**执行顺序：**

父类静态代码块🡪子类静态代码块🡪父类构造代码块、构造方法🡪子类构造代码块、构造方法

**final修饰符**

1. 类，不能被继承
2. 方法，不能被子类重写，不能修饰构造方法
3. 局部变量，使用前赋值，不能修改
4. 成员变量，只能在定义时/构造代码块/构造方法中赋值
5. 引用类型，引用不可修改，内部成员可以修改
6. 配合static使用，表示全局变量

**为什么非静态内部类中不能包含静态成员？**

非静态内部类不随外部类一起加载，只有实例化外部类后才会加载。这时，如果访问内部类的静态成员，内部类还未加载，矛盾。

**局部内部类访问外层局部变量的机制**

Java将局部变量作为参数传给了局部内部类的构造函数，将其作为内部类的成员属性封装在类中（实际上就是一个副本），保证了类的封装性。为了数据的一致性，局部变量必须设置为final。

Integer num=100;

**当-128≤num≤127时，在缓冲区（对象池）中直接创建。**

当String str1=”123”;时，会在常量池创建一个”123”，然后，

String str2=”123”;时，str2指向常量池中的”123”，

使用new String(“123”)时，会在堆中开辟新的内存空间。

Set添加元素时，判断是否重复的方法

1. **hashCode()**  hashCode不同，则肯定不一样
2. **equals(obj)** hashCode相同，再用equals比较

集合排序的两种方案

1. 定义一个类，**实现Comparator<T>接口**，需实现方法
2. 集合中的对象元素所属的类**实现Comparable接口**，需实现方法

该对象小于、等于、大于指定对象时，分别返回负整数、0、正整数

线程状态和生命周期

可运行(Runnable)

新建(new)

join()的线程执行完毕

notify()、notifyAll()

wait()、sleep()超时

I/O请求完成

join() wait() sleep() I/O请求

阻塞(Blocked)

线程执行完毕或异常终止

时间片用完

获取CPU使用权

正在运行(Running)

终止(Dead)

并行和并发的区别，关键一点在于并行强调【同时】

**线程死锁的四个条件**

1）互斥条件：一个资源每次只能被一个进程使用，即在一段时间内某 资源仅为一个进程所占有。此时若有其他进程请求该资源，则请求进程只能等待；

2）请求与保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源 已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放；

3）不可剥夺条件:进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能 由获得该资源的进程自己来释放（只能是主动释放)；

4）循环等待条件: 若干进程间形成首尾相接循环等待资源的关系。

**浅拷贝和深拷贝**

浅拷贝，需要实现Clonable接口，重写clone()方法。引用类型的成员变量并没有被复制；

深拷贝，两种实现方式：

1. 成员变量中引用类型对应的类也要实现Clonable接口，并在重写clone方法时，引用类型变量使用对应的clone()方法克隆；
2. 实现Serializable接口，通过对象的序列化和反序列化实现真正的克隆。

public static <T extends Serializable> T deepCloneObject(T object) {  
 T deepClone = null;  
 ByteArrayOutputStream bos = **null**;  
 ObjectOutputStream oos = **null**;  
 ByteArrayInputStream bis = **null**;  
 ObjectInputStream ois = **null**;  
 **try** {  
 bos = **new** ByteArrayOutputStream();  
 oos = **new** ObjectOutputStream(bos);  
 oos.writeObject(object);  
 bis = **new** ByteArrayInputStream(bos.toByteArray());  
 ois = **new** ObjectInputStream(bis);  
 deepClone = (T)ois.readObject();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 **try** {  
 **if** (bos != **null**) {  
 bos.close();  
 }  
 } **catch** (IOException ex) {  
 ex.printStackTrace();  
 }  
 **try** {  
 **if** (oos != **null**) {  
 oos.close();  
 }  
 } **catch** (IOException ex) {  
 ex.printStackTrace();  
 }  
 **try** {  
 **if** (bis != **null**) {  
 bis.close();  
 }  
 } **catch** (IOException ex) {  
 ex.printStackTrace();  
 }  
 **try** {  
 **if** (ois != **null**) {  
 ois.close();  
 }  
 } **catch** (IOException ex) {  
 ex.printStackTrace();  
 }  
 }  
 **return** deepClone;  
}

**序列化Serializable**

作用：保存内存中各种对象的状态，并且可以把保存的对象状态再读出来；为了不同jvm之间共享实例对象的一种解决方案

场景：将对象持久化到文件或数据库中；使用套接字再网路上传输对象；通过RMI传输对象。

**泛型**

即参数化类型，将类型由原来的具体的类型参数化，类似于方法中的变量参数。

**特性：**泛型只在程序源码中存在，在编译后会采取去泛型化的措施，泛型信息不会进入到运行时阶段。泛型类型在逻辑上可以看成是多个不同的类型，实际上都是相同的基本类型。

**泛型的使用**

**1 泛型类**

public class Generic<T> {…}

泛型的类型参数只能是类类型，不能是简单类型；

不能对确切的泛型类型使用instanceof操作，否则会报错

**2 泛型接口**

public interface Generator<T> {…}

**3 泛型通配符**

Generic<Integer>不是Generic<Number>的子类，如何解决此类问题，就要用到通配符?

Generic<? extends Number>

**4 泛型方法**

泛型类，是在实例化的时候指明泛型的具体类型；泛型方法，是在调用方法的时候指明泛型的具体类型。

public **<T>** T genericClass(Class<T> tClass) {…}

只有声明了<T>的方法才是泛型方法。

在泛型类中声明泛型方法，即使都使用<T>，泛型方法中的T是一个全新的类型，可以和泛型类中的T不是同一种类型。

泛型作为可变参数时，可以传入任何类型：public **<T>** void printMsg(**T...** args) {…}

如果静态方法要使用泛型的话，必须将静态方法定义为泛型方法。直接引用类上定义的泛型会报错。

指导原则：如果使用泛型方法能够将整个类泛型化，就应该使用泛型方法。

**泛型的上下边界**

泛型的上下边界添加，必须与泛型的声明在一起。

public class Generic**<T extends Number>** {…}

public **<T extends Number>** T showKeyName(Generic<T> container) {…}

**泛型数组的限制**

数组的类型不可以是类型变量，除非采用通配符的方式。

List**<?>**[] ls = new ArrayList**<?>**[10];

**注解Annotation**

**注解如同标签**，对代码中某些鲜活的个体贴上标签。它**不会改变代码本身**，只是某些工具的工具。处理提取和Annotation的代码统称为**APT**（Annotation Processing Tool）。

**给谁用的？给编译器或者APT用的**。

**注解的定义**

public **@interface** TestAnnotation {

}

**注解的应用**

@TestAnnotation

public class Test {…}

**元注解**

注解到注解上的注解；基本注解，可以应用到其他注解上；也是一张标签，用于给其他标签添加解释说明。

**5种元注解：**

**@Retention 指明生命周期**

取值如下：

RetentionPolicy.SOURCE 只在源码阶段保留，在编译时丢弃；

RetentionPolicy.CLASS 只保留到编译进行的时候，不会被加载到JVM；

RetentionPolicy.RUNTIME 保留到程序运行时，会被加载到JVM

**@Documented 将注解中的元素包含到Javadoc中去**

**@Target 注解运用的场景**

取值如下：

ElementType.ANNOTATION\_TYPE 可以给一个注解进行注解

ElementType.CONSTRUCTOR 构造方法

ElementType.FIELD 属性

ElementType.LOCAL\_VARIABLE 局部变量

ElementType.METHOD 方法

ElementType.PACKAGE 包

ElementType.PARAMETER 方法内的参数

ElementType.TYPE 类型（类、接口、枚举）

**@Inherited 子类可以继承父类的注解**

**@Repeatable 1.8加入，注解的值可以同时取多个，@Repeateable后面括号的类相当于相当于一个容器注解**

**注解的属性**

以“无形参的方法”形式来声明

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

public @interface TestAnnotation {

int id();

String msg();

}

属性可以有默认值，用default关键字指定

int id() default -1;

**Java预置的注解**

@Deprecated

@Override

@SuppressWarnings

@SafeVarargs 1.7加入，阻止编译器产生unchecked这样的警告

@FunctionalInterface 1.8加入，函数式接口，具有一个方法的普通接口，如Runnable

**通过反射提取注解**

判断是否应用了某个注解

public Boolean isAnnotationPresent(Class<? extends Annotation> annotationClass) {}

获取Annotation对象

public <A extends Annotation> A getAnnotation(Class<A> annotationClass) {}

public Annotation[] getAnnotations() {}

**注解的使用场景**

-提供信息给编译器：编译器可以利用注解来探测错误和警告信息

-编译阶段的处理：软件工具可以利用注解信息来生成代码、Html文档或其他处理

-运行时的处理：在运行时接受代码的提取

**代理**

用户调用Proxy，Proxy是中介者，它增强了真正的生产者。

**静态代理**

代理模式**在不修改被代理对象的基础上，通过扩展代理类，进行一些功能的附加与增强**。代理类和被代理类应该共同实现一个接口，或者共同继承某个类。

**动态代理**

通过Proxy的静态方法newProxyInstance创建代理，参数分别是1)ClassLoader类加载器、2)interfaces代理的接口、3)InvocationHandler对象。

内部通过ClassLoader和接口数组用工厂方法生成proxy class，名称是**包名+$Proxy+id序列号**

在实现InvocationHandler的类中的invoke方法中完成对方法的增强。

**IO**

字节序列是某种编码时，把字节序列变成字符串，也需要用这种编码方式，否则会出现乱码。

常用的编码方式

GBK 中文2字节，英文1个字节

UTF-8 中文3个字节，英文1个字节

UTF-16be 中文2个字节，英文1个字节

**File**类只用于表示文件（目录）的信息，不能用于文件内容的访问。

**RandomAccessFile**

对文件内容的访问类，可以读写文件，可以访问文件的任意位置。

两种模式 ”rw”读写 “r”只读

调用write方法写入字符时，只能写入一个字节，如果要写入的字符大于一个字节，只写后8位。可以传入一个byte数组，或者调用writeInt()等方法。

调用read方法时，先把指针移到头部raf.seek(0);可以把文件中的内容读取到字节数组中。

**字节流**

1. InputStream、OutputStream抽象类
2. EOF=End 读到-1就读到结尾
3. 输入流基本方法

int b = in.read();按字节读取，-1是EOF

in.read(byte[] buf)

in.read(byte[] buf, int start, int size)

1. 输出流基本方法

out.write(int b);将b的低8位写出到流

out.write(byte[] buf);

out.write(byte[] buf, int start, int end);

1. FileInputStream具体实现了在文件上读取数据
2. FileOutputStream具体实现了向文件写出byte数据的方法

实现文件拷贝的核心代码

while((b=in.read(buf,0,buf.length)) != -1) {

out.write(buf,0,b);

out.flush();

}

1. DataOutputStream/DataInputStream 对流功能的扩展，可以更加方便的读取int，long，字符等类型数据

writeInt() writeDouble() writeUTF() readInt()等

1. BufferedInputStream/BufferedOutputStream 提供了带缓冲区的操作，建议打开缓冲，这种流模式提高了IO性能。

**字符流**

1. 文本文件，是文本(char)序列按照某种编码方案序列化为byte的存储方案。
2. Reader/Writer

字符的处理，一次处理一个字符，底层仍然是基本的字节序列

InputStreamReader 完成byte流解析为char流，按照编码解析

OutputStreamWriter 提供char流到byte流，按照编码处理

1. FileReader/FileWriter 构造函数参数为String（文件路径）或者File对象
2. 字符流过滤器 BufferedReader/BufferedWriter/PrintWriter 可以一次读一行

readLine()

while((line=br.readLine())!=null) {…}

**对象的序列化，反序列化**

**序列化流**ObjectOutputStream/ObjectInputStream writeObject/readObject

**序列化接口**（Serializable）对象必须实现序列化接口，才能进行序列化，否则将出现异常

用transient修饰的成员变量默认不会被序列化。如果想自己实现序列化和反序列化操作，需要实现writeObject和readObject方法。

**序列化中子类和父类构造函数的调用问题**

如果父类实现了序列化接口，子类就没必要实现序列化接口。

对子类对象进行反序列化操作时，如果父类没有实现序列化接口，其父类的构造函数会被调用。