————————————————语言基础———————————————————

**九种基本数据类型的大小，以及他们的封装类，占用大小。**

Int Integer 4byte 0

char Character **2bye** [] null utf16，一个unicode两个character,超过16bit的字符，char无法正常存储，string中用两个char存储

long Long 8byte 0L

double Double 8byte 0.0d

float Float 4byte 0.0f

bool Boolean (jvm虽然定义了boolean这种数据类型，但是只对它提供了非常有限的支持。在Java虚拟机中没有任何供boolean值专用的字节码指令 4byte或1byte) false

byte Byte 1byte (byte)0

**short Short 2byte (short)0**

void Void

类的成员是基本类型（封装类是类不行）会有一个初始值

局部变量使用前必须初始化

**Switch能否用string做参数？**

在Java5以前，switch(expr)中，exper只能是**byte，short，char，int**类型。从Java5开始，java中引入了枚举类型，即**enum**类型。从Java7开始，exper还可以是**String**类型。

**equals与==的区别**

==比的是引用

Equals调用的是equals方法，如果equals没被声明则会调用父类的equals方法，object的equals比较的是引用

**Object有哪些公用方法？**

protected Object clone() 克隆一个对象

boolean equals(Object obj) 判等

protected void finalize() 垃圾回收器回收对象空间时调用

Class<?> getClass() 得到类的引用

int hashCode() 类的hash值

void notify()

void notifyAll()

String toString()

void wait()

void wait(long timeout)

void wait(long timeout, int nanos)

**Java的四种引用，强软弱虚，用到的场景。**

强引用 以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。

Counter counter = new Counter(); // strong reference

WeakReference<Counter> weakCounter = new **WeakReference**<Counter>(counter); //weak reference//weakCounter弱引用

SoftReference soft= new SoftReference(counter) ; //soft reference

ReferenceQueue refQueue = new ReferenceQueue(); //引用会在队列中等待被清楚

PhantomReference<Counter> phantom = new PhantomReference<Counter>( counter, refQueue);//虚引用

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；

在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：**虚引用必须和引用队列** （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的**内存被回收之前采取必要的行**动。

**Hashcode的作用。**

hashCode是用于查找使用的（根据hashcode得到存储位置），而equals是用于比较两个对象的是否相等的

1、hashCode的存在主要是用于查找的快捷性，如Hashtable，HashMap等，hashCode是用来在散列存储结构中确定对象的存储地址的；

2、容器中如果两个对象相同，就是适用于equals([Java](http://lib.csdn.net/base/java).lang.Object) 方法，那么这两个对象的hashCode一定要相同，非容器中hashcode没关系；

3、如果对象的equals方法被重写，那么对象的hashCode也尽量重写，并且产生hashCode使用的对象，一定要和equals方法中使用的一致，否则就会违反上面提到的第2点；

4、两个对象的hashCode相同，并不一定表示两个对象就相同，也就是不一定适用于equals(java.lang.Object) 方法。

**String、StringBuffer与StringBuilder的区别。（都是final）**

String对象是不可变的 String类的声明是：public **final** string的”+=”实际调用多个append创建多个（中间对象）空间来保存新的字符串————操作效率低（底层循环调用stringbuilder，stringbuilder 核心成员一个char数组一个count,append是通过getchars实现的）。字符串是不可变的，所以在它创建的时候hashcode就被缓存了，不需要重新计算。

效率：StringBuilder > StringBuffer > String

StringBuilder：线程非安全的 StringBuffer：线程安全的

StringBuffer与StringBuilder，他们是字符串变量，是可改变的对象

1.如果要操作少量的数据用 = String

2.单线程操作字符串缓冲区 下操作大量数据 = StringBuilder

3.多线程操作字符串缓冲区 下操作大量数据 = StringBuffer

**Final**

对于基本类型，final使数字恒定不变。而对于对象引用，final使引用恒定不变(对象自身可变)。

构造器

顺序

1. 基类构造器
2. 按声明顺序调用成员初始化方法
3. 调用导出类构造器

问题：若没有构造器会有默认构造器，若有非默认则jvm不会帮着建。

构造器不可以被覆盖只可以被重载。

**try catch finally，try里有return，finally还执行么？**

1、不管有木有出现异常，finally块中代码都会执行；

2、当try和catch中有return时，finally仍然会执行；

3、finally是在return后面的表达式运算后执行的 ，return的值会被暂存

4、finally中最好不要包含return，否则程序会提前退出，返回值不是try或catch中保存的返回值。

（先把return的值放到缓存，然后执行finally，然后再取出）

**Excption与Error包结构。**



1. Throwable Throwable是 Java 语言中所有错误或异常的超类。

2. Exception Exception及其子类是 Throwable 的一种形式，它指出了合理的应用程序想要捕获的条件。

3. RuntimeException RuntimeException是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类。 编译器不会检查RuntimeException异常。可以不捕获向上抛，如果一直没有处理，则jvm会自动处理（停止线程，打印异常），非运行期异常必须捕获**或**者在方法声明。

4. Error 和Exception行为模式一样，但编程中不应该自己去抛出error的子类应该把它视为一种编译错误， Error也是Throwable的子类。 它用于指示合理的应用程序不应该试图捕获的严重问题，大多数这样的错误都是异常条件。 和RuntimeException一样， **编译器也不会检查Error**。

编译器强制普通异常**必须try..catch处理或用throws声明继续抛给上层调用方法处理**，所以普通异常也称为**checked异常**，而系统异常可以处理也可以不处理，所以，编译器不强制用try..catch处理或用throws声明，所以**系统异常也称为unchecked异常**。

常见runtimeexception

NullPointerException - 空指针引用异常

ClassCastException - 类型强制转换异常。

IllegalArgumentException - 传递非法参数异常。

ArithmeticException - 算术运算异常

ArrayStoreException - 向数组中存放与声明类型不兼容对象异常 IndexOutOfBoundsException - 下标越界异常

NegativeArraySizeException - 创建一个大小为负数的数组错误异常

**Java面向对象的三个特征与含义。**

继承

封装 隐藏对象的属性和实现细节，仅对外公开接口（规范化）

非类的开发者不应该直接修改使用类的属性，而应该通过开发者暴露的public方法调用

多态

**参数列表**

参数的类型，参数的个数，参数的顺序

**Override和Overload的含义的区别。**

Override是覆盖，代表子类重写父类的方法

覆盖

1、“**三同一不低**” 子类和父类的**方法名称**，**参数列表**，**返回类型**必须**完全相同**，子类方法的访问修饰符的权限不能比父类**低**。

2、子类方法不能抛出比父类方法**更多的异常**。

3、被覆盖的方法不能是**final**类型的。（不会覆盖只会新建一个同名的）

4、被覆盖的方法不能为**private**。（不会覆盖只会新建一个同名的

5、被覆盖的方法不能为**static**。因为静态方法是在编译的时候把静态方法和类的引用类型进行匹配。子静态父不静或反之抛错。都静态保留父类。

Overload是重载或过载

重载

重载（overload）：对于类的方法（包括从**父类中继承的方法**），**方法名相同，参数列表不同的方法之间就构成了重载关系**。重载可发生在同一个类中，也可发生在子类与父类中

**Interface与abstract类的区别**

Interface所有的方法都是抽象方法，**域都是public static final的**(维持所有类的行为一致)，使用方法都是public

含一个抽象方法就是Abstract类，

只能继承一个抽象类 可以实现多个接口

Abstract ————is a(复用) interface————like a（规范）

**什么是擦除**

擦除其实是一个语法糖，泛型类其实在编译后就已经是裸类型不带任何类型了。

实现是在编译过程中在相应的地方加入了强制类型转换。

**foreach与正常for循环效率对比**

直接for循环效率最高，其次是迭代器和 ForEach操作。 其实 ForEach 编译成 字节码之后，使用的是迭代器实现的

**访问权限**



**多态**

在面向对象语言中，接口的多种不同的实现方式即为多态。

**允许将子类类型的**[**指针**](http://baike.baidu.com/item/%E6%8C%87%E9%92%88)**赋值给父类类型的指针**

**把不同的子类对象都当作父类来看**，可以屏蔽不同子类对象之间的差异（**后期绑定**）

**原理：覆盖->invokevirtual指令->动态分派**

**强制类型转换**

关于强转

java程序运行后,**对象的类型就不会发生改变**。而强制类型转换只不过是用某种类型去引用内存中的实体,这种类型只是编译时类型,与运行后的实体并没有联系

**深克隆浅克隆**

【浅克隆】，通常只是对克隆的实例进行复制，但里面的其他子对象，都是共用的，clone方法就是。

【深克隆】，克隆的时候会复制它的子对象的引用，里面所有的变量和子对象都是又额外拷贝了一份，对象和子对象都调用clone。

**Java8重大改进**

Lambada：

Lambda表达式实际上是一个匿名函数。

// 1. 不需要参数,返回值为 5

() -> 5

// 2. 接收一个参数(数字类型),返回其2倍的值

x -> 2 \* x

// 3. 接受2个参数(数字),并返回他们的差值

(x, y) -> x – y

// 4. 接收2个int型整数,返回他们的和

(int x, int y) -> x + y

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回void)

(String s) -> System.out.print(s)

Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println( e ) );

forEach是iterable类JAVA8新增的接口

其他：（用lambda表达式实现Runnable）

Java 8使用两个新概念扩展了**接口**的含义：

**默认方法和静态方法**。

默认方法：

接口中 default RerurnValue functionName() {具体实现; }，接口中直接实现，继承类不需要实现。（好处：当需求变更要增加接口方法时，子类不要随之改变）

接口中还能实现静态方法。

容器类 **optional**:可以存放null

**Optional.ofNullable(xx); //存储xx**

**Optional.of(xx);**

**xx.get()//得到存储的值 xx.orElse(value)//得到存储的值 若空返回value**

**方法引用**

**方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。与lambda联合使用**

**Native**

调用本地方法（jni），<http://www.cnblogs.com/Alandre/p/4456719.html>

**内部类（静态内部类）**

功能：完善多重继承，实现闭包

访问权限：成员内部类，就是作为外部类的成员，可以直接使用外部类的所有成员和方法，即使是private的。同时外部类要访问内部类的所有**成员**变量/方法，则需要通过**内部类的对象**来获取。(**需要通过生成外部类对象来生成Out.In in = new Out().new In();**)

静态内部类只能访问外部类的静态成员，生成（new）一个静态内部类不需要外部类成员：这是静态内部类和成员内部类的区别。静态内部类的对象可以直接生成： Outer.Inner in=new Outer.Inner()；而**不需要通过生成外部类对象来生成**。这样实际上使静态内部类成为了一个顶级类。可以定义私有静态内部类。

**多重继承**

接口：

Class X implements A,B{}

内部类：

public class MainExample

{

private class test1 extends Example1

{

public String name()

{

return super.name();

}

}

private class test2 extends Example2

{

public int age()

{

return super.age();

}

}

public String name()

{

return new test1().name();

}

public int age()

{

return new test2().age();

}

public static void main(String args[])

{

MainExample mi=new MainExample();

System.out.println("姓名:"+mi.name());

System.out.println("年龄:"+mi.age());

}

}

类三，里面分别实现了两个内部类 test1,和test2 ，test1类又继承了Example1，test2继承了Example2，这样我们的类三MainExample就拥有了Example1和Example2的方法和属性，也就间接地实现了多继承。

序列化(序列化 (Serialization)将对象的**状态信息**转换为可以存储或传输的形式的过程)

作用：

1. 远程通信传送对象
2. 持久化保存对象

注意：序列化的是对象的状态信息，不是类，类的方法等信息在class文件中，故要反序列化还依然要加载class文件。

只有实现了Serializable或Externalizable接口的类的对象才能被序列化，否则抛出异常。

java.io.ObjectOutputStream：writeObject(Object obj)对参数指定的obj对象进行序列化

java.io.ObjectInputStream：readObject()方法源输入流中读取字节序列，再把它们反序列化成为一个对象，并将其返回

ObjectOutputStream oout = new ObjectOutputStream(new FileOutputStream(file));

Person person = new Person("John", 101, Gender.MALE);

oout.writeObject(person);

oout.close();

ObjectInputStream oin = new ObjectInputStream(new FileInputStream(file));

Object newPerson = oin.readObject(); // 没有强制转换到Person类型

oin.close();

System.out.println(newPerson);

在序列化对象时，不仅会序列化当前对象本身，**还会对该对象引用的其它对象也进行序列化**，同样地，这些其它对象引用的另外对象也将被序列化，以此类推。

JDK中提供了另一个序列化接口—Externalizable

当某个字段被声明为**transient**后，默认序列化机制就会忽略该字段。

对象反序列化时，如果父类未实现序列化接口，则反序列出的对象会**再次调用父类的构造函数**（默认构造函数）来完成属于父类那部分内容的初始化。

深复制，反序列化还原后的对象地址与原来的的地址不同序列化前后对象的地址不同了，但是内容是一样的，而且对象中包含的引用也相同。

**静态成员属于类级别的，所以不能序列化**。序列化信息中不包含这个静态成员域

如果在本地修改了一个静态变量的值，在同一段代码里序列化再反序列化一个值，它是被修改的。因为本地的该类成员变量已经被修改。如果是远程被反序列化则静态变量会是初始值。

序列化算法一般会按步骤做如下事情：

1. 将对象实例相关的类元数据输出。
2. 递归地输出类的超类描述直到不再有超类。
3. 类元数据完了以后，开始从最顶层的超类开始输出对象实例的实际数据值。
4. 从上至下递归输出实例的数据

Transitent:不序列化

类在实现序列化的时候，如果**该类实现了writeObject和readObject这两个方法**那么就会调用该类的实现，如果没有的话就会使用defaultWriteObject()和defaultReadObject()，而HashMap就是自己实现了writeObject和readObject方法，自己对table做了处理。

HashMap要自己对table做处理呢,HashMap存储是根据Key的hash值来计算出，键值对应该放在数组的哪个位置，但是在不同的JVM中，得到的hash值不一定相同

反射

**JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法**；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

RTTI（Run-Time Type Information)：这项工作由**class对象**来完成的。

**加载类的时候就会初始化java.lang.Class对象**

**类对象c=Class.forName(‘XXX’)**//jni方法，返回clas对象引用

**对象.getClass()//jni方法,java类加载的时候 ,返回class对象引用**

**类.class 返回class对象引用**

Class类与java.lang.reflect类库一起对反射的概念进行了支持

c.newInstance()实例化对象

java.lang.reflect 包含field，method,constructor类

例：method [] methods = c.getMethods();

1. Class cls = Class.forName("chb.test.reflect.Student");
2. Method setMethod = cls.getDeclaredMethod("setAge",**int**.**class**);
3. setMethod.invoke(cls.newInstance(), 15);

jdbc使用反射：**解耦**

反射是动态加载的，程序运行期间生成指定类的对象，这样就可以程序运行期间生成不同的数据库驱动程序对象。省去了固定写死一个数据库驱动对象（通过new 构造函数的方法），造成每更换一个数据库，要重新编译代码的问题。

Class.forName("com.mysql.jdbc.Driver");

配置字符串可以写入xml中，就实现了解耦

参数传递：

类、数组引用传参

基础类型按值传参

1：一个方法不能***修改一个基本数据类型的参数***（即数值型或布尔型）。

2．一个方法可以改变一个**对象（数组）参数的状态**。

1. 一个方法不能让对象**参数（数组）引用一个新的对象**。

泛型:

声明时：

泛型类：Public class xxx<T>{}，调用 new xxx<X>();

泛型方法：Public interface xxx <T>{} 使用：class xxxx implements xxx<X>{}

泛型方法： public <T> void f(T x)

调用：使用泛型方法时，不必指明参数类型，编译器会自己找出具体的类型。泛型方法除了定义不同，调用就像普通方法一样

**Static方法无法访问泛型类的类型参数。**

动态代理

一：jdk动态代理

interface Interace{

doSomethind();

}

Class RealObject extentd Interface{

Dosomething(){//}

}

调用过程：

RealObject real = new RealObject();

Interface **proxy** = (Interface)**Proxy**.**newProxyInstance**(

Interface.class.getClassLoader();//**一个类加载器**

New Class[]{Interface.class}//**代理实现的列表接口 ，数组**

New DynamicProxyHandler(real)) //InvocationHandler的一个实现);

Proxy.doSomething()

**InvocationHandler**主要实现 **invoke**(Object proxy,Method method,Object [] args)和它的构造器

//实现

Class **DynamicProxyHandler implments InvocationHandler {**

**Private Object proxied;**

**Public** DynamicProxyHandler(Object proxied){

This.proxied= proxied

}

**invoke(Object proxy,Method method,Object [] args){**

//其他逻辑

**method.invoke(proxied,args);**

}

**}**

原理：JDK的动态代理，就是在程序运行的过程中，根据被**代理的接口**来动态生成代理类的class文件（字节码）。代理类中有接口的每个方法，实际都是调用实例化InvocationHandler的invoke方法（**基于反射**）

并加载运行的过程。newProxyInstance这个方法的第二个参数上，我们给这个代理对象提供了一组什么接口，那么我这个代理对象就会实现了这组接口并生成相应的方法

<http://rejoy.iteye.com/blog/1627405>

二：CGLib

其原理是通过字节码技术为一个类创建子类，并在子类覆盖其中方法实现增强，顺势织入横切逻辑。但因为采用的是继承，所以不能对**final**修饰的类进行代理。（没看过）。

常可以使用Java的动态代理创建代理，但当要代理的类**没有实现接口或者为了更好的性能**，CGLIB是一个好的选择。

CGLIB底层：使用字节码处理框架ASM，来转换字节码并生成新的类。不鼓励直接使用ASM，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉。

CGLIB缺点：**对于final方法，无法进行代理**。

**Assert（断言）**

assertion(断言)在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制。在实现中，assertion就是在程序中的一条语句，它对一个boolean表达式进行检查，一个正确程序必须保证这个boolean表达式的值为true；如果该值为false，说明程序已经处于不正确的状态下，assert将给出警告或退出

assert i==5;

Enumeration接口和Iterator接口的区别有哪些？

Enumeration速度是Iterator的2倍，同时占用更少的内存。但是，Iterator远远比Enumeration安全，因为其他线程不能够修改正在被iterator遍历的集合里面的对象。同时，Iterator允许调用者删除底层集合里面的元素，这对Enumeration来说是不可能的。