码,Properties经常作为配置文件使用

**6.JDK1.5的部分特性**

自动封箱拆箱、增强for循环、静态导入、可变参数、枚举、泛型、反射、内省、动态代理、注解

**7.静态导入/静态导包**

import static 包名.类名.方法名; --- 表示导入指定包中的指定的静态方法

优点：能够一定程度的提高使用效率

缺点：书写不便；降低程序的可阅读性；如果在本类中存在了方法签名一致的静态方法，会导致静态导入无效

**8.可变参数**

用...定义一个可变参数---可变参数本质上就是一个数组，所以可以用操作数组的方式来操作一个可变参数。---可以传入任意多个参数也可以传入数组。---可以只传入一个参数也可以不传参数---对参数个数不做限定

可变参数**只能定义一个**，并且必须定义到**参数列表的末尾**

**9.枚举**

取值情况相对固定并且能够一一列举---枚举

用**enum**定义枚举类。枚举常量之间用 , 隔开---**枚举常量必须定义在枚举类的首行**。

枚举类中的**构造方法默认是私有的只能是私有的**---构造方法可以重载

枚举类中**可以定义一切的方法和属性，甚至包括抽象方法**

在Java中，所有的枚举的顶级父类是**java.lang.Enum**

switch-case的()需要一个表达式，表达式的值可以是byte/short/char/int, **jdk1.7允许使用String**， jdk1.5开始允许使用**枚举常量**

枚举常亮要求必须定义在枚举类的首行

public static final Season Spring=new Season();

**10测试---单元测试**

不是Java原生提供的而是第三方厂商提供的现在被广泛应用的测试手段

@Test ---在要测试的方法上添加此注解 --- 在使用的时候要求 “三无”：**无参无返回值非静态**

@Before---一般用于完成一些初始化的操作，@After---用于善后

**11Debug---调试模式**

day21

**1.线程**

进程：计算机中执行的任务---服务一般没有界面

线程：进程中的一个小任务  QQ/微信、下载软件、记事本、JVM。

注意：计算机的每一个核在每一个时刻都只能处理一个进程中的一个线程---宏观上是并行，微观上是串行

扩展：Windows中默认是用一个核处理，Linux下是有几个核就用几个核

**2.定义线程**

1. 继承Thread，重写run方法，将逻辑放到run方法中，开启线程用的是start方法

2. **实现Runnable，重写run方法**，需要利用Thread对象来启动这个线程, t.start();

3. 实现Callable<T>，重写call方法 --- 现阶段知道即可

**3.多线程的并发安全问题**

因为多个线程之间相互抢占而导致出现不符合常理的现象就称之为多线程的并发安全问题

线程的执行是相互抢占资源的。---谁抢到谁就执行。

线程的抢占是发生在线程执行的每一步过程中。

同步代码块---synchronized---意味着一个线程在执行了其他线程不能执行---需要锁对象---锁对象要求被所有的线程都认识：共享资源、类的字节码、this

同步：一段逻辑在一个时间段内只允许一个线程执行

异步：一段逻辑在一个时间段内只允许多个线程执行

同步一定安全---是

不安全一定异步---是

安全不一定同步---宏观角度---微观上安全一定同步

**4.死锁**

由于锁导致多个线程相互等着致使程序无法继续往下执行

导致的原因：多线程、较多的共享资源、锁的嵌套

避免死锁的方法：减少线程数量、减少共享资源、统一锁对象、避免锁嵌套

**5.等待唤醒机制**

利用的wait和notify来控制线程的执行顺序---等待唤醒机制

线程在等待的时候是去线程池中等待的。线程池本质上是一个存储线程的队列。---遵循先进先出的原则

如果出现了多个线程，只唤醒一个线程就不够用了，所以用notifyAll唤醒所有的线程

注意：等待唤醒机制必须结合锁对象来使用---等待唤醒和锁是对应的

**总结：sleep和wait有什么区别？**

wait():,可以不指定等待时间，也可以指定等待时间。如果没有指定等待时间需要手动唤醒。wait必须结合锁使用。释放执行权，释放锁。它是一个成员方法。而当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备,获取对象锁进入运行状态。

1、这两个方法来自**不同的类**分别是，sleep来自Thread类，和wait来自Object类。

sleep是Thread的静态类方法，谁调用的谁去睡觉，即使在a线程里调用了b的sleep方法，实际上还是a去睡觉，要让b线程睡觉要在b的代码中调用sleep。

2、最主要是sleep方法没有**释放锁**，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

sleep不出让系统资源；wait是进入线程等待池等待，出让系统资源，其他线程可以占用CPU。一般wait不会加时间限制，因为如果wait线程的运行资源不够，再出来也没用，要等待其他线程调用notify/notifyAll唤醒等待池中的所有线程，才会进入就绪队列等待OS分配系统资源。sleep(milliseconds)可以用时间指定使它自动唤醒过来，如果时间不到只能调用interrupt()强行打断。

Thread.Sleep(0)的作用是“触发操作系统立刻重新进行一次CPU竞争”。

3、**使用范围**：wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用

   synchronized(x){

      x.notify()

     //或者wait() }

4、sleep必须捕获**异常**，而wait，notify和notifyAll不需要捕获异常

**6.守护线程**

Java分为两种线程：用户线程和守护线程

所谓守护线程是指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾回收(GC)线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因 此，当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。反过来说，只要任何非守护线程还在运行，程序就不会终止。

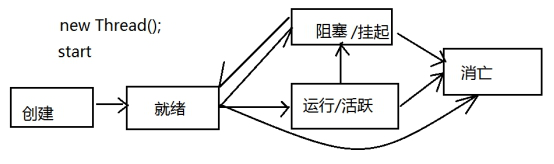
守护其他线程，被守护的线程结束的时候，守护线程无论完成与否都要结束

一个线程要么是守护线程要么是被守护的线程---当最后一个被守护的线程结束才会导致所有的守护线程结束.

**7.线程的优先级**

优先级一共有1-10这十个等级。理论上数字越大优先级越高，也就意味着线程抢到资源的几率就越大,(就越可能被cpu调度执行该线程,被线程调度器调用的几率就越高), 实际中相邻两个优先级的差别很不明显。至少相差5个优先级及以上才会相对明显.

**8.线程的状态**



**8.单例模式**

在全局过程中只有一个实例,是24种设计模式之一

模式：处理……的套路

设计模式：……的套路

1.饿汉式：对象在定义的时候就已经初始化

2.懒汉式：在真正使用这个对象的时候才初始化

饿汉式会增加加载过程，懒汉式能够避免这个问题。(饿汉式即使没有用到这个对象也要初始化，降低加载速率。)

**懒汉式会出现线程安全问题，饿汉式不会出现线程安全问题**

3.同步锁方式：在获取对象的方法身上加上synchronized

 扩展：掌握单例模式的七种方式

day22

**1.网络编程**

基于网络进行数据传输---IO

网络模型：

物理层 数据链路层 网络层--IP 传输层---UDP/TCP 会话层 表示层 应用层 --- http,ftp,POP3,SMTP...

IP地址：IPv4---用了4组数字表示一个地址，每一组数字的大小是0-255---IPv6

cmd--ipconfig

动态IP--节省成本。

端口号：0-65535，0-1024在开发过程中不要使用

域名：每一级域名之间用 . 隔开

DNS解析服务器：将域名和地址进行对应

SocketAddress抽象类

InetSocketAddress

**2.UDP**

基于流的。不建立连接，不可靠，传输速率相对较快。需要对数据进行封包，每个包不超过64K。---DatagramSocket, DatagramPacket --- 适用于要求传输速度不要求可靠性的场景， 视频聊天

发送端:

1. 创建套接字对象

2. 准备数据包，将数据放到数据包中并且要绑定发送的地址

3. 发送数据

4. 关流

接收端：

1. 创建套接字对象，绑定要监听的接收端口

2. 准备数据包

3. 接收数据

4. 关流

5. 将数据从数据包进行解析

练习：单人聊天---一个线程表示发送端，另一个线程表示接收端

255.255.255.255在UDP中表示的是一个广播地址

**3.TCP**

基于流的。需要建立连接，经过三次握手，可靠，传输速率相对较低。理论上不对数据的大小进行限制。---适用于要求可靠性而不要求传输速率的场景  文件传输、文字聊天

客户端：---Socket

1. 创建客户端的套接字对象

2. 发起连接，绑定地址

3. 获取输出流对象，写出数据

4. 通知服务器端数据写完

5. 关流

服务器端：---ServerSocket

1. 创建服务器端的套接字对象，并且绑定监听的端口号

2. 接收连接，获取到一个Socket对象

3. 获取输入流，读取数据

4. 通知客户端数据已经读完

5. 关流

注意：receive/connect/accept/read/write都会产生阻塞

扩展：BIO --- Blocking IO --- 同步式阻塞式IO

NIO --- New IO --- NonBlocking IO --- 同步式非阻塞式IO --- JDK1.4

AIO --- Asynchronous IO --- 异步式非阻塞式IO --- jdk1.8

练习：文件的上传

**4.注解**

解释程序---给机器看的

用@interface来定义注解---jdk1.5出现的

@Override

@SuppressWarnings---压制警告

元注解：修饰注解的注解

@Target---限制注解的使用范围的--- 如果注解没有指定范围，则表示任意一个范围都可以使用

@Retention --- 限制注解出现的时间的,即定义注解的保留时间

@Documented --- 表示当前这个注解能够生成到文档中

@Inherited --- 表示此注解能够限制子类,允许子类继承父类的注解

day23

**1.反射**

**如何去获取一个Class对象？**

1. 通过类名.class方式来获取一个字节码对象

2. 通过对象.getClass()方式来获取这个对象的实际类型对应的字节码

3. 通过Class.forName(类的全路径名)方式来获取指定类的字节码

**获取实例对象**

1. 如果这个类中提供了无参构造，那么可以直接利用字节码对象.newInstance()来获取对应的实例对象 String

2. 如果想使用这个类中含参构造来创建对应的实例对象，那么需要先获取这个类中对应形式的构造函数，然后再调用newInstance(Object...)来创建实例对象 Integer

3. 如果在获取这个类中的属性/方法/构造方法的时候，这些属性/方法是非公有的，那么在执行之前需要进行暴力破解---setAccessiable

**2.JVM的一些运行参数**

**栈内存**(Stack)：执行代码块---计算---每一个线程独有

**堆内存**(Heap)：存储对象 --- 被所有线程共享的

新生代(Young Generation)：伊甸园区(eden)和幸存区(Survivor)

老生代(Old Generation)

**持久代**(Permanent)/**方法区**(Method Area)：存储---被所有线程所共享的

静态常量池：存储类信息 --- 字节码

运行时常量池：字面量(数值、字符、字符串、逻辑值和null)和自定义常量

**本地方法栈**(Native Stack)：执行本地方法（native）---每一个线程独有的

**寄存器/PC计数器**(programming counter)：用于存储和调度线程中的指令---每一个线程都都有一个PC计数器

所有的栈所占用的内存不能超过物理内存的1/3或者是2G

堆内存所占用的大小不能超过物理内存的1/4或者是2G

**3.垃圾收集算法**

**标记清除Mark-Sweep**：先找到无用的对象，然后将对象标记，最后清理标记的对象---清理相对较快，但是会产生内存的碎片化

**复制Copying**：将内存分为两块，使用其中的一块，如果要回收，就将不需要回收的对象挪到另一块去，然后重建要回收的这一块区域 --- 不会产生内存的碎片化，但是耗费内存并且耗费时间

**标记整理**Mark-Compact：既能保证清理速度又能保证内存不会产生碎片化

jdk1.7采取的回收器是G1回收，G1回收算法针对新生代和老生代采取不同的回收机制。

笔试题总结：

对于数组int[] x,y[]均已初始化，下面各项正确的是：BC

A. y[0] = x[0];

B. y[0] = x;

C. y[0][0] = x[0];

D. y[0][0] = x;

E. y[0] = x[0][0];

解析：用int类型的数组定义了一个一维数组x，二维数组Y[],答案A一维数组的元素不能直接赋值给二维数组的元素，二维数组的元素应该是一个一维数组，答案B，一维数组赋值给二维数组作为元素，答案C一维数组的元素可以赋值给二维数组的具体元素，答案D二维数组的具体元素只能是一个具体类型的值，不能是一个数组，答案E ，x是一个一维数组。

构造代码块

在类内方法外用{}括起来的一段代码---构造代码块/初始化代码块---在创建对象的时候先于构造方法执行

封装

体现形式：方法，属性的私有化---将类中的属性用private修饰，限制属性的使用范围，对外提供操作和获取这个的属性的公共方法，在这些方法中可以根据场景来限制属性的值，使之更符合场景要求；内部类

优点：提高代码的复用性，保证数据的合法性