方法的使用

本节目标

1. 掌握理解方法的使用方法

1. 方法的基本用法

1.1 什么是方法(method)

方法就是一个代码片段. 类似于 C 语言中的 "函数".

方法存在的意义(不要背, 重在体会):

1. 是能够模块化的组织代码(当代码规模比较复杂的时候).
2. 做到代码被重复使用, 一份代码可以在多个位置使用.
3. 让代码更好理解更简单.
4. 直接调用现有方法开发, 不必重复造轮子.

回忆一个之前写过的代码: 计算 1! + 2! + 3! + 4! + 5!

```
int sum = 0;
for (int i = 1; i <= 5; i++) {
    int tmp = 1;
    for (int j = 1; j <= i; j++) {
        tmp *= j;
    }
    sum += tmp;
}
System.out.println("sum = " + sum);
```

这个代码中使用双重循环, 比较容易写错.

接下来我们可以使用方法来优化这个代码.

1.2 方法定义语法

基本语法

```
// 方法定义
public static 方法返回值 方法名称 ([参数类型 形参 ...]){
    方法体代码;
    [return 返回值];
}

// 方法调用
返回值变量 = 方法名称(实参...);
```

代码示例: 实现一个方法实现两个整数相加

```
class Test {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;

        // 方法的调用
        int ret = add(a, b);
        System.out.println("ret = " + ret);
    }

    // 方法的定义
    public static int add(int x, int y) {
        return x + y;
    }
}

// 执行结果
ret = 30
```

注意事项

1. public 和 static 两个关键字在此处具有特定含义, 我们暂时不讨论, 后面会详细介绍.
2. 方法定义时, 参数可以没有. 每个参数要指定类型
3. 方法定义时, 返回值也可以没有, 如果没有返回值, 则返回值类型应写成 void
4. 方法定义时的参数称为 "形参", 方法调用时的参数称为 "实参".
5. 方法的定义必须在类之中, 代码书写在调用位置的上方或者下方均可.
6. Java 中没有 "函数声明" 这样的概念.

1.3 方法调用的执行过程

基本规则

- 定义方法的时候, 不会执行方法的代码. 只有调用的时候才会执行.
- 当方法被调用的时候, 会将实参赋值给形参.
- 参数传递完毕后, 就会执行到方法体代码.
- 当方法执行完毕之后(遇到 return 语句), 就执行完毕, 回到方法调用位置继续往下执行.

- 一个方法可以被多次调用.

代码示例1 计算两个整数相加

```
class Test {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        System.out.println("第一次调用方法之前");
        int ret = add(a, b);
        System.out.println("第一次调用方法之后");
        System.out.println("ret = " + ret);

        System.out.println("第二次调用方法之前");
        ret = add(30, 50);
        System.out.println("第二次调用方法之后");
        System.out.println("ret = " + ret);
    }

    public static int add(int x, int y) {
        System.out.println("调用方法中 x = " + x + " y = " + y);
        return x + y;
    }
}

// 执行结果
一次调用方法之前
调用方法中 x = 10 y = 20
第一次调用方法之后
ret = 30
第二次调用方法之前
调用方法中 x = 30 y = 50
第二次调用方法之后
ret = 80
```

代码示例: 计算 1! + 2! + 3! + 4! + 5!

```
class Test {
    public static void main(String[] args) {
        int sum = 0;
        for (int i = 1; i <= 5; i++) {
            sum += factor(i);
        }
        System.out.println("sum = " + sum);
    }

    public static int factor(int n) {
        System.out.println("计算 n 的阶乘中! n = " + n);
        int result = 1;
```

```
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }
}

// 执行结果
计算 n 的阶乘中! n = 1
计算 n 的阶乘中! n = 2
计算 n 的阶乘中! n = 3
计算 n 的阶乘中! n = 4
计算 n 的阶乘中! n = 5
sum = 153
```

使用方法, 避免使用二重循环, 让代码更简单清晰.

1.4 实参和形参的关系(重要)

代码示例: 交换两个整型变量

```
class Test {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        swap(a, b);
        System.out.println("a = " + a + " b = " + b);
    }

    public static void swap(int x, int y) {
        int tmp = x;
        x = y;
        y = tmp;
    }
}

// 运行结果
a = 10 b = 20
```

原因分析

刚才的代码, 没有完成数据的交换.

对于**基础类型**来说, 形参相当于实参的拷贝. 即 **传值调用**

```
int a = 10;
int b = 20;

int x = a;
int y = b;

int tmp = x;
x = y;
y = tmp;
```

可以看到, 对 x 和 y 的修改, 不影响 a 和 b.

解决办法: 传引用类型参数 (例如数组来解决这个问题)

> 这个代码的运行过程, 后面学习数组的时候再详细解释.

```
class Test {
    public static void main(String[] args) {
        int[] arr = {10, 20};
        swap(arr);
        System.out.println("a = " + arr[0] + " b = " + arr[1]);
    }

    public static void swap(int[] arr) {
        int tmp = arr[0];
        arr[0] = arr[1];
        arr[1] = tmp;
    }
}

// 运行结果
a = 20 b = 10
```

1.5 没有返回值的方法

方法的返回值是可选的. 有些时候可以没有的.

代码示例

```
class Test {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        print(a, b);
    }

    public static void print(int x, int y) {
        System.out.println("x = " + x + " y = " + y);
    }
}
```

另外, 如刚才的交换两个整数的方法, 就是没有返回值的.

2. 方法的重载

有些时候我们需要用一个函数同时兼容多种参数的情况, 我们就可以使用到方法重载.

2.1 重载要解决的问题

代码示例

```
class Test {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int ret = add(a, b);
        System.out.println("ret = " + ret);

        double a2 = 10.5;
        double b2 = 20.5;
        double ret2 = add(a2, b2);
        System.out.println("ret2 = " + ret2);
    }

    public static int add(int x, int y) {
        return x + y;
    }
}

// 编译出错
Test.java:13: 错误: 不兼容的类型: 从double转换到int可能会有损失
                double ret2 = add(a2, b2);
                                  ^
```

由于参数类型不匹配, 所以不能直接使用现有的 add 方法.

那么是不是应该创建这样的代码呢?

代码示例

```
class Test {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int ret = addInt(a, b);
        System.out.println("ret = " + ret);

        double a2 = 10.5;
        double b2 = 20.5;
        double ret2 = addDouble(a2, b2);
        System.out.println("ret2 = " + ret2);
    }

    public static int addInt(int x, int y) {
        return x + y;
    }

    public static double addDouble(double x, double y) {
        return x + y;
    }
}
```

这样的写法是对的(例如 Go 语言就是这么做的), 但是 Java 认为 `addInt` 这样的名字不友好, 不如直接就叫 `add`

2.2 使用重载

代码示例

```
class Test {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int ret = add(a, b);
        System.out.println("ret = " + ret);

        double a2 = 10.5;
        double b2 = 20.5;
        double ret2 = add(a2, b2);
        System.out.println("ret2 = " + ret2);

        double a3 = 10.5;
        double b3 = 10.5;
        double c3 = 20.5;
        double ret3 = add(a3, b3, c3);
        System.out.println("ret3 = " + ret3);
    }
```

```
    public static int add(int x, int y) {
        return x + y;
    }

    public static double add(double x, double y) {
        return x + y;
    }

    public static double add(double x, double y, double z) {
        return x + y + z;
    }
}
```

方法的名字都叫 add. 但是有的 add 是计算 int 相加, 有的是 double 相加; 有的计算两个数字相加, 有的是计算三个数字相加.

同一个方法名字, 提供不同版本的实现, 称为 **方法重载**

2.3 重载的规则

针对同一个类:

- 方法名相同
- 方法的参数不同(参数个数或者参数类型)
- 方法的返回值类型不影响重载.

代码示例

```
class Test {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int ret = add(a, b);
        System.out.println("ret = " + ret);
    }

    public static int add(int x, int y) {
        return x + y;
    }

    public static double add(int x, int y) {
        return x + y;
    }
}

// 编译出错
Test.java:13: 错误: 已在类 Test中定义了方法 add(int,int)
        public static double add(int x, int y) {
                            ^
1 个错误
```

当两个方法的名字相同, 参数也相同, 但是返回值不同的时候, 不构成重载.

3. 方法递归

3.1 递归的概念

一个方法在执行过程中调用自身, 就称为 "递归".

递归相当于数学上的 "数学归纳法", 有一个起始条件, 然后有一个递推公式.

> 例如, 我们求 N!
>
> 起始条件: N = 1 的时候, N! 为 1. 这个起始条件相当于递归的结束条件.
>
> 递归公式: 求 N! , 直接不好求, 可以把问题转换成 N! => N * (N-1)!

代码示例: 递归求 N 的阶乘

```
public static void main(String[] args) {
    int n = 5;
    int ret = factor(n);
    System.out.println("ret = " + ret);
}

public static int factor(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factor(n - 1); // factor 调用函数自身
}

// 执行结果
ret = 120
```

3.2 递归执行过程分析

递归的程序的执行过程不太容易理解, 要想理解清楚递归, 必须先理解清楚 "方法的执行过程", 尤其是 "方法执行结束之后, 回到调用位置继续往下执行".

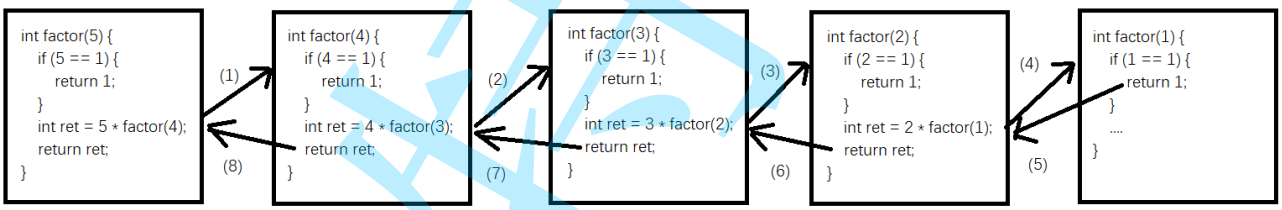
代码示例: 递归求 N 的阶乘, 加上日志版本

```
public static void main(String[] args) {
    int n = 5;
    int ret = factor(n);
    System.out.println("ret = " + ret);
}
```

```java
public static int factor(int n) {
    System.out.println("函数开始, n = " + n);
    if (n == 1) {
        System.out.println("函数结束, n = 1 ret = 1");
        return 1;
    }
    int ret = n * factor(n - 1);
    System.out.println("函数结束, n = " + n + " ret = " + ret);
    return ret;
}

// 执行结果
函数开始, n = 5
函数开始, n = 4
函数开始, n = 3
函数开始, n = 2
函数开始, n = 1
函数结束, n = 1 ret = 1
函数结束, n = 2 ret = 2
函数结束, n = 3 ret = 6
函数结束, n = 4 ret = 24
函数结束, n = 5 ret = 120
ret = 120
```

执行过程图



程序按照序号中标识的 (1) -> (8) 的顺序执行.

> 关于 "调用栈"
>
> 方法调用的时候, 会有一个 "栈" 这样的内存空间描述当前的调用关系. 称为调用栈.
>
> 每一次的方法调用就称为一个 "栈帧", 每个栈帧中包含了这次调用的参数是哪些, 返回到哪里继续执行等信息.
>
> 后面我们借助 IDEA 很容易看到调用栈的内容.

3.3 递归练习

代码示例1 按顺序打印一个数字的每一位(例如 1234 打印出 1 2 3 4)

```
public static void print(int num) {
    if (num > 9) {
        print(num / 10);
    }
    System.out.println(num % 10);
}
```

代码示例2 递归求 1 + 2 + 3 + ... + 10

```
public static int sum(int num) {
    if (num == 1) {
        return 1;
    }
    return num + sum(num - 1);
}
```

代码示例3 写一个递归方法，输入一个非负整数，返回组成它的数字之和. 例如，输入 1729, 则应该返回1+7+2+9，它的和是19

```
public static int sum(int num) {
    if (num < 10) {
        return num;
    }
    return num % 10 + sum(num / 10);
}
```

代码示例4 求斐波那契数列的第 N 项

斐波那契数列介绍: https://baike.sogou.com/v267267.htm?fromTitle=%E6%96%90%E6%B3%A2%E9%82%A3%E5%A5%91%E6%95%B0%E5%88%97

```
public static int fib(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    return fib(n - 1) + fib(n - 2);
}
```

当我们求 fib(40) 的时候发现, 程序执行速度极慢. 原因是进行了大量的重复运算.

```
class Test {
    public static int count = 0;    // 这个是类的成员变量. 后面会详细介绍到.

    public static void main(String[] args) {
```

```
        System.out.println(fib(40));
        System.out.println(count);
    }

    public static int fib(int n) {
        if (n == 1 || n == 2) {
            return 1;
        }
        if (n == 3) {
            count++;
        }
        return fib(n - 1) + fib(n - 2);
    }
}

// 执行结果
102334155
39088169    // fib(3) 重复执行了 3 千万次.
```

可以使用循环的方式来求斐波那契数列问题, 避免出现冗余运算.

```
public static int fib(int n) {
    int last2 = 1;
    int last1 = 1;
    int cur = 0;
    for (int i = 3; i <= n; i++) {
        cur = last1 + last2;
        last2 = last1;
        last1 = cur;
    }
    return cur;
}
```

此时程序的执行效率大大提高了.

3.4 递归小结

递归是一种重要的编程解决问题的方式.

有些问题天然就是使用递归方式定义的(例如斐波那契数列, 二叉树等), 此时使用递归来解就很容易.

有些问题使用递归和使用非递归(循环)都可以解决. 那么此时更推荐使用循环, 相比于递归, 非递归程序更加高效.

作业

1. 理解方法的基本用法, 方法重载, 方法递归, 并写一篇博客总结.
2. 实现代码: 递归求 N 的阶乘

3. 实现代码: 递归求 1 + 2 + 3 + ... + 10
4. 实现代码: 按顺序打印一个数字的每一位(例如 1234 打印出 1 2 3 4)
5. 实现代码: 写一个递归方法，输入一个非负整数，返回组成它的数字之和.
6. 实现代码: 求斐波那契数列的第 N 项
7. 实现代码: 求解汉诺塔问题(提示, 使用递归)

> 汉诺塔问题是一个经典的问题。汉诺塔（Hanoi Tower），又称河内塔，源于印度一个古老传说。
>
> 大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着64片黄金圆盘。
>
> 大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。
>
> 并且规定，任何时候，在小圆盘上都不能放大圆盘，且在三根柱子之间一次只能移动一个圆盘。
>
> 问应该如何操作?

8. 实现代码: 青蛙跳台阶问题(提示, 使用递归)

> 一只青蛙一次可以跳上 1 级台阶，也可以跳上2 级。求该青蛙跳上一个n 级的台阶总共有多少种跳法