1. **面相对象的特征**：

**抽象**：数据抽象和行为抽象。是将一类对象的共同特征总结出来构造类的过程。只关注对象有哪些属性和行为，不关心这些行为的细节。

用到java上，就比方，我要定义一个方法和类。但还没确定怎么去实现它的具体一点的子方法，那我就可以用抽象类或接口

**继承**：子类继承父类。

**封装**：把数据和数据的方法绑定起来。对数据的访问只能通过已定义的接口。 比如，类中的private数据，它的get和set方法

**多态性**：同一个行为 具有多个不同表现形式或形态的能力。即，同样的对象应用 调用 同样的方法，却做了不同的操作。比如： Animol a = new Cat(); Animol b = new Dog(). a.eat(); b.eat(); 同样是Animol的引用，却做了不同的操作。

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，再去调用子类的同名方法。

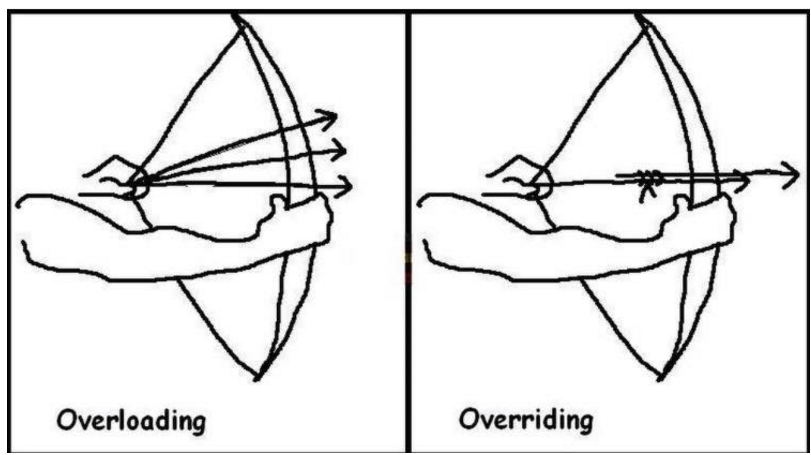
多态的好处：可以使程序有良好的扩展，并可以对所有类的对象进行通用处理。

运行时多态：override 重写，是运行时多态。子类的方法重写父类的方法---在编译阶段，只是检查参数的引用类型。然而在运行时，Java虚拟机(JVM)指定对象的类型并且运行该对象的方法。 编译时，Animol 有方法 eat()，运行时,运行 cat和 dog的eat()

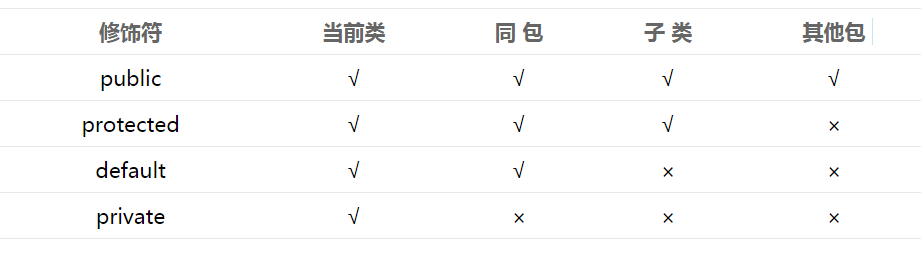
编译时多态：overload 重载，是编译时多态。重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。最常用的地方就是构造器的重载。

**方法重载是一个类的多态性表现,而方法重写是子类与父类的一种多态性表现**。





**2. 访问修饰符public,private,protected,以及不写（默认）时的区别？**



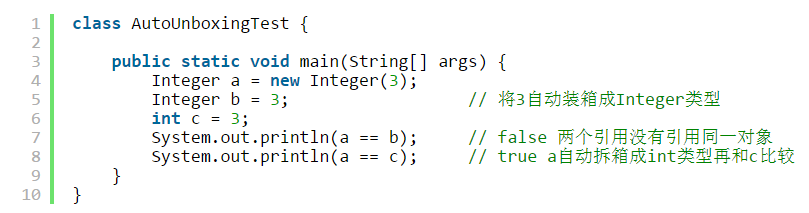
**3、String 是最基本的数据类型吗？**  
答：不是。Java中的基本数据类型只有8个：byte、short、int、long、float、double、char、boolean；除了基本类型（primitive type）和枚举类型（enumeration type），剩下的都是引用类型（reference type）。

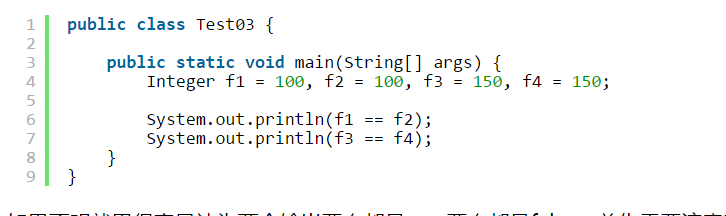
**4、float f=3.4;是否正确？**  
答:不正确。3.4是双精度数，将双精度型（double）赋值给浮点型（float）属于下转型（down-casting，也称为窄化）会造成精度损失，因此需要强制类型转换float f =(float)3.4; 或者写成float f =3.4F;。

**5、short s1 = 1; s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1;有错吗？**  
答：对于short s1 = 1; s1 = s1 + 1;由于1是int类型，因此s1+1运算结果也是int 型，需要强制转换类型才能赋值给short型。而short s1 = 1; s1 += 1;可以正确编译，因为s1+= 1;相当于s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

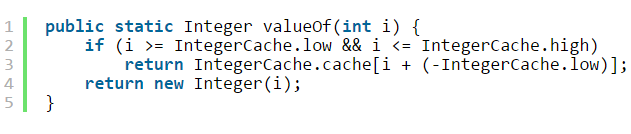
**6、Java有没有goto？**  
答：goto 是Java中的保留字，在目前版本的Java中没有使用。（根据James Gosling（Java之父）编写的《The Java Programming Language》一书的附录中给出了一个Java关键字列表，其中有goto和const，但是这两个是目前无法使用的关键字，因此有些地方将其称之为保留字，其实保留字这个词应该有更广泛的意义，因为熟悉C语言的程序员都知道，在系统类库中使用过的有特殊意义的单词或单词的组合都被视为保留字）

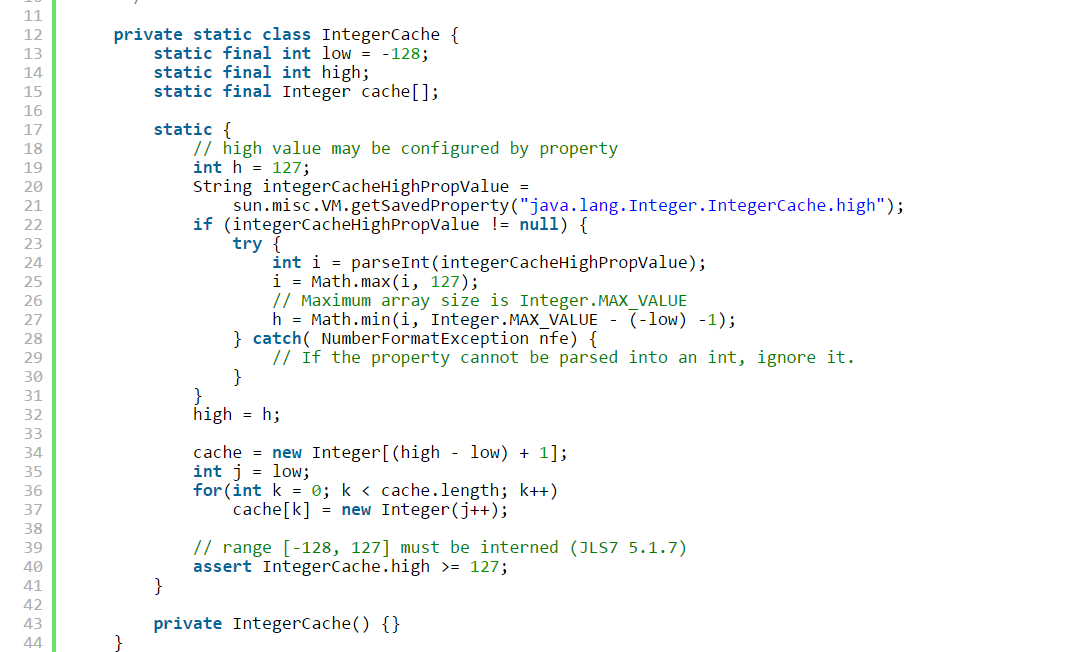
**7、int和Integer有什么区别？**  
答：Java是一个近乎纯洁的面向对象编程语言，但是为了编程的方便还是引入了基本数据类型，但是为了能够将这些基本数据类型当成对象操作，Java为每一个基本数据类型都引入了对应的包装类型（wrapper class），int的包装类就是Integer，从Java 5开始引入了自动装箱/拆箱机制，使得二者可以相互转换。  
Java 为每个原始类型提供了包装类型：  
- 原始类型: boolean，char，byte，short，int，long，float，double  
- 包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double





如果不明就里很容易认为两个输出要么都是true要么都是false。首先需要注意的是f1、f2、f3、f4四个变量都是Integer对象引用，所以下面的==运算比较的不是值而是引用。装箱的本质是什么呢？当我们给一个Integer对象赋一个int值的时候，会调用Integer类的静态方法valueOf，如果看看valueOf的源代码就知道发生了什么。

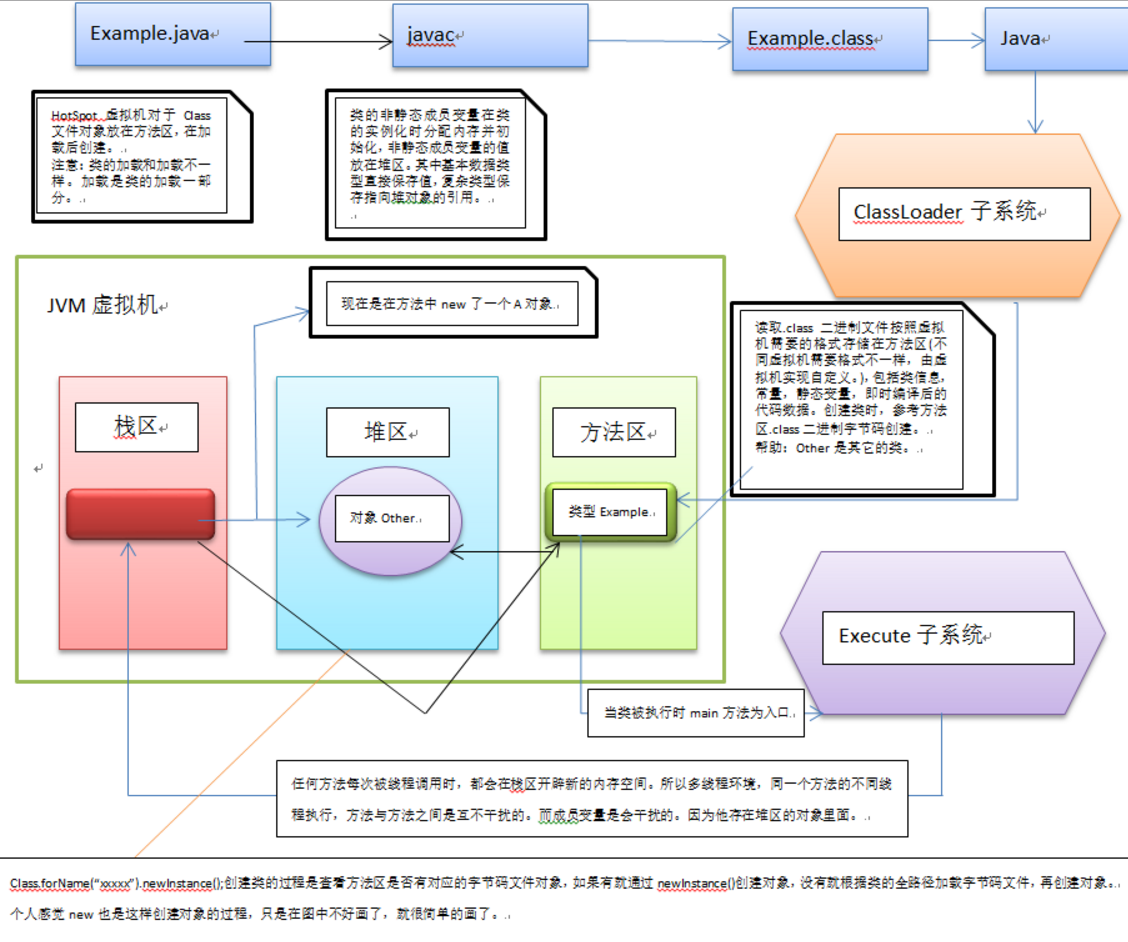


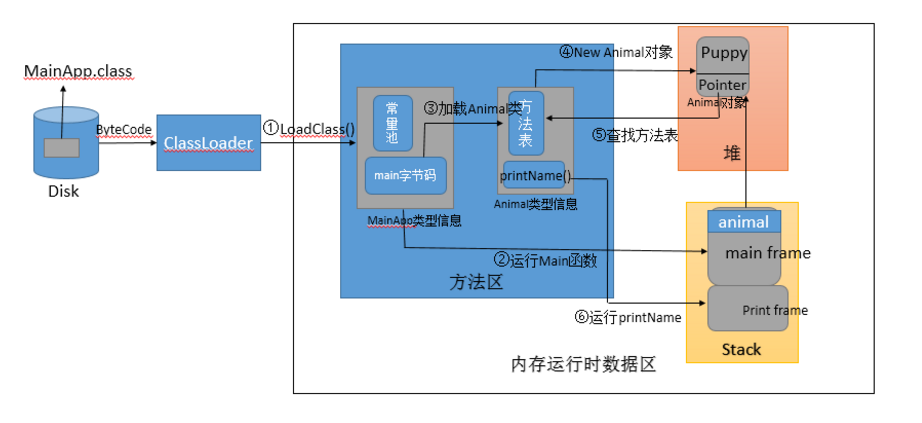


简单的说，如果整型字面量的值在-128到127之间，那么不会new新的Integer对象，而是直接引用常量池中的Integer对象，所以上面的面试题中f1==f2的结果是true，而f3==f4的结果是false

**8、&和&&的区别？**  
答：&运算符有两种用法：(1)按位与；(2)逻辑与。&&运算符是短路与运算。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是true整个表达式的值才是true。&&之所以称为短路运算是因为，如果&&左边的表达式的值是false，右边的表达式会被直接短路掉，不会进行运算。很多时候我们可能都需要用&&而不是&，例如在验证用户登录时判定用户名不是null而且不是空字符串，应当写为：username != null &&!username.equals(“”)，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立，根本不能进行字符串的equals比较，否则会产生NullPointerException异常。注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。

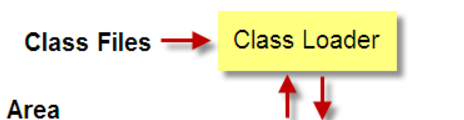
**9、解释内存中的栈(stack)、堆(heap)和静态区(static area)的用法。---------------**《JVM Specification 》java内存的书

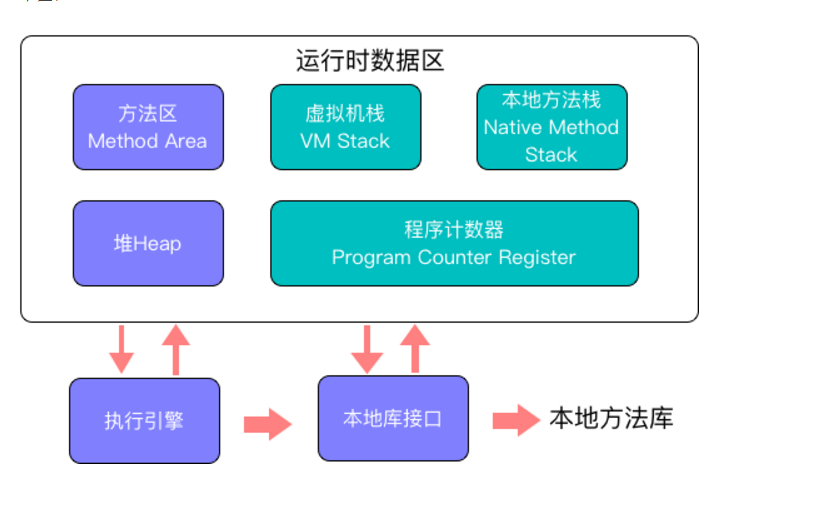






**Java虚拟机内存模型：**





**## Class Loader 类加载器**

类加载器的作用是加载类文件到内存，比如编写一个HelloWord.java 程序，然后通过javac 编译成class 文件，那怎么才能加载到内存中被执行呢？Class Loader 承担的就是这个责任，那不可能随便建立一个.class 文件就能被加载的，Class Loader 加载的class 文件是有格式要求

需要详细了解的话，可以仔细阅读《JVM Specification 》的第四章“The class File Format ”，这里不再详细说明。

友情提示：Class Loader 只管加载，只要符合文件结构就加载，至于说能不能运行，则不是它负责的，那是由Execution Engine 负责的。

**## Execution Engine 执行引擎**

执行引擎也叫做解释器(Interpreter) ，负责解释命令，提交操作系统执行。

**## Native Interface 本地接口**

本地接口的作用是融合不同的编程语言为Java 所用，它的初衷是融合C/C++ 程序，Java 诞生的时候是C/C++ 横行的时候，要想立足，必须有一个聪明的、睿智的调用C/C++ 程序，于是就在内存中专门开辟了一块区域处理标记为native 的代码，它的具体做法是Native Method Stack 中登记native 方法，在Execution Engine 执行时加载native libraies 。目前该方法使用的是越来越少了，除非是与硬件有关的应用，比如通过Java 程序驱动打印机，或者Java 系统管理生产设备，在企业级应用中已经比较少见，因为现在的异构领域间的通信很发达，比如可以使用Socket 通信，也可以使用Web Service 等等，不多做介绍。

**## Runtime data area 运行数据区**

运行数据区是整个JVM 的重点。我们所有写的程序都被加载到这里，之后才开始运行，Java 生态系统如此的繁荣，得益于该区域的优良自治。

整个JVM 框架由加载器加载文件，然后执行器在内存中处理数据，需要与异构系统交互是可以通过本地接口进行，瞧，一个完整的系统诞生了！

### 程序计数器

寄存器里面有一个叫指令寄存器，用来储存现在正在被运行的指令。想象一下，在JVM中怎么办，程序寄存器就是这样的功能。

程序技术器是一块较小的内存空间，它可以看做是当前线程所执行的字节码的行号指示器。“字节码”就是Java程序被编译之后的形态，JVM有字节码解释器，这个解释器要解释程序的哪段，就由这个程序技术器来决定的

### Java虚拟机栈

Java虚拟机栈也是线程私有的，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。每个方法在执行的同时会创建栈帧，保存

* 局部变量表
* 操作数栈
* 动态链接
* 方法出口  
  每个方法从调用直到执行完成的过程，就对应整一个栈帧在虚拟机栈中入栈和出栈的过程。  
  平时咱们在讨论的时候总会提到“栈”和“堆”这两种内存区域，那么其中的栈，就是这里所指的栈，更细一点说，就是虚拟机栈中局部变量的部分。
* 在虚拟机栈中，使用栈指针来访问处理器。我们都学过栈这种数据结构，它是一种快速有效的 **分配** 存储的方法,存储速度仅次于寄存器，堆栈指针若向下移动，则分配新的内存，若向上移动则释放那些内存。由于Java编译器需要预先去生成相应的内存空间，所以，当我们尝试创建程序的时候，Java编译器必须知道被存储在栈内的所有数据的确切大小和声明周期。以便可以像上面描述的那样去分配内存空间。

答：java源文件 编译后的.class字节码文件的主要内容是 **常量池**和**方法字节码** 常量池记录的是代码出现过的所有token(类名，成员变量名等等)以及符号引用（方法引用，成员变量引用等等）；方法字节码放的是类中各个方法的字节码

编译后的加载，把.class字节码文件 加载到内存，即 把类信息加载到 方法区。

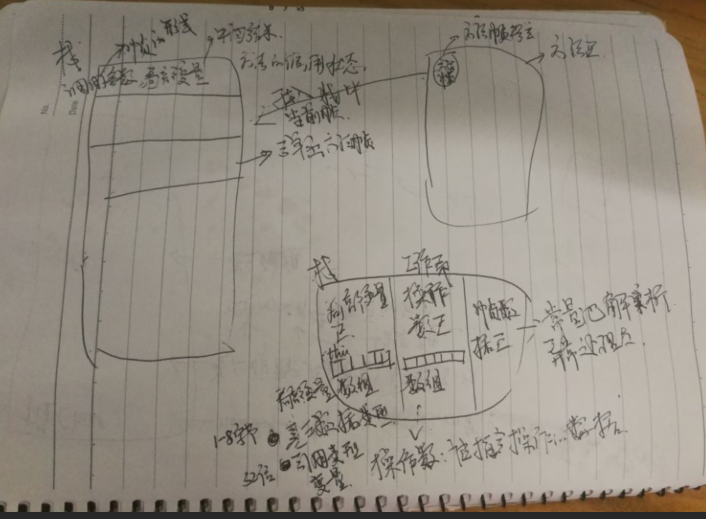
jvm可以允许用户和程序指定方法区的初始大小，最小和最大尺寸。

类实例化一个对象后，对象保存指向方法区的该类的类型信息（其中包含有方法表，java动态绑定的底层实现）的引用。

调用对象的方法时，JVM找到对象的指针，在根据该对象的持有的引用 ，定位到**方法区**中的 该类的 **类型信息**的**方法表**。。获得方法的字节码地址，然后执行方法

特别说明：java类中所有public和protected的实例方法都采用**动态绑定机制**，所有私有方法、静态方法、构造器及初始化方法<clinit>都是采用静态绑定机制。而使用动态绑定机制的时候会用到**方法表**，静态绑定时并不会用到。

**栈：**



在Java中，一个线程就会有一个线程的JVM栈与之对应，因为不同的线程执行逻辑显然不同，因此都需要一个独立的JVM栈来存放该线程的执行逻辑。

   Java栈内存，以帧的形式存放本地方法的调用状态，包括方法调用的参数、局部变量、中间结果等（方法都是以方法帧的形式存放在方法区的），每调用一个方法就将对应该方法的方法帧压入Java栈，成为当前方法帧。当调用结束(返回)时，就弹出该帧。

 Java 栈内存的组成：  
局部变量区、操作数栈、帧数据区组成。  
（1）：局部变量区为一个以字为单位的数组，每个数组元素对应一个局部变量的 值。调用方法时，将方法的局部变量组成一个数组，通过索引来访问。若为非静态方法，则加入一个隐含的引用参数this,该参数指向调用这个方法的对象。而 静态方法则没有this参数。因此，对象无法调用静态方法。

由此，我们可以知道，方法什么时候设计为静态，什么时候为非静态？

|  |
| --- |
| 前面已经说过，对象是类的一个实例，各个对象结构相同，只是属性不同。 而静态方法是对象无法调用的。 所以，静态方法适合那些工具类中的工具方法，这些类只是用来实现一些功能，也不需要产生对象，通过设置对象的属性来得到各个不同的个体。 |

（2）：操作数栈也是一个数组，但是通过栈操作来访问。所谓操作数是那些被指令操作的数据。当需要对参数操作时如a=b+c,就将即将被操作的参数压栈，如将b 和c 压栈，然后由操作指令将它们弹出，并执行操作。虚拟机将操作数栈作为工作区。  
（3）：帧数据区处理常量池解析，异常处理等

1.定义一个基本数据类型的变量，

2.一个对象的引用

3.函数调用的现场保存(是指线程调用方法在栈内开辟的内存空间？变量的copy??)（局部变量，当一段代码或者一个方法调用完毕后，栈中为这段代码所提供的基本数据类型或者对象的引用立即被释放）

**堆：**1.通过new关键字和构造器创建的对象和数组放在堆空间；(非静态的成员变量，在实例化时开辟空间初始化值)。。对象不能直接访问，必须由栈中的引用类型变量指向堆中的对象

**静态区：**方法区里面的一块区域

1程序中的字面量（literal）如直接书写的100、”hello”（静态成员变量）

2.2.常量都是放在静态区中。

**常量池：**

Java中的常量池，实际上分为两种形态：静态常量池和运行时常量池。

 1）所谓静态常量池，即\*.class文件中的常量池，class文件中的常量池不仅仅包含字符串(数字)字面量，还包含类、方法的信息，占用class文件绝大部分空间。

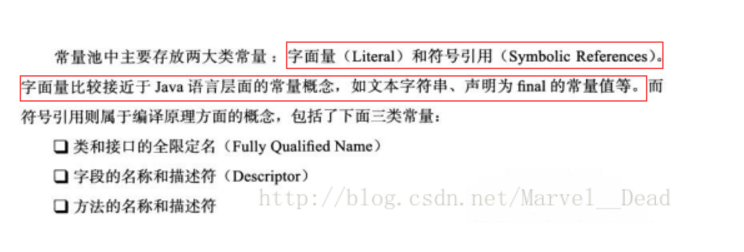
2） 而运行时常量池，则是jvm虚拟机在完成类装载操作后，将class文件中的常量池载入到内存中，并保存在方法区中，我们常说的常量池，就是指方法区中的运行时常量池。

 Java中八种**基本类型**的包装类的大部分都实现了常量池技术，它们是Byte、Short、Integer、Long、Character、Boolean，另外两种浮点数类型的包装类(Float、Double)则没有实现。另外Byte,Short,Integer,Long,Character这5种整型的包装类也只是在对应值在-128到127时才可使用对象池。

一个Class字节码文件的Class字节码文件对象只有一个常量池，常量池就是这个类型用到的常量的一个有序集合，包括实际的常量(string,   
integer, 和floating point常量)和对类型，域和方法的符号引用。池中的数据项象数组项一样，是通过索引访问的。

常量池被所有线程共享。。常量池是方法区的一部分内存。在编译期间就将一部分数据存放于该区域，包含基本数据类型如int、long等以final声明的常量值，和String字符串、特别注意的是对于方法运行期位于栈中的局部变量String常量的值可以通过 String.intern()方法将该值置入到常量池中。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**

因为常量池存储了一个类型所使用到的所有类型，域和方法的符号引用，所以它在java程序的动态链接中起了核心的作用。





常量池的好处

常量池是为了避免频繁的创建和销毁对象而影响系统性能，其实现了对象的共享。例如字符串常量池，在编译阶段就把所有的字符串文字放到一个常量池中。

（1）节省内存空间：常量池中所有相同的字符串常量被合并，只占用一个空间。

（2）节省运行时间：比较字符串时，==比equals()快。对于两个引用变量，只用==判断引用是否相等，也就可以判断实际值是否相等

对于通过 new 产生一个字符串（假设为 ”china” ）时，会先去常量池中查找是否已经有了 ”china” 对象，如果没有则在常量池中创建一个此字符串对象，然后堆中再创建一个常量池中此 ”china” 对象的拷贝对象。这也就是有道面试题： String s = new String(“xyz”); 产生几个对象？一个或两个，如果常量池中原来没有 ”xyz”, 就是两个。

**方法区:** 包含静态区。是各个线程共享的内存区域，它用于存储class二进制文件，包含了虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。它有个名字叫做Non-Heap(非堆)，目的是与Java堆区分开。

方法区是线程安全的。由于所有的线程都共享方法区，所以，方法区里的数据访问必须被设计成线程安全的。例如，假如同时有两个线程都企图访问方法区中的同一个类，而这个类还没有被装入JVM，那么只允许一个线程去装载它，而其它线程必须等待 ！

一个类里面的静态成员变量和静态成员方法，它是存储在方法区的，静态成员变量是在方法区的静态域里面，而静态成员方法是在方法区的class二进制信息里面(.class文件和方法区里面的二进制信息不一样，读取.class文件按照虚拟机需要的格式存储在方法区。这种格式包括数据结构方面)，静态成员和静态成员方法使用时不用创建对象，即类加载初始化后就可以使用，并且是线程共享的。

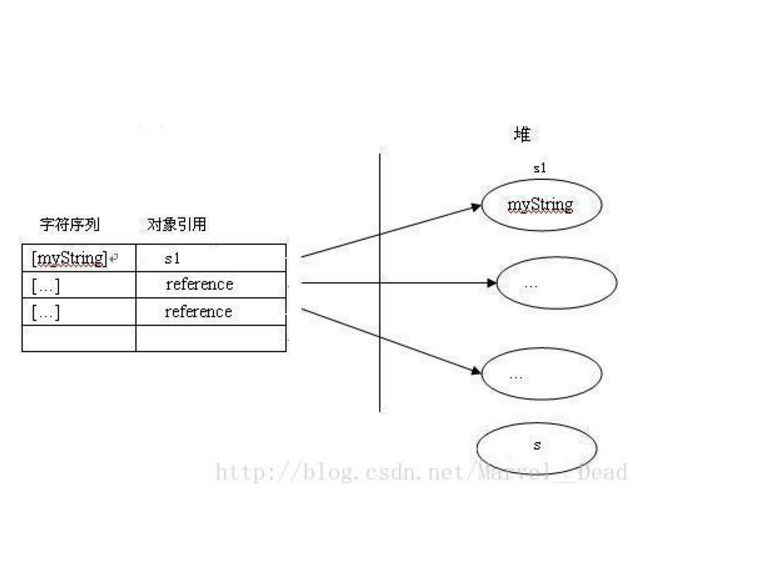
栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，理论上整个内存没有被其他进程使用的空间甚至硬盘上的虚拟内存都可以被当成堆空间来使用。



上面的语句中变量str放在栈上，用new创建出来的字符串对象放在堆上，而”hello”这个字面量放在静态区。

内部字符串常量列表

**String 对象与字符串常量的关系：**



由此我们可以知道反射的基础：

|  |
| --- |
| 在装载类的时候，加入方法区中的所有信息，最后都会形成Class类的实例，代表这个被装载的类。方法区中的所有的信息，都是可以通过这个Class类对象反射得到。我们知道对象是类的实例，类是相同结构的对象的一种抽象。同类的各个对象之间，其实是拥有相同的结构（属性），拥有相同的功能（方法），各个对象的区别只在于属性值的不同。     同样的，我们所有的类，其实都是Class类的实例，他们都拥有相同的结构-----Field数组、Method数组。而各个类中的属性都是Field属性的一个具体属性值，方法都是Method属性的一个具体属性值。 |

**10、Math.round(11.5) 等于多少？Math.round(-11.5)等于多少？**  
答：Math.round(11.5)的返回值是12，Math.round(-11.5)的返回值是-11。四舍五入的原理是在参数上加0.5然后进行下取整。

Math.round :Returns the closest long to the argument, with ties rounding to positive infinity..即，返回一个最近的一个long值，也就是，四舍五入算法为Math.floor(x+0.5),即将原来的数字加上0.5后再向下取整

Math. Ceil 天花板，向上取整，Math.ceil（11.3）的结果为12，Math.ceil(-11.6)的结果为-11；

Math. Floor 该方法就表示向下取整，Math.floor(11.6)的结果是11，Math.floor(-11.4)的结果-12；