

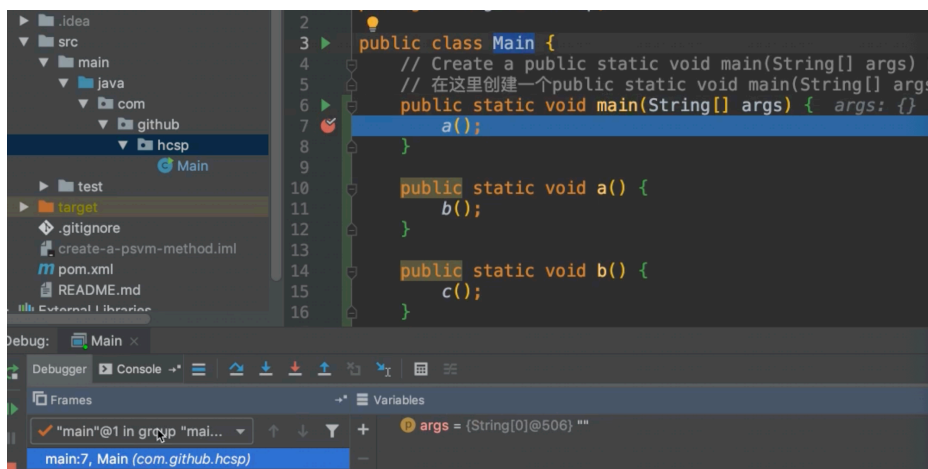


方法、静态方法 静态成员变量

▼ 设计类的本质是设计类的成员，类的成员分为

Field = 属性 = 成员变量

Method = 方法 = 成员方法



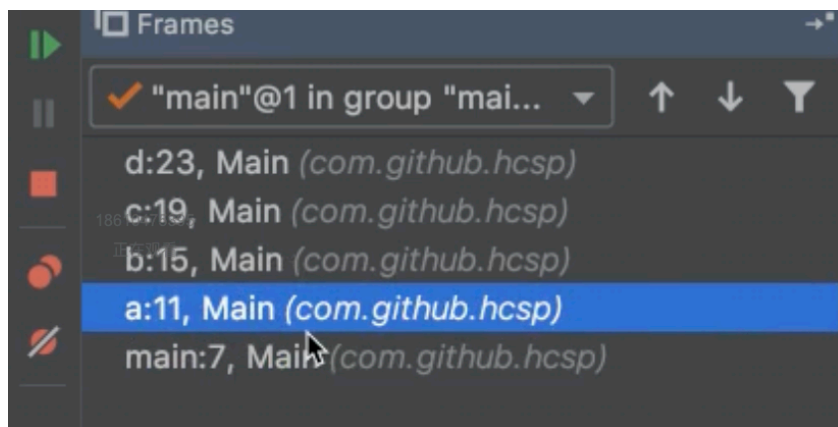
debug的流程：

main 📍 当前方法名

7 📍 当前的行数

Main 📍 当前的类名

(com.github.hcsp) 📍 当前的package名



这是调用栈，展示了：
main 方法调用了 a 方法
a 方法调用了 b 方法
b 方法调用了 c 方法
c 方法调用了 d 方法

✅ 函数 = 一次声明，多次调用

只需要**写一次函数的代码**，就可以在程序中**随时调用、重复使用**

```
public static int add(int a, int b) {  
    return a + b;  
}
```

调用几次，它就会执行几次：

```
add(1, 2); // 输出 3  
add(10, 20); // 输出 30
```

✅ 每次调用都是独立的一次执行过程 🙌 独立的栈帧（Frame）

每次调用：

- JVM 会在“**调用栈**”中创建一个**独立的栈帧（Frame）**
- 函数内部的参数和局部变量，都是**临时的、互不干扰的**

```
public static int square(int x) {  
    return x * x;  
}  
  
public static void main(String[] args) {  
    int a = square(2); // x=2, 返回4  
    int b = square(3); // x=3, 返回9  
}
```

- 第一次调用 `square(2)`，x 是 2，执行完就消失
- 第二次调用 `square(3)`，x 是 3，与上次没关系

✅ 每一次运行都会开辟一个新的临时空间

调用 `square(2)`：

x = 2	
return 4	

调用 `square(3)`：

x = 3	
return 9	

两次调用，是**完全独立的空间和流程**，互不影响

✅ 函数调用背后的栈帧（Stack Frame）与 JVM 调用栈

🧠 基础概念：

每次函数调用时，JVM 会在调用栈中创建一个“栈帧”（stack frame）

这个栈帧保存了该函数的参数、局部变量、返回地址等执行上下文

📦 什么是“栈帧”？

可以理解为：

每调用一个函数，JVM 就在“栈”上新建一层小盒子，叫**栈帧**，执行完就把这个盒子弹掉👉**出栈**

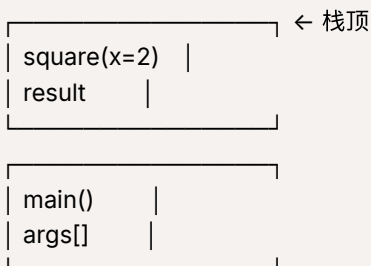
```
public static int square(int x) {  
    int result = x * x;  
    return result;  
}  
  
public static void main(String[] args) {  
    int a = square(2);  
    int b = square(3);  
}
```

📦 JVM 函数调用栈结构图（栈顶在上方）

◆ 1 初始：main 方法刚开始执行



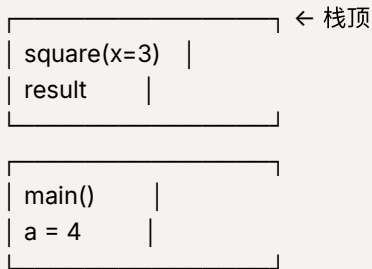
◆ 2 调用 square(2)



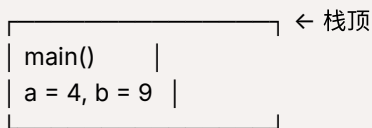
◆ 3 square(2) 执行完，返回 4，出栈



◆ 4 调用 square(3)



◆ 5 square(3) 返回 9，出栈，main 方法继续执行



- 栈的特性：**先进后出 (LIFO)**
- 栈顶总是当前正在运行的函数
- 每个函数调用 → 创建一个“栈帧”
- 栈帧包含：**参数、局部变量、返回地址**
- 函数执行完 → 栈帧出栈，释放内存
- 所以函数调用👉一次一次的独立空间

✓ Java 程序流程：

- 每个 .java 文件中通常只有一个 public 类，**类名要和文件名一样**

| public 类叫 Main，文件就要叫 Main.java


文件名：Person.java

```
public class Person {
```


```
String name;
void sayHello() { ... }
}
```

局部变量作用域

一个局部变量的作用域，是包围它的第一个 { } 块内部

也就是说，这个变量只能在它所定义的那对大括号  内部被访问和使用，出了这个块就“失效”了


作用域在方法内部

```
public static void test() {
    int x = 5;           // 局部变量 x 的作用域是 test 方法内
    System.out.println(x); //  可用
}
```

```
System.out.println(x);    //  报错：x 不在作用域内
```

变量在哪声明，就在哪生效，出了大括号就“死”掉

局部变量只能在函数内部使用，那如果我们有好几个函数想要共享一个变量怎么办？

 静态成员变量


静态成员变量 == 它是当前类的一个成员，它会持续存在，它独立于函数存在，它存在在jvm中某个区域，在当前类的任何地方都可以操作这个静态成员变量

“成员变量”

成员变量是：定义在类里的变量，但在方法之外

它们属于对象本身，每个对象都有一份

成员变量的定义与使用

```
public class Person {
    //  成员变量（属于对象）
    String name;
    int age;

    public void sayHello() {
        System.out.println("你好，我是 " + name + "，我 " + age + " 岁了");
    }
}
```

```

public static void main(String[] args) {
    Person p1 = new Person();
    p1.name = "小明";
    p1.age = 18;
    p1.sayHello(); // 输出：你好，我是 小明，我 18 岁了

    Person p2 = new Person();
    p2.name = "小红";
    p2.age = 20;
    p2.sayHello(); // 输出：你好，我是 小红，我 20 岁了
}
}

```

每个对象都有自己的成员变量，互不干扰

✓ 成员变量的特点

1. 定义在类中，但方法体外
2. 随着对象创建而初始化
3. 有默认值（比如 `int` 是 0，`boolean` 是 `false`）
4. 可以被所有方法使用（属于整个对象）

✓ Java 三类变量总览

类型	定义位置	所属对象	生命周期	访问方式	默认值	static
成员变量	类中、方法外	属于每个对象	对象创建时生成	对象.变量名	有	✗ 否
静态成员变量	类中、方法外 + <code>static</code>	属于类	类加载时生成	类名.变量名	有	✓ 是
局部变量	方法内部、参数、代码块内	属于方法	方法调用时生成	只能在方法中访问	✗ 无	✗ 否

```

public class Student {

    static int totalStudents = 0; // 静态变量（全班共用）

    String name;                // 成员变量（每个学生一份）

    public Student(String name) { // 构造器
        int id = 100;            // 局部变量（只在这个方法内有效）

        this.name = name;
        totalStudents++; // 静态变量累加
        System.out.println("创建学生：" + name + " ID: " + id);
    }
}

```

```

    }

    public void showInfo() {
        System.out.println("我是：" + name);
    }

    public static void showTotal() {
        System.out.println("总学生人数：" + totalStudents);
    }


    public static void main(String[] args) {
        Student s1 = new Student("小明");
        Student s2 = new Student("小红");

        s1.showInfo(); // 小明
        s2.showInfo(); // 小红

        Student.showTotal(); // 总学生人数：2
    }
}

```

Java 程序入口 `public static void main(String[] args)`

- 每个 Java 程序 **必须** 有一个类包含 `psvm`：
- 这是程序“开始运行”的地方
- JVM 会自动找到这个方法并运行它
-  一个 `.java` 文件中只能有一个 `public` 类
 - 这个类的名字要和文件名一致

```

public class Main {
    public static void main(String[] args) {
        // 程序入口
    }
}

```

可以在 `main` 方法中 创建其他类的对象，并调用它们的方法

```

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.name = "小明";
        p.sayHello();
    }
}

```

```
}  
}
```

✓ Java 程序运行流程：

1. 编译器从 `Main.java` 中的 `main` 方法开始运行
2. 在 `main` 可以：
 - 创建对象
 - 调用方法
 - 访问静态类/静态方法
3. 其他 `.java` 文件定义的类，可以在 `main` 中使用

✓ 静态成员变量 static

◆ 静态变量是什么？

- 用 `static` 修饰的变量，属于整个类，而不是对象
- 所有对象共享一份静态变量

◆ 静态变量特点：

- 被所有对象共享
- 即使没有对象也可以用：`类名.变量名`
- 生命周期从类加载开始到程序退出
- JVM 只为它分一次内存，节省资源

◆ 非静态变量（成员变量）：

- 每个对象有自己一份
- 每创建一个对象，就分配一份实例变量的内存
- 更加灵活，但占用多些内存

✓ 对象 & 构造器

◆ 类（class）：模板，定义属性 + 行为

◆ 对象（object）：通过类创建的具体个体

◆ 构造器（constructor）：

- 是一种特殊方法，用来“初始化”对象
- 构造器名称必须和类名相同
- 没有返回类型
- 构造器可以带参数，来给对象赋初值


```
public class Dog {
    String breed;
    int age;

    public Dog(String b, int a) {
        breed = b;
        age = a;
    }
}
```

✓ 创建类与对象的步骤

1. 每个类写在独立的 `.java` 文件中，比如 `Cat.java` 定义类 `Cat`
2. 在主程序入口 `Main.java` 中，创建 `Cat` 类的对象：

```
Cat cat = new Cat(); // 调用构造器
```

关键点：用 `new` 关键字 + 构造器 创建对象

◆ 没写构造器时，Java 会自动帮你加一个“无参构造器”

```
public Dog() {}
```

✓ `this` 是什么？

- `this` 表示“当前对象”
- 构造器中经常用 `this.变量名 = 参数名;` 来区分变量和参数

```
public class Cat {
    String name;
    public Cat(String name) {
        this.name = name;
        this.name 是成员变量
        name 是参数
    }
}
```

✓ 访问静态变量

◆ 静态变量可以在没有对象的情况下访问

```
public class Main {
    static int i = 0;
    public static void main(String[] args) {
```

```
        System.out.println(i); // 直接访问
    }
}
```

◆ 在别的类中访问其他类的静态变量

```
Main.i = 10; // 类名 + 点操作
```

✅ 类中有哪些内容

Java 类中可以包含：

- ✅ 静态变量（class-level）
- ✅ 静态方法（class-level）
- ✅ 成员变量（每个对象一份）
- ✅ 成员方法（针对对象行为）
- ✅ 构造器（创建对象时用）

static 是类的，非 static 是对象的

✅ 一、方法（Method）

语法格式：

```
[访问修饰符] 返回类型 方法名(参数...) {
    // 方法体
    return 返回值;
}
```

示例：

```
public class Person {
    String name;           成员变量

    public void sayHello() {  成员方法
        System.out.println("你好，我是 " + name);
    }
}
```

```
Person p = new Person();
p.name = "小明";
p.sayHello(); // 输出：你好，我是 小明
```

✅ 静态方法 (static method)

使用 static 修饰的方法，属于类本身，可以不创建对象就调用

注意：

- 静态方法不能访问非静态成员变量，因为它没有对象上下文
- 可以访问其他 static 成员

示例：

```
public class MathUtils {  
    public static int add(int a, int b) {  
        return a + b;  
    }  
}
```

调用：

```
int result = MathUtils.add(5, 3);  
System.out.println(result); // 输出 8
```

✅ 成员方法 vs 静态方法 vs 静态变量

```
public class Demo {  
    String name;        // 成员变量  
    static int count = 0; // 静态变量  
  
    public Demo(String name) {  
        this.name = name;  
        count++; // 每次创建新对象，计数+1  
    }  
  
    public void greet() { // 成员方法  
        System.out.println("你好，我是 " + name);  
    }  
  
    public static void showCount() { // 静态方法  
        System.out.println("当前对象总数：" + count);  
    }  
  
    public static void main(String[] args) {  
        Demo d1 = new Demo("Alice");  
        Demo d2 = new Demo("Bob");  
  
        d1.greet(); // 成员方法  
        d2.greet(); // 成员方法  
    }  
}
```

```

        Demo.showCount(); // 静态方法，通过类名调用
    }
}

```

项目	实例方法（普通方法）	静态方法	静态变量
属于	对象（实例）	类	类
是否需创建对象	是	否	否
可访问内容	成员变量、静态变量	只能访问静态变量	全部共享
典型用途	实例行为（如：走路）	工具函数（如：加法）	全局计数/常量等

▲ 注意：`static` 方法中不能用 `this`

```

public class Example {
    String name;
    static int count = 0;

    public static void printName() {
        System.out.println(this.name); ❌ 静态方法里不能用 this
    }
}

```

```

public class Person {
    String name = "张三";    // 成员变量
    static int totalCount = 0; // 静态变量

    // ✅ 成员方法（对象方法）
    public void sayHello() {
        System.out.println("你好，我是 " + name);
    }

    // ✅ 静态方法（类方法）
    public static void showTotal() {
        System.out.println("总人数是：" + totalCount);
    }

    public static void main(String[] args) {
        // ✅ 调用静态方法：不需要对象
        Person.showTotal();

        // ✅ 创建对象
        Person p1 = new Person();
        p1.name = "小明";
        p1.sayHello(); // 调用成员方法，必须通过对象

        // ✅ 静态变量也可通过对象访问（不推荐）
    }
}

```

```
        System.out.println(p1.totalCount); // 不推荐  
    }  
}
```