static关键字：

static是一个关键字，翻译成:静态的。下面我们来看看static关键字的特性：

> - static修饰的变量，叫做静态变量。当所有对象的某个属性的值是相同的，建议将该属性定义为静态变量，来节省内存开销。
> - 静态变量在类加载时初始化，存储在堆中。
> - static修饰的方法叫做静态方法。
> - 所有静态变量和静态方法，统一使用“类名.”调用。虽然可以使用“引用.”来调用，但实际运行时和对象无关，所以不建议这样写，因为这样写会给其他人造成疑惑。
> - 使用“引用.”访问静态相关的，即使引用为null，也不会出现空指针异常。
> - 静态方法中不能使用this关键字。因此无法直接访问实例变量和调用实例方法。
> - 静态代码块在类加载时执行，一个类中可以编写多个静态代码块，遵循自上而下的顺序依次执行。
> - 静态代码块代表了类加载时刻，如果你有代码需要在此时刻执行，可以将该代码放到静态代码块中。
> - static还可以修饰内部类，修饰内部类，它就是一个静态内部类

1.1 static关键字修饰变量

现在我们来看一个类：

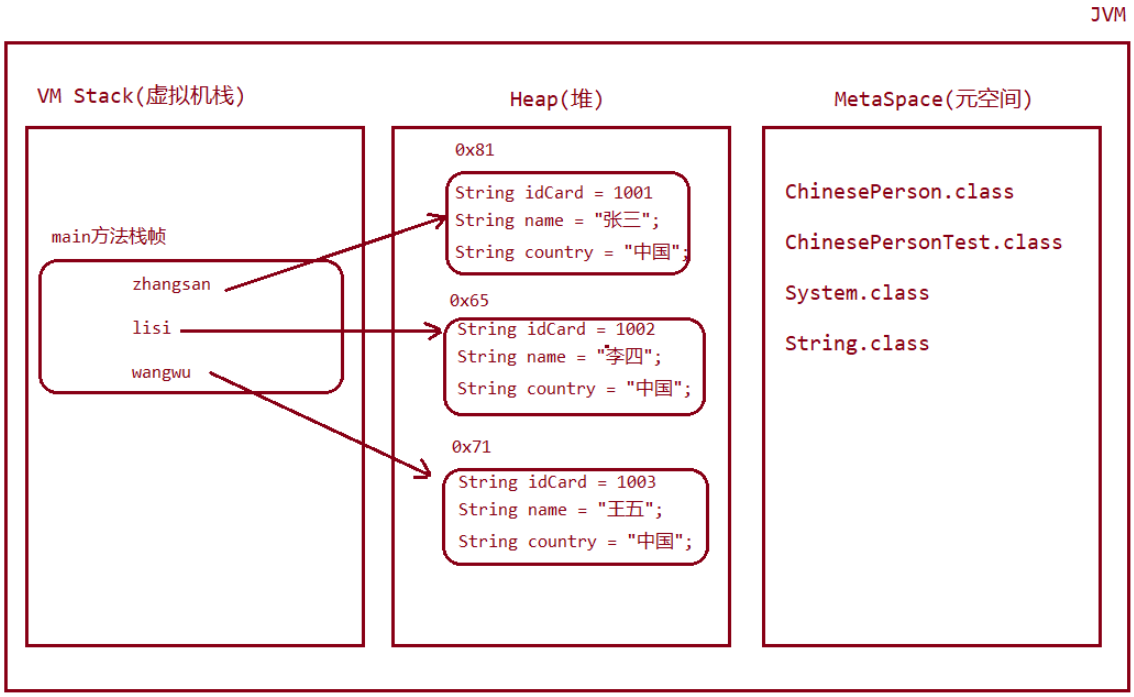
```
package com.xq.demo1;

/**
 * static关键字：
 *      1. static翻译为静态的
 *      2. static修饰的变量：静态变量
 *      3. static修饰的方法：静态方法
 *      4. 所有static修饰的，访问的时候，直接采用“类名.”，不需要new对象。
 *      5. 什么情况下把成员变量定义为静态成员变量？
 *          当一个属性是对象级别的，这个属性通常定义为实例变量。（实例变量是一个对象一份。100个对象就应该有100个空间）
 */
public class ChinesePerson { // 中国人类

    /*int i; // 实例变量
    static int j; // 静态变量

    public static void main(String[] args) {
        int i; // 局部变量
    }*/

    // 身份证号
    String idCard;

    // 姓名
    String name;

    // 国籍
    String country = "中国";

    public ChinesePerson(String idCard, String name) {
```

```
        this.idCard = idCard;
        this.name = name;
    }

    public void display(){
        System.out.println("身份证号："+this.idCard+"，姓名："+this.name+"，国籍："
+ this.country);
    }
}
```

现在我们来写一个测试类：

```
public class ChineseTest {
    public static void main(String[] args) {
        // 创建中国人对象3个
        ChinesePerson zhangsan = new ChinesePerson("1001", "张三");
        ChinesePerson lisi = new ChinesePerson("1002", "李四");
        ChinesePerson wangwu = new ChinesePerson("1003", "王五");

        zhangsan.display();
        lisi.display();
        wangwu.display();
    }
}
```

下面我们基于上面的代码，画出3个对象的内存图：



通过以上的内存图，我们发现什么问题：

当country国籍定义为实例变量的时候，每个对象中都一个country变量但是country的值永远都是"中国"，这个值不会因为对象的变化而发生变化。所有的ChinesePerson类型的对象的这个属性值都是一样的,如果还是将其定义为实例变量，有点浪费内存空间。

现在我们尝试对我们的代码进行优化：

```
/**
 * static关键字：
```

```
 *       1. static翻译为静态的
 *       2. static修饰的变量：静态变量
 *       3. static修饰的方法：静态方法
 *       4. 所有static修饰的，访问的时候，直接采用“类名.”，不需要new对象。
 *       5. 什么情况下把成员变量定义为静态成员变量？
 *            当一个属性是对象级别的，这个属性通常定义为实例变量。（实例变量是一个对象一份。100个对象就应该有100个空间）
 *            当一个属性是类级别的（所有对象都有这个属性，并且这个属性的值是一样的），建议将其定义为静态变量，在内存空间上只有一份。节省内存开销。
 *            这种类级别的属性，不需要new对象，直接通过类名访问。
 */
public class ChinesePerson { // 中国人类

    // 身份证号
    String idCard;

    // 姓名
    String name;

    // 国籍(使用static关键字修饰的变量就是静态变量)
    static String country = "中国";

    public ChinesePerson(String idCard, String name) {
        this.idCard = idCard;
        this.name = name;
    }

    public void display(){
        // 在编译器上，this.country会报一个警告，提示我们可以通过类名调用
        System.out.println("身份证号："+this.idCard+"，姓名："+this.name+"，国籍：" + this.country);
    }
}
```

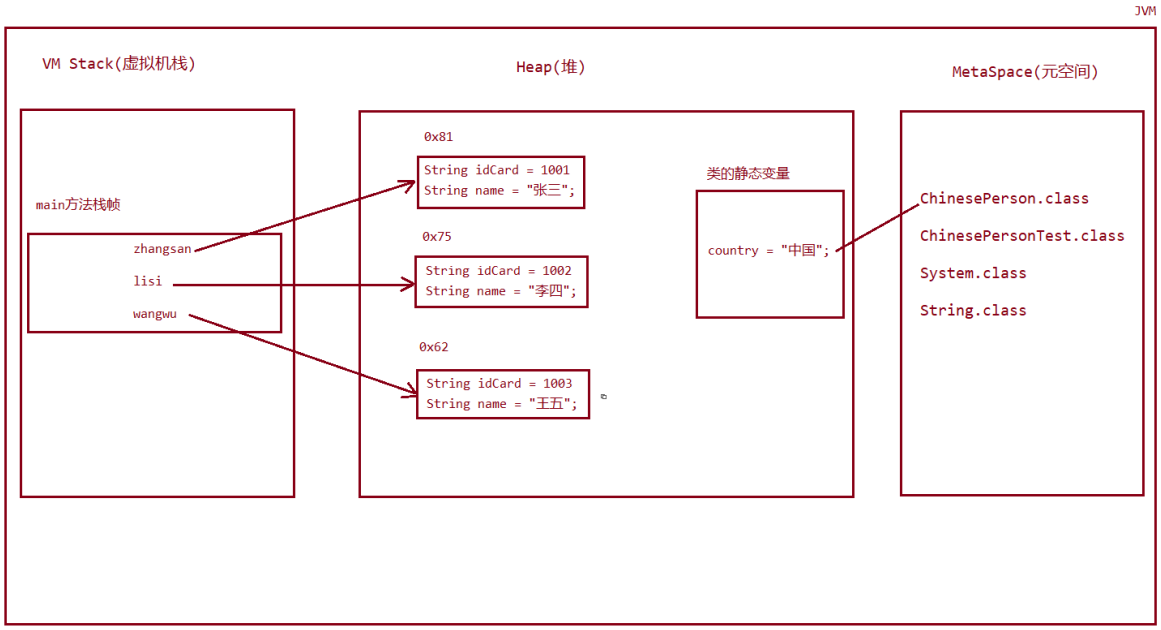
在测试类里面，我们对静态变量进行访问：

```
public class ChineseTest {
    public static void main(String[] args) {
        // 创建中国人对象3个
        ChinesePerson zhangsan = new ChinesePerson("1001", "张三");
        ChinesePerson lisi = new ChinesePerson("1002", "李四");
        ChinesePerson wangwu = new ChinesePerson("1003", "王五");

        zhangsan.display();
        lisi.display();
        wangwu.display();

        System.out.println("国籍：" + ChinesePerson.country);
    }
}
```

现在问题来了：**静态变量存储在哪里？静态变量在什么时候初始化？（什么时候开辟空间）**

答案：JDK8之后：静态变量存储在堆内存当中。类加载时初始化，并且只会被JVM加载一次。

加了static变量的内存图怎么画呢：



当country变量声明为静态变量时，和对象就没有关系了。并且静态变量country在类加载时初始化。

静态变量可以采用“引用.”来访问吗？可以（但不建议：会给程序员造成困惑，程序员会认为country是一个实例变量。）建议还是使用“类名.”来访问。这是正规的。

```
System.out.println(zhangsan.country);
System.out.println(lisi.country);
System.out.println(wangwu.country);
```

静态变量也可以用“引用.”访问，但是实际运行时和对象无关。所以以下程序也不会出现空指针异常。

```
System.out.println(zhangsan.country);
System.out.println(lisi.country);
System.out.println(wangwu.country);
```

什么时候会出现空指针异常？一个空引用访问实例相关的，都会出现空指针异常。

```
System.out.println(zhangsan.name); // 会出现空指针异常
```

1.2 static关键字修饰方法

在ChinesePerson中定义一个静态方法：

```
public static void test(){
    System.out.println("静态方法test执行了");
}
```

静态方法也可以使用"引用."的方式来访问：

```
zhangsan.test(); // 如果引用对象为null，也不会出现空指针异常。
```

但是这种访问是不建议的。静态方法就应该使用“类名.”来访问。

```
ChinesePerson.test();
```

在静态方法中，只能访问静态的资源，如果直接访问非静态的资源，会报错。

```
public class ChinesePerson { // 中国人类

    // 身份证号
    String idCard;

    // 姓名
    String name;

    // 国籍(使用static关键字修饰的变量就是静态变量)
    static String country = "中国";

    public ChinesePerson(String idCard, String name) {
        this.idCard = idCard;
        this.name = name;
    }

    public void display(){
        // 在编译器上，this.country会报一个警告，提示我们可以通过类名调用
        System.out.println("身份证号："+this.idCard+"，姓名："+this.name+"，国籍："
+ this.country);
    }

     // 静态方法
    public static void test(){
        System.out.println("静态方法test执行了");

        // 这个不行
        //display();
        //System.out.println(name);

        // 这些可以
        System.out.println(Chinese.country);
        System.out.println(country); // 在同一个类中，类名. 可以省略。

        // 这个可以
        // ChinesePerson.test2();
        test2();
    }

    public static void test2(){

    }
}
```

为什么？**在Java中，静态资源在类加载的过程中被加载。**当JVM加载一个类时，会先加载该类的静态资源，包括静态变量和静态方法，然后再加载非静态资源。

在静态方法中使用非静态资源，如果此时静态资源还没有被加载，这样调用就会有问题，所以在语法层面上，不允许在静态方法里面，直接访问非静态资源(除非你在静态方法中创建一个对象，通过对象进行调用)。

为了加深理解，我们再写一个例子，在ChinesePerson中定义如下：

```
// 实例方法
public void doSome(){
    // 标准的访问方式
    /*  System.out.println(this.k);
        System.out.println(Chinese.f);
        this.doOther1();
        Chinese.doOther2();
     */

    // 省略的方式
    System.out.println(k);
    System.out.println(f);
    doOther1();
    doOther2();
}

// 实例变量
int k = 100;

// 静态变量
static int f = 1000;

// 实例方法
public void doOther1(){
    System.out.println("do other1....");
}

// 静态方法
public static void doOther2(){
    System.out.println("do other2....");
}
```

1.3 静态代码块

语法格式:

```
static{
   // 定义代码
}
```

下面我们来看一个例子:

```
public class StaticTest01 {
    // 静态代码块
    static {
        System.out.println("静态代码块1执行了");
    }

    public static void main(String[] args) {
        System.out.println("main 方法执行了!");
    }

}
```

运行程序，程序的执行结果如下:

```
静态代码块1执行了
main 方法执行了!
```

静态代码块在main方法之前执行，说明：静态代码块在类加载时执行，并且只执行一次。

静态代码块是可以写多个的，并且**遵循自上而下的顺序，依次执行**：

```
public class StaticTest01 {
    // 静态代码块1
    static {
        System.out.println("静态代码块1执行了");
    }

    // 静态代码块2
    static {
        System.out.println("静态代码块2执行了");
    }

    public static void main(String[] args) {
        System.out.println("main 方法执行了!");
    }

     // 静态代码块3
    static {
        System.out.println("静态代码块3执行了");
    }

}
```

运行程序，程序的执行结果：

```
静态代码块1执行了
静态代码块2执行了
静态代码块3执行了
main 方法执行了!
```

在静态代码块中能访问实例变量吗？不行，因为静态代码块在类加载的时候执行，此时实例对象还不存在。

```
public class StaticTest01 {

     // 实例变量
    String name = "zhangsan";

    // 静态代码块1
    static {
        // 在静态代码块中访问实例对象
        // System.out.println(name); // 报错
        System.out.println("静态代码块1执行了");
    }
}
```

在静态代码块中是可以访问静态变量的，因为静态变量恰好在类加载时初始化的。

```
public class StaticTest01 {
```

```
    // 实例变量
    String name = "zhangsan";

    // 静态变量
    static int i = 100;

    // 静态代码块1
    static {
        // 在静态代码块中访问实例对象
        // System.out.println(name); // 报错
        // 可以访问
        System.out.println(i);
        System.out.println("静态代码块1执行了");
        // 报错，加载顺序按照代码自上而下的顺序初始化。此时j还没有被初始化
        System.out.println(j);
    }

   // 静态变量
   static int j = 100;
}
```

搞清楚静态代码块的用法之后，**静态代码块什么时候用?**

如果我们想要在类加载的时候，用于执行一**次性的初始化操作**。就可以使用静态代码块。比如读取配置文件、建立数据库连接等。静态代码块可以在类加载时预加载一些资源，以提高程序的性能。例如，可以在静态代码块中预加载一些常用的数据，减少后续操作的延迟。

1.4 静态内部类

static关键字还可以修饰一个类。当在一个类的内部嵌套另外一个类的时候，这个内部类就可以使用static关键字修饰。我们也称为这个内部类是静态内部类。**这意味着它不需要外部类的实例就可以被创建和使用。**

静态内部类不能访问外部类的非静态成员（实例变量和实例方法），但可以访问外部类的静态成员（静态变量和静态方法）以及它自己的成员。

下面我们来看看其具体用法：

```
public class Outer { // 外部类

    public static void out(){
        System.out.println("out方法是外部类的静态方法");
    }

    public void show(){
        System.out.println("这是外部类的一个非静态方法");
    }

    static class Inner{ // 静态内部类
        public void inner(){ // 在内部类的普通方法中可以访问外部类的静态方法
            out();
            // show(); // 报错
        }

        public static void staticInner(){ // 在内部类的静态方法中可以访问外部类的静态方法
```

```
            out();
            // show(); // 报错
        }
    }
}
```

定义一个测试类：

```
public class TestInnerClass {
    public static void main(String[] args) {
        Outer.Inner.staticInner(); // 调用静态内部类中的静态方法
        System.out.println(Outer.Inner.username);  // 调用静态内部类中的静态资源

        Outer.Inner inner = new Outer.Inner();
        inner.inner();  // 调用静态内部类中的非静态方法
    }
}
```

静态内部类的应用：实现单例设计模式

```
public class Singleton {

    private Singleton(){

    }

    private static class SingletonHandler{
        private static Singleton singleton = new Singleton();
    }

    public static Singleton getInstance(){
        return SingletonHandler.singleton;
    }
}
```

测试：

```
public class TestSingleton {
    public static void main(String[] args) {
        Singleton o1 = Singleton.getInstance();
        Singleton o2 = Singleton.getInstance();
        System.out.println(o1 == o2); // true
    }
}
```

分析：

- 在这里，Singleton的实例singleton被设置为静态内部类SingletonHandler的静态成员
- 当我们第一次调用getInstance方法，访问了SingletonHandler.singleton，由于SingletonHandler是静态内部类，那么根据静态内部类的特性，就会触发该静态内部类的加载，即JVM就会帮我们完成静态成员singleton的创建，这样子，JVM帮我们保证了线程安全，而且这还是懒加载的方式（需要才创建）
- 后续再调用getInstance方法，由于INSTANCE已经创建好了，所以直接返回即可

这种实现单例设计模式的好处就是：在并发场景下面是线程安全的，并且实现了懒加载。