1、Java 变量

Java 中主要有如下几种类型的变量
局部变量
类变量（静态变量）-- 属于类
成员变量（非静态变量）-- 属于对象

2、关于枚举

```
package com.scu.lly;

public class EnumTest {

    /**
* 颜色枚举
*/
    enum ColorEnum{
        RED,
        GREEN,
        BLUE
    }
    /**
* 性别枚举
* 可用中文字符，不能单独使用数字
* （枚举值组成：字母、下划线）
*/
    enum SexEnum{
        男,
        女,
```

```
        MAN,
        WOWAM
    }
    /**
* 1、带有构造方法的枚举，构造方法为只能为private(默认可不写private)；
* 2、含带参构造方法的枚举，枚举值必须赋值；
* 3、枚举中有了其他属性或方法之后，枚举值必须定义在最前面，且需要在最后一个枚举值后面加分号";"
*/
    enum CarEnum{
        BMW("宝马",1000000),
        JEEP("吉普",800000),
        MINI("mini",200000);
        private String name;
        /**
* 从这里可以看出虽然枚举值不能直接由数字组成，但是我们可以给该枚举类添加一个int 类型的值，通过构造方法给其赋值，相当于间接的可以使用数值
*/
        private int price;
        private CarEnum(String name,int price){
            this.name = name;
            this.price = price;
        }
        //添加setter、getter 方法
```

```
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        public int getPrice() {
            return price;
        }
        public void setPrice(int price) {
            this.price = price;
        }
    }
    /**
* 由于枚举类都继承了Enum类，故我们定义的enum都不能在继承其他类了，但是可以实现其他接口
*/
    enum CarSetEnum implements Car{
        BMW("宝马"),
        JEEP("吉普"),
        MINI("mini");
        private String name;
        private CarSetEnum(String name){
            this.name = name;
```

```
        }
        //添加setter、getter 方法
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        @Override
        public void setCarName(String name) {
            this.name = name;
        }
    }
    public static void main(String[] args){
        ColorEnum color = ColorEnum.BLUE;
        switch(color){
            case RED:
            System.out.println("红色");
            break;
            case GREEN:
            System.out.println("绿色");
            break;
            case BLUE:
```

```
            System.out.println("蓝色");
            break;
        }
        getSelectedColor(color);
        //测试含构造方法的枚举
        System.out.println("吉普信息: "+CarEnum.JEEP.getName() + ":" +CarEnum.JEEP.p
rice);
        for (CarEnum car : CarEnum.values()){
            System.out.println(car.name);
            System.out.println(car.getPrice());
        }
        //测试实现接口的枚举
        CarSetEnum.BMW.setName("加长宝马");
        System.out.println(CarSetEnum.BMW.getName());
    }
    public static ColorEnum getSelectedColor(ColorEnum color){
        ColorEnum result;
        switch(color){
            case RED:
            System.out.println("红色");
            break;
            case GREEN:
            System.out.println("绿色");
```

```
            break;
            case BLUE:
            System.out.println("蓝色");
            break;
        }
        result = color;
        return result;
    }
    interface Car{
        public void setCarName(String name);
    }
}
```

3、访问控制修饰符

修饰符	说明
private	私有的，在同一类内可见。
默认没写	在同一包(包括子类和非子类)内可见。默认不使用任何修饰符。
protected	受保护的，对同一包内的类和所有子类可见。
public	共有的，对所有类可见。

主要是默认和 protected 这两个修饰符，总结起来就是：

- 默认的：同一包下可访问；
- protected：同一包和所有子类可访问；

（1）这里的可见、可访问指的是能不能通过 ”类的对象.变量名“的方式访问，这是因为除 static 声明的变量属于类变量外，其他的都属于实例变量，是属于某个对象的！

如，Person p = new Person(); p.age 直接访问 age 变量，对于那些私有的变量，很多情况下会对外提供 public 的 setter 和 getter 方法来供外部访问。

（2）要注意的是，对于有继承关系的子类来说，比如 class A extends B，A 直接继承拥有了默认的(在同一包下)、protected、public 的这个字段，可以直接使用该字段，而不用通过再次的实例化父类或"父类对象.字段"的形式访问，因为在实例化 A 类的时候父类 B 已经实例化好了。特别的，对于 protected 来说，如下形式是编译不能通过的。

```
package com.a

public class A extends B{

    public void test(){

        B b = new B();

        String str = b.age;

        //错误！不同包下的子类不能通过实例出来的父类获取 protected 的变量

        String str2 = age;

        //正确，A 类继承了 B，直接拥有了该字段

        String str3 = b.birthday;

        //正确，birthday 为 public

    }

}

package com.b

public class B{
```

```
    protected String age = "20";

    public String birthday = "1995";

}
```

结论就是：

在上面那个表格修饰符的约束前提下，对于非继承类关系，需要使用“实例变量.变量名的形式访问”（static 类变量除外）；

对于有继承关系的子类来说， 子类直接继承了拥有了默认的(在同一包下)、protected、public 的这个字段，可以直接使用该字段。

4、UTF-8 和 GBK 编码转换

下面哪段程序能够正确的实现了 GBK 编码字节流到 UTF-8 编码字节流的转换：byte[] src,dst;

- A、dst=String.fromBytes(src，"GBK").getBytes("UTF-8")
- B、dst=new String(src，"GBK").getBytes("UTF-8")
- C、dst=new String("GBK"，src).getBytes()
- D、dst=String.encode(String.decode(src，"GBK"))，"UTF-8")

正确答案：B

操作步骤就是先解码再编码，先通过 GBK 编码还原字符串，在该字符串正确的基础上得到“UTF-8”所对应的字节串。

5、try、catch、finally 执行顺序问题

下面函数将返回？

```
publicstaticintfunc (){

    try{

        return1;
```

```
    }
    catch(Exception e){
        return2;
    }
    finally{
        return3;
    }
}
```

- A、1
- B、2
- C、3
- D、编译错误

正确答案：C

（1）try catch 中只要有 finally 语句都要执行（有特例：如果 try 或 catch 里面有 exit（0）就不会执行 finally 了）；

（2）finally 语句在 try 或 catch 中的 return 语句执行之后返回之前执行，且 finally 里的修改语句不能影响 try 或 catch 中 return 已经确定的返回值；若 finally 里也有 return 语句则覆盖 try 或 catch 中的 return 语句直接返回；

（3）在遵守第（2）条 return 的情况下，执行顺序是：try-->catch（如果有异常的话）-->finally；

6、静态代码块、子类、父类初始化顺序

如下代码的输出结果：

```
package com.scu.lly;

public class HelloB extends HelloA {
```

```
    public HelloB() {
        System.out.println("-----------HelloB 构造方法-----------");
    }
    {
        System.out.println("I’m B class");
    }
    static{
        System.out.println("static B");
    }
    public static void main(String[] args){
        new HelloB();
    }
}
class HelloA{
    public HelloA(){
        System.out.println("-----------HelloA 构造方法-----------");
    }
    {
        System.out.println("I’m A class");
    }
    static{
        System.out.println("static A");
    }
```

```
}
```

输出结果：

```
static A

static B

I’m A class

----------HelloA 构造方法------------

I’m B class

----------HelloB 构造方法------------
```

执行顺序：1.静态代码块 --> 2.普通代码块 --> 3.构造方法

需要明白的是，1 是类级别的，2 和 3 是实例级别的，所以在父子类关系中，上述的执行顺序为：

父类静态代码块-->子类静态代码块-->父类普通代码块-->父类构造方法-->子类代码块-->子类构造方法；

也就是上到下（父类到子类）先走完 类级别的（静态的）--> 再依次走完父类的所有实例级别代码 --> 再走子类所有实例级别代码

7、关于 null 对象、static 变量和方法

有关下述 Java 代码描述正确的选项是____。

```
publicclassTestClass {

    privatestaticvoidtestMethod(){

        System.out.println("testMethod");

    }

    publicstaticvoidmain(String[] args) {
```

```
        ((TestClass)null).testMethod();

    }

}
```

- A、编译不通过
- B、编译通过，运行异常，报 NullPointerException
- C、编译通过，运行异常，报 IllegalArgumentException
- D、编译通过，运行异常，报 NoSuchMethodException
- E、编译通过，运行异常，报 Exception
- F、运行正常，输出 testMethod

正确答案：F

静态方法是属于类的，静态方法在对象实例创建前就已经存在了，它的使用不依赖于对象是否被创建。当我们通过类的实例来调用时，最后实际上还是将对象实例转换成了类去掉用该静态方法，所以这里的 null 只是迷惑大家的跟它没有什么关系。

这里 ((TestClass)null).testMethod();也可以写成 TestClass t = null; t.testMethod();同样可以正确输出。null 可以被强制转换成任意类型对象，虽然这个时候 t 被赋为了空，但这个“空对象”也是属于 TestClass 的，那么这个“空对象”也就可以去堆上的静态方法区调用 testMethod()方法了。

如果这里 testMethod 把 static 去掉，该 testMethod 方法就变成了实例对象的方法了。这时，可以编译通过，但是会报空指针。

同理，对于 static 变量也是一样的。比如 TestClass 中有如下变量：private static String str = "abc"; 我们通过 TestClass t = null; System.out.println(t.str); 同样可以正确输出。

8、关于线程启动

下列方法中哪个是执行线程的方法？ （）

- A、run()
- B、start()

- C、sleep()
- D、suspend()

正确答案：A

start()方法启动一个线程，使其处于就绪状态，得到了 CPU 就会执行，而调用 run()方法，就相当于是普通的方法调用，会在主线程中直接运行，此时没有开启一个线程。如下：

```
Thread t = new Thread(new Runnable() {

    @Override

    public void run() {

        try {

            Thread.sleep(2000);

        }

        catch (InterruptedException e) {

            e.printStackTrace();

        }

        System.out.println("----------线程中 run---------");

    }

}

);

System.out.println("-------主线程 11111------");

t.run();

System.out.println("-------主线程 222------");
```

这里调用 Thread 的 run()方法，此时相当于是普通的方法调用，并没有开启线程，直接在主线程中运行 Thread 中的 Runnable 方法，睡眠 2 秒后打印"----------线程中 run---------"，在执行 t.run 后面的打印。所以此时的输出为：

```
-------主线程 11111------

----------线程中 run---------

-------主线程 222------
```

若将 t.run()改为 t.start()，此时会开启一个线程，并使该线程处于就绪状态，得到 CPU 后开始执行，调用 t.start()后主线程继续执行下面的打印，所以此时的输出为：

```
-------主线程 11111------

-------主线程 222------

----------线程中 run---------
```

9、关于内部类

往 OuterClass 类的代码段中插入内部类声明, 哪一个是错误的:

```
public class OuterClass{

    private float f=1.0f;

    //插入代码到这里

}
```

A、

```
class InnerClass{

    public static float func(){

        return f;
```

```
    }
}
```

B、

```
abstract class InnerClass{
    public abstract float func(){
    }
}
```

C、

```
static class InnerClass{
    protected static float func(){
        return f;
    }
}
```

D、

```
public class InnerClass{
    static float func(){
        return f;
    }
}
```

正确答案：ABCD

静态的内部类才可以定义 static 方法，排除 AD；
B 抽象方法中不能有方法体；
C，静态方法不能够引用非静态变量。

10、Final 修饰符、volatile 修饰符

Final 修饰符，用来修饰类、方法和变量，final 修饰的类不能够被继承，修饰的方法可以被继承，重载，但是不能被子类重写（即重新定义）

（1）final 变量：
被声明为 final 的对象的引用不能指向不同的对象。但是 final 对象里的数据可以被改变。也就是说 final 对象的引用不能改变，但是里面的值可以改变。比如：

```
final Person p = new Person();

p.name = "aaa";

p.name = "bbb";
```

但是，如果是 final String str = "aaa"; str = "bbb";//错误编译不能通过，因为此时 str 的引用已经改变了！

（2）final 修饰方法
Final 修饰的方法可以被子类继承，但是不能被子类修改（重写）。

声明 final 方法的主要目的是防止该方法的内容被修改。

volatile 修饰符，Volatile 修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。一个 volatile 对象引用可能是 null。

11、StringBuffer 和 StringBuilder

和 String 类不同，StringBuffer 和 StringBuilder 类的对象能够被多次的修改，并且不产生新的未使用对象。

StringBuilder 类和 StringBuffer 之间的最大不同在于 StringBuilder 的方法不是线程安全的（不能同步访问）。

由于 StringBuilder 相较于 StringBuffer 有速度优势，所以多数情况下建议使用 StringBuilder 类。然而在应用程序要求线程安全的情况下，则必须使用 StringBuffer 类。

12、可变参数

JDK 1.5 开始，Java 支持传递同类型的可变参数给一个方法，一个方法中只能指定一个可变参数，它必须是方法的最后一个参数。任何普通的参数必须在它之前声明。

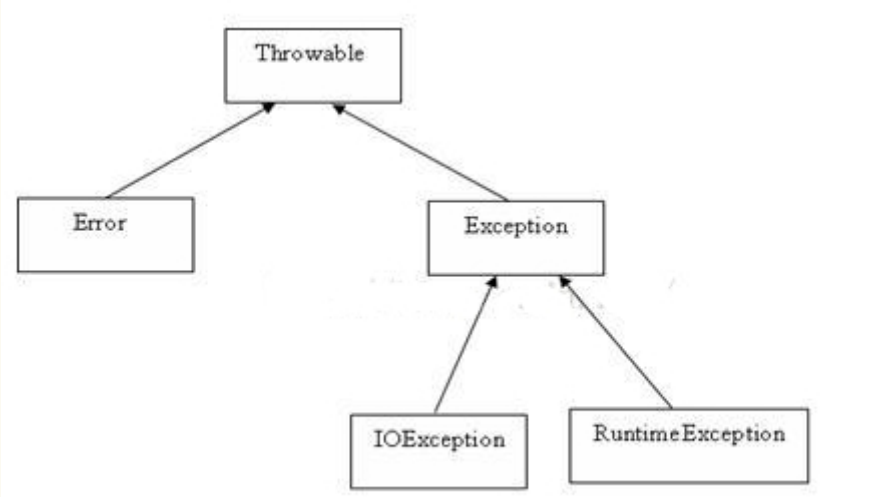
如，public void getStr(String ...str){}//正确

public void getStr2(String ...str,int a){}//错误，一个方法中可变参数只能有一个，它必须是方法的最后一个参数。

13、关于异常分类

所有的异常类是从 java.lang.Exception 类继承的子类。

Exception 类是 Throwable 类的子类。除了 Exception 类外，Throwable 还有一个子类 Error 。Error 用来指示运行时环境发生的错误。层次关系如图：



- 检查性异常: 不处理编译不能通过
- 非检查性异常：不处理编译可以通过，如果有抛出直接抛到控制台。
- 运行时异常（RuntimeException）： 继承自 RuntimeException 类的就是非检查性异常

- 非运行时异常： 就是检查性异常

下面的表中列出了 **Java** 的非检查性异常（**RuntimeException**）。

ArithmeticException	当出现异常的运算条件时，抛出此异常。例如，一个整数"除以零"时，抛出此类的一个实例。
ArrayIndexOutOfBoundsException	用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。
ArrayStoreException	试图将错误类型的对象存储到一个对象数组时抛出的异常。
ClassCastException	当试图将对象强制转换为不是实例的子类时，抛出该异常。
IllegalArgumentException	抛出的异常表明向方法传递了一个不合法或不正确的参数。
IllegalMonitorStateException	抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。
IllegalStateException	在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。
IllegalThreadStateException	线程没有处于请求操作所要求的适当状态时抛出的异常。
IndexOutOfBoundsException	指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
NegativeArraySizeException	如果应用程序试图创建大小为负的数组，则抛出该异常。
NullPointerException	当应用程序试图在需要对象的地方使用 `null` 时，抛出该异常
NumberFormatException	当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。
SecurityException	由安全管理器抛出的异常，指示存在安全侵犯。
StringIndexOutOfBoundsException	此异常由 `String` 方法抛出，指示索引或者为负，或者超出字符串的大小。
UnsupportedOperationException	当不支持请求的操作时，抛出该异常。

下面的表中列出了 **Java** 定义在 **java.lang** 包中的检查性异常类。

异常	描述
ClassNotFoundException	应用程序试图加载类时，找不到相应的类，抛出该异常。
CloneNotSupportedException	当调用 `Object` 类中的 `clone` 方法克隆对象，但该对象的类无法实现 `Cloneable` 接口时，抛出该异常。
IllegalAccessException	拒绝访问一个类的时候，抛出该异常。
InstantiationException	当试图使用 `Class` 类中的 `newInstance` 方法创建一个类的实例，而指定的类对象因为是一个接口或是一个抽象类而无法实例化时，抛出该异常。
InterruptedException	一个线程被另一个线程中断，抛出该异常。
NoSuchFieldException	请求的变量不存在
NoSuchMethodException	请求的方法不存在

其他

1、下面描述属于 java 虚拟机功能的是？

- A、通过 ClassLoader 寻找和装载 class 文件
- B、解释字节码成为指令并执行，提供 class 文件的运行环境
- C、进行运行期间垃圾回收

- D、提供与硬件交互的平台

正确答案：ABCD

2、下面有关 java threadlocal 说法正确的有？

- A、ThreadLocal 存放的值是线程封闭，线程间互斥的，主要用于线程内共享一些数据，避免通过参数来传递
- B、线程的角度看，每个线程都保持一个对其线程局部变量副本的隐式引用，只要线程是活动的并且 ThreadLocal 实例是可访问的；在线程消失之后，其线程局部实例的所有副本都会被垃圾回收
- C、在 Thread 类中有一个 Map，用于存储每一个线程的变量的副本。
- D、对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而 ThreadLocal 采用了“以空间换时间”的方式

正确答案：ABCD

ThreadLocal 类用于创建一个线程本地变量
在 Thread 中有一个成员变量 ThreadLocals，该变量的类型是 ThreadLocalMap,也就是一个 Map，它的键是 threadLocal，值就是变量的副本，ThreadLocal 为每一个使用该变量的线程都提供了一个变量值的副本，每一个线程都可以独立地改变自己的副本，是线程隔离的。通过 ThreadLocal 的 get() 方法可以获取该线程变量的本地副本，在 get 方法之前要先 set,否则就要重写 initialValue()方法。
ThreadLocal 不是用来解决对象共享访问问题的，而主要是提供了保持对象的方法和避免参数传递的方便的对象访问方式。一般情况下，通过 ThreadLocal.set() 到线程中的对象是该线程自己使用的对象，其他线程是不需要访问的，也访问不到的。各个线程中访问的是不同的对象。

3、以下集合对象中哪几个是线程安全的？( )

- A、ArrayList
- B、Vector
- C、Hashtable
- D、Stack

正确答案：BCD

在集合框架中，有些类是线程安全的，这些都是 jdk1.1 中的出现的。在 jdk1.2 之后，就出现许许多多非线程安全的类。 下面是这些线程安全的同步的类：

- vector：比 arraylist 多了个同步化机制（线程安全），因为效率较低，现在已经不太建议使用。在 web 应用中，特别是前台页面，往往效率（页面响应速度）是优先考虑的。
- statck：堆栈类，继承了 Vector
- hashtable：比 hashmap 多了个线程安全
- enumeration：枚举，相当于迭代器
- 除了这些之外，其他的都是非线程安全的类和接口。比如常用的 ArrayList、HashMap 都是线程不安全的。