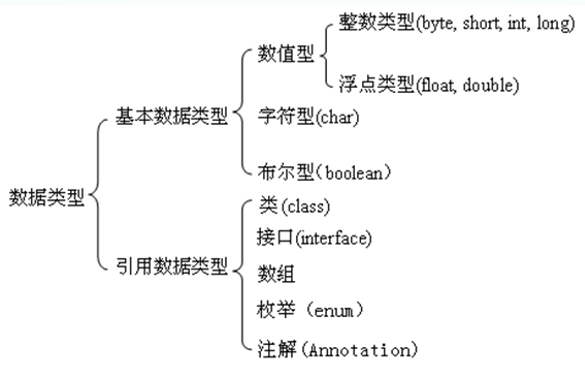
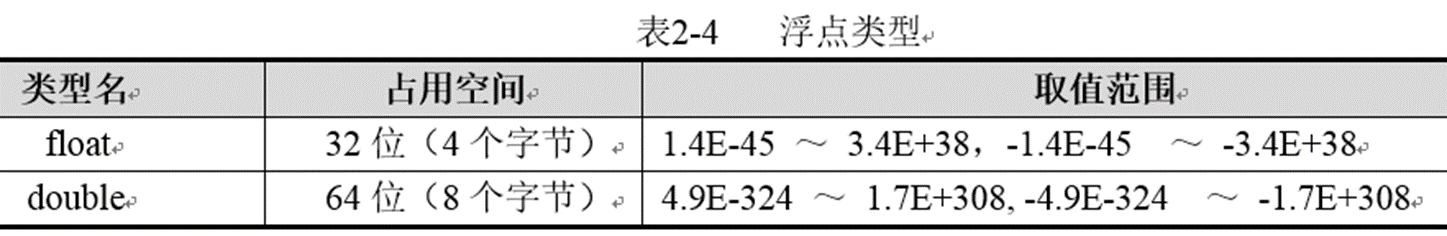
概念相关

二、Java编程基础

1. 标识符（其实就是类似变量名之类的）可以由大小写字母、数字、下划线以及美元符号$ 组成，但是数字无法作为开头！例如不合法的有：123user，class， 98.3， Hello World
2. 变量类型

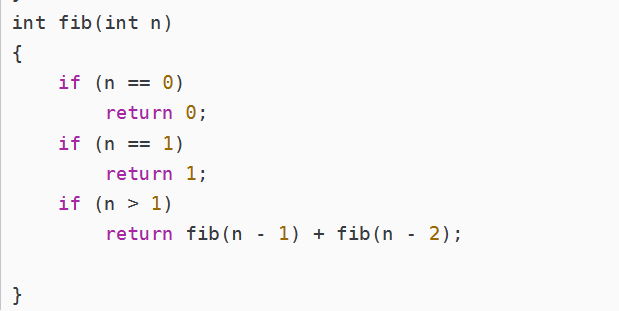






字符类型变量（char）：用于存储单一字符，每个char类型的字符变量占2字节；

1. 变量需要在它的作用范围内才可以被使用，这个作用范围称为变量的作用域。 变量一定会被定义在某一对大括号中，该大括号所包含的代码区域便是这个变量的作用域
2. 根据传入不同的参数确定调用特定重载的方法。方法的重载与返回值类型无关，它只有两个条件，一是方法名相同，二是参数个数或参数类型不相同。
3. 递归算法指的是一个方法内部调用自身的过程；必须要设置结束条件。



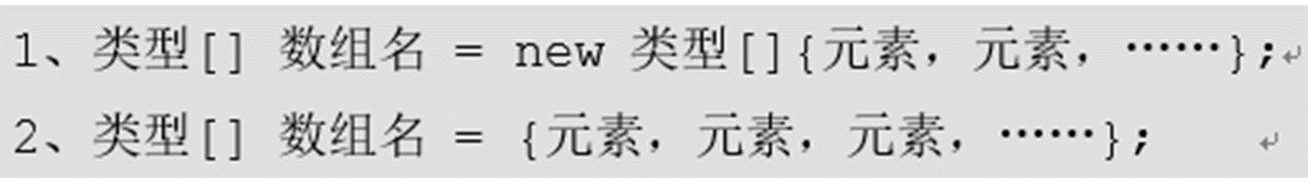
1. 数组定义：

在数组中存放任意类型的元素，同一个数组里存放的元素类型必须一致

int[] x;//声明一个int[]类型变量，提供数组的首地址

x = new int[100];//以x的首地址为始，顺延创建100元素的内存进行存储

数组的两种初始化：



数组排序：

第一步，从第一个元素开始，将相邻的两个元素依次进行比较，直到最后两个元素完成比较。如果前一个元素比后一个元素大，则交换它们的位置。整个过程完成后，数组中最后一个元素自然就是最大值，这样也就完成了第一轮比较。  
 第二步，除了最后一个元素，将剩余的元素继续进行两两比较，过程与第一步相似，这样就可以将数组中第二大的数放在了倒数第二个位置。  
 第三步，以此类推，持续对越来越少的元素重复上面的步骤，直到没有任何一对元素需要比较为止。

多维度数组定义以及初始化类似：

int[][] arr = new int[3][4];

int[][] arr = new int[3][];

int[][] arr = {{1,2},{3,4,5,6},{7,8,9}};

1. 面向对象（上）

逻辑线：

先定义一个类 --> 类的实例化（对象）以及对象的使用 --> 但是存在问题，部分成员变量需要限制或者保密 --> 所以需要进行类的封装

类默认调用无参的构造方法，但也可以进行有参的构造 --> 那么实际实例化过程中用户的需求不同，构造方法也是不同 --> 所以需要类构造方法的重载 --> 可是还有问题，如果类的构造方法的参数名和成员变量名相同该怎么办？--> 所以需要用this来特指成员变量或者方法

不同实例的部分属性可能是一样的，为了方便引入了static --> static 实现了实例化同一个类的所有成员变量的共享；实现了类本身的一些方法调用（因此是类的方法，不需要实例化后再调用方法）；实现了代码块的一次性执行，通常用于类static变量的初始化（因为一个类的不同对象的static变量共享，那么初始化一次就好了）

那么结合以上两条逻辑线（封装和静态），就可以实现类的单例模式（在类里面定义私有共享对象），在需要时利用方法将对象拿出来用即可

内部类为了更好封装且间接性实现多继承而引出：1. 成员内部类：需要将外部类先实例化再实例化内部类进行调用 2. 静态内部类：可以不创建外部类对象直接进行实例化内部类 3. 方法内部类：创建外部类对象并调用方法来实例化内部类，封装性极强。

概念：

1. 面向对象的思想中提出两个概念，即类和对象。面向对象的特点概括为封装性、继承性和多态性。
2. 类与对象的关系：类是对某一类事物的抽象描述；对象用于表示现实中该类事物的个体，是类的具象化。所以对象是类的实例，一个类可以对应多个对象。
3. 类中定义成员变量和成员方法，其中成员变量用于描述对象的特征，也被称作属性；成员方法用于描述对象的行为，可简称为方法。
4. 构造方法满足条件：方法名和类名相同；方法名的前面没有返回值类型的声明；方法中不能使用return语句返回一个值
5. 对象无法使用的情况：没有变量引用这个对象时，它将成为垃圾对象，不能再被使用；变量超出作用域变为垃圾对象；变量的值变为null（当一个变量的值为null时，则表示该变量不指向任何一个对象）
6. 静态变量被所有实例共享，可以使用“**类名**.变量名”的形式来访问。静态方法在被调用时可以不创建任何对象，所以是“**类名**.方法”。静态代码块只在类加载过程中执行一次。通常使用静态代码块来对类的成员变量进行初始化。
7. 面向对象（下）

逻辑线：

类可以继承，不需要重新构造类 --> 当然不只是添加自己的方法，也可以根据自身特性，重写父类方法 -->重写后的父类方法依然可以调用，利用关键词super（专门调用父类的变量，方法以及构造方法） --> 但是有些类，成员以及方法用于特殊情况，是不能被随意重写的（final类无法被继承，final方法无法被重写，final变量变成常量，仅能赋值一次） --> 另外有时候公司工作，上司只会提供他需要的一些功能，至于具体实现方法不明，因此需要利用抽象类进行定义父类。而且抽象类不可被直接实例化，需要先继承把抽象方法都实现了之后才能实例化-->因此如果这个类所有方法都是抽象的，Java提供了接口（Interface）-->Java提供了implements来实现接口中的所有方法（很令人兴奋的是，这个就可以避免不能多重限制的问题，可以实现多个接口，包括接口之间也能extends）

为了使得方法具有一定通用性，因此引入了类的多态 --> 因为方法的参数类型是固定唯一的，因此只能是父类参数类型，所以是利用父类变量来引用子类对象进行传递 --> 既然是用父类变量调用子类对象，那么只能用父类的方法。若想要用，需要进行强制转化（但是子类不可互相转化）

类的继承使得Java定义一个通用父类Object有了可能性，因此可以调用或者根据需要重写object的方法。

类的多态有另外的实现方法，可以使用内部类或者匿名内部类 --> 即不需要在外部定义子类来引用，而是直接在括号里面进行定义：1. 在main中定义子类实现，创建传入，这是内部类的写法 2. new一个父类，在父类里面实现抽象方法，这是匿名内部类的写法

异常：发现异常，代码需要继续执行 --> try...catch --> 但是想在发生异常之后的语句继续执行，因此需要finally, 同时在catch里面return出来 --> 每个人对异常的处理方法不同，因此源代码作者可以抛出相应的异常，让下一个人来写异常处理 利用throws关键字 -->在程序设计时，有时需要自己定义特殊异常。那么首先自定义异常继承于异常，然后再需要出现异常的地方抛出 new出来的自定义异常，使得读代码人进行修改。

概念：

1. 类的继承是指在一个现有类的基础上去构建一个新的类，构建出来的新类被称作子类，现有类被称作父类，子类会自动拥有父类所有可继承的属性和方法。如果想声明一个类继承另一个类，需要使用extends关键字。
2. 继承关系中，子类会自动继承父类中定义的方法，但有时在子类中需要对继承的方法进行一些修改，即对父类的方法进行重写。在子类中重写的方法需要和父类被重写的方法具有相同的方法名、参数列表以及返回值类型。
3. final关键字用于修饰类、变量和方法。被final修饰的类、变量和方法将具有以下特性：final修饰的类不能被继承；final修饰的方法不能被子类重写；final修饰的变量（成员变量和局部变量）是常量，只能赋值一次。
4. 抽象类是不可以被实例化的，因为抽象类中有可能包含抽象方法，抽象方法是没有方法体的，不可以被调用。如果想调用抽象类中定义的方法，则需要创建一个子类，在子类中将抽象类中的抽象方法进行实现。一个抽象类中的所有方法都是抽象的，则可以将这个类用另外一种方式来定义，即接口
5. 接口中定义的方法默认使用“public abstract”来修饰，即抽象方法。接口中的变量默认使用“public static final”来修饰，即全局常量。
6. 设计一个方法时，通常希望该方法具备一定的通用性。在同一个方法中，这种由于参数类型不同而导致执行效果各异的现象就是多态。 在Java中为了实现多态，允许使用一个父类类型的变量来引用一个子类类型的对象，根据被引用子类对象特征的不同，得到不同的运行结果。
7. Throwable有两个直接子类Error和Exception，其中Error代表程序中产生的错误，Exception代表程序中产生的异常。接下来就对这两个直接子类进行详细讲解。

Error类称为错误类，它表示Java运行时产生的系统内部错误或资源耗尽的错误，是比较严重的，仅靠修改程序本身是不能恢复执行的。举一个生活中的例子，在盖楼的过程中因偷工减料，导致大楼坍塌，这就相当于一个Error。使用java命令去运行一个不存在的类就会出现Error错误。

Exception类称为异常类，它表示程序本身可以处理的错误，在开发Java程序中进行的异常处理，都是针对Excption类及其子类。在Exception类的众多子类中有一个特殊的RuntimeException类，该类及其子类用于表示运行时异常。Exception类下所有其它的子类都用于表示编译时异常。

1. Java中允许在方法的后面使用throws关键字对外声明该方法有可能发生的异常，这样调用者在调用方法时，就明确地知道该方法有异常，并且必须在程序中对异常进行处理，否则编译无法通过。
2. 在实际开发中，经常会在程序编译时期产生一些异常，而这些异常必须要进行处理，这种异常被称为编译时期异常，也称为checked异常。另外有一种异常是在程序运行时期产生的，这种异常即使不编写异常处理代码，依然可以通过编译，因此我们称之为运行时异常，也称为unchecked异常。

在Java中，Exception类中除了RuntimeException类及其的子类都是编译时异常。编译时异常的特点是Java编译器会对其进行检查，如果出现异常就必须对异常进行处理，否则程序无法通过编译解决办法：1. 使用try…catch语句对异常进行捕获；2. 使用throws关键字声明抛出异常，调用者对其处理。

RuntimeException类及其子类都是运行时异常。运行时异常的特点是Java编译器不会对其进行检查，也就是说，当程序中出现这类异常时，即使没有使用try..catch语句捕获或使用throws关键字声明抛出，程序也能编译通过。运行时异常一般是由于程序中的逻辑错误引起的，在程序运行时无法恢复。比如通过数组的角标访问数组的元素时，如果超过了数组的最大角标(数组越界)，就会发生运行时异常。

1. 在Java中允许用户自定义异常，但自定义的异常类必须继承自Exception或其子类。throw关键字用于在方法中声明抛出异常的实例对象
2. 面向对象（下）

逻辑线：

为了使得代码可以交替并行运行，引入了多线程 --> 多线程的创建：1.extends Thread类创建，内部定义run方法，在主代码里进行start --> 但是这个由于java的单继承有限制，因此实现Runnable接口，作为Thread类的参数来创建进程 --> Runnable不仅可以解除单继承的限制，同时也能实现对同一个资源的处理-->若要删除线程，可以将其设置为后台线程。进程中若只有后台线程，进程结束。

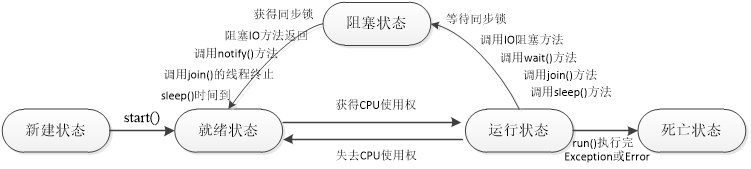
线程的生命周期：新建状态，就绪状态，执行状态，阻塞状态以及死亡状态

基于以上五种状态，首先定义线程优先使用cpu的级别（两种方式）--> 线程休眠，线程让步（区别于线程休眠，是让其变为就绪状态，重新调度） --> 线程插队

使用多线程时，会由于多个线程共用资源造成线程安全问题，因此需要在处理资源时，每一时刻只能有一个线程执行 --> 引出同步代码块 --> 定义一个lock对象，synchornized(lock):起初lock设置为1，进入一个线程后lock为0，其余无法进入，直至他出来之后设为1，其他线程才能进去，循环往复 --> 另外也可以通过同步方法实现：在方法前面使用synchronized关键字来修饰，被修饰的方法为同步方法，在某一时刻只允许一个线程访问 --> 但是同步锁也会出现死锁问题：两个线程在运行时都在等待对方的锁，这样便造成了程序的停滞，这种现象称为死锁。两个线程都需要对方所占用的锁，但是都无法释放自己所拥有的锁，于是这两个线程都处于了挂起状态，从而造成了死锁。

概念：

1. 每个独立执行的程序都可称之为一个进程，也就是“正在进行的程序”。对于一个CPU而言，在某个时间点只能运行一个程序，也就是说只能执行一个进程。在一个进程中还可以有多个执行单元同时运行，这些执行单元可以看做程序执行的一条条线索，被称为线程。
2. 单线程程序：代码都是按照调用顺序依次往下执行；多线程程序：多段程序代码交替运行的效果，则需要创建多个线程，彼此独立
3. 总结：实现Runnable接口相对于继承Thread类来说，有如下显著好处  
    1. 适合多个相同程序代码的线程去处理同一个资源的情况，把线程同程序代码、数据有效的分离，很好的体现了面向对象的设计思想。  
    2. 可以避免由于Java的单继承带来的局限性。在开发中经常碰到这样一种情况，就是使用一个已经继承了某一个类的子类创建线程，由于一个类不能同时有两个父类，所以不能用继承Thread类的方式，那么就只能采用实现Runnable接口的方式。
4. 线程整个生命周期可以分为五个阶段，分别是新建状态(New)、就绪状态(Runnable)、运行状态(Running)、阻塞状态(Blocked)和死亡状态(Terminated)，线程的不同状态表明了线程当前正在进行的活动。



新建状态(New)  
 创建一个线程对象后，该线程对象就处于新建状态，此时它不能运行，和其它Java对象一样，仅仅由Java虚拟机为其分配了内存，没有表现出任何线程的动态特征。

就绪状态(Runnable)  
 当线程对象调用了start()方法后，该线程就进入就绪状态（也称可运行状态）。处于就绪状态的线程位于可运行池中，此时它只是具备了运行的条件，能否获得CPU的使用权开始运行，还需要等待系统的调度。

运行状态(Running)  
 如果处于就绪状态的线程获得了CPU的使用权，开始执行run()方法中的线程执行体，则该线程处于运行状态。当一个线程启动后，它不可能一直处于运行状态(除非它的线程执行体足够短，瞬间就结束了)，当使用完系统分配的时间后，系统就会剥夺该线程占用的CPU资源，让其它线程获得执行的机会。需要注意的是，只有处于就绪状态的线程才可能转换到运行状态。

阻塞状态(Blocked)  
 一个正在执行的线程在某些特殊情况下，如执行耗时的输入/输出操作时，会放弃CPU的使用权，进入阻塞状态。线程进入阻塞状态后，就不能进入排队队列。只有当引起阻塞的原因被消除后，线程才可以转入就绪状态。  
阻塞状态的可能情形：

当线程试图获取某个对象的同步锁时，如果该锁被其它线程所持有，则当前线程会进入阻塞状态，如果想从阻塞状态进入就绪状态必须得获取到其它线程所持有的锁。

当线程调用了一个阻塞式的IO方法时，该线程就会进入阻塞状态，如果想进入就绪状态就必须要等到这个阻塞的IO方法返回。

当线程调用了某个对象的wait()方法时，也会使线程进入阻塞状态，如果想进入就绪状态就需要使用notify()方法唤醒该线程。

当线程调用了Thread的sleep(long millis)方法时，也会使线程进入阻塞状态，在这种情况下，只需等到线程睡眠的时间到了以后，线程就会自动进入就绪状态。

当在一个线程中调用了另一个线程的join()方法时，会使当前线程进入阻塞状态，在这种情况下，需要等到新加入的线程运行结束后才会结束阻塞状态，进入就绪状态。

死亡状态(Terminated)  
线程的run()方法正常执行完毕或者线程抛出一个未捕获的异常(Exception)、错误(Error)，线程就进入死亡状态。一旦进入死亡状态，线程将不再拥有运行的资格，也不能再转换到其它状态。

1. 线程休眠：人为地控制线程，使正在执行的线程暂停，将CPU让给别的线程，这时可以使用静态方法sleep(long millis)，该方法可以让当前正在执行的线程暂停一段时间，进入休眠等待状态。

线程让步：该方法和sleep()方法有点相似，都可以让当前正在运行的线程暂停，区别在于yield()方法不会阻塞该线程，它只是将线程转换成就绪状态，让系统的调度器重新调度一次。当某个线程调用yield()方法之后，只有与当前线程优先级相同或者更高的线程才能获得执行的机会。

线程插队：某个线程中调用其它线程的join()方法时，调用的线程将被阻塞，直到被join()方法加入的线程执行完成后它才会继续运行