

千锋Java学院出品

# 方法/函数

Java Platform Standard Edition

# 课程目标

## CONTENTS

ITEMS **1** 方法的概念

ITEMS **2** 方法的定义

ITEMS **3** 方法的组成

ITEMS **4** 方法的调用

ITEMS **5** 方法的好处

ITEMS **6** 递归

- 控制台打印：《静夜思》

床前明月光，  
-----  
疑是地上霜。  
-----  
举头望明月，  
-----  
低头思故乡。  
-----

```
public class TestFunction {  
    public static void main(String[] args) {  
        System.out.println("床前明月光");  
        System.out.println("-----");  
        System.out.println("疑是地上霜");  
        System.out.println("-----");  
        System.out.println("举头望明月");  
        for (int i = 1; i <= 10 ; i++) {  
            System.out.print("-");  
        }  
        System.out.println();  
        System.out.println("低头思故乡");  
        for (int i = 1; i <= 10 ; i++) {  
            System.out.print("-");  
        }  
        System.out.println();  
    }  
}
```

无论是直接打印，  
亦或是循环打印，  
都无法避免冗余代码。

- 要求：以现有知识，至少使用两种方式打印以上效果。

# 方法的定义

- 概念：实现特定功能的一段代码，可反复使用。

- 定义语法：

```
public static void 方法名称() {  
    //方法主体  
}
```

遵循标识符命名规范

功能代码

- 经验：将需要在多个位置重复使用的一组代码，定义在方法内部。

# 定义的位置

- 方法定义在类的内部，与main方法并列。

```
//位置1
public class TestDefinitionFunction {
    //位置2
    public static void main(String[] args) {
        //位置3
    }
    //位置4
}
//位置5
```

- 正确位置：位置2、位置4

# 定义第一个方法

```
public class TestFunction {  
    public static void main(String[] args) {  
        System.out.println("床前明月光");  
        System.out.println("疑是地上霜");  
        System.out.println("举头望明月");  
        System.out.println("低头思故乡");  
    }  
  
    //定义：打印10个分割符的函数  
    public static void printSign(){  
        for (int i = 1; i <= 10 ; i++) {  
            System.out.print("-");  
        }  
        System.out.println();  
    }  
}
```

运行结果：

床前明月光  
疑是地上霜  
举头望明月  
低头思故乡

注意：

方法的定义并没有改变执行结果。  
如何使方法执行，并达到预期效果？

# 方法的调用

```
public class TestFunction {  
    public static void main(String[] args) {  
        System.out.println("床前明月光");  
        printSign();  
        System.out.println("疑是地上霜");  
        printSign();  
        System.out.println("举头望明月");  
        printSign();  
        System.out.println("低头思故乡");  
        printSign();  
    }  
  
    //定义：打印10个分割符的函数  
    public static void printSign(){  
        for (int i = 1; i <= 10 ; i++) {  
            System.out.print("-");  
        }  
        System.out.println();  
    }  
}
```

在需要执行方法代码的位置，  
通过方法名称进行调用。

运行结果：

```
床前明月光  
-----  
疑是地上霜  
-----  
举头望明月  
-----  
低头思故乡  
-----
```

注意：调用方法时，会优先执行方法内部代码，结束后，返回到方法调用处，继续向下执行。



# 方法的参数

- 多数情况下，方法与调用者之间需要数据的交互；调用者必须提供必要的数  
据，才能使方法完成相应的功能。

输入金额



投入硬币



- 调用方法时，所传入的数据被称为“参数”。



# 形参与实参

- 定义语法:

```
public static void 方法名称( 形式参数 ){  
    //方法主体  
}
```

经验:  
“形参” 等价于 “局部变量的声明”

- 调用语法:

```
方法名称( 实际参数 );
```

经验:  
“实参” 等价于 “局部变量的赋值”

作用: 方法的参数可以让代码功能更灵活、普适性更高, 易于修改及维护。

# 单个参数

```
public class TestFunction {  
    public static void main(String[] args) {  
        System.out.println("床前明月光");  
        printSign(10);  
        System.out.println("疑是地上霜");  
        System.out.println("举头望明月");  
        System.out.println("低头思故乡");  
    }  
  
    //定义：打印count个分割符的函数  
    public static void printSign(int count){  
        for (int i = 1; i <= count ; i++) {  
            System.out.print("-");  
        }  
        System.out.println();  
    }  
}
```

实际参数：10  
调用带参方法时，必须传入实际参数，  
为形式参数赋值。

形式参数：int count  
当方法被执行时，循环count次。

# 多个参数

```
public class TestFunction {  
    public static void main(String[] args) {  
        System.out.println("床前明月光");  
        printSign(10, '#');  
        System.out.println("疑是地上霜");  
        System.out.println("举头望明月");  
        System.out.println("低头思故乡");  
    }  
  
    //定义：打印count个sign的函数  
    public static void printSign(int count, char sign){  
        for (int i = 1; i <= count; i++) {  
            System.out.print(sign);  
        }  
        System.out.println();  
    }  
}
```

实参：10, #  
调用带参方法时，依次传入实参，  
类型、个数、顺序，必须与形参对应。

形参：int count, char sign  
当方法被执行时，打印count次sign。

# 如何定义参数

- 经验：根据具体的业务需求，来定义方法的参数。



```
public static boolean login(String username , String password){  
}
```



```
public static boolean login(String name , String pwd , String checkNo){  
}
```

# 返回值与返回值类型

- 概念：方法执行后的返回结果。
- 方法执行后，一些情况下无需返回结果；另一些情况下则必须返回结果。



例如：

存款操作无需返回结果；

取款操作必须返回结果。



# 返回值与返回值类型

- 定义语法:

```
public static 返回值类型 方法名称( 形式参数列表 ){  
    //方法主体  
    return value; //返回值  
}
```

规定返回值的具体类型（基本、引用、void）

根据需求返回一个结果（值）。

- 调用语法:

```
变量 = 方法名称( );
```

变量类型与返回值类型一致。



# 返回值与返回值类型

- 需求：定义方法，计算两个整数的和，并返回结果，在main中打印。

```
public class TestResultValue {  
    public static void main(String[] args) {  
        int result = add(5,6);  
        System.out.println(result);  
    }  
  
    public static int add(int a , int b){  
        return a + b;  
    }  
}
```

接收返回值：  
方法调用的表达式，  
最终即代表了所返回的结果。

返回值类型：  
定义时，即约定了返回的结果类型。

返回值：  
与返回值类型匹配的具体结果。  
在return关键字的后面追加具体值。

# return关键字

```
public class TestResultValue {  
    public static void main(String[] args) {  
        System.out.println("两值的和: "+calc(5,2));  
        System.out.println("两值的差: "+calc(5,2));  
    }  
    public static int calc(int a , int b){  
        return a + b;  
        return a - b;  
    }  
}
```

错误：一个方法只能有一个返回值。

# return关键字

```
public class TestResultValue {  
    public static void main(String[] args) {  
        String result = isEven(10);  
        System.out.println(result);  
    }  
  
    public static String isEven(int num){  
        if(num % 2 == 0){  
            return "偶数";  
        }else{  
            return "奇数";  
        }  
    }  
}
```

当有返回值的方法存在分支结构时，  
必须保证每一条分支都具有正确的返回值。

# return关键字

- return的两种用法：
  - 应用在具有返回值类型的方法中：
    - `return value;` //表示结束当前方法，并伴有返回值，返回到方法调用处。
  - 应用在没有返回值类型（void）的方法中：
    - `return;` //表示结束当前方法，直接返回到方法调用处。

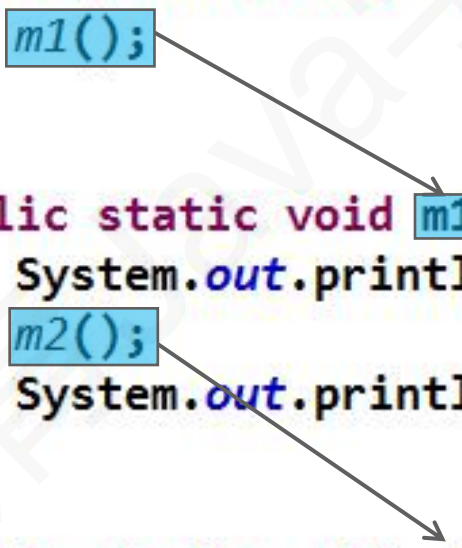
```
public static void show(){  
    for (int i = 1; i <= 100; i++) {  
        if(i == 50){  
            return;  
        }  
    }  
}
```

结束当前方法直接返回。

- 注意：一个类中可以定义多个方法，方法之间属于并列关系，不可嵌套。
- 经验：一个方法只做一件事。
- 好处：
  - 减少代码冗余。
  - 提高复用性。
  - 提高可读性。
  - 提高可维护性。
  - 方便分工合作。

# 多级调用

```
public class TestNestInvoke {  
    public static void main(String[] args) {  
        m1();  
    }  
  
    public static void m1(){  
        System.out.println("m1() - start");  
        m2();  
        System.out.println("m1() - end");  
    }  
  
    public static void m2(){  
        System.out.println("m2() - start");  
        System.out.println("m2() - end");  
    }  
}
```



运行结果:

m1() - start  
m2() - start  
m2() - end  
m1() - end

优先执行方法内部代码，  
结束后，返回到调用处，  
继续向下执行。



# 无穷递归

```
public class TestRecursionInvoke {  
  
    public static void main(String[] args) {  
        m1();  
    }  
  
    public static void m1(){  
        System.out.println("m1() - start");  
        m1();  
        System.out.println("m1() - end");  
    }  
}
```

当方法自己调用自己时，如果没有正确的出口条件，则产生无穷递归。

运行结果：

m1() - start  
m1() - start  
m1() - start  
m1() - start  
m1() - start  
m1() - start  
m1() - start  
m1() - start  
m1() - start  
m1() - start  
.....

Exception in thread "main" java.lang.StackOverflowError 内存溢出

- 什么是递归？

- 解决具有既定规律的问题时，在方法内部再次调用自身方法的一种编程方式。

- 何时使用递归？

- 当需要解决的问题可以拆分成若干个小问题，大小问题的解决方式相同，方法中自己调用自己。
- 使用循环解决的常规问题，都可以替换为递归解决。

- 如何正确使用递归？

- 设置有效的出口条件，可以让调用链上的每个方法都可以正确返回，避免无穷递归。

- 计算5的阶乘:  $5! = 5 * 4 * 3 * 2 * 1$ ;

```
public class TestFactorial {  
    public static void main(String[] args) {  
        System.out.println(factorial(5));  
    }  
  
    public static int factorial(int n){  
        int sum = 1;  
        for (int i = 2; i <= n; i++) {  
            sum *= i;  
        }  
        return sum;  
    }  
}
```

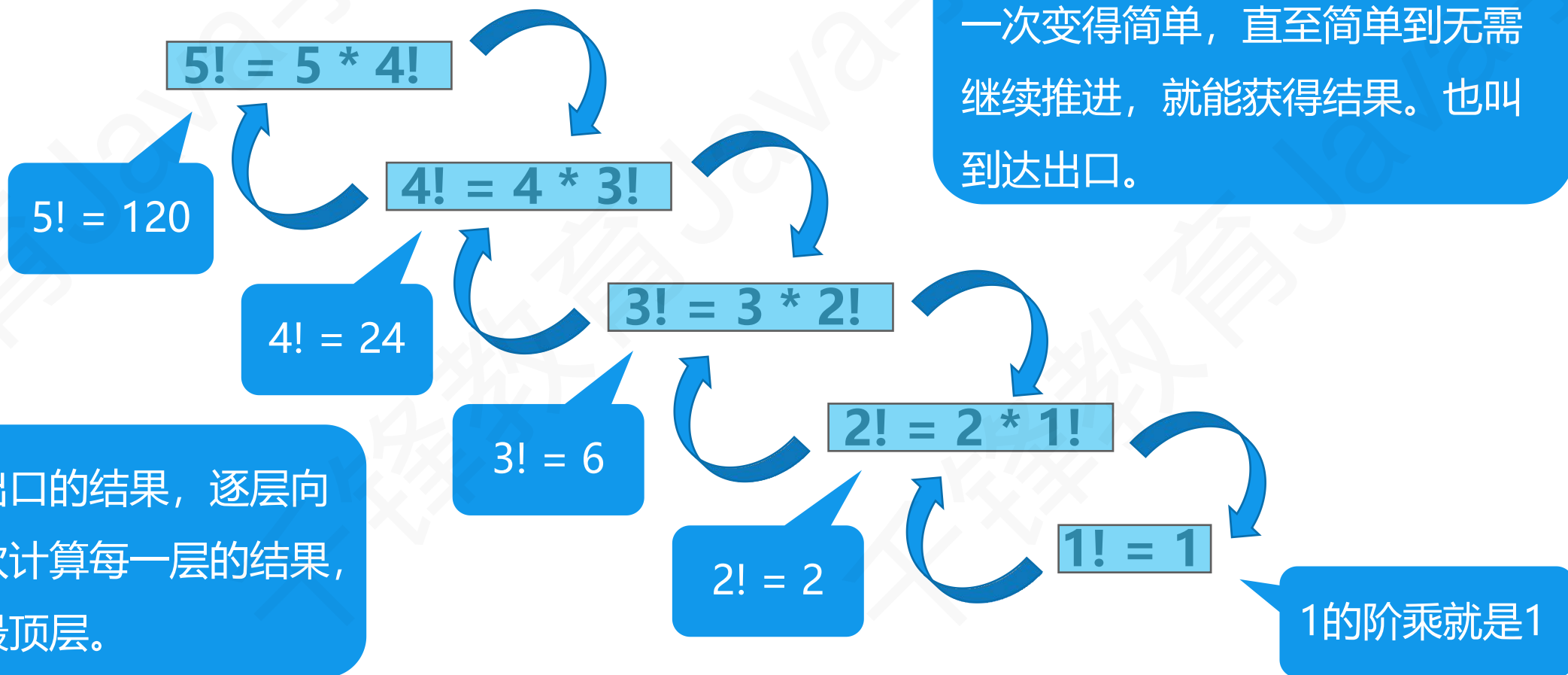
循环计算阶乘较为简单，  
依次与每个值相乘即可。

# 递归阶乘

- 阶乘的定义： $n! = n * (n-1) * (n-2) * (n-3) \dots$

**递进**：每一次推进，计算都比上一次变得简单，直至简单到无需继续推进，就能获得结果。也叫到达出口。

**回归**：基于出口的结果，逐层向上回归，依次计算每一层的结果，直至回归到最顶层。



# 递归阶乘

```
public class TestFactorial {  
    public static void main(String[] args) {  
        System.out.println(getFive(5));  
    }  
    public static int getFive(int n) { //n=5  
        return n * getFour(n-1); //5 * 4!  
    }  
    public static int getFour(int n) { //n=4  
        return n * getThree(n-1); //4 * 3!  
    }  
    public static int getThree(int n) { //n=3  
        return n * getTwo(n-1); //3 * 2!  
    }  
    public static int getTwo(int n) { //n=2  
        return n * getOne(n-1); //2 * 1!  
    }  
    public static int getOne(int n) { //n=1  
        return 1; //1! = 1  
    }  
}
```

多个方法解决问题的思路相同，  
同时遵循着相同的规律。  
如何整合成为一个方法？

# 递归阶乘

```
public class TestFactorial {  
    public static void main(String[] args) {  
        System.out.println(factorial(5));  
    }  
  
    public static int factorial(int n){  
        return n * factorial(n-1);  
    }  
}
```

当 $n=1$ 时，并没有计算出 $1!=1$ 而是继续递进，计算 $1!=1*0!$ ，所以导致永远没有出口条件，则造成无穷递归，内存溢出！

Exception in thread "main" java.lang.StackOverflowError 内存溢出



# 递归阶乘

```
public class TestFactorial {  
    public static void main(String[] args) {  
        System.out.println(factorial(5));  
    }  
  
    public static int factorial(int n){  
        if(n == 1){  
            return 1;  
        }  
        return n * factorial(n-1);  
    }  
}
```

增加出口条件,  $n = 1$  时, 无需计算阶乘, 直接返回结果1。

- 注意: 所有能以递归解决的问题, 循环都可以解决。当解决复杂问题时, 递归的实现方式更为简单。

- 使用递归完成“斐波那契数列”。

- 0 1 1 2 3 5 8 13 21 34 55 .....

- 方法的概念：

- 实现特定功能的一段代码，可反复使用。

- 方法的组成：

```
public static 返回值类型 方法名(形参列表...){  
    方法主体  
    [return 返回值]  
}
```

- 方法的调用：

- 方法名(实参列表...)

- 方法的好处：

- 减少冗余、提高复用性、可读性、可维护性、方便分工合作。

- 递归：

- 将大问题拆分成若干个小问题，大小问题的解决方法相同，有固定规律，需设置有效的出口条件，方法中自己调用自己。

# THANK YOU



做真实的自己，用良心做教育