1，JDK：Java Development Kit，java 的开发和运行环境，java 的开发工具和 jre。
2，JRE：Java Runtime Environment，java 程序的运行环境，java 运行的所需的类库+JVM(java 虚拟机)。
3，配置环境变量：让 java jdk\bin 目录下的工具，可以在任意目录下运行，原因是，将该工具所在目录告诉了系统，当使用该工具时，由系统帮我们去找指定的目录。
环境变量的配置：
1）永久配置方式：JAVA_HOME=%安装路径%\Java\jdk
path=%JAVA_HOME%\bin
2）临时配置方式：set path=%path%;C:\Program Files\Java\jdk\bin
特点：系统默认先去当前路径下找要执行的程序，如果没有，再去 path 中设置的路径下找。
classpath 的配置:
1）永久配置方式：classpath=.;c:\;e:\
2）临时配置方式：set classpath=.;c:\;e:\
注意：在定义 classpath 环境变量时，需要注意的情况
如果没有定义环境变量 classpath，java 启动 jvm 后，会在当前目录下查找要运行的类文件；
如果指定了 classpath，那么会在指定的目录下查找要运行的类文件。
还会在当前目录找吗？两种情况：
CLASSPATH 是什么？它的作用是什么？
它是 javac 编译器的一个环境变量。它的作用与 import、package 关键字有关。当你写下 improt java.util.*时，编译器面对 import 关键字时，就知道你要引入 java.util 这个 package 中的类；但是编译器如何知道你把这个 package 放在哪里了呢？所以你首先得告诉编译器这个 package 的所在位置；如何告诉它呢？就是设置 CLASSPATH 啦 :) 如果 java.util 这个 package 在 c:/jdk/ 目录下，你得把 c:/jdk/这个路径设置到 CLASSPATH 中去！当编译器面对 import java.util.*这个语句时，它先会查找 CLASSPATH 所指定的目录，并检视子目录 java/util 是否存在，然后找出名称吻合的已编译文件（.class 文件）。如果没有找到就会报错！CLASSPATH 有点像 c/c++编译器中的 INCLUDE 路径的设置哦，是不是？当 c/c++编译器遇到 include 这样的语句，它是如何运作的？哦，其实道理都差不多！搜索 INCLUDE 路径，检视文件！当你自己开发一个 package 时，然后想要用这个 package 中的类；自然，你也得把这个 package 所在的目录设置到 CLASSPATH 中去！CLASSPATH 的设定，对 JAVA 的初学者而言是一件棘手的事。所以 Sun 让 JAVA2 的 JDK 更聪明一些。你会发现，在你安装之后，即使完全没有设定 CLASSPATH，你仍然能够编译基本的 JAVA 程序，并且加以执行。
PATH 环境变量
PATH 环境变量。作用是指定命令搜索路径，在命令行下面执行命令如 javac 编译 java 程序时，它会到 PATH 变量所指定的路径中查找看是否能找到相应的命令程序。我们需要把 jdk 安装目录下的 bin 目录增加到现有的 PATH 变量中，bin 目录中包含经常要用到的可执行文件如 javac/java/javadoc 等待，设置好 PATH 变量后，就可以在任何目录下执行 javac/java 等工具了。

4，javac 命令和 java 命令做什么事情呢？
要知道 java 是分两部分的：一个是编译，一个是运行。
javac：负责的是编译的部分，当执行 javac 时，会启动 java 的编译器程序。对指定扩展名的.java 文件进行编译。 生成了 jvm 可以识别的字节码文件。也就是 class 文件，也就是 java 的运行程序。
java：负责运行的部分.会启动 jvm.加载运行时所需的类库,并对 class 文件进行执行.
一个文件要被执行,必须要有一个执行的起始点,这个起始点就是 main 函数.
标示符：
1），数字不可以开头。
2），不可以使用关键字。
变量的作用域和生存期:
1. 变量的作用域：作用域从变量定义的位置开始，到该变量所在的那对大括号结束；
生命周期：变量从定义的位置开始就在内存中活了；
变量到达它所在的作用域的时候就在内存中消失了；

数据类型：

1）：基本数据类型：byte、short、int、long、float、double、char、boolean

运算符号：
4）、逻辑运算符。
& | ^ ! && ||
逻辑运算符除了 ！外都是用于连接两个 boolean 类型表达式。
&: 只有两边都为 true 结果是 true。否则就是 false。
|:只要两边都为 false 结果是 false，否则就是 true
^:异或：和或有点不一样。
两边结果一样，就为 false。
两边结果不一样，就为 true.
& 和 &&区别： & ：无论左边结果是什么，右边都参与运算。
&&:短路与，如果左边为 false，那么右边不参数与运算。
| 和|| 区别：|：两边都运算。
||：短路或，如果左边为 true，那么右边不参与运算。
5）、位运算符:用于操作二进制位的运算符。
& | ^
<< >> >>>(无符号右移)
练习：对两个变量的数据进行互换。不需要第三方变量。
int a = 3,b = 5;-->b = 3,a = 5;
方法一：

a = a + b; a = 8;
b = a - b; b = 3;
a = a - b; a = 5;

方法二：

a = a ^ b;//
b = a ^ b;//b = a ^ b ^ b = a
a = a ^ b;//a = a ^ b ^ a = b;

练习：高效的算出 2*8 = 2<<3;

重载的定义是：在一个类中，如果出现了两个或者两个以上的同名函数，只要它们的参数的个数，或者参数的类型不同，即可称之为该函数重载了。

如何区分重载：当函数同名时，只看参数列表。和返回值类型没关系。

重写：父类与子类之间的多态性，对父类的函数进行重新定义。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写(Overriding)。

Java 内存管理

Java 内存管理：深入 Java 内存区域

Java 与 C++之间有一堵由内存动态分配和垃圾收集技术所围成的高墙，墙外面的人想进去，墙里面的人却想出来。

概述：

对于从事 C 和 C++程序开发的开发人员来说，在内存管理领域，他们既是拥有最高权力的皇帝，又是从事最基础工作的劳动人民—既拥有每一个对象的"所有权"，又担负着每一个对象生命开始到终结的维护责任。

对于 Java 程序员来说，在虚拟机的自动内存管理机制的帮助下，不再需要为每一个 new 操作去写配对的 delete/free 代码，而且不容易出现内存泄漏和内存溢出问题，看起来由虚拟机管理内存一切都很美好。不过，也正是因为 Java 程序员把内存控制的权力交给了 Java 虚拟机，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那排查错误将会成为一项异常艰难的工作。

运行时数据区域

Java 虚拟机在执行 Java 程序的过程中会把它所管理的内存划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而存在，有些区域则是依赖用户线程的启动和结束而建立和销毁。根据《Java 虚拟机规范（第 2 版）》的规定，Java 虚拟机所管理的内存将会包括以下几个运行时数据区域，如下图所示：

程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。 **由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为"线程私有"的内存。** 如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Natvie 方法，这个计数器值则为空（Undefined）。**此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。**

Java 虚拟机栈

与程序计数器一样，Java 虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是 Java 方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）**用于存储局部变量表、操作栈、动态链接、方法出口等**信息。**每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。**

经常有人把 Java 内存区分为堆内存（Heap）和栈内存（Stack），这种分法比较粗糙，Java 内存区域的划分实际上远比这复杂。这种划分方式的流行只能说明大多数程序员最关注的、与对象内存分配关系最密切的内存区域是这两块。其中所指的"堆"在后面会专门讲述，而所指的"栈"就是现在讲的虚拟机栈，或者说是虚拟机栈中的局部变量表部分。
局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference 类型），它不等同于对象本身，根据不同的虚拟机实现，它可能是一个指向对象起始地址的引用指针，也可能指向一个代表对象的句柄或者其他与此对象相关的位置）和 returnAddress 类型（指向了一条字节码指令的地址）。
其中 64 位长度的 long 和 double 类型的数据会占用 2 个局部变量空间（Slot），其余的数据类型只占用 1 个。**局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。** 在 Java 虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常；如果虚拟机栈可以动态扩展（当前大部分的 Java 虚拟机都可动态扩展，只不过 Java 虚拟机规范中也允许固定长度的虚拟机栈），当扩展时无法申请到足够的内存时会抛出 OutOfMemoryError 异常。
本地方法栈
本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的 Native 方法服务。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如 Sun HotSpot 虚拟机）直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryError 异常。
Java 堆
对于大多数应用来说，Java 堆（Java Heap）是 Java 虚拟机所管理的内存中最大的一块。Java 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。**此内存区域的唯一目的就是存放对象实例**，几乎所有的对象实例都在这里分配内存。这一点在 Java 虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配，但是随着 JIT 编译器的发展与逃逸分析技术的逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不是那么"绝对"了。
Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称做"GC 堆"（Garbage Collected Heap，幸好国内没翻译成"垃圾堆"）。如果从内存回收的角度看，由于现在收集器基本都是采用的**分代收集算法**，所以 Java 堆中还可以细分为：新生代和老年代；再细致一点的有 Eden 空间、From Survivor 空间、To Survivor 空间等。如果**从内存分配的角度看，线程共享的 Java 堆中可能划分出多个线程私有的分配缓冲区**（Thread Local Allocation Buffer，TLAB）。不过，无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分的目的是为了更好地回收内存，或者更快地分配

内存。在本章中，我们仅仅针对内存区域的作用进行讨论，Java 堆中的上述各个区域的分配和回收等细节将会是下一章的主题。

根据 Java 虚拟机规范的规定，Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。在实现时，既可以实现成固定大小的，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的（通过-Xmx 和-Xms 控制）。如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出 OutOfMemoryError 异常。

方法区

方法区（Method Area）与 Java 堆一样，是各个线程共享的内存区域，它**用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据**。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。

对于习惯在 HotSpot 虚拟机上开发和部署程序的开发者来说，很多人愿意把方法区称为"永久代"Permanent Generation），本质上两者并不等价，仅仅是因为 HotSpot 虚拟机的设计团队选择把 GC 分代收集扩展至方法区，或者说使用永久代来实现方法区而已。对于其他虚拟机（如 BEA JRockit、IBM J9 等）来说是不存在永久代的概念的。即使是 HotSpot 虚拟机本身，根据官方发布的路线图信息，现在也有放弃永久代并"搬家"至 Native Memory 来实现方法区的规划了。

Java 虚拟机规范对这个区域的限制非常宽松，除了和 Java 堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入了方法区就如永久代的名字一样"永久"存在了。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载，一般来说这个区域的回收"成绩"比较难以令人满意，尤其是类型的卸载，条件相当苛刻，但是这部分区域的回收确实是有必要的。在 Sun 公司的 BUG 列表中，　　曾出现过的若干个严重的 BUG 就是由于低版本的 HotSpot 虚拟机对此区域未完全回收而导致内存泄漏。根据 Java 虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError 异常。

运行时常量池

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是常量池（Constant Pool Table），**用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。** Java 虚拟机对 Class 文件的每一部分（自然也包括常量池）的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范上的要求，这样才会被虚拟机认可、装载和执行。但对于运行时常量池，Java 虚拟机规范没有做任何细节的要求，不同的提供商实现的虚拟机可以按照自己的需要来实现这个内存区域。不过，一般来说，除了保存 Class 文件中描述的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池中。运行时常量池相对于 Class 文件常量池的另外一个重要特征是具备动态性，Java 语言并不要求常量一定只能在编译期产生，也就是并非预置入 Class 文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是 String 类的 intern()方法。既然运行时常量池是方法区的一部分，自然会受到

方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

对象访问

介绍完 Java 虚拟机的运行时数据区之后，我们就可以来探讨一个问题：在 Java 语言中，对象访问是如何进行的？对象访问在 Java 语言中无处不在，是最普通的程序行为，但即使是最简单的访问，也会却涉及 Java 栈、Java 堆、方法区这三个最重要内存区域之间的关联关系，如下面的这句代码：

Object obj = new Object();

假设这句代码出现在方法体中，那"Object obj"这部分的语义将会反映到 Java 栈的本地变量表中，作为一个 reference 类型数据出现。而"new Object()"这部分的语义将会反映到 Java 堆中，形成一块存储了 Object 类型所有实例数据值（Instance Data，对象中各个实例字段的数据）的结构化内存，根据具体类型以及虚拟机实现的对象内存布局（Object Memory Layout）的不同，这块内存的长度是不固定的。另外，在 Java 堆中还必须包含能查找到此对象类型数据（如对象类型、父类、实现的接口、方法等）的地址信息，这些类型数据则存储在方法区中。

由于 reference 类型在 Java 虚拟机规范里面只规定了一个指向对象的引用，并没有定义这个引用应该通过哪种方式去定位，以及访问到 Java 堆中的对象的具体位置，因此不同虚拟机实现的对象访问方式会有所不同，主流的访问方式有两种：使用句柄和直接指针。 如果使用句柄访问方式，Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据和类型数据各自的具体地址信息，如下图所示：

如果使用的是直接指针访问方式，Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，reference 中直接存储的就是对象地址，如下图所示：

这两种对象的访问方式各有优势，使用句柄访问方式的最大好处就是 **reference** 中存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而 **reference** 本身不需要被修改。使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开销，由于对象的访问在 **Java** 中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本。就本书讨论的主要虚拟机 Sun HotSpot 而言，它是使用第二种方式进行对象访问的，但从整个软件开发的范围来看，各种语言和框架使用句柄来访问的情况也十分常见。

类

匿名对象使用场景：

1：当对方法只进行一次调用的时候，可以使用匿名对象。

2：当对象对成员进行多次调用时，不能使用匿名对象。必须给对象起名字。

类中怎么没有定义主函数呢？

注意：主函数的存在，仅为该类是否需要独立运行，如果不需要，主函数是不用定义的。

主函数的解释：保证所在类的独立运行，是程序的入口，被 jvm 调用。

成员变量和局部变量的区别：

1：成员变量直接定义在类中。

局部变量定义在方法中，参数上，语句中。

2：成员变量在这个类中有效。

局部变量只在自己所属的大括号内有效，大括号结束，局部变量失去作用域。

3：成员变量存在于堆内存中，随着对象的产生而存在，消失而消失。

局部变量存在于栈内存中，随着所属区域的运行而存在，结束而释放。

构造函数：用于给对象进行初始化，是给与之对应的对象进行初始化，它具有针对性，函数中的一种。

特点：

1：该函数的名称和所在类的名称相同。

2：不需要定义返回值类型。

3：该函数没有具体的返回值。
记住：所有对象创建时，都需要初始化才可以使用。
注意事项：一个类在定义时，如果没有定义过构造函数，那么该类中会自动生成一个空参数的构造函数，为了方便该类创建对象，完成初始化。如果在类中自定义了构造函数，那么默认的构造函数就没有了。
一个类中，可以有多个构造函数，因为它们的函数名称都相同，所以只能通过参数列表来区分。所以，**一个类中如果出现多个构造函数。它们的存在是以重载体现的。**
构造代码块和构造函数有什么区别？
构造代码块：是给所有的对象进行初始化，也就是说，所有的对象都会调用一个代码块。只要对象一建立。就会调用这个代码块。
构造函数：是给与之对应的对象进行初始化。它具有针对性。

执行顺序：（优先级从高到低。）静态代码块>mian 方法>构造代码块>构造方法。其中静态代码块只执行一次。构造代码块在每次创建对象是都会执行。
静态代码块的作用：比如我们在调用 C 语言的动态库时会可把.so 文件放在此处。
构造代码块的功能：（可以把不同构造方法中相同的共性的东西写在它里面）。例如：比如不论任何机型的电脑都有开机这个功能，此时我们就可以把这个功能定义在构造代码块内。
Person p = new Person();
创建一个对象都在内存中做了什么事情？
1：先将硬盘上指定位置的 Person.class 文件加载进内存。
2：执行 main 方法时，在栈内存中开辟了 main 方法的空间(压栈-进栈)，然后在 main 方法的栈区分配了一个变量 p。
3：在堆内存中开辟一个实体空间，分配了一个内存首地址值。new
4：在该实体空间中进行属性的空间分配，并进行了默认初始化。
5：对空间中的属性进行显示初始化。
6：进行实体的构造代码块初始化。
7：调用该实体对应的构造函数，进行构造函数初始化。（）
8：将首地址赋值给 p ，p 变量就引用了该实体。(指向了该对象)
封 装（面向对象特征之一）：是指隐藏对象的属性和实现细节，仅对外提供公共访问方式。
好处：将变化隔离；便于使用；提高重用性；安全性。
封装原则：将不需要对外提供的内容都隐藏起来，把属性都隐藏，提供公共方法对其访问。
this:代表对象。就是所在函数所属对象的引用。
this 到底代表什么呢？**哪个对象调用了 this 所在的函数，this 就代表哪个对象，就是哪个对象的引用。**
开发时，什么时候使用 this 呢？

在定义功能时，如果该功能内部使用到了调用该功能的对象，这时就用 this 来表示这个对象。
this 还可以用于构造函数间的调用。
调用格式：this(实际参数)；
this 对象后面跟上 . 调用的是成员属性和成员方法(一般方法)；
this 对象后面跟上 () 调用的是本类中的对应参数的构造函数。
注意：用 this 调用构造函数，必须定义在构造函数的第一行。因为构造函数是用于初始化的，所以初始化动作一定要执行。否则编译失败。

static：★★★ 关键字，是一个修饰符，用于修饰成员(成员变量和成员函数)。

特点：
1、static 变量
按照是否静态的对类成员变量进行分类可分两种：一种是被 static 修饰的变量，叫静态变量或类变量；另一种是没有被 static 修饰的变量，叫实例变量。两者的区别是：
对于静态变量在内存中只有一个拷贝（节省内存），JVM 只为静态分配一次内存，在加载类的过程中完成静态变量的内存分配，可用类名直接访问（方便），当然也可以通过对象来访问（但是这是不推荐的）。
对于实例变量，没创建一个实例，就会为实例变量分配一次内存，实例变量可以在内存中有多个拷贝，互不影响（灵活）。
2、静态方法
静态方法可以直接通过类名调用，任何的实例也都可以调用，因此静态方法中不能用 this 和 super 关键字，不能直接访问所属类的实例变量和实例方法(就是不带 static 的成员变量和成员成员方法)，只能访问所属类的静态成员变量和成员方法。因为实例成员与特定的对象关联！这个需要去理解，想明白其中的道理，不是记忆！！！
因为 static 方法独立于任何实例，因此 static 方法必须被实现，而不能是抽象的 abstract。
3、static 代码块
static 代码块也叫静态代码块，是在类中独立于类成员的 static 语句块，可以有多个，位置可以随便放，它不在任何的方法体内，JVM 加载类时会执行这些静态的代码块，如果 static 代码块有多个，JVM 将按照它们在类中出现的先后顺序依次执行它们，每个代码块只会被执行一次。
4、static 和 final 一块用表示什么
static final 用来修饰成员变量和成员方法，可简单理解为"全局常量"！
对于变量，表示一旦给值就不可修改，并且通过类名可以访问。
对于方法，表示不可覆盖，并且可以通过类名直接访问。
备注：
1，有些数据是对象特有的数据，是不可以被静态修饰的。因为那样的话，特有数据会变成对象的共享数据。这样对事物的描述就出了问题。所以，在定义静态时，必须要明确，这个数据是否是被对象所共享的。
2，**静态方法只能访问静态成员，不可以访问非静态成员。**

(这句话是针对同一个类环境下的，比如说，一个类有多个成员（属性，方法，字段），静态方法 A，那么可以访问同类名下其他静态成员，你如果访问非静态成员就不行)
因为静态方法加载时，优先于对象存在，所以没有办法访问对象中的成员。
3，**静态方法中不能使用 this，super 关键字。**
因为 **this 代表对象**，而静态在时，有可能没有对象，所以 this 无法使用。
4，主函数是静态的。
成员变量和静态变量的区别：
1，成员变量所属于对象。所以也称为实例变量。
静态变量所属于类。所以也称为类变量。
2，成员变量存在于堆内存中。
静态变量存在于方法区中。
3，成员变量随着对象创建而存在。随着对象被回收而消失。
静态变量随着类的加载而存在。随着类的消失而消失。
4，成员变量只能被对象所调用 。
静态变量可以被对象调用，也可以被类名调用。
所以，成员变量可以称为对象的特有数据，静态变量称为对象的共享数据。
静态代码块：就是一个有静态关键字标示的一个代码块区域。定义在类中。
作用：可以完成类的初始化。静态代码块随着类的加载而执行，而且只执行一次（new 多个对象就只执行一次）。如果和主函数在同一类中，优先于主函数执行。
final
根据程序上下文环境，Java 关键字 final 有"这是无法改变的"或者"终态的"含义，它可以修饰非抽象类、非抽象类成员方法和变量。你可能出于两种理解而需要阻止改变、设计或效率。
final 类不能被继承，没有子类，final 类中的方法默认是 final 的。
final 方法不能被子类的方法覆盖，但可以被继承。
final 成员变量表示常量，只能被赋值一次，赋值后值不再改变。
final 不能用于修饰构造方法。
注意：父类的 private 成员方法是不能被子类方法覆盖的，因此 private 类型的方法默认是 final 类型的。
1、final 类
final 类不能被继承，因此 final 类的成员方法没有机会被覆盖，默认都是 final 的。在设计类时候，如果这个类不需要有子类，类的实现细节不允许改变，并且确信这个类不会载被扩展，那么就设计为 final 类。
2、final 方法
如果一个类不允许其子类覆盖某个方法，则可以把这个方法声明为 final 方法。
使用 final 方法的原因有二：
第一、把方法锁定，防止任何继承类修改它的意义和实现。
第二、高效。编译器在遇到调用 final 方法时候会转入内嵌机制，大大提高执行效率。
3、final 变量（常量）
用 final 修饰的成员变量表示常量，值一旦给定就无法改变！

final 修饰的变量有三种：静态变量、实例变量和局部变量，分别表示三种类型的常量。

从下面的例子中可以看出，一旦给 final 变量初值后，值就不能再改变了。

另外，final 变量定义的时候，可以先声明，而不给初值，这中变量也称为 final 空白，无论什么情况，编译器都确保空白 final 在使用之前必须被初始化。但是，final 空白在 final 关键字 final 的使用上提供了更大的灵活性，为此，一个类中的 final 数据成员就可以实现依对象而有所不同，却有保持其恒定不变的特征。

4、final 参数

当函数参数为 final 类型时，你可以读取使用该参数，但是无法改变该参数的值。

生成 Java 帮助文档：命令格式：javadoc –d 文件夹名 –auther –version *.java

/** //格式

*类描述

*@author 作者名

*@version 版本号

*/

/**

*方法描述

*@param 参数描述

*@return 返回值描述

*/

继 承（面向对象特征之一）

java 中对于继承，java 只支持单继承。java 虽然不直接支持多继承，但是可实现多接口。

1：成员变量。

当子父类中出现一样的属性时，子类类型的对象，调用该属性，值是子类的属性值。

如果想要调用父类中的属性值，需要使用一个关键字：**super**

This：代表是本类类型的对象引用。

Super：代表是子类所属的父类中的内存空间引用。

注意：子父类中通常是不会出现同名成员变量的，因为父类中只要定义了，子类就不用在定义了，直接继承过来用就可以了。

2：成员函数。

当子父类中出现了一模一样的方法时，建立子类对象会运行子类中的方法。好像父类中的方法被覆盖掉一样。所以这种情况，是函数的另一个特性：**重写**

3：构造函数。

发现子类构造函数运行时，先运行了父类的构造函数。为什么呢?

原因：子类的**所有构造函数中的第一行**，其实都有一条隐身的语句 super();

super(): 表示父类的构造函数，并会调用于参数相对应的父类中的构造函数。而 super():是在调用父类中空参数的构造函数。

为什么子类对象初始化时，都需要调用父类中的函数？(为什么要在子类构造函数的第一行加入这个 super()?)

因为子类继承父类，会继承到父类中的数据，所以必须要看父类是如何对自己的数据进行初始化的。所以子类在进行对象初始化时，先调用父类的构造函数，这就是**子类的实例化过程**。

注意：子类中所有的构造函数都会默认访问父类中的空参数的构造函数，因为每一个子类构造内第一行都有默认的语句 super();

如果父类中没有空参数的构造函数，那么子类的构造函数内，必须通过 super 语句指定要访问的父类中的构造函数。

如果子类构造函数中用 this 来指定调用子类自己的构造函数，那么被调用的构造函数也一样会访问父类中的构造函数。

问题：

super()和 this()是否可以同时出现的构造函数中？

两个语句只能有一个定义在第一行，所以只能出现其中一个。

super()或者 this():为什么一定要定义在第一行？

因为 super()或者 this()都是调用构造函数，构造函数用于初始化，所以初始化的动作要先完成。

在方法覆盖时，注意两点：

1：子类覆盖父类时，必须要保证，子类方法的权限必须大于等于父类方法权限可以实现继承。否则，编译失败。（举个例子，在父类中是 public 的方法，如果子类中将其降低访问权限为 private，那么子类中重写以后的方法对于外部对象就不可访问了，这个就破坏了继承的含义）

2：覆盖时，要么都静态，要么都不静态。 (静态只能覆盖静态，或者被静态覆盖)

继承的一个弊端：打破了封装性。对于一些类，或者类中功能，是需要被继承，或者复写的。

这时如何解决问题呢？介绍一个关键字，**final**。

final 特点：（详细解释见前面）

1：这个关键字是一个修饰符，可以修饰类，方法，变量。

2：被 final 修饰的类是一个最终类，不可以被继承。

3：被 final 修饰的方法是一个最终方法，不可以被覆盖。

4：被 final 修饰的变量是一个常量，只能赋值一次。

抽象类: abstract

抽象类的特点：

1：抽象方法只能定义在抽象类中，抽象类和抽象方法必须由 abstract 关键字修饰（可以描述类和方法，不可以描述变量）。

2：抽象方法只定义方法声明，并不定义方法实现。

3：抽象类不可以被创建对象(实例化)。

4：只有通过子类继承抽象类并覆盖了抽象类中的所有抽象方法后，该子类才可以实例化。否则，该子类还是一个抽象类。

抽象类的细节：

1：抽象类中是否有构造函数？有，用于给子类对象进行初始化。

2：抽象类中是否可以定义非抽象方法？

可以。其实，抽象类和一般类没有太大的区别，都是在描述事物，只不过抽象类在描述事物时，有些功能不具体。所以抽象类和一般类在定义上，都是需

要定义属性和行为的。只不过，比一般类多了一个抽象函数。而且比一般类少了一个创建对象的部分。

3：抽象关键字 abstract 和哪些不可以共存？final ， private , static

4：抽象类中可不可以不定义抽象方法？可以。抽象方法目的仅仅为了不让该类创建对象。

接 口：★★★★★

1：是用关键字 **interface** 定义的。

2：接口中包含的成员，最常见的有全局常量、抽象方法。

注意：接口中的成员都有固定的修饰符。

成员变量：public static final

成员方法：public abstract

```
interface Inter{
    public static final int x = 3;
    public abstract void show();
}
```

3：接口中有抽象方法，说明**接口不可以实例化。接口的子类必须实现了接口中所有的抽象方法后，该子类才可以实例化。否则，该子类还是一个抽象类。**

4：类与类之间存在着继承关系，类与接口中间存在的是**实现**关系。

继承用 extends ；实现用 implements ；

5：接口和类不一样的地方，就是，**接口可以被多实现**，这就是多继承改良后的结果。java 将多继承机制通过多现实来体现。

6：一个类在继承另一个类的同时，还可以实现多个接口。所以**接口的出现避免了单继承的局限性**。还可以将类进行功能的扩展。

7：其实 java 中是有多继承的。接口与接口之间存在着继承关系，**接口可以多继承接口**。

java 类是单继承的。classB Extends classA

java 接口可以多继承。Interface3 Extends Interface0, Interface1, interface……

不允许类多重继承的主要原因是，如果 A 同时继承 B 和 C，而 b 和 c 同时有一个 D 方法，A 如何决定该继承那一个呢？

但接口不存在这样的问题，接口全都是抽象方法继承谁都无所谓，所以接口可以继承多个接口。

抽象类与接口：

抽象类：一般用于描述一个体系单元，将一组共性内容进行抽取，特点：可以在类中定义抽象内容让子类实现，可以定义非抽象内容让子类直接使用。它里面定义的都是一些体系中的**基本内容**。

接口：一般用于定义对象的**扩展功能**，是在继承之外还需这个对象具备的一些功能。

抽象类和接口的共性：都是不断向上抽取的结果。

抽象类和接口的区别：

1：抽象类只能被继承，而且只能单继承。

接口需要被实现，而且可以多实现。

2：抽象类中可以定义非抽象方法，子类可以直接继承使用。

接口中都是抽象方法，需要子类去实现。

3：抽象类使用的是 is a 关系。
接口使用的 like a 关系。
4：抽象类的成员修饰符可以自定义。
接口中的成员修饰符是固定的。全都是 public 的。

多 态★★★★★

多 态★★★★★（面向对象特征之一）：函数本身就具备多态性，某一种事物有不同的具体的体现。
体现：父类引用或者接口的引用指向了自己的子类对象。//Animal a = new Cat();父类可以调用子类中覆写过的（父类中有的方法）
多态的好处：提高了程序的扩展性。继承的父类或接口一般是类库中的东西，（如果要修改某个方法的具体实现方式）只有通过子类去覆写要改变的某一个方法，这样在通过将父类的应用指向子类的实例去调用覆写过的方法就行了！
多态的弊端：当父类引用指向子类对象时，虽然提高了扩展性，但是只能访问父类中具备的方法，不可以访问子类中特有的方法。(前期不能使用后期产生的功能，即访问的局限性**)**
多态的前提：
1：必须要有关系，比如继承、或者实现。
2：通常会有覆盖操作。
如果想用子类对象的特有方法，如何判断对象是哪个具体的子类类型呢？
可以可以通过一个关键字 **instanceof ;//判断对象是否实现了指定的接口或继承了指定的类**
格式：**<对象 instanceof 类型>** ，判断一个对象是否所属于指定的类型。
Student instanceof Person = true;//student 继承了 person 类
---
java.lang.Object
Object：所有类的直接或者间接父类，Java 认为所有的对象都具备一些基本的共性内容，这些内容可以不断的向上抽取，最终就抽取到了一个最顶层的类中的，该类中定义的就是所有对象都具备的功能。
具体方法：
boolean equals(Object obj)：用于比较两个对象是否相等，**其实内部比较的就是两个对象地址。**
2，String toString()：将对象变成字符串；默认返回的格式：类名@哈希值 = getClass().getName() + '@' + Integer.toHexString(hashCode())
为了对象对应的字符串内容有意义，可以通过复写，建立该类对象自己特有的字符串表现形式。

```
public String toString(){
    return "person : "+age;
}
```

3，Class getClass()：获取任意对象运行时的所属字节码文件对象。
4，int hashCode()：返回该对象的哈希码值。支持此方法是为了提高哈希表的性能。将该对象的内部地址转换成一个整数来实现的。

通常 equals，toString，hashCode，在应用中都会被复写，建立具体对象的特有的内容。

内部类：如果 A 类需要直接访问 B 类中的成员，而 B 类又需要建立 A 类的对象。这时,为了方便设计和访问，直接将 A 类定义在 B 类中。就可以了。A 类就称为**内部类**。内部类可以直接访问外部类中的成员。而外部类想要访问内部类，必须要建立内部类的对象。

```
class Outer{
    int num = 4;
    class Inner {
        void show(){
            System.out.println("inner show run "+num);
        }
    }
    public void method(){
        Inner in = new Inner();//创建内部类的对象。
        in.show();//调用内部类的方法。 //内部类直接访问外部类成员，用自己的实例对象；
    }                                   //外部类访问内部类要定义内部类的对象；
}
```

当内部类定义在外部类中的成员位置上，可以使用一些成员修饰符修饰 private、static。

1：默认修饰符。

直接访问内部类格式：外部类名.内部类名 变量名 = 外部类对象.内部类对象;

Outer.Inner in = new Outer.new Inner();//这种形式很少用。

但是这种应用不多见，因为内部类之所以定义在内部就是为了封装。**想要获取内部类对象通常都通过外部类的方法来获取。这样可以对内部类对象进行控制。**

2：私有修饰符。

通常内部类被封装，都会被私有化，因为封装性不让其他程序直接访问。

3：静态修饰符。

如果内部类被静态修饰，相当于外部类，会出现访问局限性，只能访问外部类中的静态成员。

注意；**如果内部类中定义了静态成员，那么该内部类必须是静态的。**

内部类编译后的文件名为："外部类名$内部类名.java"；

为什么内部类可以直接访问外部类中的成员呢？

那是因为内部中都持有一个外部类的引用。这个是引用是 **外部类名.this**

内部类可以定义在外部类中的成员位置上，也可以定义在外部类中的局部位置上。

当内部类被定义在局部位置上，只能访问局部中被 final 修饰的局部变量。

匿名内部类（对象）：没有名字的内部类。就是内部类的简化形式。一般只用一次就可以用这种形式。匿名内部类其实就是一个**匿名子类对象**。**想要定义匿名内部类：需要前提，内部类必须继承一个类或者实现接口。**
匿名内部类的格式：new 父类名&接口名(){ 定义子类成员或者覆盖父类方法 }.方法。
匿名内部类的使用场景：
当函数的参数是接口类型引用时，如果接口中的方法不超过 3 个。可以通过匿名内部类来完成参数的传递。
其实就是在创建匿名内部类时，该类中的封装的方法不要过多，最好两个或者两个以内。

//面试

```
//1
new Object(){
    void show(){
        System.out.println("show run");
    }
}.show();                          //写法和编译都没问题
//2
Object obj = new Object(){
    void show(){
        System.out.println("show run");
    }
};
obj.show();                        //写法正确，编译会报错
```

1 和 2 的写法正确吗？有区别吗？说出原因。
写法是正确，1 和 2 都是在通过匿名内部类建立一个 Object 类的子类对象。
区别：
第一个可是编译通过，并运行。
第二个编译失败，因为匿名内部类是一个子类对象，当用 Object 的 obj 引用指向时，就被提升为了 Object 类型，而编译时会检查 Object 类中是否有 show 方法，此时编译失败。

异 常：★★★★

--java.lang.Throwable：
Throwable：可抛出的。
|--Error：错误，一般情况下，不编写针对性的代码进行处理，通常是 jvm 发生的，需要对程序进行修正。
|--Exception：异常，可以有针对性的处理方式
这个体系中的所有类和对象都具备一个独有的特点；就是可抛性。
可抛性的体现：就是这个体系中的类和对象都可以被 throws 和 throw 两个关键字所操作。
throw 与 throws 区别：

throws 是用来声明一个方法可能抛出的所有异常信息，而 throw 则是指抛出的一个具体的异常类型。此外 throws 是将异常声明但是不处理，而是将异常往上传，谁调用我就交给谁处理。
throw 用于抛出异常对象，后面跟的是异常对象；throw 用在函数内。
throws 用于抛出异常类，后面跟的异常类名，可以跟多个，用逗号隔开。
throws 用在函数上。
throws 格式：方法名（参数）throws 异常类 1，异常类 2，.....
throw：就是自己进行异常处理，处理的时候有两种方式，要么自己捕获异常（也就是 try catch 进行捕捉），要么声明抛出一个异常（就是 throws 异常 ~~）。
处理方式有两种：1、捕捉；2、抛出。
对于捕捉：java 有针对性的语句块进行处理。

```
try {
   需要被检测的代码；
}
catch(异常类 变量名){
   异常处理代码；
}
fianlly{
   一定会执行的代码；
}
```

定义异常处理时，什么时候定义 try，什么时候定义 throws 呢？
功能内部如果出现异常，如果内部可以处理，就用 try；
如果功能内部处理不了，就必须声明出来，让调用者处理。使用 throws 抛出，交给调用者处理。谁调用了这个功能谁就是调用者；
自定义异常的步骤：
1：定义一个子类继承 Exception 或 RuntimeException，让该类具备可抛性(既可以使用 throw 和 throws 去调用此类)。
2：通过 throw 或者 throws 进行操作。
异常的转换思想：当出现的异常是调用者处理不了的，就需要将此异常转换为一个调用者可以处理的异常抛出。
try catch finally 的几种结合方式：

1，

try
catch
finally

这种情况，如果出现异常，并不处理，但是资源一定关闭，所以 **try finally 集合只为关闭资源**。
记住：finally 很有用，主要用户关闭资源。无论是否发生异常，资源都必须进行关闭。
System.exit(0); //退出 jvm，只有这种情况 finally 不执行。
注意：

如果父类或者接口中的方法没有抛出过异常，那么子类是不可以抛出异常的，如果子类的覆盖的方法中出现了异常，只能 try 不能 throws。
如果这个异常子类无法处理，已经影响了子类方法的具体运算，这时可以在子类方法中，通过 throw 抛出 RuntimeException 异常或者其子类，这样，子类的方法上是不需要 throws 声明的。

多线程：★★★★

返回当前线程的名称：**Thread.currentThread().getName()**
线程的名称是由：Thread-编号定义的。编号从 0 开始。
线程要运行的代码都统一存放在了 **run 方法**中。
线程要运行必须要通过类中指定的方法开启。**start 方法**。（启动后，就多了一条执行路径）
start 方法：1）、启动了线程；2）、让 jvm 调用了 run 方法。
Thread 类中 run()和 start()方法的区别：
start()：用 start 方法来启动线程，真正实现了多线程运行，这时无需等待 run 方法体代码执行完毕而直接继续执行下面的代码。通过调用 Thread 类的 start()方法来启动一个线程，这时此线程处于就绪（可运行）状态，并没有运行，一旦得到 cpu 时间片，就开始执行 run()方法，这里方法 run()称为线程体，它包含了要执行的这个线程的内容，Run 方法运行结束，此线程随即终止。
run()：run()方法只是类的一个普通方法而已，如果直接调用 Run 方法，程序中依然只有主线程这一个线程，其程序执行路径还是只有一条，还是要顺序执行，还是要等待 run 方法体执行完毕后才可继续执行下面的代码，这样就没有达到写线程的目的。
总结：start()方法最本质的功能是从 CPU 中申请另一个线程空间来执行 run()方法中的代码,它和当前的线程是两条线,在相对独立的线程空间运行,也就是说,如果你直接调用线程对象的 run()方法,当然也会执行,但那是 在当前线程中执行,run()方法执行完成后继续执行下面的代码.而调用 start()方法后,run()方法的代码会和当前线程并发(单 CPU)或并行 (多 CPU)执行。所以请记住一句话：调用线程对象的 run 方法不会产生一个新的线程，虽然可以达到相同的执行结果,但执行过程和执行效率不同
创建线程的第一种方式：继承 Thread ，由子类复写 run 方法。
步骤：
1，定义类继承 Thread 类；
2，目的是复写 run 方法，将要让线程运行的代码都存储到 run 方法中；
3，通过创建 Thread 类的子类对象，创建线程对象；
4，调用线程的 start 方法，开启线程，并执行 run 方法。
线程状态：
被创建：start()
运行：具备执行资格，同时具备执行权；
冻结：sleep(time),wait()—notify()唤醒；线程释放了执行权，同时释放执行资格；
临时阻塞状态：线程具备 cpu 的执行资格，没有 cpu 的执行权；
消亡：stop()

创建线程的第二种方式：实现一个接口 Runnable。
步骤：
1，定义类实现 Runnable 接口。
2，覆盖接口中的 run 方法（用于封装线程要运行的代码）。
3，通过 Thread 类创建线程对象；
4，将实现了 Runnable 接口的子类对象作为实际参数传递给 Thread 类中的构造函数。
为什么要传递呢？因为要让线程对象明确要运行的 run 方法所属的对象。
5，调用 Thread 对象的 start 方法。开启线程，并运行 Runnable 接口子类中的 run 方法。

```
Ticket t = new Ticket();
    /*
    直接创建 Ticket 对象，并不是创建线程对象。
    因为创建对象只能通过 new Thread 类，或者 new Thread 类的子类才可以。
    所以最终想要创建线程。既然没有了 Thread 类的子类，就只能用 Thread 类。
    */
    Thread t1 = new Thread(t); //创建线程。
    /*
    只要将 t 作为 Thread 类的构造函数的实际参数传入即可完成线程对象和 t 之间的关联
    为什么要将 t 传给 Thread 类的构造函数呢？其实就是为了明确线程要运行的代码 run 方法。
    */
    t1.start();
```

为什么要有 Runnable 接口的出现？
1：通过继承 Thread 类的方式，可以完成多线程的建立。但是这种方式有一个局限性，如果一个类已经有了自己的父类，就不可以继承 Thread 类，因为**java 单继承**的局限性。
可是该类中的还有部分代码需要被多个线程同时执行。这时怎么办呢？
只有对该类进行额外的功能扩展，java 就提供了一个接口 Runnable。这个接口中定义了 run 方法，其实 run 方法的定义就是为了存储多线程要运行的代码。
所以，通常创建线程都用第二种方式。
因为实现 Runnable 接口可以避免单继承的局限性。
2：其实是将不同类中需要被多线程执行的代码进行抽取。将多线程要运行的代码的位置单独定义到接口中。为其他类进行功能扩展提供了前提。
所以 Thread 类在描述线程时，内部定义的 run 方法，也来自于 Runnable 接口。

实现 Runnable 接口可以避免单继承的局限性。而且，继承 Thread，是可以对 Thread 类中的方法，进行子类复写的。但是不需要做这个复写动作的话，只为定义线程代码存放位置，实现 Runnable 接口更方便一些。所以 **Runnable 接口将线程要执行的任务封装成了对象**。

//面试

```
new Thread(new Runnable(){ //匿名
    public void run(){
        System.out.println("runnable run");
    }
})
{
    public void run(){
        System.out.println("subthread run");
    }
}.start(); //结果：subthread run
```

---

synchronized 关键字（一）

一、当两个并发线程访问同一个对象 object 中的这个 synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。

二、然而，当一个线程访问 object 的一个 synchronized(this)同步代码块时，另一个线程仍然可以访问该 object 中的非 synchronized(this)同步代码块。

三、尤其关键的是，当一个线程访问 object 的一个 synchronized(this)同步代码块时，其他线程对 object 中所有其它 synchronized(this)同步代码块的访问将被阻塞。

四、第三个例子同样适用其它同步代码块。也就是说，当一个线程访问 object 的一个 synchronized(this)同步代码块时，它就获得了这个 object 的对象锁。结果，其它线程对该 object 对象所有同步代码部分的访问都被暂时阻塞。

五、以上规则对其它对象锁同样适用.

```
package ths;
public class Thread1 implements Runnable {
public void run() {
synchronized(this) {
for (int i = 0; i < 5; i++) {
System.out.println(Thread.currentThread().getName()+"synchronized loop " +
i);
}
}
}
}
```

synchronized 关键字（二）

synchronized 关键字，它包括两种用法：synchronized 方法和 synchronized 块。

1. synchronized 方法：通过在方法声明中加入 synchronized 关键字来声明 synchronized 方法。如：
public synchronized void accessVal(int newVal);
synchronized 方法控制对类成员变量的访问：每个类实例对应一把锁，每个 synchronized 方法都必须获得调用该方法的类实例的锁方能执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行状态。这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态（因为至多只有一个能够获得该类实例对应的锁），从而有效避免了类成员变量的访问冲突（只要所有可能访问类成员变量的方法均被声明为 synchronized）。
在 Java 中，不光是类实例，每一个类也对应一把锁，这样我们也可将类的静态成员函数声明为 synchronized ，以控制其对类的静态成员变量的访问。
synchronized 方法的缺陷：若将一个大的方法声明为 synchronized 将会大大影响效率，典型地，若将线程类的方法 run() 声明为 synchronized ，由于在线程的整个生命期内它一直在运行，因此将导致它对本类任何 synchronized 方法的调用都永远不会成功。当然我们可以通过将访问类成员变量的代码放到专门的方法中，将其声明为 synchronized ，并在主方法中调用来解决这一问题，但是 Java 为我们提供了更好的解决办法，那就是 synchronized 块。
2. synchronized 块：通过 synchronized 关键字来声明 synchronized 块。语法如下：
synchronized(syncObject) {
//允许访问控制的代码
}
synchronized 块是这样一个代码块，其中的代码必须获得对象 syncObject （如前所述，可以是类实例或类）的锁方能执行，具体机制同前所述。由于可以针对任意代码块，且可任意指定上锁的对象，故灵活性较高。
对 synchronized(this)的一些理解
一、当两个并发线程访问同一个对象 object 中的这个 synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。
二、然而，当一个线程访问 object 的一个 synchronized(this)同步代码块时，另一个线程仍然可以访问该 object 中的非 synchronized(this)同步代码块。
三、尤其关键的是，当一个线程访问 object 的一个 synchronized(this)同步代码块时，其他线程对 object 中所有其它 synchronized(this)同步代码块的访问将被阻塞。
四、第三个例子同样适用其它同步代码块。也就是说，当一个线程访问 object 的一个 synchronized(this)同步代码块时，它就获得了这个 object 的对象锁。结果，其它线程对该 object 对象所有同步代码部分的访问都被暂时阻塞。
五、以上规则对其它对象锁同样适用。
解决安全问题的原理：
只要将操作共享数据的语句在某一时段让一个线程执行完，在执行过程中，其他线程不能进来执行就可以解决这个问题。
如何保障共享数据的线程安全呢？

java 中提供了一个解决方式：就是同步代码块。
格式：
synchronized(对象) { //任意对象都可以。这个对象就是共享数据。
需要被同步的代码；
}

同步：★★★★★

好处：解决了线程安全问题。Synchronized
弊端：相对降低性能，因为判断锁需要消耗资源，产生了死锁。
同步的第二种表现形式：　　**//对共享资源的方法定义同步**
同步函数：其实就是将同步关键字定义在函数上，让函数具备了同步性。
同步函数是用的哪个锁呢？　　//synchronized(this)用以定义需要进行同步的某一部分代码块
通过验证，函数都有自己所属的对象 this，所以同步函数所使用的锁就是 this 锁。**This.方法名**
当同步函数被 static 修饰时，这时的同步用的是哪个锁呢？
静态函数在加载时所属于类，这时有可能还没有该类产生的对象，但是该类的字节码文件加载进内存就已经被封装成了对象，这个对象就是**该类的字节码文件对象**。
所以静态加载时，只有一个对象存在，那么静态同步函数就使用的这个对象。这个对象就是 **类名.class**
同步代码块和同步函数的区别？
同步代码块使用的锁可以是任意对象。
同步函数使用的锁是 this，静态同步函数的锁是该类的字节码文件对象。
在一个类中只有一个同步的话，可以使用同步函数。如果有多同步，必须使用同步代码块，来确定不同的锁。所以同步代码块相对灵活一些。

★考点问题：请写一个延迟加载的单例模式？写懒汉式；当出现多线程访问时怎么解决？加同步，解决安全问题；效率高吗？不高；怎样解决？通过双重判断的形式解决。
//懒汉式：延迟加载方式。
当多线程访问懒汉式时，因为懒汉式的方法内对共性数据进行多条语句的操作。所以容易出现线程安全问题。为了解决，加入同步机制，解决安全问题。但是却带来了效率降低。
为了效率问题，通过双重判断的形式解决。

```
class Single{
    private static Single s = null;
    private Single(){}
    public static Single getInstance(){ //锁是谁？字节码文件对象；
        if(s == null){
            synchronized(Single.class){
                if(s == null)
```

```
                s = new Single();
            }
        }
        return s;
    }
}
```

等待唤醒机制：涉及的方法：
wait:将同步中的线程处于冻结状态。释放了执行权，释放了资格。同时将线程对象存储到线程池中。
notify：唤醒线程池中某一个等待线程。
notifyAll:唤醒的是线程池中的所有线程。
注意：
1：**这些方法都需要定义在同步中**。
2：因为这些方法必须要标示所属的锁。
你要知道 A 锁上的线程被 wait 了,那这个线程就相当于处于 A 锁的线程池中，只能 A 锁的 notify 唤醒。
3：这三个方法都定义在 Object 类中。为什么操作线程的方法定义在 Object 类中？
因为这三个方法都需要定义同步内，并标示所属的同步锁，既然被锁调用，而锁又可以是任意对象，那么能被任意对象调用的方法一定定义在 Object 类中。
wait 和 sleep 区别： 分析这两个方法：从执行权和锁上来分析：
wait：可以指定时间也可以不指定时间。不指定时间，只能由对应的 notify 或者 notifyAll 来唤醒。
sleep：必须指定时间，时间到自动从冻结状态转成运行状态(临时阻塞状态)。
wait：线程会释放执行权，而且线程会释放锁。
sleep：线程会释放执行权，但不是不释放锁。
线程的停止：通过 stop 方法就可以停止线程。但是这个方式过时了。
停止线程：原理就是：让线程运行的代码结束，也就是结束 run 方法。
怎么结束 run 方法？一般 run 方法里肯定定义循环。所以只要结束循环即可。
第一种方式：**定义循环的结束标记。**
第二种方式：如果线程处于了冻结状态，是不可能读到标记的，这时就需要**通过 Thread 类中的 interrupt 方法，将其冻结状态强制清除**。让线程恢复具备执行资格的状态，让线程可以读到标记，并结束。
---------< java.lang.Thread >----------
interrupt()：中断线程。
setPriority(int newPriority)：更改线程的优先级。
getPriority()：返回线程的优先级。
toString()：返回该线程的字符串表示形式，包括线程名称、优先级和线程组。
Thread.yield()：暂停当前正在执行的线程对象，并执行其他线程。
setDaemon(true)：将该线程标记为守护线程或用户线程。将该线程标记为守护线程或用户线程。当正在运行的线程都是守护线程时，Java 虚拟机退出。该方法必须在启动线程前调用。

join：临时加入一个线程的时候可以使用 join 方法。
当 A 线程执行到了 B 线程的 join 方式。A 线程处于冻结状态，释放了执行权，B 开始执行。A 什么时候执行呢？只有当 B 线程运行结束后，A 才从冻结状态恢复运行状态执行。
LOCK 的出现替代了同步：lock.lock();……lock.unlock();
Lock 接口：多线程在 JDK1.5 版本升级时，推出一个接口 Lock 接口。
解决线程安全问题使用同步的形式，(同步代码块，要么同步函数)其实最终使用的都是锁机制。
到了后期版本，直接将锁封装成了对象。线程进入同步就是具备了锁，执行完，离开同步，就是释放了锁。
在后期对锁的分析过程中，发现，获取锁，或者释放锁的动作应该是锁这个事物更清楚。所以将这些动作定义在了锁当中，并把锁定义成对象。
所以**同步是隐示的锁操作，而 Lock 对象是显示的锁操作**，它的出现就替代了同步。
在之前的版本中使用 Object 类中 wait、notify、notifyAll 的方式来完成的。那是因为同步中的锁是任意对象，所以操作锁的等待唤醒的方法都定义在 Object 类中。
而现在锁是指定对象 Lock。所以查找等待唤醒机制方式需要通过 Lock 接口来完成。而 Lock 接口中并没有直接操作等待唤醒的方法，而是将这些方式又单独封装到了一个对象中。这个对象就是 **Condition**，将 Object 中的三个方法进行单独的封装。并提供了功能一致的方法 **await()、signal()、signalAll()**体现新版本对象的好处。
< java.util.concurrent.locks > Condition 接口：await()、signal()、signalAll();

```
class BoundedBuffer {

final Lock lock = new ReentrantLock();

final Condition notFull = lock.newCondition();

final Condition notEmpty = lock.newCondition();

final Object[] items = new Object[100];

int putptr, takeptr, count;

public void put(Object x) throws InterruptedException {

lock.lock();

try {

while (count == items.length)
```

```
notFull.await();
items[putptr] = x;
if (++putptr == items.length) putptr = 0;
++count;
notEmpty.signal();
}
    finally {
lock.unlock();
}
}
public Object take() throws InterruptedException {
lock.lock();
try {
while (count == 0)
notEmpty.await();
Object x = items[takeptr];
if (++takeptr == items.length) takeptr = 0;
--count;
notFull.signal();
return x;
}
finally {
```

```
lock.unlock();

}

}

}
```

集合框架

集合框架：★★★★★，用于存储数据的容器。

对于集合容器，有很多种。因为每一个容器的自身特点不同，其实原理在于每个容器的内部数据结构不同。
集合容器在不断向上抽取过程中。出现了集合体系。
在使用一个体系时，原则：参阅顶层内容。建立底层对象。

--< java.util >-- List 接口：
List 本身是 Collection 接口的子接口，具备了 Collection 的所有方法。现在学习 List 体系特有的共性方法，查阅方法发现 List 的特有方法都有索引，这是该集合最大的特点。
List：有序(元素存入集合的顺序和取出的顺序一致)，元素都有索引。元素可以重复。
|--ArrayList：底层的数据结构是数组,线程不同步，ArrayList 替代了 Vector，查询元素的速度非常快。
|--LinkedList：底层的数据结构是链表，线程不同步，增删元素的速度非常快。
|--Vector：底层的数据结构就是数组，线程同步的，Vector 无论查询和增删都巨慢。
可变长度数组的原理：
当元素超出数组长度，会产生一个新数组，将原数组的数据复制到新数组中，再将新的元素添加到新数组中。
ArrayList：是按照原数组的 50%延长。构造一个初始容量为 10 的空列表。
Vector：是按照原数组的 100%延长。

--< java.util >-- Set 接口：
数据结构：数据的存储方式；
Set 接口中的方法和 Collection 中方法一致的。Set 接口取出方式只有一种，**迭代器**。
|--HashSet：底层数据结构是哈希表，线程**是不同步的。无序，高效；**

HashSet 集合保证元素唯一性：通过元素的 hashCode 方法，和 equals 方法完成的。

当元素的 hashCode 值相同时，才继续判断元素的 equals 是否为 true。

如果为 true，那么视为相同元素，不存。如果为 false，那么存储。

如果 hashCode 值不同，那么不判断 equals，从而提高对象比较的速度。

|--LinkedHashSet：有序，hashset 的子类。

|--TreeSet：对 Set 集合中的元素的进行指定顺序的排序。**不同步**。TreeSet 底层的数据结构就是二叉树。

对于 ArrayList 集合，判断元素是否存在，或者删元素底层依据都是 equals 方法。

对于 HashSet 集合，判断元素是否存在，或者删除元素，底层依据的是 hashCode 方法和 equals 方法。

Map 集合：

|--Hashtable：底层是哈希表数据结构，是**线程同步**的。不可以存储 null 键，null 值。

|--HashMap：底层是哈希表数据结构，是**线程不同步**的。可以存储 null 键，null 值。替代了 Hashtable.

|--TreeMap：底层是二叉树结构，可以对 map 集合中的键进行指定顺序的排序。

Map 集合存储和 Collection 有着很大不同：

Collection 一次存一个元素；Map 一次存一对元素。

Collection 是单列集合；Map 是双列集合。

Map 中的存储的一对元素：一个是键，一个是值，**键与值之间有对应(映射)关系。**

特点：要保证 map 集合中键的唯一性。

5，想要获取 map 中的所有元素：

原理：map 中是没有迭代器的，collection 具备迭代器，只要将 map 集合转成 Set 集合，可以使用迭代器了。之所以转成 set，是因为 map 集合具备着键的唯一性，其实 set 集合就来自于 map，set 集合底层其实用的就是 map 的方法。

把 map 集合转成 set 的方法：

Set keySet();

Set entrySet();//取的是键和值的映射关系。

Entry 就是 Map 接口中的内部接口；

为什么要定义在 map 内部呢？entry 是访问键值关系的入口，是 map 的入口，访问的是 map 中的键值对。

取出 map 集合中所有元素的方式一：keySet()方法。

可以将 map 集合中的键都取出存放到 set 集合中。对 set 集合进行迭代。迭代完成，再通过 get 方法对获取到的键进行值的获取。

```
Set keySet = map.keySet();

            Iterator it = keySet.iterator();
```

```
            while(it.hasNext()) {

                    Object key = it.next();

                    Object  value  = map.get(key);

                    System.out.println(key+":"+value);

            }
```

取出 **map** 集合中所有元素的方式二：**entrySet()**方法。

```
Set  entrySet = map.entrySet();

            Iterator it = entrySet.iterator();

            while(it.hasNext()) {

                    Map.Entry me = (Map.Entry)it.next();

                    System.out.println(me.getKey()+"::::"+me.getValue())
;

            }
```

将非同步集合转成同步集合的方法：**Collections** 中的 XXX

```
synchronizedXXX(XXX);

List  synchronizedList(list);

Map  synchronizedMap(map);

public  static  <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {

return  new  SynchronizedMap<K,V>(m);

}
```

原理：定义一个类，将集合所有的方法加同一把锁后返回。

List list = Collections.synchronizedList(new ArrayList());

Map<String,String> synmap = Collections.synchronizedMap(map);

Collection 和 **Collections** 的区别：

Collections 是个 java.util 下的类，是针对集合类的一个工具类,提供一系列静态方法,实现对集合的查找、排序、替换、线程安全化（将非同步的集合转换成同步的）等操作。
Collection 是个 java.util 下的接口，它是各种集合结构的父接口，继承于它的接口主要有 Set 和 List,提供了关于集合的一些操作,如插入、删除、判断一个元素是否其成员、遍历等。

自动拆装箱：java 中数据类型分为两种 ： 基本数据类型 引用数据类型(对象)
在 java 程序中所有的数据都需要当做对象来处理，针对 8 种基本数据类型提供了包装类，如下：
int --> Integer
byte --> Byte
short --> Short
long --> Long
char --> Character
double --> Double
float --> Float
boolean --> Boolean
jdk5 以前基本数据类型和包装类之间需要互转：
基本---引用 Integer x = new Integer(x);
引用---基本 int num = x.intValue();
1)、Integer x = 1; x = x + 1; 经历了什么过程？**装箱 à 拆箱 à 装箱**；
2)、为了优化，虚拟机为包装类提供了缓冲池，**Integer 池的大小 -128~127 一个字节的大小**；
3)、**String 池：**Java 为了优化字符串操作 提供了一个缓冲池；

泛型：jdk1.5 版本以后出现的一个安全机制。表现格式：**< >**
好处：
1：将运行时期的问题 ClassCastException 问题转换成了编译失败，体现在编译时期，程序员就可以解决问题。
2：避免了强制转换的麻烦。
泛型中的通配符：可以解决当具体类型不确定的时候，这个通配符就是 **?** ；当操作类型时，不需要使用类型的具体功能时，只使用 Object 类中的功能。那么可以用 ？ 通配符来表未知类型。

反射技术
反射技术：其实就是动态加载一个指定的类，并获取该类中的所有的内容。并将字节码文件中的内容都封装成对象，这样便于操作这些成员。简单说：**反射技术可以对一个类进行解剖。**
反射的好处：大大的增强了程序的扩展性。
反射的基本步骤：
1、获得 Class 对象，就是获取到指定的名称的字节码文件对象。
2、实例化对象，获得类的属性、方法或构造函数。

3、访问属性、调用方法、调用构造函数创建对象。

获取这个 Class 对象，有三种方式：

1：通过每个对象都具备的方法 getClass 来获取。弊端：必须要创建该类对象，才可以调用 getClass 方法。

2：每一个数据类型(基本数据类型和引用数据类型)都有一个**静态的属性 class**。弊端：必须要先明确该类。

前两种方式不利于程序的扩展，因为都需要在程序使用具体的类来完成。

3：使用的 Class 类中的方法，**静态的 forName 方法**。

指定什么类名，就获取什么类字节码文件对象，这种方式的扩展性最强，只要将类名的字符串传入即可。

```
// 1. 根据给定的类名来获得 用于类加载
String classname = "cn.itcast.reflect.Person";// 来自配置文件
Class clazz = Class.forName(classname);// 此对象代表 Person.class
// 2. 如果拿到了对象，不知道是什么类型 用于获得对象的类型
Object obj = new Person();
Class clazz1 = obj.getClass();// 获得对象具体的类型
// 3. 如果是明确地获得某个类的 Class 对象 主要用于传参
Class clazz2 = Person.class;
```

反射的用法：

1）、需要获得 java 类的各个组成部分，首先需要获得类的 Class 对象，获得 Class 对象的三种方式：

Class.forName(classname) 用于做类加载

obj.getClass() 用于获得对象的类型

类名.class 用于获得指定的类型，传参用

2)、反射类的成员方法：

```
Class clazz = Person.class;
Method method = clazz.getMethod(methodName, new
Class[]{paramClazz1, paramClazz2});
method.invoke();
```

3)、反射类的构造函数：

```
Constructor con = clazz.getConstructor(new Class[]{paramClazz1,
paramClazz2,...})
con.newInstance(params...)
```

4)、反射类的属性：

```
Field field = clazz.getField(fieldName);
field.setAccessible(true);
field.setObject(value);
```

获取了字节码文件对象后，最终都需要创建指定类的对象：

创建对象的两种方式(其实就是对象在进行实例化时的初始化方式)：

1，调用空参数的构造函数：使用了 Class 类中的 **newInstance()**方法。

2，调用带参数的构造函数：先要获取指定参数列表的构造函数对象，然后通过该构造函数的**对象的 newInstance(实际参数)** 进行对象的初始化。

综上所述，第二种方式，必须要先明确具体的构造函数的参数类型，不便于扩展。**所以一般情况下，被反射的类，内部通常都会提供一个公有的空参数的构造函数。**

```
   // 如何生成获取到字节码文件对象的实例对象。
      Class clazz = Class.forName("cn.itcast.bean.Person");//类加载
// 直接获得指定的类型
      clazz = Person.class;
      // 根据对象获得类型
      Object obj = new Person("zhangsan", 19);
      clazz = obj.getClass();
      Object obj = clazz.newInstance();//该实例化对象的方法调用就是指定类中的空参数构造函数，给创建对象进行初始化。当指定类中没有空参数构造函数时，该如何创建该类对象呢？请看 method_2();
   public static void method_2() throws Exception {
      Class clazz = Class.forName("cn.itcast.bean.Person");
      //既然类中没有空参数的构造函数,那么只有获取指定参数的构造函数,用该函数来进行实例化。
      //获取一个带参数的构造器。
      Constructor constructor = clazz.getConstructor(String.class,int.class);
      //想要对对象进行初始化，使用构造器的方法 newInstance();
      Object obj = constructor.newInstance("zhagnsan",30);
      //获取所有构造器。
      Constructor[] constructors = clazz.getConstructors();//只包含公共的
      constructors = clazz.getDeclaredConstructors();//包含私有的
      for(Constructor con : constructors) {
         System.out.println(con);
      }
   }
```

反射指定类中的方法：

```
   //获取类中所有的方法。
   public static void method_1() throws Exception {
      Class clazz = Class.forName("cn.itcast.bean.Person");
      Method[] methods = clazz.getMethods();//获取的是该类中的公有方法和父类中的公有方法。
      methods = clazz.getDeclaredMethods();//获取本类中的方法，包含私有方法。
      for(Method method : methods) {
         System.out.println(method);
      }
   }
   //获取指定方法；
   public static void method_2() throws Exception {
```

```
        Class clazz = Class.forName("cn.itcast.bean.Person");
        //获取指定名称的方法。
        Method method = clazz.getMethod("show", int.class,String.class);
        //想要运行指定方法，当然是方法对象最清楚，为了让方法运行，调用方法对象的 invoke 方法即可，但是方法运行必须要明确所属的对象和具体的实际参数。
        Object obj = clazz.newInstance();
        method.invoke(obj, 39,"hehehe");//执行一个方法
    }
    //想要运行私有方法。
    public static void method_3() throws Exception {
        Class clazz = Class.forName("cn.itcast.bean.Person");
        //想要获取私有方法。必须用 getDeclearMethod();
        Method method = clazz.getDeclaredMethod("method", null);
        // 私有方法不能直接访问，因为权限不够。非要访问，可以通过暴力的方式。
        method.setAccessible(true);//一般很少用，因为私有就是隐藏起来，所以尽量不要访问。
    }
    //反射静态方法。
    public static void method_4() throws Exception {
        Class clazz = Class.forName("cn.itcast.bean.Person");
        Method method = clazz.getMethod("function",null);
        method.invoke(null,null);
    }
```