**（一）**：JVM内存管理及垃圾回收：

程序计数器：这是一块比较小的内存，不在Ram上，而是直接划分在CPU上的，程序员无法直接操作它，它的作用是：JVM在解释字节码文件（.class）时，存储当前线程所执行的字节码的行号.多线程就是通过现程轮流切换而达到的，同一时刻，一个内核只能执行一个指令，所以，对于每一个程序来说，必须有一个计数器来记录程序的执行进度，这样，当现程恢复执行的时候，才能从正确的地方开始，所以，每个线程都必须有一个独立的程序计数器，这类计数器为线程私有的内存。如果一个线程正在执行一个Java方法，则计数器记录的是字节码的指令的地址，如果执行的一个Native方法，则计数器的记录为空，此内存区是唯一一个在Java规范中没有任何OutOfMemoryError情况的区域。

堆内存：是垃圾回收的主要区域，在32位系统上最大为2G，64位系统上无限制。可通过-Xms和-Xmx控制，-Xms为JVM启动时申请的最小Heap内存，-Xmx为JVM可申请的最大Heap内存。

 方法区：是所有线程共享的内存区域，用于存储已经被JVM加载的类信息、常量、静态变量等数据，一般来说，方法区属于持久代，也难怪Java规范将方法区描述为堆的一个逻辑部分，但是它不是堆。此处引入方法区中一个重要的概念：运行时常量池。主要用于存放在编译过程中产生的字面量（字面量简单理解就是常量）和引用。一般情况，常量的内存分配在编译期间就能确定，但不一定全是，有一些可能就是运行时也可将常量放入常量池中。

在JVM内存管理之外的一个内存区：直接内存。在JDK1.4中新加入类NIO类，引入了一种基于通道与缓冲区的I/O方式，它可以使用Native函数库直接分配堆外内存，即我们所说的直接内存，这样在某些场景中会提高程序的性能。

Native方法：native是一个计算机函数，一个Native Method（原生方法）就是一个Java调用非[Java](https://baike.baidu.com/item/Java/85979)代码的接口。方法的实现由非Java语言实现，比如C或C++。

垃圾回收：不需要进行垃圾回收的：程序计数器、JVM栈、本地方法栈。因为它们的生命周期是和线程同步的，随着线程的销毁，它们占用的内存会自动释放，所以只有方法区和堆需要进行GC。

垃圾回收算法：根搜索算法：从一个叫GC Roots的对象开始，向下搜索，如果一个对象不能到达GC Roots对象的时候，说明它已经不再被引用，即可被进行垃圾回收，如果类重写了finalize()方法，且没有被系统调用过，那么系统会调用一次finalize()方法，以完成最后的工作，在这期间，如果可以将对象重新与任何一个和GC Roots有引用的对象相关联，则该对象可以“重生”，如果不可以，那么就说明彻底可以被回收了。

引入了强、软、若、虚四种引用，被标记为这四种引用的对象，在GC时分别有不同的意义：

a> 强引用(Strong Reference).就是为刚被new出来的对象所加的引用，它的特点就是，永远不会被回收。

b> 软引用(Soft Reference).声明为软引用的类，是可被回收的对象，如果JVM内存并不紧张，这类对象可以不被回收，如果内存紧张，则会被回收。此处有一个问题，既然被引用为软引用的对象可以回收，为什么不去回收呢？其实我们知道，Java中是存在缓存机制的，就拿字面量缓存来说，有些时候，缓存的对象就是当前可有可无的，只是留在内存中如果还有需要，则不需要重新分配内存即可使用，因此，这些对象即可被引用为软引用，方便使用，提高程序性能。

c> 弱引用(Weak Reference).弱引用的对象就是一定需要进行垃圾回收的，不管内存是否紧张，当进行GC时，标记为弱引用的对象一定会被清理回收。

d> 虚引用(Phantom Reference).虚引用弱的可以忽略不计，JVM完全不会在乎虚引用，其唯一作用就是做一些跟踪记录，辅助finalize函数的使用。

需要回收的无用类需满足如下要求：1> 该类的所有实例对象都已经被回收。

  2> 加载该类的ClassLoader已经被回收。 3> 该类对应的反射类java.lang.Class对象没有被任何地方引用。

gc 只能清除在堆上分配的内存(纯java语言的所有对象都在堆上使用new分配内存)，而不能清除栈上分配的内存（当使用JNI技术时,可能会在栈上分配内存，例如java调用c程序，而该c程序使用malloc分配内存时）。因此，如果某些对象被分配了栈上的内存区域，那gc就管不着了，对栈上的对象进行内存回收就要靠finalize()。

编程习惯：

（1）避免在循环体中创建对象，即使该对象占用内存空间不大。

（2）尽量及时使对象符合垃圾回收标准。

（3）不要采用过深的继承层次。

（4）访问本地变量优于访问类中的变量。

**（二）**：Camparable接口和Comparator的区别：

一个类实现了Camparable接口则表明这个类的对象之间是可以相互比较的，这个类对象组成的集合就可以直接使用sort方法排序。用法：

在User中继承Comparable接口，重写compareTo方法，直接对User进行排序。

1. User[] users = **new** User[] { **new** User("a", 30), **new** User("b", 20) };
2. Arrays.sort(users);

Comparator可以看成一种算法的实现，将算法和数据分离，Comparator也可以在下面两种环境下使用：

1、类的设计师没有考虑到比较问题而没有实现Comparable，可以通过Comparator来实现排序而不必改变对象本身

2、可以使用多种排序标准，比如升序、降序等。用法：

重写一个继承了Comparator接口的类，之后用这个类的规则对User队形进行排序

1. User[] users = **new** User[] { **new** User("a", 30), **new** User("b", 20) };
2. Arrays.sort(users, **new** UserComparator());

**（三）**：static 关键字的用途：　核心：方便在没有创建对象的情况下来进行调用（方法/变量）。

被static关键字修饰的方法或者变量不需要依赖于对象来进行访问，只要类被加载了，就可以通过类名去进行访问。因此是没有this方法的。在静态方法中不能访问类的非静态成员变量和非静态成员方法，因为非静态成员方法/变量都是必须依赖具体的对象才能够被调用。

static可以用来修饰类的成员方法、类的成员变量，另外可以编写static代码块来优化程序性能。

main方法必须是static的。因为程序在执行main方法的时候没有创建任何对象，因此只有通过类名来访问。

static成员变量的初始化顺序按照定义的顺序进行初始化。

static形成静态代码块以优化程序性能，static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。因此，很多时候会将一些只需要进行一次的初始化操作都放在static代码块中进行。static是不允许用来修饰局部变量的。

static相关笔试：

public class Test extends Base{

    static{

        System.out.println("test static");

    }

    public Test(){

        System.out.println("test constructor");

    }

    public static void main(String[] args) {

        new Test();

    }

}

class Base{

    static{

        System.out.println("base static");

    }

    public Base(){

        System.out.println("base constructor");

    }

}

结果：base static

test static

base constructor

test constructor

在执行main方法之前，必须先加载Test类，而在加载Test类的时候发现Test类继承自Base类，因此会转去先加载Base类，在加载Base类的时候，发现有static块，便执行了static块。在Base类加载完成之后，便继续加载Test类，然后发现Test类中也有static块，便执行static块。在加载完所需的类之后，便开始执行main方法。在main方法中执行new Test()的时候会先调用父类的构造器，然后再调用自身的构造器。如果将构造器改为普通方法则不会调用

public class Test {

    Person person = new Person("Test");

    static{

        System.out.println("test static");

    }

    public Test() {

        System.out.println("test constructor");

    }

    public static void main(String[] args) {

        new MyClass();

    }

}

class Person{

    static{

        System.out.println("person static");

    }

    public Person(String str) {

        System.out.println("person "+str);

    }

}

class MyClass extends Test {

    Person person = new Person("MyClass");

    static{

        System.out.println("myclass static");

    }

    public MyClass() {

        System.out.println("myclass constructor");

    }

}

结果：test static

myclass static

person static

person Test

test constructor

person MyClass

myclass constructor

首先加载Test类，因此会执行Test类中的static块。接着执行new MyClass()，而MyClass类还没有被加载，因此需要加载MyClass类。在加载MyClass类的时候，发现MyClass类继承自Test类，但是由于Test类已经被加载了，所以只需要加载MyClass类，那么就会执行MyClass类的中的static块。在加载完之后，就通过构造器来生成对象。而在生成对象的时候，必须先初始化父类的成员变量，因此会执行Test中的Person person = new Person()，而Person类还没有被加载过，因此会先加载Person类并执行Person类中的static块，接着执行父类的构造器，完成了父类的初始化，然后就来初始化自身了，因此会接着执行MyClass中的Person person = new Person()，最后执行MyClass的构造器。

（四）：哈希表是由数组+链表组成的，一个长度为16的数组中，每个元素存储的是一个链表的头结点。那么这些元素是按照什么样的规则存储到数组中呢。一般情况是通过hash(key)%len获得，也就是元素的key的哈希值对数组长度取余得到。

第一个键值对A进来，通过计算其key的hash得到的i=0，记做:Entry[0] = A。一会后又进来一个键值对B，通过计算其i也等于0，现在怎么办？HashMap会这样做:B.next = A,Entry[0] = B,如果又进来C,i也等于0,那么C.next = B,Entry[0] = C；这样我们发现i=0的地方其实存取了A,B,C三个键值对,他们通过next这个属性链接在一起,也就是说数组中存储的是最后插入的元素。

synchronized关键字加锁的原理，其实是对对象加锁，不论你是在方法前加synchronized还是语句块前加，锁住的都是对象整体，但是ConcurrentHashMap的同步机制和这个不同，它不是加synchronized关键字，而是基于lock操作的，这样的目的是保证同步的时候，锁住的不是整个对象。事实上，ConcurrentHashMap可以满足concurrentLevel个线程并发无阻塞的操作集合对象。