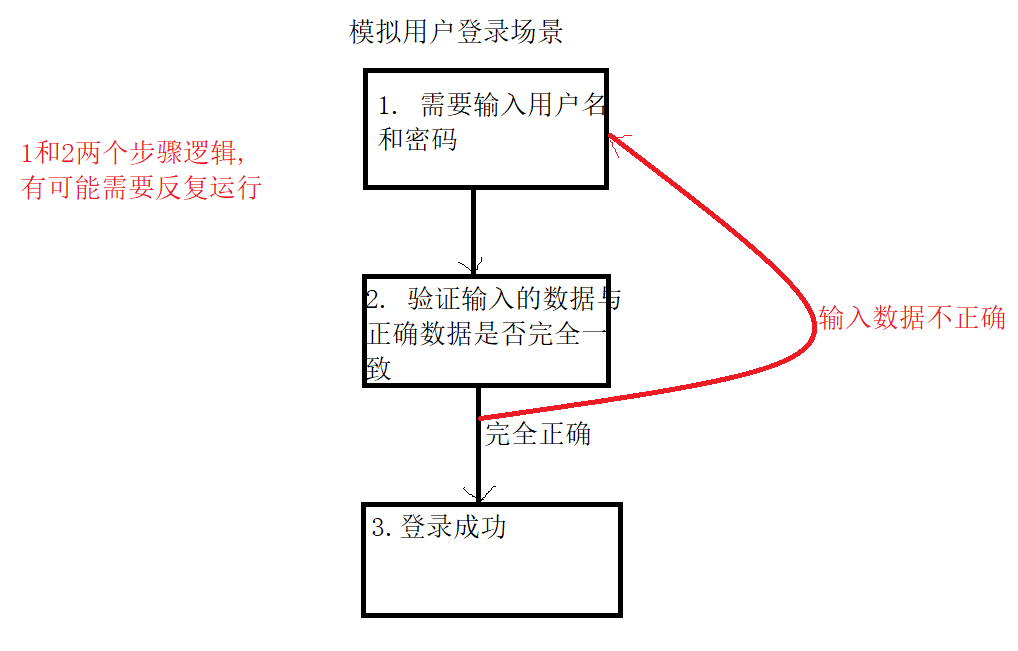
# day04

## 循环结构

1. 如果代码中有需要反复运行的相同的或者类似的功能逻辑, 可以使用循环语句实现



1. 主要实现循环语法

for循环

while循环

### for循环语句

1. for循环语法格式:

for( 初始化语句; 循环判断条件; 初始化语句变化 ){

需要反复多次执行的代码逻辑; // 循环体

}

1. 解释:
2. for : 关键字, 表示循环的概念
3. 初始化语句 : 定义出一个变量, 变量用于表示循环的起点或者用于记录循环的次数
4. 循环判断条件 : boolean类型表达式, 可以决定循环何时结束
5. 初始化语句变化 : 从循环的起点每次循环结束之后,向着循环的终点(循环结束)方向进行数据变化(自增最常见初始化变量语句变化方式)
6. for循环执行过程:
7. 先执行初始化语句, 并且初始化语句只执行一次
8. 执行循环判断条件, 如果判断条件为false, 那么循环结束
9. 如果循环判断条件为true, 那么执行循环体
10. 执行初始化语句变化
11. 回到第二点

需求1 : 打印1-5所有数字

需求2 : 打印5-1所有数字

需求3 : 打印1-100累加和

需求1代码

|  |
| --- |
| package com.ujiuye.fordemo;  public class Demo01\_For循环语法结构 {  public static void main(String[] args) {  // 需求 : 打印1-5所有数字  /\*System.out.println(1);  System.out.println(2);  System.out.println(3);  System.out.println(4);  System.out.println(5);\*/    for(int i = 1; i <= 5; i++) {  // 打印数据  System.out.println(i);  }  }  } |

需求2代码

|  |
| --- |
| package com.ujiuye.fordemo;  public class Demo02\_For循环练习1 {  public static void main(String[] args) {  // 需求2 : 打印5-1所有数字  for(int i = 5; i >= 1; i--) {  // 打印数字  System.out.println(i);  }  }  } |

需求3代码

|  |
| --- |
| package com.ujiuye.fordemo;  public class Demo03\_For循环练习2 {  public static void main(String[] args) {  // 需求3 : 打印1-100累加和  /\*  \* 分析 :  \* 1) 需要获取到1-100中的每一个数字(设计出一个1-100数字循环)  \* 2) 将获取到的每一个整数在循环中进行累加  \*  \* \*/  // sum变量用于表示1-100所有数据的累加和结果  // 在for循环小括号中定义的变量 i , 只能在for循环的大括号中使用, 出了for循环大括号变量超出了作用范围  int sum = 0;  for(int i = 1; i <= 100; i++) {  // 循环的i值就是1-100的每一个整数  sum = sum + i;// sum += i;  }  System.out.println(sum);  // System.out.println(i);  }  } |

### while循环语句

1. while语法结构:

while(boolean类型判断条件){

需要反复多次运行代码逻辑;// 循环体

}

可以通过自己设计, 转换成与for循环相似处理

int i = 1;// 初始化语句设定

while(boolean类型判断条件){

需要反复多次运行代码逻辑;// 循环体

初始化语句变化;

}

1. 运行过程:
2. while : 关键字, 表示循环概念
3. 只要while循环小括号中的boolean类型表达式结果为true,那么循环体就可以执行; 如果boolean类型表达式结果为false, 那么while循环结束

需求1 : 打印1-5所有整数

需求2 : 打印出1-100中所有可以被7整除的数字

需求1代码

|  |
| --- |
| package com.ujiuye.fordemo;  public class Demo04\_While循环语法结构 {  public static void main(String[] args) {  // 需求1 : 打印1-5所有整数  for(int i = 1; i <= 5; i++) {  // 打印数据  System.out.println("for---"+i);  }    int j = 1;  while(j <= 5) {  System.out.println("while---"+j);  // 千万不要忘记做初始化语句j的自增动作  j++;  }  System.out.println(j);// 6  }  } |

需求2代码:

|  |
| --- |
| package com.ujiuye.fordemo;  public class Demo05\_While循环练习 {  public static void main(String[] args) {  // 需求2 : 打印出1-100中所有可以被7整除数字  /\*  \* 分析 :  \* 1) 获取到1-100中的每一个整数, 才能知道这个数能不能被7整除(设计1-100循环)  \* 2) 在循环中, 获取到每一个数, 使用逻辑验证 % 7 == 0 ,那么输出  \*/    int i = 1;  while(i <= 100) {  if(i % 7 == 0) {  System.out.println(i);  }  // 千万不要忘记做初始化语句i的自增动作  i++;  }    System.out.println("--------------");    for(int w = 1; w <= 100; w++) {  if(w % 7 == 0) {  System.out.println(w);  }  }  }  } |

### 死循环

1. 死循环 : 表示循环次数无限, 循环不结束; 只要保证循环的判断条件一直为true, 那么这个循环就可以一直执行, 不结束, 代码中尽量不要出现死循环
2. for循环和while循环设计出死循环代码

for( ; ; ){ // for循环的判断条件不写, 默认为true

}

while(true){//设计方式在实际开发中常用,使用于循环次数未知,与循环跳出语句配合使用

}

注意 : 死循环之后代码会报错, 因为代码永远不会执行到

代码

|  |
| --- |
| package com.ujiuye.fordemo;  public class Demo06\_死循环 {  public static void main(String[] args) {  /\*int i = 0;  while(true) {  System.out.println(i);  i++;  }\*/    for(; ;) {  System.out.println("abc");  }    // System.out.println(123); 死循环之后代码因为永远无法执行, 因此系统报错提示  }  } |

### 循环跳转语句

1. 循环跳转语句 : 使用在循环中, 可以对循环做控制, 或结束循环的语句, 称为跳转语句, 主要由两个关键字实现
2. continue : 关键字, 表示继续,下一次的概念, 用于表示结束本次循环,继续执行下一次循环

break : 关键字, 表示破坏, 结束; 用于循环中, 当循环执行到break,那么循环马上结束(可以用于避免死循环问题)

注意 : 上述两个关键字在for循环和while循环中效果一致

continue需求 : 打印出1-100中所有偶数数据

break需求 : 模拟用户登录界面, 正确用户名为 : 123 , 正确的密码为: 456 , 需要客户通过键盘录入登录的用户名和密码, 如果输入数据正确, 显示登录成功; 如果输入数据有误, 可以支持无限次输入用户名和密码, 直到输入正确的数据为止

continue需求代码

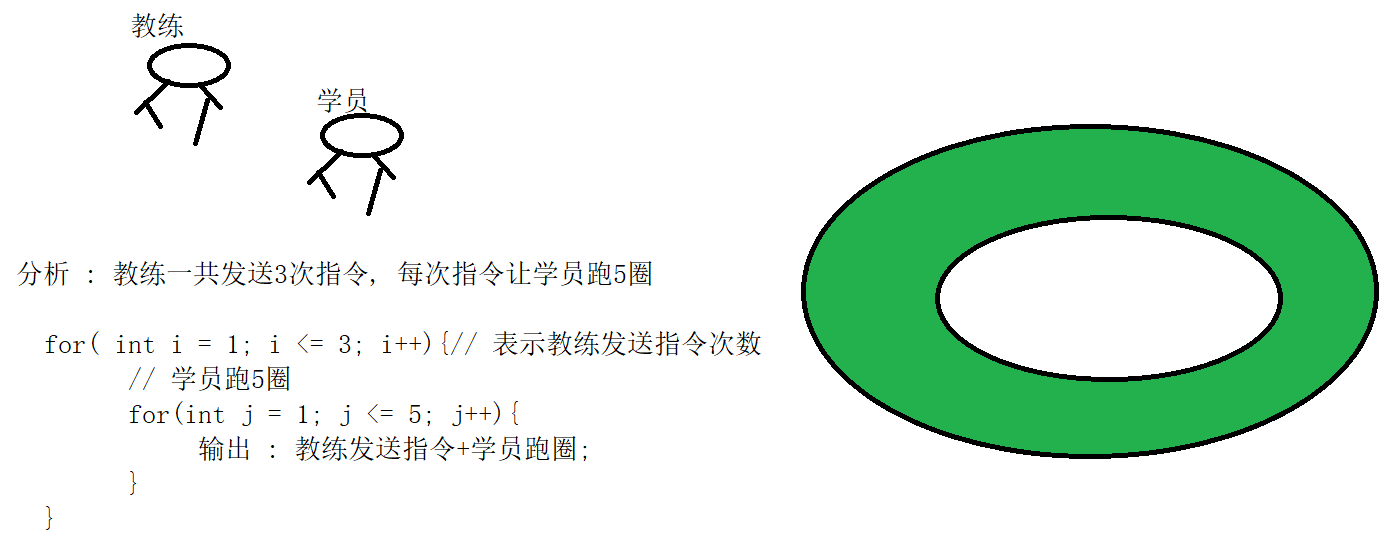
|  |
| --- |
| package com.ujiuye.fordemo;  public class Demo07\_Continue {  public static void main(String[] args) {  // continue需求 : 打印出1-100中所有偶数数据  for(int i = 1; i <= 100; i++) {  if(i % 2 != 0) {  // 奇数数据不需要打印, 本次循环可以结束  continue;  }  System.out.println(i);  }  }  } |

break需求代码

|  |
| --- |
| package com.ujiuye.fordemo;  import java.util.Scanner;  public class Demo08\_Break {  public static void main(String[] args) {  // 1.定义出正确用户名  int name = 123;  // 2.定义出正确密码  int password = 456;  Scanner sc = new Scanner(System.in);    while(true) {  // inName表示通过键盘录入的用户名  System.out.println("请输入用户名:");  int inName = sc.nextInt();  // inPassword表示通过键盘录入的密码  System.out.println("请输入密码");  int inPassword = sc.nextInt();    if(inName == name && inPassword == password) {  System.out.println("登录成功!");  // break结束循环  break;  }else {  System.out.println("输入数据有误,请重新登录");  }  }  }  } |

### 嵌套循环

1. 嵌套循环 : 一个循环, 循环体恰巧又是另外一个循环



1. 嵌套循环执行过程 : 外层循环执行一次, 内层循环完全执行完毕; 外层循环判断条件为false, 嵌套循环结束
2. 嵌套循环中, for循环嵌套for循环是最常见的嵌套方式

for(初始化语句1 ; 判断条件1; 初始化语句1 变化 ){// 外层循环

for(初始化语句2 ; 判断条件2; 初始化语句2 变化 ){// 内层循环

循环体;

}

}

代码

|  |
| --- |
| package com.ujiuye.fordemo;  public class Demo09\_嵌套循环 {  public static void main(String[] args) {  for(int i = 1; i <= 3; i++){// 表示教练发送指令次数  // 学员跑5圈  for(int j = 1; j <= 5; j++){  System.out.println("教练第" + i + "次发出指令, 目前学员跑到了第" + j + "圈");  }  }  }  } |

需求1 : 打印出如下图形

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

代码

|  |
| --- |
| package com.ujiuye.fordemo;  public class Demo10\_嵌套循环矩形星星案例 {  /\*需求1 : 打印出如下图形  \*\*\*\*\*  \*\*\*\*\*  \*\*\*\*\*  \*\*\*\*\*  \*  \* 提示 : 不要把一行星星看做一个整体, 每一个 \* 看成一个独立的个体  \* 类比于 : 教练发送4次指令(对应一共4行数据打印), 每一次指令中, 都跑5圈(每行有5个星星)  \*  \*\*/  public static void main(String[] args) {  /\*  \* 第一种实现方式  \*  \* for(int i = 1; i <= 4; i++) {// i值表示循环次数  System.out.println("\*\*\*\*\*");  }\*/    // 第二种实现方式 : 嵌套循环  for( int i = 1; i <= 4; i++) {// 外层循环用于设计星星的打印行数  for(int j = 1; j <= 5; j++) {// 内层循环用于反复打印出星星  // println 表示输出数据, 数据后添加一次回车换行  // print 只表示输出数据  System.out.print("\*");  }  // 添加一次回车换行, 换到第二行  System.out.println();  }  }  } |

需求2 :打印出如下图形

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

代码

|  |
| --- |
| package com.ujiuye.fordemo;  public class Demo11\_嵌套循环直角星星案例 {  /\* 需求2 :打印出如下图形  \*  \*\*  \*\*\*  \*\*\*\*  \*\*\*\*\*  \*  \* 分析 :  \* 第一行, 打印1个星  \* 第二行, 打印2个星  \* 第三行, 打印3个星  \* 第四行, 打印4个星  \* 第五行, 打印5个星  \*  \* 规律 : 每一行打印的星星数量, 与当前的行数一致  \*\*/  public static void main(String[] args) {  for(int i = 1; i <= 5; i++) {// 外层循环表示目前打印的行数  for(int j = 1; j <= i; j++) {  System.out.print("\*");  }  System.out.println();  }  }  } |

### break关键字在嵌套循环中使用

1. break关键字在嵌套循环中使用, 结束的是内层循环

break关键字每次只能结束一层循环, 结束的就是包含break关键字本身, 离break最近的一层循环

需求 : 教练一共发送三次指令, 每次要求学员跑5圈; 实际上学员每次指令只跑3圈就回去接收下一次指令, 将教练预期跑圈和学员实际跑圈过程实现出来

代码

|  |
| --- |
| package com.ujiuye.fordemo;  public class Demo12\_Break关键字在嵌套循环中使用 {  public static void main(String[] args) {  /\*需求 : 教练一共发送三次指令, 每次要求学员跑5圈;  实际上学员每次指令只跑3圈回去接收下一次指令,  将教练预期跑圈和学员实际跑圈过程实现出来\*/    for(int i = 1; i <= 6; i++) {  for(int j = 1; j <= 5; j++) {  System.out.println("教练第" + i + "次发出指令, 目前学员跑到了第" + j + "圈");  if(j == 3) {  break;  }  }  }  }  } |

## 方法

1. 方法 : 表示具有指定功能的代码段, 用于解决开发中的需求, 需要, 或者常见问题
2. 方法封装 : 如果有代码逻辑需要多次使用, 那么可以将这部分代码使用一对大括号包裹起来, 给大括号起个名字, 以后通过这个名字表示这段代码功能
3. 方法封装好处:
4. 提高代码的复用性(复用就表示反复使用)
5. 提高代码封装性(封装概念, 也可以提高代码安全性)
6. 简化对于代码设计难度

### 方法的定义

1. 方法定义的语法结构:

修饰符 返回值类型 方法名(参数列表){

方法体;

return 语句;

}

1. 说明:
2. 修饰符 : 目前与main方法修饰符保持一致即可 public static
3. 返回值类型 : 当运行方法完毕之后, 思考, 是否需要将方法的运行结果返回给方法的调用者, 如果需要反馈结果, 将方法运行结果所属数据类型作为方法返回值类型; 如果不需要反馈结果, 使用关键字void表示, void表示空, 表示方法没有任何返回值类型
4. 方法名 : 符合标识符规范, 小驼峰原则, 从第二个英文单词开始首字母大写
5. 参数列表 : 运行方法功能时, 是否需要外界提供数据资源, 如果需要, 那么将需要的数据个数和数据类型以参数列表形式罗列, 数据类型 变量名, 数据类型 变量名...
6. 方法体 : 实现方法功能的代码逻辑
7. return语句 : return 关键字, 表示返回

a : 如果方法有返回值类型, 必须使用return关键字将方法的运行结果返回到方法调用者处

b : 如果方法没有返回值类型, 可以不写return, 也可以写成return;

注意 : 方法功能彼此独立, 必须单独定义, 不能嵌套

需求1 : 定义出一个方法功能, 求任何两个整数的和

需求2 : 定义出一个方法功能, 比较两个浮点类型数据是否相等

需求3 : 定义出一个方法功能, 打印1-n整数(1-n任意整数打印, n为正整数)

代码

|  |
| --- |
| package com.ujiuye.function;  public class Demo01\_方法定义和使用 {  public static void main(String[] args) {  }  // 需求1 : 定义出一个方法功能, 求任何两个整数的和  //This method must return a result of type int  public static int getSum(int x, int y) {  int w = x + y;  return w;  }    // 需求2 : 定义出一个方法功能, 比较两个浮点类型数据是否相等  public static boolean equal(double x, double y) {  boolean boo = x == y;  return boo;  }    // 需求3 : 定义出一个方法功能, 打印1-n整数(1-n任意整数打印, n为正整数)  public static void print(int n) {  for(int i = 1; i <= n; i++) {  System.out.println(i);  }  // return;  }  } |

### 方法的调用

1. 方法需要调用才能运行, 方法不会自动运行
2. 方法调用有3种形式:
3. 直接调用 : 使用于方法没有返回值类型(有返回值类型方法也可以直接调用), 方法修饰符与main方法修饰符一致, 且与main方法定义在同一个类中, 方法名(实际参数);
4. 输出调用 : 适用于方法有返回值类型, 可以在标准输出语句中进行方法调用, 效果是直接将方法的结果进行输出, System.out.println(方法名(实际参数));
5. 赋值调用 : 适用于方法有返回值类型,调用方法之后, 可以将方法返回值给某一个变量进行赋值, 在代码中, 可以继续使用变量(最常用, 最灵活和优秀方法调用方式)

代码

|  |
| --- |
| package com.ujiuye.function;  public class Demo01\_方法定义和使用 {  public static void main(String[] args) {  // 调用方法(使用方法): 需要传递实际参数值  // 1. 直接调用, print方法没有返回值类型  print(6);  // 2. 输出调用  System.out.println(getSum(3,5));// 8  // 3. 赋值调用  boolean boo = equal(3.14,3.14);  System.out.println(boo);// true  }  /\*修饰符 返回值类型 方法名(参数列表){  方法体;  return 语句;  }\*/  // 需求1 : 定义出一个方法功能, 求任何两个整数的和  //This method must return a result of type int  public static int getSum(int x, int y) {  int w = x + y;  return w;  }    // 需求2 : 定义出一个方法功能, 比较两个浮点类型数据是否相等  public static boolean equal(double x, double y) {  boolean boo = x == y;  return boo;  }    // 需求3 : 定义出一个方法功能, 打印1-n整数(1-n任意整数打印, n为正整数)  public static void print(int n) {  for(int i = 1; i <= n; i++) {  System.out.println(i);  }  // return;  }  } |

### 方法定义注意事项

1. 方法必须单独进行定义, 不能嵌套, 方法定义没有前后顺序, 没有优先级别, 都是平级
2. 参数列表:

a : 定义方法时参数列表称为形式参数(形参), 形式参数设计方式 数据类型 变量名 , 数据类型 变量名...

b : 方法调用时参数列表称为实际参数(实参), 方法调用时实际参数个数和类型上需要与形式参数保持一致

1. return 语句:

a :return 关键字, 表示返回, 也表示方法结束, 方法执行到return关键字, 表示方法运行完了

b : 如果方法有返回值类型, 必须return, 并且return的结果必须与方法返回值类型一致

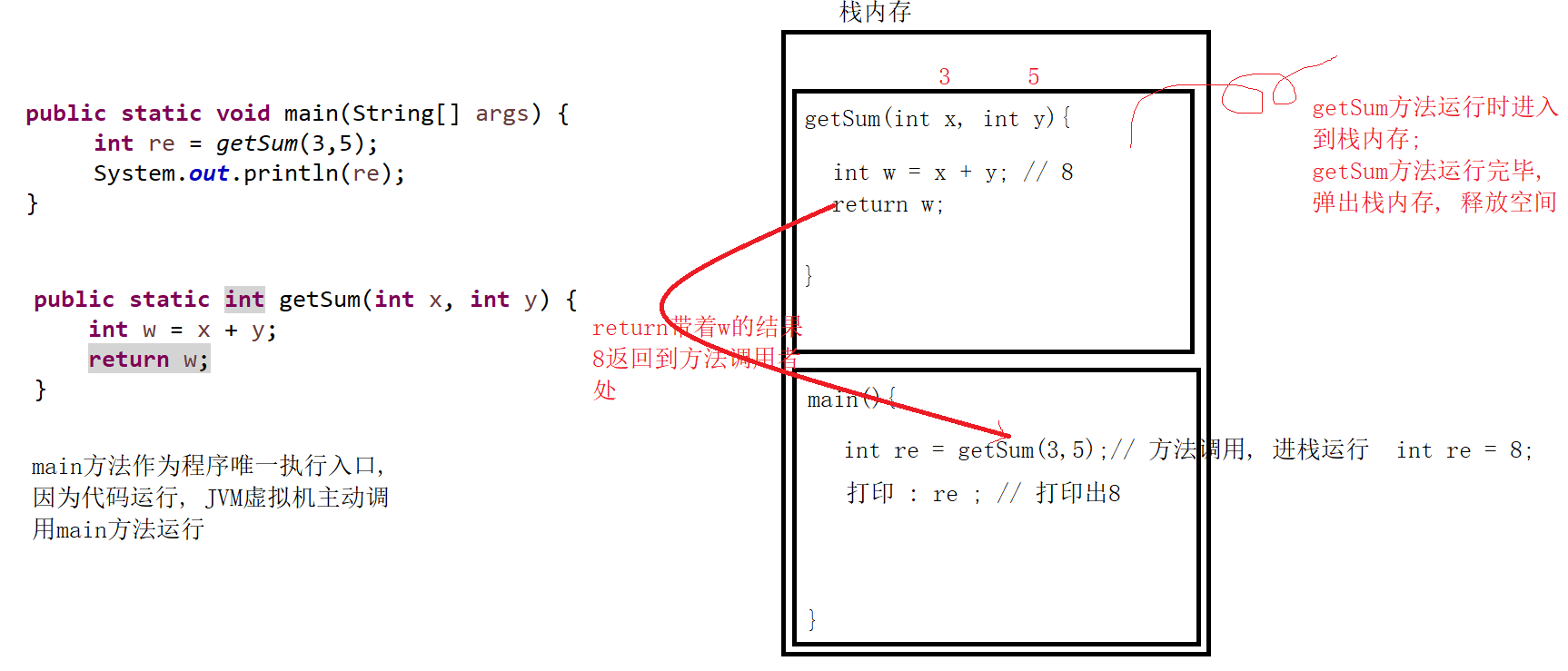
c : 如果方法没有返回值类型, 可以没有return , 可以写成return;

d : return将结果返回到方法调用者处

### 方法在内存中理解

Java代码在JVM虚拟机中运行, JVM虚拟机就是运行Java代码的容器, 而JVM运行时,JVM本身就需要在内存中开辟空间, JVM将其占有的内存空间划分成5块空间区域, 其中一块空间, 称为栈内存

栈内存 : 代码中的方法运行, 需要在栈内存中开辟空间运行; 方法调用时进入栈内存中开辟空间, 方法运行完毕, 出栈内存, 占有空间释放(方法弹栈死亡)



### 方法重载

1. 重载 : Overload(英文单词,表示超载)

场景 : 客户要求定义出几个方法:

1. 要求做两个任意整数求和
2. 要求做三个整数求和
3. 要求做2个double浮点类型求和

问题 : 在功能上类似的1 , 2 , 3方法, 客户认为方法名称太多, 记不住, 要求让类似功能具有相同方法名, 通过参数列表不同区分多个方法, 可以实现, 称为方法重载

1. 方法重载:

定义在同一个类中, 方法名称相同, 方法参数列表不同, 与方法返回值类型无关, 如此多个方法之间称为方法重载

参数列表不同体现:

a : 参数个数不同

b : 参数数据类型不同

c : 不同类型参数之间的排列顺序不同

代码

|  |
| --- |
| package com.ujiuye.function;  public class Demo02\_方法重载 {  public static void main(String[] args) {  System.out.println(getSum(3,5));  System.out.println(getSum(3,5,6));  System.out.println(getSum(3.5,5.0));  }    // 1)要求做两个任意整数求和  public static int getSum(int x, int y) {  return x + y;  }    // 2) 要求做三个整数求和  public static int getSum(int x, int y ,int z) {  return x + y + z;  }    // 3) 要求做2个double浮点类型求和  public static double getSum(double x, double y) {  return x + y;  }    public static double get(int x, double y) {  return x + y;  }    public static double get(double x, int y) {  return x + y;  }  } |