类和对象（下）

包

包 (package) 是组织类的一种方式.

使用包的主要目的是保证类的唯一性.

> 例如, 你在代码中写了一个 Test 类. 然后你的同事也可能写一个 Test 类. 如果出现两个同名的类, 就会冲突, 导致代码不能编译通过.

导入包中的类

Java 中已经提供了很多现成的类供我们使用. 例如

```
public class Test {
    public static void main(String[] args) {
        java.util.Date date = new java.util.Date();
        // 得到一个毫秒级别的时间戳
        System.out.println(date.getTime());
    }
}
```

可以使用 `java.util.Date` 这种方式引入 `java.util` 这个包中的 `Date` 类.

但是这种写法比较麻烦一些, 可以使用 `import` 语句导入包.

```
import java.util.Date;
public class Test {
    public static void main(String[] args) {
        Date date = new Date();
        // 得到一个毫秒级别的时间戳
        System.out.println(date.getTime());
    }
}
```

如果需要使用 `java.util` 中的其他类, 可以使用 `import java.util.*`

```
import java.util.*;
public class Test {
    public static void main(String[] args) {
        Date date = new Date();
        // 得到一个毫秒级别的时间戳
        System.out.println(date.getTime());
    }
}
```

但是我们更建议显式的指定要导入的类名. 否则还是容易出现冲突的情况.

```
import java.util.*;
import java.sql.*;
public class Test {
    public static void main(String[] args) {
        // util 和 sql 中都存在一个 Date 这样的类, 此时就会出现歧义, 编译出错
        Date date = new Date();
        System.out.println(date.getTime());
    }
}

// 编译出错
Error:(5, 9) java: 对Date的引用不明确
  java.sql 中的类 java.sql.Date 和 java.util 中的类 java.util.Date 都匹配
```

在这种情况下需要使用完整的类名

```
import java.util.*;
import java.sql.*;
public class Test {
    public static void main(String[] args) {
        java.util.Date date = new java.util.Date();
        System.out.println(date.getTime());
    }
}
```

注意事项: `import` 和 C++ 的 `#include` 差别很大. C++ 必须 `#include` 来引入其他文件内容, 但是 Java 不需要. import 只是为了写代码的时候更方便. `import` 更类似于 C++ 的 `namespace` 和 `using`

静态导入

使用 `import static` 可以导入包中的静态的方法和字段.

```
import static java.lang.System.*;
public class Test {
    public static void main(String[] args) {
        out.println("hello");
    }
}
```

使用这种方式可以更方便的写一些代码, 例如

```java
import static java.lang.Math.*;

public class Test {
    public static void main(String[] args) {
        double x = 30;
        double y = 40;
        // 静态导入的方式写起来更方便一些.
        // double result = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
        double result = sqrt(pow(x, 2) + pow(y, 2));
        System.out.println(result);
    }
}
```

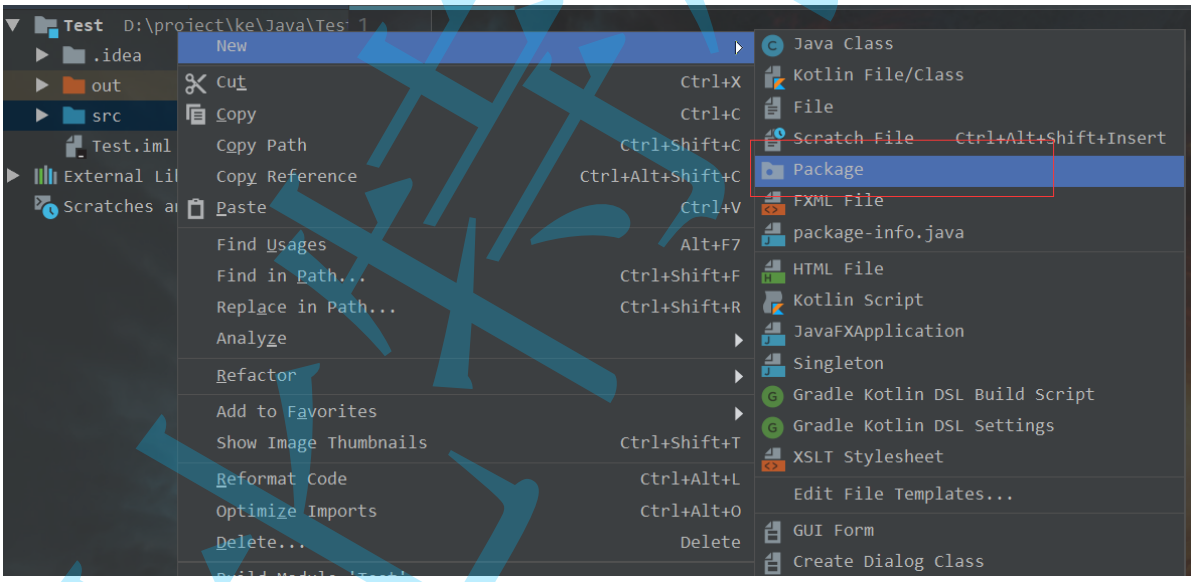
将类放到包中

基本规则

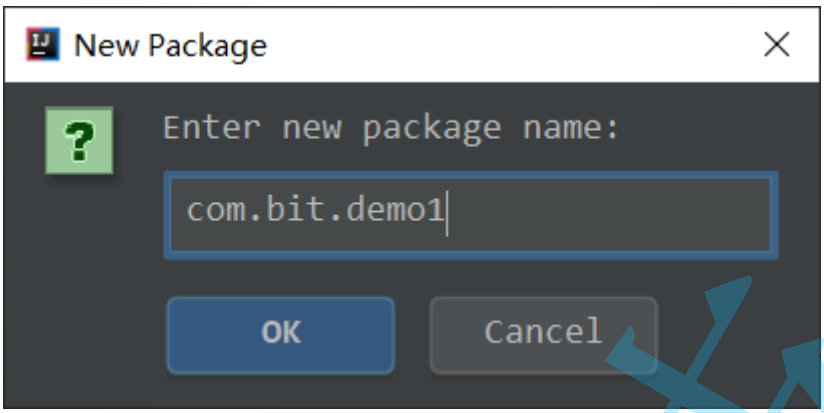
- 在文件的最上方加上一个 package 语句指定该代码在哪个包中.
- 包名需要尽量指定成唯一的名字, 通常会用公司的域名的颠倒形式(例如 `com.bit.demo1`).
- 包名要和代码路径相匹配. 例如创建 `com.bit.demo1` 的包, 那么会存在一个对应的路径 `com/bit/demo1` 来存储代码.
- 如果一个类没有 package 语句, 则该类被放到一个默认包中.

操作步骤

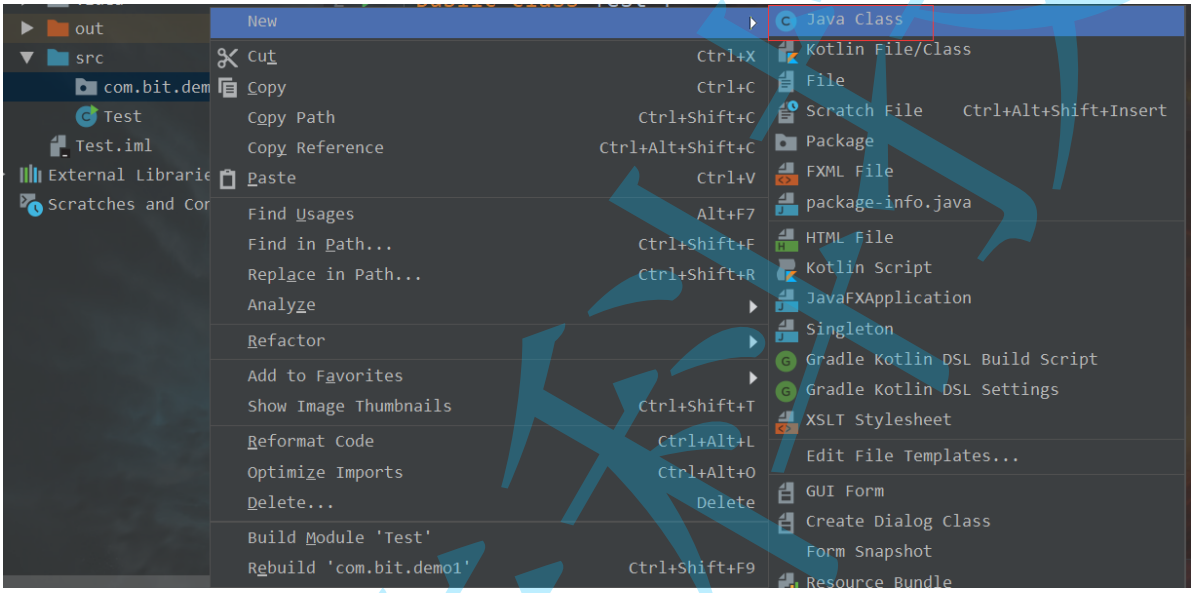
1) 在 IDEA 中先新建一个包: 右键 src -> 新建 -> 包



2) 在弹出的对话框中输入包名, 例如 `com.bit.demo1`



3) 在包中创建类, 右键包名 -> 新建 -> 类, 然后输入类名即可.



4) 此时可以看到我们的磁盘上的目录结构已经被 IDEA 自动创建出来了

5) 同时我们也看到了, 在新创建的 `Test.java` 文件的最上方, 就出现了一个 package 语句

包的访问权限控制

我们已经了解了类中的 public 和 private. private 中的成员只能被类的内部使用.

如果某个成员不包含 public 和 private 关键字, 此时这个成员可以在包内部的其他类使用, 但是不能在包外部的类使用.

下面的代码给了一个示例. Demo1 和 Demo2 是同一个包中, Test 是其他包中.

Demo1.java

```
package com.bit.demo;

public class Demo1 {
    int value = 0;
}
```

Demo2.java

```
package com.bit.demo;

public class Demo2 {
    public static void Main(String[] args) {
        Demo1 demo = new Demo1();
        System.out.println(demo.value);
    }
}

// 执行结果, 能够访问到 value 变量
10
```

Test.java

```
import com.bit.demo.Demo1;

public class Test {
    public static void main(String[] args) {
        Demo1 demo = new Demo1();
        System.out.println(demo.value);
    }
}

// 编译出错
Error:(6, 32) java: value在com.bit.demo.Demo1中不是公共的; 无法从外部程序包中对其进行访问
```

常见的系统包

1. java.lang:系统常用基础类(String、Object),此包从JDK1.1后自动导入。
2. java.lang.reflect:java 反射编程包;
3. java.net:进行网络编程开发包。
4. java.sql:进行数据库开发的支持包。
5. java.util:是java提供的工具程序包。(集合类等) **非常重要**

6. java.io:I/O编程开发包。

访问限定符

刚才我们发现, 如果把字段设为 private, 子类不能访问. 但是设成 public, 又违背了我们 "封装" 的初衷.

两全其美的办法就是 protected 关键字.

- 对于类的调用者来说, protected 修饰的字段和方法是不能访问的
- 对于类的 **子类** 和 **同一个包的其他类** 来说, protected 修饰的字段和方法是可以访问的

```
// Animal.java
public class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat(String food) {
        System.out.println(this.name + "正在吃" + food);
    }
}

// Bird.java
public class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    public void fly() {
        // 对于父类的 protected 字段, 子类可以正确访问
        System.out.println(this.name + "正在飞 ~(￣▽￣)~");
    }
}

// Test.java 和 Animal.java 不在同一个 包 之中了.
public class Test {
    public static void main(String[] args) {
        Animal animal = new Animal("小动物");
        System.out.println(animal.name); // 此时编译出错, 无法访问 name
    }
}
```

小结: Java 中对于字段和方法共有四种访问权限

- private: 类内部能访问, 类外部不能访问
- 默认(也叫包访问权限): 类内部能访问, 同一个包中的类可以访问, 其他类不能访问.
- protected: 类内部能访问, 子类和同一个包中的类可以访问, 其他类不能访问.
- public : 类内部和类的调用者都能访问

No	范围	private	default	protected	public
1	同一包中的同一类	✓	✓	✓	✓
2	同一包中的不同类		✓	✓	✓
3	不同包中的子类			✓	✓
4	不同包中的非子类				✓

> **什么时候下用哪一种呢?**
>
> 我们希望类要尽量做到 "封装", 即隐藏内部实现细节, 只暴露出 **必要** 的信息给类的调用者.
>
> 因此我们在使用的时候应该尽可能的使用 **比较严格** 的访问权限. 例如如果一个方法能用 private, 就尽量不要用 public.
>
> 另外, 还有一种 **简单粗暴** 的做法: 将所有的字段设为 private, 将所有的方法设为 public. 不过这种方式属于是对访问权限的滥用, 还是更希望同学们能写代码的时候认真思考, 该类提供的字段方法到底给 "谁" 使用(是类内部自己用, 还是类的调用者使用, 还是子类使用).

static

> 1、修饰属性
>
> 2、修饰方法
>
> 3、代码块(本课件中会介绍)
>
> 4、修饰类(后面讲内部类会讲到)

a) 修饰属性，Java静态属性和类相关, 和具体的实例无关. 换句话说, 同一个类的不同实例共用同一个静态属性.

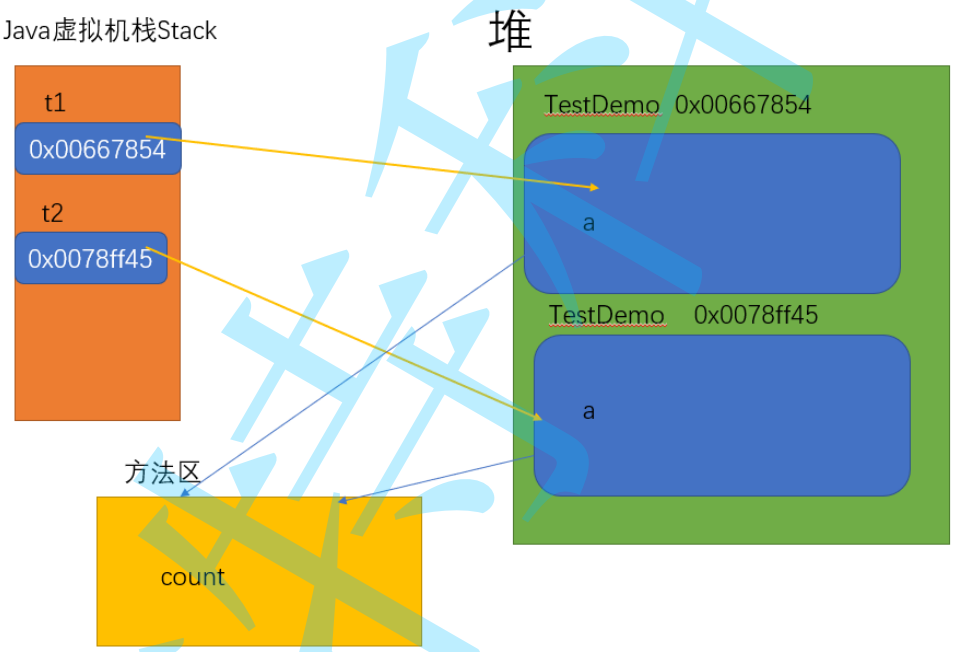
```
class TestDemo{
    public int a;
    public static int count;
}

public class Main{

    public static void main(String[] args) {
        TestDemo t1 = new TestDemo();
        t1.a++;
        TestDemo.count++;
        System.out.println(t1.a);
```

```
        System.out.println(TestDemo.count);
        System.out.println("============");
        TestDemo t2 = new TestDemo();
        t2.a++;
        TestDemo.count++;
        System.out.println(t2.a);
        System.out.println(TestDemo.count);
    }
}
```

输出结果为：

```
1
1
============
1
2
```

示例代码内存解析：count被static所修饰，所有类共享。且不属于对象，访问方式为：类名 . 属性。



b) 修饰方法

如果在任何方法上应用 `static` 关键字，此方法称为静态方法。

- 静态方法属于类，而不属于类的对象。
- 可以直接调用静态方法，而无需创建类的实例。
- 静态方法可以访问静态数据成员，并可以更改静态数据成员的值。

```
class TestDemo{
    public int a;
    public static int count;

    public static void change() {
        count = 100;
        //a = 10; error  不可以访问非静态数据成员
    }
```

```
}

public class Main{

    public static void main(String[] args) {
        TestDemo.change(); //无需创建实例对象 就可以调用
        System.out.println(TestDemo.count);
    }
}
```

输出结果：

```
100
```

注意事项1: 静态方法和实例无关, 而是和类相关. 因此这导致了两个情况:

- 静态方法不能直接使用非静态数据成员或调用非静态方法(非静态数据成员和方法都是和实例相关的).
- this和super两个关键字不能在静态上下文中使用(this 是当前实例的引用, super是当前实例父类实例的引用, 也是和当前实例相关).

注意事项2

- 我们曾经写的方法为了简单, 都统一加上了 static. 但实际上一个方法具体要不要带 static, 都需要是情形而定.
- main 方法为 static 方法.

代码块

字段的初始化方式有：

1. 就地初始化
2. 使用构造方法初始化
3. 使用代码块初始化

前两种方式前面已经学习过了, 接下来我们介绍第三种方式, 使用代码块初始化.

6.1 什么是代码块

使用`{}`定义的一段代码.

根据代码块定义的位置以及关键字，又可分为以下四种：

- 普通代码块
- 构造块
- 静态块
- 同步代码块（后续讲解多线程部分再谈）

6.2 普通代码块

普通代码块：定义在方法中的代码块.

```
public class Main{
    public static void main(String[] args) {
        { //直接使用{}定义，普通方法块
            int x = 10 ;
            System.out.println("x1 = " +x);
        }
        int x = 100 ;
        System.out.println("x2 = " +x);
    }
}

// 执行结果
x1 = 10
x2 = 100
```

这种用法较少见/.

6.3 构造代码块

构造块：定义在类中的代码块(不加修饰符)。也叫：**实例代码块**。构造代码块一般用于初始化实例成员变量。

```
class Person{
    private String name;//实例成员变量
    private  int age;
    private String sex;

    public Person() {
        System.out.println("I am Person init()!");
    }

    //实例代码块
    {
        this.name = "bit";
        this.age = 12;
        this.sex = "man";
        System.out.println("I am instance init()!");
    }

    public void show(){
        System.out.println("name: "+name+" age: "+age+" sex: "+sex);
    }

}

public class Main {
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.show();
    }
```

```
}

// 运行结果
I am instance init()!
I am Person init()!
name: bit age: 12 sex: man
```

注意事项: 实例代码块优先于构造函数执行。

6.4 静态代码块

使用static定义的代码块。**一般用于初始化静态成员属性。**

```
class Person{

    private String name;//实例成员变量
    private  int age;
    private String sex;
    private static int count = 0;//静态成员变量     由类共享数据   方法区

    public Person(){
        System.out.println("I am Person init()!");
    }

     //实例代码块
    {
        this.name = "bit";
        this.age = 12;
        this.sex = "man";
        System.out.println("I am instance init()!");
    }

    //静态代码块
    static {
        count = 10;//只能访问静态数据成员
        System.out.println("I am static init()!");
    }

    public void show(){
        System.out.println("name: "+name+" age: "+age+" sex: "+sex);
    }

}

public class Main {
    public static void main(String[] args) {
        Person p1 = new Person();
         Person p2 = new Person();//静态代码块是否还会被执行?
    }
}
```

注意事项

- 静态代码块不管生成多少个对象，其只会执行一次，且是最先执行的。
- 静态代码块执行完毕后, 实例代码块（构造块）执行，再然后是构造函数执行。

内部类

在 Java 中，可以将一个类定义在另一个类里面或者一个方法里面，这样的类称为内部类。广泛意义上的内部类一般来说包括这四种：静态内部类、匿名内部类、成员内部类和局部内部类。

注意：定义在class 类名{}花括号外部的，即使是在一个文件里，也不是内部类了。比如这样：

```
public class A{

}
class B{

}
```

再来看看内部类都是怎样的：

静态内部类

先来看看静态内部类是怎么定义的：

```
public class Test {

    static class B{

    }
}
```

和静态变量、静态方法类似，静态内部类也是和当前类（Test）绑定。

使用时，也是通过Test类来调用，如

```
public class Test {

    static class B{

    }
    public static void main(String[] args) {
        B b1 = new Test.B();
        //在当前类Test中使用时，和静态变量，静态方法类似，也可以把Test.B()省略写为B()
        B b2 = new B();
    }
}
```

匿名内部类

匿名内部类的定义，是在一个方法或是代码块中定义的类，并且没有显示申明类的名称，比如这样：

```
public class Test {

    public static void main(String[] args) {
        //定义了一个匿名内部类
        A a = new A(){

        };
    }
}
class A{

}
```

匿名内部类是使用的非常多的一种内部类，和A a = new A(); 这样的实例操作不同，后边还有一个大括号，表示可以重写方法，其实是定义了另外一个类（没有显示的类名，所以叫匿名）。经常用在需要实例化某个对象，但需要重写方法时，比如new接口，抽象类就是使用匿名内部类较多的方式。

```
public class Test {

    public static void main(String[] args) {
        //定义了一个匿名内部类
        X x = new X(){
            @Override
            public void callback() {

            }
        };
    }
}
interface X{
    void callback();
}
```

成员内部类（了解）

成员内部类，顾名思义，是作为对象的一个成员来定义的类：

```
public class Test {

    class C{

    }
}
```

和成员变量、实例方法类似，成员内部类也是和当前类的实例对象绑定的，类似Test类对象的成员。需要通过对象来使用：

```
public class Test {

    public static void main(String[] args) {
        C c = new Test().new C();
    }

    class C{

    }
}
```

成员内部类使用的比较少，作为了解即可。

局部内部类（了解）

联想到局部变量，局部内部类的作用域也是和局部变量类似，是在方法或是代码块中定义。这点和匿名内部类的作用域一样：

```
public class Test {

    public static void main(String[] args) {
        class D extends A{

        }
        System.out.println(new D());
    }

}
class A{

}
```

局部内部类和匿名内部类的写法非常类似，只是显示声明了类的名称，一般也很少使用，做简单了解即可。

类和对象的内存布局

观察以下代码, 分析内存布局.

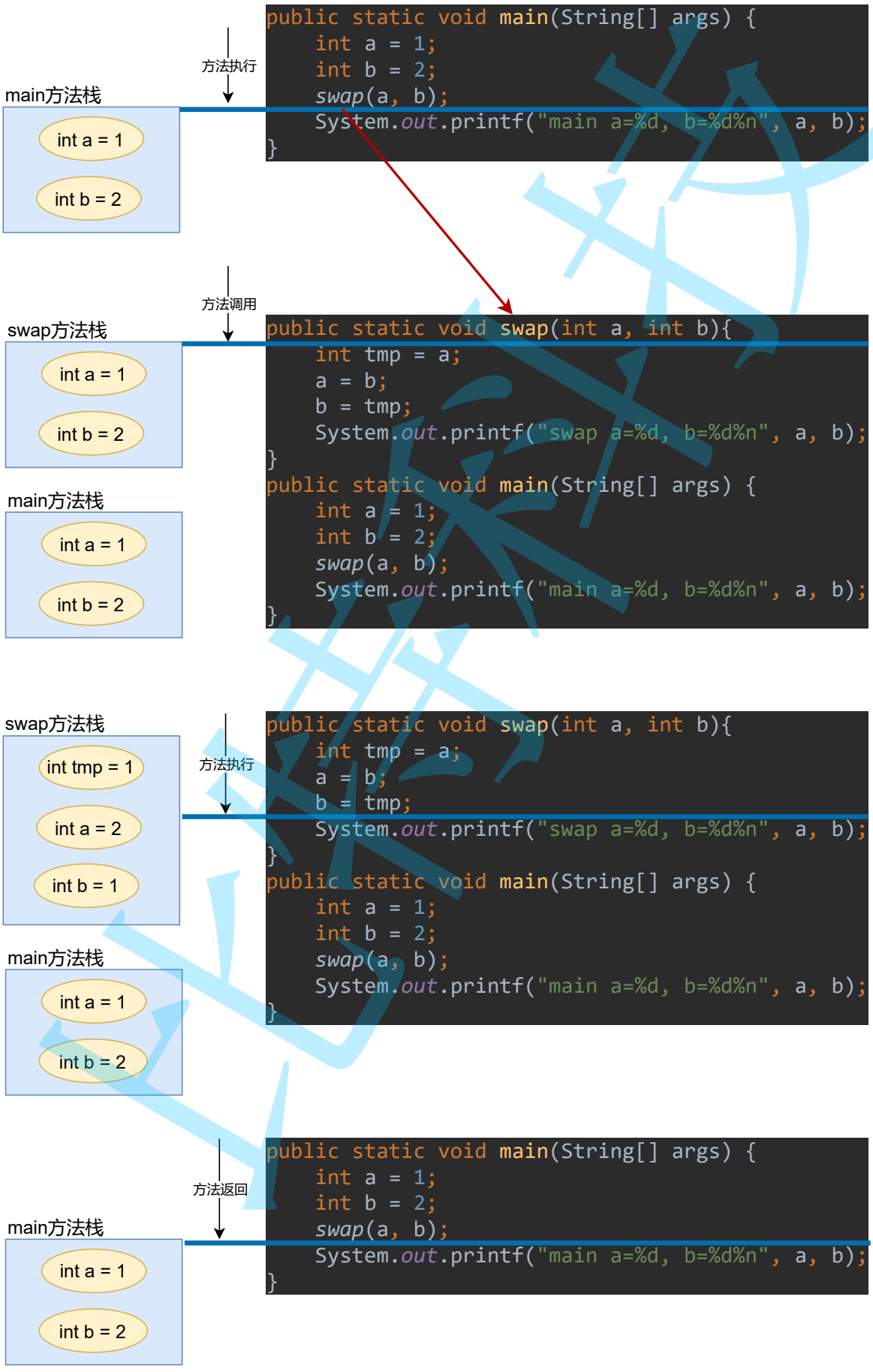
局部变量和方法栈帧

```
public class Main{
    public static void swap(int a, int b){
        int tmp = a;
        a = b;
        b = tmp;
        System.out.printf("swap a=%d, b=%d%n", a, b);
    }
    public static void main(String[] args) {
        int a = 1;
        int b = 2;
        swap(a, b);
        System.out.printf("main a=%d, b=%d%n", a, b);
    }
}
```

输出结果为

> swap a=2, b=1 main a=1, b=2

内存布局为：



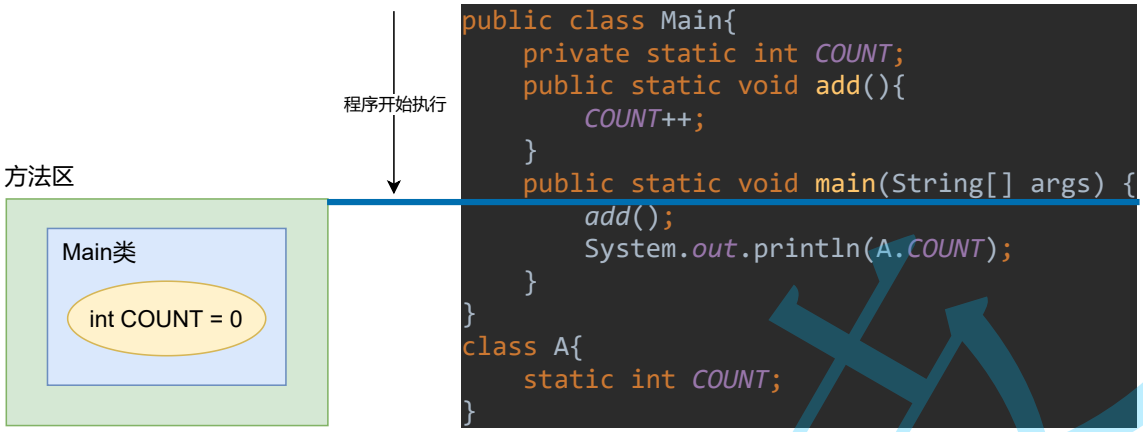
整体看，某个方法进入时就有自己的方法栈帧，传入参数都是局部变量，方法退出后，所有栈帧内保存的局部变量就销毁了。

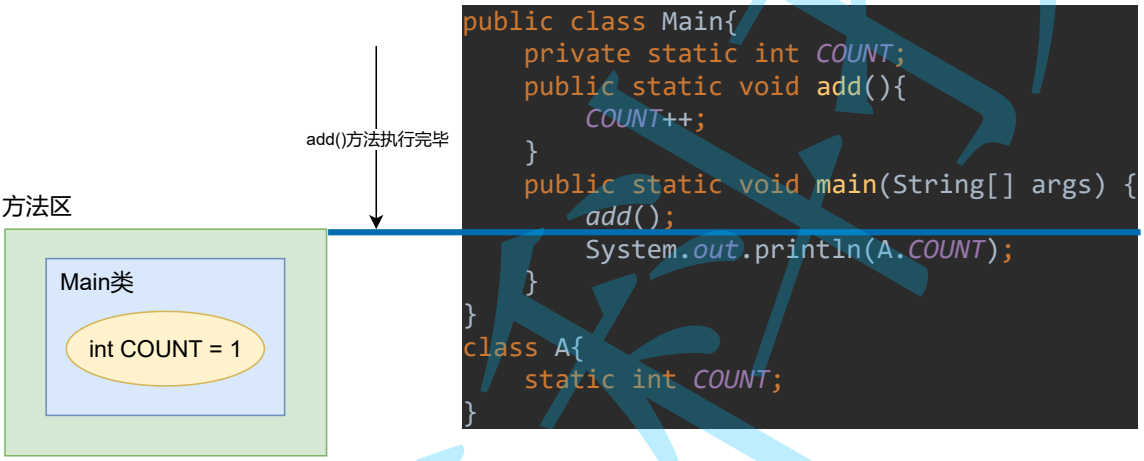
注意，这里的方法不分实例方法还是静态方法，都是满足上面的结论。

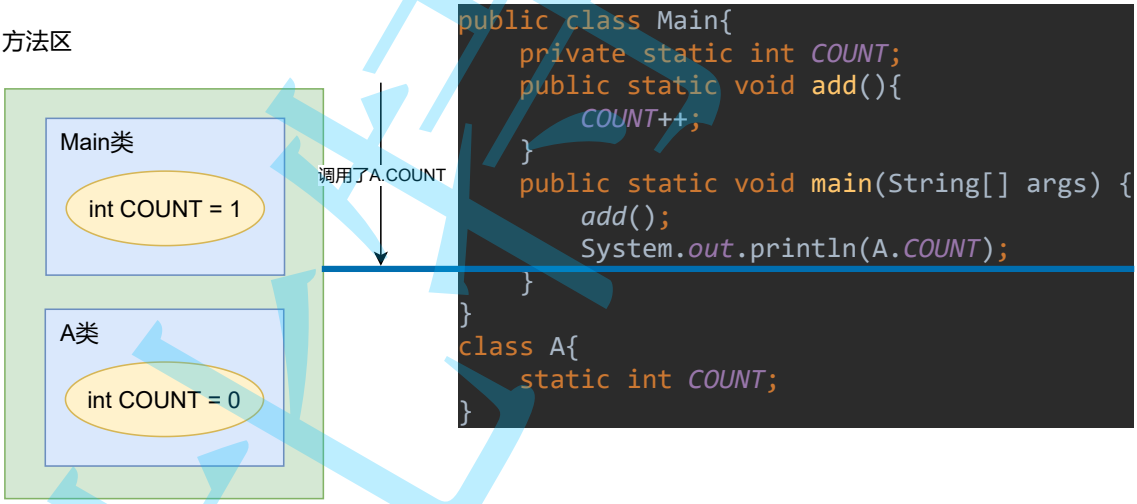
类和类变量

```
public class Main{
    private static int COUNT;
    public static void add(){
        COUNT++;
    }
    public static void main(String[] args) {
        add();
        System.out.println(A.COUNT);
    }
}
class A{
    static int COUNT;
}
```

内存布局为







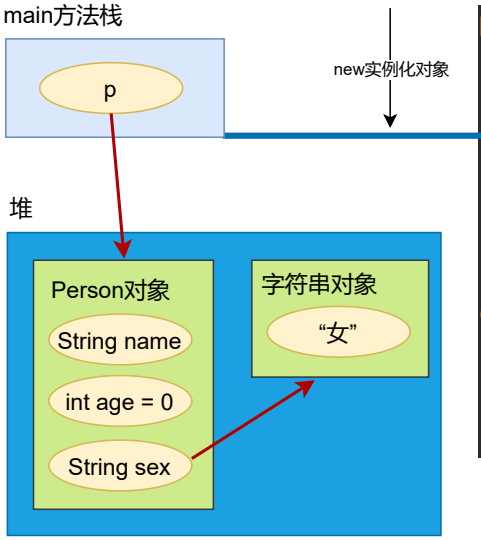
对于类和类变量来说，一定要加载到类以后才会加载到方法区，并会进行初始化赋值。比如这里没有执行到A.COUNT时，A类不会加载到方法区。

没有显示的赋值操作，对于基本类型的赋值就是基本类型的默认值，如这里的static int COUNT，会赋值为0。对于引用类型就是null，代表没有指向任何对象，如static String S，默认就是null。

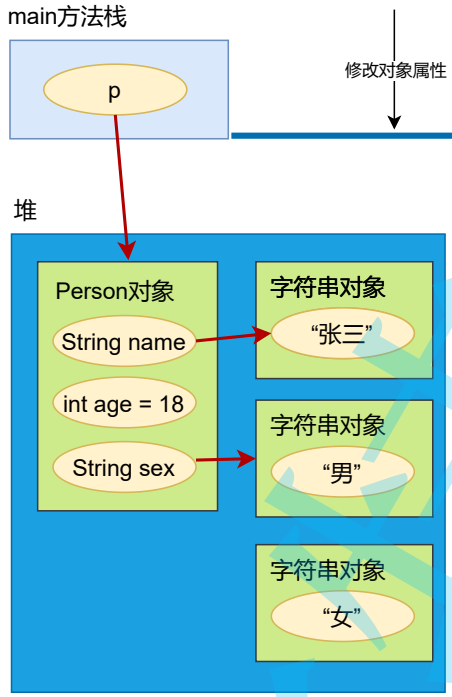
对象和成员变量

```
public class Main{

    public static void main(String[] args) {
        Person p = new Person();
        p.name = "张三";
        p.age = 18;
        p.sex = "男";
    }
}
class Person{
    String name;
    int age;
    String sex = "女";
}
```

内存布局为




```java
public class Main{

    public static void main(String[] args) {
        Person p = new Person();
        p.name = "张三";
        p.age = 18;
        p.sex = "男";
    }
}
class Person{
    String name;
    int age;
    String sex = "女";
}
```




```java
public class Main{

    public static void main(String[] args) {
        Person p = new Person();
        p.name = "张三";
        p.age = 18;
        p.sex = "男";
    }
}
class Person{
    String name;
    int age;
    String sex = "女";
}
```

对象要在成员变量初始化赋值完成，构造方法调用执行初始化操作以后，才算创建完成。

对象存储在堆里边，成员变量存储在堆里边的对象中。如果是成员变量是引用类型，会指向引用的对象。