1、创建对象的几种方式：

1）直接 new 一个对象

变形：**Xxx 静态方法中私有构造器**或**XxxBuilder/XxxFactory 的静态方法**；

2）Class 的 newInstance()：反射的方式，只能**调用空参的构造器**，权限必须是 **public**；

3）Constructor 的 newInstance(Xxx)：反射的方式，可以调用**空参、带参**的构造器，**权限没有要求**，实用性更广；

4）使用 clone()：不调用任何构造器，**当前类需要实现 Cloneable 接口**，实现 clone()，默认**浅拷贝**，即指向同一个实例对象；

5）使用反序列化：从文件中、数据库中、网络中获取**一个对象的二进制流，反序列化**为内存中的对象；

6）第三方库 Objenesis，利用了 **asm 字节码技术**，动态生成 Constructor 对象。

2、从执行步骤角度分析对象的创建过程：

1）判断对象对应类是否加载、链接、初始化；

虚拟机遇到**一条 new 指令**，首先去**检查这个指令的参数**能否**在元空间的常量池中**定位到一个类的**符号引用**，并且检查这个符号引用代表的类**是否已经被加载、解析和初始化**。

>如果**没有**，那么在双亲委派模式下，使用当前类加载器以 ClassLoader+包名+类名为 Key 进行**查找对应的.class 文件**。

>如果**没有找到文件**，则抛出 **ClassNotFoundException 异常**。

>如果**找到**，则进行**类加载**，并**生成对应的 Class 类对象**。

2）为对象分配内存；

首先**计算对象占用空间**大小，接着**在堆中划分一块内存**给新对象，如果实例成员变量是**引用变量**，仅**分配**引用变量空间即可，即 **4 个字节大小**。（分配方法：指针碰撞和空闲列表）

3）处理并发安全问题；（两种方式）

>**CAS**（Compare And Swap）**失败重试、区域加锁**：保证指针更新操作的原子性；

>**TLAB** 把内存分配的动作按照**线程划分在不同的空间之中**进行。

4）初始化分配到的空间；

内存分配结束，虚拟机将分配到的内存空间都**初始化为零值**（不包括对象头）。保证**对象的实例字段**在 Java 代码中**可以不用赋初始值就可以直接使用**，程序**能访问到**这些字段的数据类型**所对应的零值**。

5）设置对象的对象头；

将对象的**所属类**（即类的元数据信息）、**对象的 HashCode** 和**对象的 GC 信息**、**锁信息**等数据**存储**在对象的**对象头中**。这个过程的具体设置方式取决于 JVM 实现。

6）执行 init 方法进行初始化；

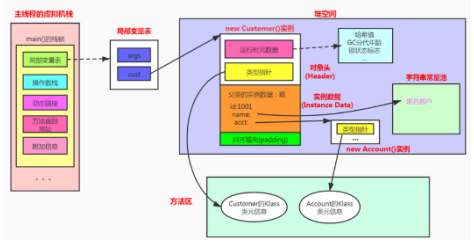
初始化成员变量，执行实例化代码块，调用类的构造方法，并把**堆内对象的首地址**赋值给**引用变量**。

3、指针碰撞和空闲列表

1）如果***内存规整***，使用**指针碰撞**为对象分配内存。**所有用过的内存**在一边，**空闲的内存**在另外一边，**中间放**着一个**指针作为分界点**的指示器，**分配内存**就仅仅是把**指针向空闲那边挪动一段**与对象大小相等的距离。

2）如果***内存不规整***，已使用的内存和未使用的内存是相互交错的，使用**空闲列表分配**。空闲列表**记录哪些内存块是可用的**，再分配的时候从列表中**找到一块足够大的空间**划分给对象实例，并**更新列表上的内容**。

4、对象的内存布局有：对象头、实例数据、对齐填补。



5、对象头（运行时元数据和类型指针）

1）对象自身的运行时元数据：

>哈希值：对象在堆空间中都有一个**首地址值，栈空间的引用**根据这个地址**指向堆中的对象**；

>GC 分代年龄计数器；

>锁状态标志，在同步中**判断该对象是否是锁**；

>**线程持有的锁**；

>线程偏向 ID；

>偏向时间戳。

2）类型指针：指向元数据区的类元数据 InstanceKlass，**确定该对象所属的类型**。

【注意】如果对象是一个**数组，对象头中**还必须**有一块**用于**记录数组的长度**的数据。

6、实例数据和对齐填补

1）实例数据：是对象**真正存储的有效信息**，包括程序代码中定义的**各种类型的字段**（包括从父类**继承下来**的和**本身拥有**的字段）。

2）对齐填充：不是必须的，也没特别含义，仅仅起到**占位符的作用**。

7、对象的访问定位（通过栈上的引用访问）

通过栈帧中的对象引用访问到其内部的对象实例有两中方式：

1）使用句柄访问

堆中划分出一块内存来做**句柄池，栈帧引用**中**存储**对象的**句柄池地址**，句柄中包含**对象实例**与**类型数据**各自具体**的地址信息**。

好处：对象被移动时只会**改变句柄中实例数据指针，栈帧引用本身不需要被修改**，但是占用额外内存。

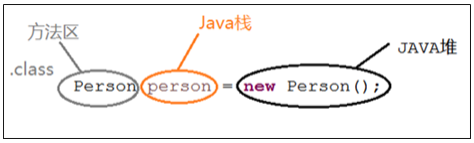
2）使用直接指针访问

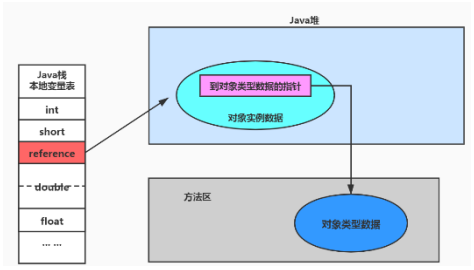
栈帧引用存储的就是**对象的地址**，如果只是**访问对象本身**的话，就**不需要多一次间接访问**的开销。

好处：**速度更快**，java 中对象访问频繁，每次访问都**节省了一次指针定位的时间**开销。

8、栈、堆、方法区的关系

变量的引用放在**栈**里，**类本身信息**放在**方法区**里，new 出来的**对象**放在**堆**空间中。**栈中引用**指向堆空间中的**对象**，堆里**对象中的类型指针**指向方法区中**所属的类**。





【注意】方法区逻辑上属于堆的一部分，但在实际实现中把它看作是独立于堆的内存空间。

9、方法区介绍

1）和堆一样是**线程共享**的内存区域；

2）在 JVM 启动时被创建，和堆一样实际物理**内存可以不连续**，空间**大小可固定可扩展**；

3）方法区大小决定**系统可以保存多少个类**，如果系统定义了太多的类，导致**方法区溢出**；

4）**关闭 JVM** 就会**释放**这个区域的内存。

10、方法区都存什么？

1）类型信息

2）域(Field)信息

3）方法(Method)信息

4）静态变量

5）运行时常量池

字节码文件中的常量池表中存放了**编译期生成**的**各种字面量**与**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。

【注意】常量池中会有：数量值、字符串值、类引用、字段引用、方法引用。

11、方法区和永久代/元空间是什么关系呢？

永久代（jdk1.8 之前）和元空间（jdk1.8 及之后）是对方法区的两种实现方式。

12、为什么要将永久代（PermGen）替换为元空间（MetaSpace）呢？

为了避免 OOM 异常。因为通常使用 **PermSize** 和 **MaxPermSize** 设置**永久代的大小**就决定了**永久代的上限**，但是**不是总能知道**应该**设置为多大合适**，如果使用默认值很容易遇到 OOM 错误。当**使用元空间**时，可以**加载多少类的元数据**据就不再由 MaxPermSize 控制，而**由系统的实际可用空间**来**控制**。

13、JDK 1.7 为什么要将字符串常量池移动到堆中？

因为**永久代**（方法区实现）的 **GC 回收效率太低**，只有在**整堆收集（Full GC）**的时候**才会被执行 GC**。Java 程序中通常会有**大量的**被创建的**字符串等待回收**，将**字符串常量池**放到**堆中**，能够更**高效及时地回收**字符串内存。