**面向对象的特征有哪些？**

1. 抽象
   1. 将一类对象的共同特征总结出类，构造类的过程，只关注对象的属性和行为，并不关注这些行为的细节是什么。
2. 继承
   1. 从已有类（父类，基类）得到继承信息创建新类（子类，派生类）。好处可扩展性
3. 封装
   1. 通常把数据和操作数据的方法绑定，对数据的访问只能通过已定义的接口
   2. 封装就是隐藏一切可以隐藏的东西，只向外界提供最简单的接口，
4. 多态性
   1. 允许不同子类型的对象对同一消息做出不同的响应。同样的对象引用调出同样的方法但是做了不同的事情
      1. 编译时：方法重载overloading前绑定，发生在一个类中，方法同名，参数列表不同，对返回值没有特殊要求
      2. 运行时：重写发生在子类和父类之间
         1. 方法重写overriding后绑定（子类继承父类并重写父类中已有的或抽象的方法）
         2. 对象造型（父类型引用，引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的**行为）**

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **修饰符** | **当前类** | **同包** | **子类** | **其他包** |
| **Public** | **√** | **√** | **√** | **√** |
| **Protected** | **√** | **√** | **√** | **×** |
| **Default** | **√** | **√** | **×** | **×** |
| **private** | **√** | **×** | **×** | **×** |

1、

**8种基本数据类型**

byte。short、int long float double char boolean 基本数据类型

Byte Short Int Long Float Double Char Boolean 包装类型(wrapper class)

从java1.5开始引入自动装箱、拆箱，使二者可以转换

**public** **class** InterViewDemo {

**public** **static** **void** main(String[] args) {

Integer a=**new** Integer(3);

Integer b=3;

**int** c=3;

System.***out***.println("a==b:"+(a==b));

System.***out***.println("a==c:"+(a==c));

}

}

结果：

a==b:false 两个引用没有引用同一对象

a==c:true a自动装箱成int类型再和c比较

**注意：如果整型的数值在127和-128之间不会new新的integer对象，如果超出则新建**

Integer a=100,b=100,c=500,d=500;

System.***out***.println("a==b:"+(a==b));

System.***out***.println("d==c:"+(d==c));

结果：a==b:true

d==c:false

**引用类型**

除了基本类型(primitive)和枚举类型(enumeration type)，剩下的都是引用类型(reference type)

**&和&&的区别**

1、&按位与

如果左边的是false就会继续运行右边的

2、逻辑与

如果左边的是false，就不会执行右边的

逻辑或(|)和短路或的差别也是如此

**解释内存中的栈(stack)堆（heap）和静态区（static area）的用法**

1. 通常我们定一个基本数据类型的变量，一个对象的引用，函数调用的现场保存都使用内存中的栈空间
2. 通过new关键字和构造器创建的对象放在堆空间中
3. 程序中的字面量（literal），
   1. 如直接书写的100，”字符串”和常量都是放在静态区中
   2. 理论上整个内存没有被其他进程使用甚至是硬盘上的虚拟内存都可以被当成堆空间使用
   3. String str=new Stirng(“hello”);
   4. 上面的语句变量str放在栈上，new创建出来的字符串对象放在堆上，hello这字面量放在静态区
      1. 补充：从java6某个版本开始有一项“逃逸分析”的技术，可以将一些局部对象放在栈上以提升对象的操作性

**Switch（expr）表达式**

在 java5以前expr只能是byte short char int，从1.5以后引入了枚举类型，从1.7开始还可以是字符串，但是long是个版本都不行

构造器不能被继承，不能被重写，但是可以重载

**Equals方法和hashcode方法的规定**

1. 如果两个对象相同(equals方法返回true)，那么他的hashcode值一定相等
2. 如果两个对象的hashcode值相同，他们不一定相同

**Equals方法的介绍**

1. equals方法必须满足自反性(x.equals(x)必须返回true)
2. 对称性（x.equals(y)返回true时，y.equals(x)也必须返回true）
3. 传递性(x.equals(y)和y.equals(z)都返回true时，x.equals(z)也必须返回true)
4. 一致性（当x/y引用的对象信息没有被修改时，多次调用x.equals(y)应该得到同样的返回值），而且对于任何非null值得引用x，x.equals(null)必须返回false

**实现高质量的equals方法的诀窍**

1. 使用==操作符检查参数是否为这个对象的引用
2. 使用instanceof操作符检查从参数是否为正确的类型
3. 对于类中的关键属性是否与之相匹配
4. 编写完equals方法后，为自己它是否满足对称性、传递性、一致性
5. 重写equals时总是要重写hashCode
6. 不要将equals方法参数中的Object对象替换为其他的类型，在重写时不要忘记@Override注解

**值传递和值引用**

1. 值传递：
   1. 当一个对象被当做参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果
   2. Java语言的方法调用只支持参数的值传递
   3. 当一个对象实例作为一个参数被传递到方法中，参数的值就是对该对象的引用，对象的属性可以在被调用过程中改变，但对对象引用的改变是不会影响到调用者的

**String StringBuilder StringBuffer**

Java提供了两种类型的字符串String和StringBuilder/StringBuffer，他们都可以存储和操作字符串

1. String是只读字符串，String引用的字符串是不能被修改的
2. StringBuilder和StringBuffer表示的字符串是可以修改的，
3. StringBuilder是java5中引入的，他和StringBuffer的方法完全相同，区别在于StringBuilder是在单线程环境下使用的，因为他的方法都没有被synchronized修饰，因此效率也是比StringBuffer高

**JVM加载class文件的机制**

1. Jvm中类的加载是由类加载器ClassLoader和他的子类实现的，java中的类加载器是一个重要的java运行时系统组件，负责在运行时查找和装入类文件中的类
2. 由于java的跨平台性，经过编译的java源程序并不是一个可执行文件，而是一个或多个类文件，当java程序需要使用某个类时，jvm会确保这个类已经被加载、链接（验证、准备、解析）和初始化，
3. 类的加载是指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，，然后产生与加载类对应的class对象，加载完之后类还不是完整，此时的类还不能用，当类被加载后进入连接阶段。这一阶段包括验证、准备（为静态变量分配内存并初始化）和解析（将符号引用替换为直接引用）三个步骤。最后jvm对类进行初始化，包括：
   1. 如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类
   2. 如果类中存在初始化语句，就依次执行这些初始化语句
4. 类的加载是由类的加载器完成的，类加载器包括根加载器(BootStrap)、扩展加载器（Extension）、系统加载器(System)和用户自定义类加载器(java.long.ClassLoder的子类)。从jdk1.2开始，类加载过程采用了父类委托机制(PDM)。Pdm更好的保证了java平台的安全性，在该机制中，jvm自带的bootStrap是根加载器，其他的加载器都有且仅有一个父类加载器，类的加载首先请求父类加载器加载，父类加载器无能无力时才由其子类加载器自行加载。Jvm不会向Java程序提供对BootStrap的引用。
   1. BootStrap： 一般用本地代码实现，负责加载JVM基础核心类库(rt.jar)
   2. Extension：从java.ext.dirs系统属性所指定的目录中加载类库，他的附加载器是BootStrap
   3. System：又叫应用类加载器，其父类是Extension。他是应用最广泛的类加载器，他从环境变量classpath或者系统属性java.class.path所指定的目录中记载类，是用户自定义加载器的默认父加载器

**Char 可以存放Unicode字符**

Java中使用的编码是unicode（不选择任何特定的编码，直接使用字符在字符集中的编号，这是统一的唯一的方法），一个char 占两个字节（16bit）

1. 使用Unicode意味着字符在jvm内部和外部有不同的表现形式，在jvm内部都是Unicode，当这个字符从jvm内部转移到外部时（例如存到文件系统中），需要进行编码转换。
2. Java中有字节流和字符流，以及在字符流和字节流之间进行转换的转换流（InputStreamReader/OutputStreamReader），这两个类是字节流和字符流之间的适配器类，承担了编码转换的任务，

**抽象类abstract和接口interface**

1. 抽象类和接口都不能被实例化，但可以定义抽象类和接口类型的引用。
2. 一个类如果继承了某个抽象类或者实现了某个接口都需要对其中的抽象方法全部进行实现，
3. 接口比抽象类更加抽象，因为抽象类中可以定义构造器，可以有抽象方法和具体方法，接口中不能定义构造方法和具体方法
4. 抽象类中的成员可以是private 默认 protected public ，接口中的成员都是public
5. 抽象中可以定义成员变量，但是接口中定义的成员变量都是常量。
6. 有抽象方法的类必须被声明为抽象类，但是抽象类不一定要有抽象方法

**静态嵌套类Static Nested Class和内部类Inner Class**

1、Static Nested Class被声明为静态的内部类，可以不依赖外部类实例被实例化，通常的内部类需要在外部类实例化后才能实例化

下面看一下例子：

class Outer{

class Inner{}

public static void main(String[] args){

new Inner();

}

}

这个是错误的。正确的是new Outer().new Inner();

解释：java中非静态内部类对象的创建要依赖其外部类对象，main()是static修饰，静态方法中没有this，也就是没有所谓的外部类对象，因此无法创建内部类对象，如果要在静态方法中穿件内部类对象，必须先实例化外部类对象，再实例化内部类对象（new Outer().new Inner()）

**Java存在内存泄漏**

1. 理论上java中有垃圾回收机制GC，不会出现内存泄漏（这也是java被广泛应用于服务器端编程的一个重要原因），然而在实际开发中可能存在无用但可达的对象，这些对象不能被GC回收，也会导致内存泄露
   1. 例如Hibernate的Session（一级缓存）中的对象属于持久态，GC是不会回收这些对象的，如果不及时关闭或者清空(flush)一级缓存就可能导致内存泄露

**GC垃圾回收机制**

1. 内存处理是编程人员容易出问题的地方，忘记或者错误的内存回收，会导致系统的不稳定或者崩溃，
2. Java提供的GC功能，可以自动检测对象是否超过作用域从而达到自动回收内存的目的，java语言没有提供释放已分配内存的显示操作方法
3. 要请求垃圾收集，调用System.gc();或者是Runtime.getRuntime().gc()，但JVM可以屏蔽掉显示的垃圾回收调用
4. 与垃圾回收相关的jvm参数
   1. –Xms / -Xmx 堆的初始大小/最大大小
   2. –Xmn 堆中年轻代的大小
   3. –XX:DisableExplicitGC 让System.gc()不产生任何作用
   4. –XX:+PrintGCDetails 打印GC的细节
   5. –XX:+PrintGCDateStamps 打印GC操作额时间戳
   6. –XX:NewSize / XX:MaxNewSize 设置新生代大小/最大大小
   7. –XX:NewRatio 可以设置老生代和新生代的比例
   8. –XX:PrintTenuringDistribution设置每次新生代GC后输出幸存者乐园中对象年龄的分布
   9. –XX:InitialTenuringThreshold / -XX:MaxTenuringThreshold:设置老生代阈值的初始值和最大值
   10. –XX:TargetSurvivorRatio 设置幸存区的目标使用率

**Java中的final关键字**

1. 修饰类：表示不能被继承
2. 修饰方法：方法不能被重写
3. 修饰变量：变量只能一次赋值以后值不能被修改

**静态变量和实例变量**

1. 静态变量是被static修饰符修饰的变量，也称之为类变量，它属于类，不属于类的任何一个对象，一个类中不管创建多少个对象，静态变量在内存中有且仅有一个copy，可以让多个对象共享内存
2. 实例变量必须依存某一个实例，需要先创建对象才能访问它

**克隆对象**

1. 实现Cloneable接口并重写Object类中的clone()
2. 实现Serializable接口，通过对象的序列化和反序列化，可以实现真正的深度克隆

注意：基于序列化和反序列化实现的克隆不仅是深度克隆，更重要的是通过泛型限定，可以检查出要克隆的对象是否支持序列化，这项检查是编译器完成的，不是在运行时抛出异常，这种方案优于Object类的clone()。让问题在编译的时候暴露出来总是优于把问题留在运行时

**什么时候用断言assert**

1. 断言是开发中常用的调试方式，很多开发语言都支持，断言用于保证程序最基本、关键的正确性，断言检查通常在开发和测试时开启，为了保证程序的执行率，断言是包含布尔表达式的语句，在执行这个语句的时候假定该表达式为true，如果表达式的值为false，那么系统会报告一个AssertionError。
2. 断言有两种形式：
   1. Assert Expression1;总是产生一个布尔值
   2. Assert Expression1:Expression2;得出一个值得任意表达式，这个值用于生成显示更多调试信息的字符串消息
   3. assert(a>0);
   4. 断言不应该以任何方式改变程序的状态。如果希望在不满足某些条件是阻止代码的执行，就可以考虑用断言来阻止

**throws throw try catch finally的使用**

1. java通过面向对象的方法进行异常处理，把各种不同的异常进行分类并提供了良好的接口。
2. java中每个异常都是一个对象，他是throwable类或者其子类的实例。当一个方法出现异常便抛出一个异常对象，该对象包含异常信息，，调用这个对象的方法可以捕获到这个异常并处理，
3. 一般情况下用try执行一段程序，如果系统会抛出throw一个异常对象，可以通过他的类型来捕获catch，或通过finally语句块处理。
4. Try用来指定一块预防所有异常的程序，catch子句紧跟在try块后面，用来指定捕获异常的类型，throw用来明确的抛出一个异常，throws用来声明一个方法可能抛出的异常，finally语句块保证里面的代码总要被执行

**多线程的问题**

1. new Thread的弊端
   1. 每次新建对象，性能差
   2. 线程缺乏统一管理，可能无限制新建线程，