



# Java

极夜酱

# 目录

<b>1</b>	<b>Hello World!</b>	<b>1</b>
1.1	Hello World! . . . . .	1
1.2	数据类型 . . . . .	5
1.3	输入输出 . . . . .	8
1.4	表达式 . . . . .	11
<b>2</b>	<b>分支</b>	<b>15</b>
2.1	逻辑运算符 . . . . .	15
2.2	if . . . . .	17
2.3	switch . . . . .	20
<b>3</b>	<b>循环</b>	<b>22</b>
3.1	while . . . . .	22
3.2	for . . . . .	27
3.3	break or continue? . . . . .	32
<b>4</b>	<b>数组</b>	<b>35</b>
4.1	数组 . . . . .	35
4.2	字符串 . . . . .	42
4.3	字符串方法 . . . . .	49
<b>5</b>	<b>函数</b>	<b>54</b>
5.1	函数 . . . . .	54
5.2	作用域 . . . . .	59
5.3	递归 . . . . .	61
<b>6</b>	<b>面向对象</b>	<b>68</b>
6.1	封装 . . . . .	68
6.2	构造方法 . . . . .	73
6.3	继承 . . . . .	78

6.4	抽象类 . . . . .	85
6.5	多态 . . . . .	87
6.6	接口 . . . . .	90
<b>7</b>	<b>异常</b>	<b>92</b>
7.1	异常 . . . . .	92
7.2	自定义异常 . . . . .	97





图 1.1: 计算  $\sum_{i=1}^{100} i$  的流程图

### 1.1.2 Hello World!

Hello World 是学习编程的第一个程序，它的作用是向屏幕输出"Hello World!"。

**Hello World!**

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Hello World!");
4     }
5 }
```

### 运行结果

Hello World!

`public class HelloWorld` 是一个公共类，类名需要大写，与文件名相同。

`main()` 是程序的入口，程序运行后会首先执行 `main()` 中的代码。`System.out.println()` 的功能是在屏幕上输出一个字符串（string）。最后的分号用于表示一条语句的结束，注意不要使用中文的分号。

不同编程语言的 Hello World 写法大同小异，可以看出编程语言的基本结构是相似的。

### C

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello World!\n");
5     return 0;
6 }
```

### C++

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
6     cout << "Hello World!" << endl;
7     return 0;
8 }
```

### Python

```
1 print("Hello World!")
```

### 1.1.3 注释 (Comment)

注释就是对代码的解释和说明，它并不会程序所执行。注释能提高程序的可读性，让人更加容易了解代码的功能。

注释一般分为单行注释和多行注释：

1. 单行注释：以//开头，该行之后的内容视为注释。
2. 多行注释：以/\* 开头，\*/结束，中间的内容视为注释。

#### 注释

```
1  /*
2  * Author: Terry
3  * Date: 2022/11/16
4  */
5
6  public class Comment {
7      public static void main(String[] args) {
8          System.out.println("Hello World!"); // display "Hello World!"
9      }
10 }
```

## 1.2 数据类型

### 1.2.1 数据类型 (Data Types)

在计算机中，每个数据一般都有一个对应的类型，基础数据类型包括：

#### 1. 整型

- 字节型 byte
- 短整型 short
- 整型 int
- 长整型 long

#### 2. 浮点型

- 单精度浮点数 float
- 双精度浮点数 double

#### 3. 字符型 char

#### 4. 布尔型 boolean

不同的数据类型所占的内存空间大小不同，因此所能表示的数值范围也不同。

数据类型	大小	取值范围
byte	1 字节	$-2^7 \sim 2^7 - 1$
short	2 字节	$-2^{15} \sim 2^{15} - 1$
int	4 字节	$-2^{31} \sim 2^{31} - 1$
long	4 字节	$-2^{63} \sim 2^{63} - 1$
float	4 字节	7 位有效数字
double	8 字节	15 位有效数字
char	2 字节	$-2^{15} \sim 2^{15} - 1$
boolean	1 字节	{true, false}



### 1.2.2 变量 (Variable)

变量是用来存储数据的内存空间，每个变量都有一个类型，变量中只能存储对应类型的数据。

```
1 int num = 10;
2 double salary = 8232.56;
```

变量的命名需要符合规范：

1. 由字母、数字和下划线组成，不能以数字开头
2. 不可以使用编程语言中预留的关键字
3. 使用英语单词，顾名思义

关键字是编程语言内置的一些名称，具有特殊的用处和意义，因此不应该作为变量名，防止产生歧义。

abstract	do	implements	protected	throws
boolean	double	import	public	transient
break	else	instanceof	return	true
byte	extends	int	short	try
case	false	interface	static	void
catch	final	long	strict	volatile
char	finally	native	super	while
class				

表 1.1: 关键字

### 1.2.3 常量 (Constant)

变量的值在程序运行过程中可以修改，但有一些数据的值是固定的，为了防止这些数据被随意改动，可以将这些数据定义为常量。

在数据类型前加上 final 关键字，即可定义常量，常量一般使用大写表示。如果在程序中尝试修改常量，将会报错。

### 常量

```
1 public class Constant {  
2     public static void main(String[] args) {  
3         final double PI = 3.14159;  
4         PI = 4;  
5     }  
6 }
```

### 运行结果

The final local variable PI cannot be assigned.

## 1.3 输入输出

### 1.3.1 println()

System.out.println() 的功能是向屏幕输出指定格式的文本，但是有些需要输出的字符在编程语言中具有特殊含义，因此这些特殊的字符，需要经过转义后输出。

转义字符	描述
\\	反斜杠 \
\'	单引号 '
\"	双引号 "
\n	换行
\t	制表符

表 1.2: 转义字符

#### 转义字符

```
1 public class EscapeCharacter {  
2     public static void main(String[] args) {  
3         System.out.println("\"Hello\nWorld\"");  
4     }  
5 }
```

#### 运行结果

```
"Hello  
World"
```

在对变量的值进行输出时，可以使用 + 进行连接多个部分。

#### 长方形面积

```

1 public class Rectangle {
2     public static void main(String[] args) {
3         int length = 10;
4         int width = 5;
5         double area = length * width;
6         System.out.println(
7             "Area = " + length + " * " + width + " = " + area
8         );
9     }
10 }

```

#### 运行结果

Area = 10 \* 5 = 50.0

### 1.3.2 Scanner

有时候一些数据需要从键盘输入，Scanner 类可以读取对应类型的数据，并赋值给相应的变量。使用 Scanner 类需要导入 java.util.Scanner，并创建 Scanner 类的对象，然后调用对应方法读取数据。

在读取数据前，通常会使用 print() 先输出一句提示信息，告诉用户需要输入什么数据。

在对变量的值进行格式化输出时，可以在 printf() 中使用对应类型的占位符。

数据类型	占位符
int	%d
float	%f
double	%f
char	%c

表 1.3: 占位符

## 圆面积

```
1 import java.util.Scanner;
2
3 public class Circle {
4     public static void main(String[] args) {
5         final double PI = 3.14159;
6
7         Scanner scanner = new Scanner(System.in);
8
9         System.out.print("Radius: ");
10        double r = scanner.nextDouble();
11
12        double area = PI * Math.pow(r, 2);
13        System.out.printf("Area = %.2f\n", area);
14
15        scanner.close();
16    }
17 }
```

### 运行结果

Radius: 5

Area = 78.54

Math 库中定义了一些常用的数学函数，例如 `pow(x, y)` 可用于计算  $x$  的  $y$  次方。

## 1.4 表达式

### 1.4.1 算术运算符

大部分编程语言中的除法与数学中的除法意义不同。

当相除的两个数都为整数时，那么就会进行整除运算，因此结果仍为整数，例如  $21 / 4 = 5$ 。

如果相除的两个数中至少有一个为浮点数时，那么就会进行普通的除法运算，结果为浮点数，例如  $21.0 / 4 = 5.25$ 。

取模（modulo）运算符% 用于计算两个整数相除之后的余数，例如  $22 \% 3 = 1$ 、 $4 \% 7 = 4$ 。

#### 逆序三位数

```
1 import java.util.Scanner;
2
3 public class Reverse {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Enter a 3-digit integer: ");
8         int num = scanner.nextInt();
9
10        int a = num / 100;
11        int b = num / 10 % 10;
12        int c = num % 10;
13
14        System.out.println("Reversed: " + (c * 100 + b * 10 + a));
15
16        scanner.close();
17    }
18 }
```

### 运行结果

Enter a 3-digit integer: 520

Reversed: 25

## 1.4.2 复合运算符

使用复合运算符可以使表达式更加简洁。例如  $a = a + b$  可以写成  $a += b$ ,  $--$ 、 $*=$ 、 $/=$ 、 $\%=$  等复合运算符的使用方式同理。

当需要给一个变量的值加/减 1 时,除了可以使用  $a += 1$  或  $a -= 1$  之外,还可以使用  $++$  或  $--$  运算符,但是  $++$  和  $--$  可以出现在变量之前或之后:

表达式	含义
$a++$	执行完所在语句后自增 1
$++a$	在执行所在语句前自增 1
$a--$	执行完所在语句后自减 1
$--a$	在执行所在语句前自减 1

表 1.4: 自增/自减运算符

### 自增/自减运算符

```
1 public class Operator {
2     public static void main(String[] args) {
3         int n = 10;
4
5         System.out.println(n++);
6         System.out.println(++n);
7         System.out.println(n--);
8         System.out.println(--n);
```

```
9     }  
10 }
```

#### 运行结果

```
10  
12  
12  
10
```

### 1.4.3 隐式类型转换

在计算机计算的过程中，只有类型相同的数据才可以进行运算。例如整数 + 整数、浮点数/浮点数等。

但是很多时候，我们仍然可以对不同类型的数据进行运算，而并不会产生错误，例如整数 + 浮点数。这是由于编译器会自动进行类型转换。在整数 + 浮点数的例子中，编译器会将整数转换为浮点数，这样就可以进行运算了。

编译器选择将整数转换为浮点数，而不是将浮点数转换为整数的原因在于，浮点数相比整数能够表示的范围更大。例如整数 8 可以使用 8.0 表示，而浮点数 9.28 变为整数 9 后就会丢失精度。

隐式类型转换最常见的情形就是除法运算，这也是导致整数/整数 = 整数、整数/浮点数 = 浮点数的原因。

### 1.4.4 显式类型转换

有些时候编译器无法自动进行类型转换，这时就需要我们手动地强制类型转换。

#### 显式类型转换



```
1 public class Casting {  
2     public static void main(String[] args) {  
3         int total = 821;  
4         int num = 10;  
5         double average = (double)total / num;  
6         System.out.printf("Average = %.2f\n", average);  
7     }  
8 }
```

#### 运行结果

Average = 82.10

# Chapter 2 分支

## 2.1 逻辑运算符

### 2.1.1 关系运算符

编程中经常需要使用关系运算符来比较两个数据的大小，比较的结果是一个布尔值 (boolean)，即 True (非 0) 或 False (0)。

在编程中需要注意，一个等号 = 表示赋值运算，而两个等号 == 表示比较运算。

数学符号	关系运算符
<	<
>	>
≤	<=
≥	>=
=	==
≠	!=

### 2.1.2 逻辑运算符

逻辑运算符用于连接多个关系表达式，其结果也是一个布尔值。

1. 逻辑与 &&：当多个条件全部为 True，结果为 True。

条件 1	条件 2	条件 1 && 条件 2
T	T	T
T	F	F
F	T	F
F	F	F

2. 逻辑或 ||: 多个条件至少有一个为 True 时, 结果为 True。

条件 1	条件 2	条件 1    条件 2
T	T	T
T	F	T
F	T	T
F	F	F

3. 逻辑非!: 条件为 True 时, 结果为 False; 条件为 False 时, 结果为 True。

条件	! 条件
T	F
F	T

## 2.2 if

### 2.2.1 if

if 语句用于判断一个条件是否成立，如果成立则进入语句块，否则不执行。

年龄

```
1 import java.util.Scanner;
2
3 public class Age {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Enter your age: ");
8         int age = scanner.nextInt();
9
10        if (age > 0 && age < 18) {
11            System.out.println("Minor");
12        }
13
14        scanner.close();
15    }
16 }
```

运行结果

```
Enter your age: 17
Minor
```

### 2.2.2 if-else

if-else 的结构与 if 类似，只是在 if 语句块中的条件不成立时，执行 else 语句块中的语句。

## 闰年

```
1 import java.util.Scanner;
2
3 public class Leap {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Enter a year: ");
8         int year = scanner.nextInt();
9
10        /*
11         * A year is a leap year if it is
12         * 1. exactly divisible by 4, and not divisible by 100;
13         * 2. or is exactly divisible by 400
14         */
15        if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0) {
16            System.out.println("Leap year");
17        } else {
18            System.out.println("Common year");
19        }
20
21        scanner.close();
22    }
23 }
```

### 运行结果

Enter a year: 2020

Leap year

### 2.2.3 if-else if-else

当需要对更多的条件进行判断时，可以使用 if-else if-else 语句。

#### 字符

```
1 import java.util.Scanner;
2
3 public class Character {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Enter a character: ");
8         char c = scanner.next().charAt(0);
9
10        if (c >= 'a' && c <= 'z') {
11            System.out.println("Lowercase");
12        } else if (c >= 'A' && c <= 'Z') {
13            System.out.println("Uppercase");
14        } else if (c >= '0' && c <= '9') {
15            System.out.println("Digit");
16        } else {
17            System.out.println("Special character");
18        }
19        scanner.close();
20    }
21 }
```

#### 运行结果

```
Enter a character: T
Uppercase
```

## 2.3 switch

### 2.3.1 switch

switch 结构用于根据一个整数值，选择对应的 case 执行。需要注意的是，当对应的 case 中的代码被执行完后，并不会像 if 语句一样跳出 switch 结构，而是会继续向后执行，直到遇到 break。

#### 计算器

```
1 import java.util.Scanner;
2
3 public class Calculator {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Enter an expression: ");
8         int num1 = scanner.nextInt();
9         char operator = scanner.next().charAt(0);
10        int num2 = scanner.nextInt();
11        scanner.close();
12
13        switch (operator) {
14            case '+':
15                System.out.printf(
16                    "%d + %d = %d\n", num1, num2, num1 + num2
17                );
18                break;
19            case '-':
20                System.out.printf(
21                    "%d - %d = %d\n", num1, num2, num1 - num2
22                );
23                break;
24            case '*':
25                System.out.printf(
26                    "%d * %d = %d\n", num1, num2, num1 * num2
```

```
27         );
28         break;
29     case '/':
30         System.out.printf(
31             "%d / %d = %d\n", num1, num2, num1 / num2
32         );
33         break;
34     default:
35         System.out.println("Error! Operator is not supported");
36         break;
37     }
38 }
39 }
```

#### 运行结果

Enter an expression: 5 \* 8

5 \* 8 = 40



# Chapter 3 循环

## 3.1 while

### 3.1.1 while

while 循环会对条件进行判断，如果条件成立，就会执行循环体，然后再次判断条件，直到条件不成立。

while 循环的次数由循环变量的变化决定，因此 while 循环一般都包括对循环变量的初值、判断和更新。

```
1 int i = 1;           // initial value
2 while (i <= 5) {     // condition
3     System.out.println("In loop: i = " + i);
4     i++;             // update
5 }
6 System.out.println("After loop: i = " + i);
```

while 循环的特点是先判断、再执行，因此循环体有可能会执行一次或多次，也有可能一次也不会执行。

#### 平均身高

```
1 import java.util.Scanner;
2
3 public class Height {
4     public static void main(String[] args) {
5         final int NUM_PEOPLE = 5;
6
7         Scanner scanner = new Scanner(System.in);
8
9         int total = 0;
```

```
10
11     int i = 1;
12     while (i < NUM_PEOPLE) {
13         System.out.print("Enter person " + i + "'s height: ");
14         double height = scanner.nextDouble();
15         total += height;
16         i++;
17     }
18     scanner.close();
19
20     double average = total / NUM_PEOPLE;
21     System.out.printf("Average height: %.2f\n", average);
22 }
23 }
```

#### 运行结果

```
Enter person 1's height: 160.8
Enter person 2's height: 175.2
Enter person 3's height: 171.2
Enter person 4's height: 181.3
Enter person 5's height: 164
Average height: 170.50
```

### 3.1.2 do-while

do-while 循环是先执行一轮循环体内的代码后，再检查循环的条件是否成立。如果成立，则继续下一轮循环；否则循环结束。

do-while 循环是先执行、再判断，因此它至少会执行一轮循环。do-while 一般应用在一些可能会需要重复，但必定会发生一次的情景下。例如登录账户，用户输入账户和密码后，检查是否正确，如果正确，那么就成功登录；否则继续循环让用户重新输入。

需要注意，do-while 循环的最后有一个分号。

```
1 do {  
2     // code  
3 } while(condition);
```

### 整数位数

```
1 import java.util.Scanner;  
2  
3 public class Digits {  
4     public static void main(String[] args) {  
5         Scanner scanner = new Scanner(System.in);  
6  
7         System.out.print("Enter an integer: ");  
8         int num = scanner.nextInt();  
9         scanner.close();  
10  
11         int n = 0;  
12  
13         do {  
14             num /= 10;  
15             n++;  
16         } while (num != 0);  
17  
18         System.out.println("Digits: " + n);  
19     }  
20 }
```

### 运行结果

```
Enter an integer: 123  
Digits: 3
```

## 猜数字

```
1 import java.util.Random;
2 import java.util.Scanner;
3
4 public class Guess {
5     public static void main(String[] args) {
6         Scanner scanner = new Scanner(System.in);
7         Random random = new Random();
8
9         // generate random number between 1 and 100
10        int answer = random.nextInt(100) + 1;
11
12        int num = 0;
13        int cnt = 0;
14
15        do {
16            System.out.print("Guess a number: ");
17            num = scanner.nextInt();
18            cnt++;
19
20            if (num > answer) {
21                System.out.println("Too high");
22            } else if (num < answer) {
23                System.out.println("Too low");
24            }
25        } while (num != answer);
26
27        System.out.println("Correct! You guessed " + cnt + " times.");
28        scanner.close();
29    }
30 }
```

### 运行结果

```
Guess a number: 50
Too high
Guess a number: 25
Too low
Guess a number: 37
Too low
Guess a number: 43
Too high
Guess a number: 40
Too high
Guess a number: 38
Too low
Guess a number: 39
Correct! You guessed 7 times.
```

## 3.2 for

### 3.2.1 for

while 循环将循环变量的初值、条件和更新写在了三个地方，但是这样不容易明显地看出循环变量的变化。

for 循环将循环变量的初值、条件和更新写在了了一行内，中间用分号隔开。对于指定次数的循环一般更多地会采用 for 循环，而对于不确定次数的一般会采用 while 循环。

```
1 for (int i = 0; i < 5; i++) {  
2     System.out.println("i = " + i);  
3 }
```

#### 累加

```
1 public class Sum {  
2     public static void main(String[] args) {  
3         int sum = 0;  
4         for (int i = 1; i <= 100; i++) {  
5             sum += i;  
6         }  
7         System.out.println("Sum = " + sum);  
8     }  
9 }
```

#### 运行结果

Sum = 5050

#### 斐波那契数列



```

1 import java.util.Scanner;
2
3 public class Fibonacci {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Enter the number of terms: ");
8         int n = scanner.nextInt();
9         scanner.close();
10
11         if (n == 1) {
12             System.out.println("1");
13         } else if (n == 2) {
14             System.out.println("1, 1");
15         } else {
16             int num1, num2, val;
17             num1 = 1;
18             num2 = 1;
19             System.out.print("1, 1");

```

```

20
21         for (int i = 3; i <= n; i++) {
22             val = num1 + num2;
23             System.out.print(", " + val);
24             num1 = num2;
25             num2 = val;
26         }
27         System.out.println();
28     }
29 }
30 }

```

#### 运行结果

```

Enter the number of terms: 10
1, 1, 2, 3, 5, 8, 13, 21, 34, 55

```

### 3.2.2 嵌套循环

循环也可以嵌套使用，外层循环每执行一次，内层循环就会执行多次。

```

1 for(int i = 0; i < 2; i++) {
2     for(int j = 0; j < 3; j++) {
3         System.out.println("i = " + i + ", j = " + j);
4     }
5 }

```



### 运行结果

i = 0, j = 0  
i = 0, j = 1  
i = 0, j = 2  
i = 1, j = 0  
i = 1, j = 1  
i = 1, j = 2

### 九九乘法表

1*1=1	1*2=2	1*3=3	1*4=4	1*5=5	1*6=6	1*7=7	1*8=8	1*9=9
2*1=2	2*2=4	2*3=6	2*4=8	2*5=10	2*6=12	2*7=14	2*8=16	2*9=18
3*1=3	3*2=6	3*3=9	3*4=12	3*5=15	3*6=18	3*7=21	3*8=24	3*9=27
4*1=4	4*2=8	4*3=12	4*4=16	4*5=20	4*6=24	4*7=28	4*8=32	4*9=36
5*1=5	5*2=10	5*3=15	5*4=20	5*5=25	5*6=30	5*7=35	5*8=40	5*9=45
6*1=6	6*2=12	6*3=18	6*4=24	6*5=30	6*6=36	6*7=42	6*8=48	6*9=54
7*1=7	7*2=14	7*3=21	7*4=28	7*5=35	7*6=42	7*7=49	7*8=56	7*9=63
8*1=8	8*2=16	8*3=24	8*4=32	8*5=40	8*6=48	8*7=56	8*8=64	8*9=72
9*1=9	9*2=18	9*3=27	9*4=36	9*5=45	9*6=54	9*7=63	9*8=72	9*9=81

```
1 public class Multiplication {  
2     public static void main(String[] args) {  
3         for (int i = 1; i <= 9; i++) {  
4             for (int j = 1; j <= 9; j++) {  
5                 System.out.printf("%d*%d=%d\t", i, j, i * j);  
6             }  
7             System.out.println();  
8         }  
9     }  
10 }
```

## 打印图案

```
1 *
2 **
3 ***
4 ****
5 *****
```

```
1 public class Stars {
2     public static void main(String[] args) {
3         for (int i = 1; i <= 5; i++) {
4             for (int j = 1; j <= i; j++) {
5                 System.out.print("*");
6             }
7             System.out.println();
8         }
9     }
10 }
```

## 3.3 break or continue?

### 3.3.1 break

break 可用于跳出当前的 switch 或循环结构。在一些情况下，在循环的中途已经完成了某个目标，没有必要再进行剩余的循环，这时就可以使用 break 跳出循环。

例如在判断一个数  $n$  是否为素数时，利用循环逐个判断  $2 \sim n - 1$  之间的数是否能整除  $n$ 。只要发现其中有一个数能整除  $n$ ，就证明  $n$  不是素数，可以跳出循环，不必再进行剩余的检查。

素数

```
1 import java.util.Scanner;
2
3 public class Prime {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Enter an integer: ");
8         int n = scanner.nextInt();
9         scanner.close();
10
11         boolean is_prime = true;
12         for (int i = 2; i <= Math.sqrt(n); i++) {
13             if (n % i == 0) {
14                 is_prime = false;
15                 break;
16             }
17         }
18
19         if (is_prime) {
20             System.out.println(n + " is a prime number");
21         } else {
22             System.out.println(n + " is not a prime number");
```

```
23     }
24 }
25 }
```

#### 运行结果

```
Enter an integer: 17
17 is a prime number
```

### 3.3.2 continue

continue 与 break 使用方法类似，但是它并不是跳出循环，而是跳过本轮循环，直接开始下一轮循环。

#### 正数平方和

```
1 import java.util.Scanner;
2
3 public class Multiples {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         int n = 10;
8         System.out.print("Enter " + n + " integers: ");
9
10        int sum_square = 0;
11        for (int i = 0; i < n; i++) {
12            int num = scanner.nextInt();
13            if (num < 0) {
14                continue;
15            }
16            sum_square += num * num;
17        }
18    }
```

```
19         System.out.println(  
20             "Sum of squares of positive integers: " + sum_square  
21         );  
22         scanner.close();  
23     }  
24 }
```

#### 运行结果

Enter 10 integers: 5 7 -2 0 4 -4 -9 3 9 5

Sum of squares of positive integers: 205

# Chapter 4 数组

## 4.1 数组

### 4.1.1 数组 (Array)

数组能够存储一组类型相同的元素，数组在声明时必须指定它的大小（容量），数组的大小是固定的，无法在运行时动态改变。数组通过下标（index）来访问某一位置上的元素，下标从 0 开始。

```
1 int[] arr = new int[5];
```

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
--------	--------	--------	--------	--------

如果在声明数组时没有指定数组的大小，那么将根据初始化的元素个数来确定。

```
1 int[] arr = {3, 6, 8, 2, 4, 0, 1, 7};
```

通过下标可以访问数组中的元素，下标的有效范围是 0 ~ 数组的长度 - 1，如果使用不合法的下标就会导致数组越界。

```
1 System.out.println(arr[0]);    // 3
2 System.out.println(arr[3]);    // 2
3 System.out.println(arr[7]);    // 7
```

当数组的容量比较大时，可以使用循环来初始化数组。

```
1 int[] arr = new int[10];
2
3 for(int i = 0; i < 10; i++) {
4     arr[i] = i + 1;
5 }
```

```
1 import java.util.Scanner;
2
3 public class Search {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Enter the number of elements: ");
8         int n = scanner.nextInt();
9
10        int[] arr = new int[n];
11        System.out.print("Enter the elements: ");
12        for (int i = 0; i < n; i++) {
13            arr[i] = scanner.nextInt();
14        }
15
16        System.out.print("Enter the key: ");
17        int key = scanner.nextInt();
18        scanner.close();
19
20        boolean found = false;
21
22        for (int i = 0; i < n; i++) {
23            if (arr[i] == key) {
24                found = true;
25                break;
26            }
27        }
28
29        if (found) {
30            System.out.println(key + " exists.");
31        } else {
32            System.out.println(key + " not found.");
33        }
34    }
35 }
```

### 运行结果

Enter the number of elements: 5  
Enter the elements: 4 8 9 2 3  
Enter the key: 2  
2 exists.

### 最大值/最小值

```
1 public class MaxMin {  
2     public static void main(String[] args) {  
3         int[] num = {7, 6, 2, 9, 3, 1, 4, 0, 5, 8};  
4         int n = num.length;  
5         int max = num[0];  
6         int min = num[0];  
7  
8         for (int i = 1; i < n; i++) {  
9             if (num[i] > max) {  
10                max = num[i];  
11            }  
12            if (num[i] < min) {  
13                min = num[i];  
14            }  
15        }  
16  
17        System.out.println("Max = " + max);  
18        System.out.println("Min = " + min);  
19    }  
20 }
```



#### 运行结果

Max = 9

Min = 0

### 4.1.2 for-each

for-each 循环是一种更加简洁的 for 循环，可以用于遍历访问数组中的每一个元素。

#### 平方和

```
1 public class SquareSum {  
2     public static void main(String[] args) {  
3         int[] arr = {7, 6, 2, 9, 3};  
4         int sum = 0;  
5         for (int elem : arr) {  
6             sum += elem * elem;  
7         }  
8         System.out.println("Square sum = " + sum);  
9     }  
10 }
```

#### 运行结果

Square sum = 179

### 4.1.3 二维数组 (2-Dimensional Array)

二维数组由行和列两个维度组成，行和列的下标同样也都是从 0 开始。在声明二维数组时，需要指定行和列的大小。二维数组可以看成是由多个一维数组组成的，因此二维数组中的每个元素都是一个一维数组。

```

1 int[][] arr = new int[3][4];
2 int[][] arr = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};

```

arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]
arr[1][0]	arr[1][1]	arr[1][2]	arr[1][3]
arr[2][0]	arr[2][1]	arr[2][2]	arr[2][3]

在初始化二维数组时，为了能够更直观地看出二维数组的结构，可以将每一行单独写在一行中。

```

1 int[][] arr = {
2     {1, 2, 3, 4},
3     {5, 6, 7, 8},
4     {9, 10, 11, 12},
5 };

```

对于容量较大的二维数组，可以通过两层循环进行初始化。

```

1 int[][] arr = new int[3][4];
2
3 for (int i = 0; i < 3; i++) {
4     for (int j = 0; j < 4; j++) {
5         arr[i][j] = 0;
6     }
7 }

```

## 矩阵运算

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1-0 & 3-0 \\ 1-7 & 0-5 \\ 1-2 & 2-1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ -6 & -5 \\ -1 & 1 \end{bmatrix}$$

```

1 public class Matrix {
2     public static void main(String[] args) {
3         int[][] A = {
4             {1, 3},
5             {1, 0},
6             {1, 2}
7         };
8         int[][] B = {
9             {0, 0},
10            {7, 5},
11            {2, 1}
12        };
13        int[][] C = new int[3][2];
14
15        System.out.println("Matrix Addition");
16        for (int i = 0; i < 3; i++) {
17            for (int j = 0; j < 2; j++) {
18                C[i][j] = A[i][j] + B[i][j];
19                System.out.printf("%3d", C[i][j]);
20            }
21            System.out.println();
22        }
23
24        System.out.println("Matrix Subtraction");
25        for (int i = 0; i < 3; i++) {
26            for (int j = 0; j < 2; j++) {
27                C[i][j] = A[i][j] - B[i][j];
28                System.out.printf("%3d", C[i][j]);
29            }
30            System.out.println();

```

```
31     }
32     }
33 }
```

#### 运行结果

Matrix Addition

1 3

8 5

3 3

Matrix Subtraction

1 3

-6 -5

-1 1

## 4.2 字符串

### 4.2.1 ASCII

美国信息交换标准代码 ASCII (American Standard Code for Information Interchange) 一共定义了 128 个字符。

ASCII	字符	ASCII	字符	ASCII	字符	ASCII	字符
0	NUL	32	(space)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	TB	55	7	87	W	119	w

24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

## ASCII

```

1 public class ASCII {
2     public static void main(String[] args) {
3         for (int i = 0; i < 128; i++) {
4             System.out.println(String.format("%c - %d", i, i));
5         }
6     }
7 }

```

### 4.2.2 字符串 (String)

一个字符串由若干的字符组成，因此可以使用字符数组表示。

```

1 char[] str = {'p', 'r', 'o', 'g', 'r', 'a', 'm'};

```

但是这样的写法比较麻烦，也不容易对字符串进行操作，因此可以使用更为常用的 String 类来保存字符串。

```

1 String str = "program";

```

占位符%s 可以对字符串进行格式化输出操作，通过 Scanner 类的 next() 和 nextLine() 方法可以读取字符串，其中 next() 只读取到空格为止，而 nextLine()

会读取到回车为止。

### 字符串输入输出

```
1 import java.util.Scanner;
2
3 public class StringIO {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Enter string 1: ");
8         String str1 = scanner.nextLine();
9         System.out.println(str1);
10
11        System.out.print("Enter string 2: ");
12        String str2 = scanner.next();
13        System.out.printf("%s", str2);
14
15        scanner.close();
16    }
17 }
```

### 运行结果

```
Enter string 1: hello world
hello world
Enter string 2: hello world
hello
```

### 字符统计

```
1 import java.util.Scanner;
2
3 public class Frequency {
```

```

4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Enter a string: ");
8         String str = scanner.nextLine();
9         System.out.print("Character to search: ");
10        char c = scanner.next().charAt(0);
11
12        int cnt = 0;
13        for (int i = 0; i < str.length(); i++) {
14            if (str.charAt(i) == c) {
15                cnt++;
16            }
17        }
18
19        System.out.printf(
20            "'%c' appears %d times in \"%s\".\n", c, cnt, str
21        );
22        scanner.close();
23    }
24 }

```

#### 运行结果

```

Enter a string: this is a test
Character to search: t
't' appears 3 times in "this is a test".

```

### 4.2.3 包装类 (Wrapper Class)

包装类在基本数据类型上做一层包装，使得基本数据类型可以进行互相转换的操作。



基本数据类型	包装类型
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

表 4.2: 包装类

将基本数据类型转换为包装类型的过程称为装箱 (boxing)，反之将包装类型转换为基本数据类型的过程称为拆箱 (unboxing)。

```
1 Integer num1 = 123;  
2 int num2 = num1;
```

很多时候需要将字符串与基本数据类型进行转换，这时就需要用到包装类的方法。

在将基本数据类型转换为字符串的过程中，可以使用 String 类的 valueOf() 方法。

```
1 int num = 10;  
2 String str = String.valueOf(num);
```

在将字符串转换为基本数据类型的过程中，可以使用包装类的 valueOf() 或 parseXXX() 方法。

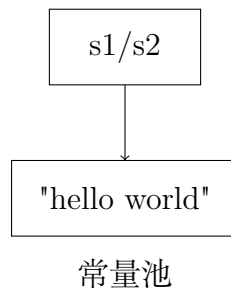
```
1 Integer num1 = Integer.valueOf("10");  
2 System.out.println(num1);  
3  
4 int num2 = Integer.parseInt("10");  
5 System.out.println(num2);
```

需要注意的是，在将字符串转换为基本数据类型时，可能会转换失败。例如将一个非整数的字符串转换为整数，就会抛出 `NumberFormatException` 异常。

#### 4.2.4 字符串内存

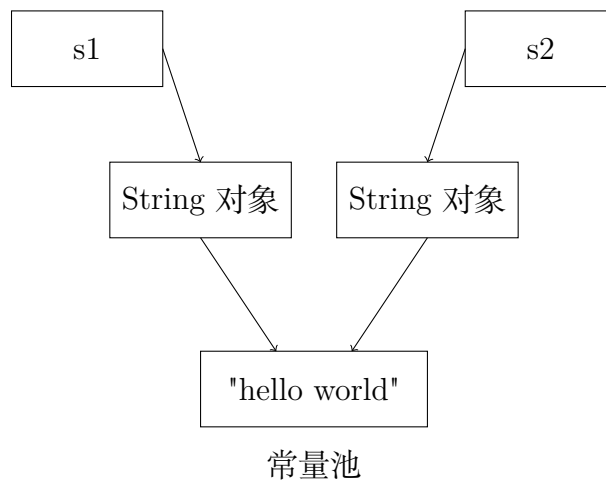
字符串的存储方式与基本数据类型不同，字符串是存储在常量池中的。当使用双引号创建字符串时，会将常量池中已经存在的字符串赋值给该字符串，而不是创建一个新的字符串。

```
1 String s1 = "hello world";  
2 String s2 = "hello world";
```



但是，如果通过 `new` 关键字创建字符串，就会创建一个新的字符串对象。

```
1 String s1 = new String("hello world");  
2 String s2 = new String("hello world");
```



在判断两个字符串是否相等时，应该使用 `equals()` 方法，而不应该使用 `==` 运算符。因为 `==` 运算符是判断两个字符串对象的地址是否相同，而 `equals()` 方法是

判断两个字符串的内容是否相等。

### 字符串比较

```
1 public class StringCompare {  
2     public static void main(String[] args) {  
3         String s1 = new String("hello world");  
4         String s2 = new String("hello world");  
5         System.out.println(s1 == s2);  
6         System.out.println(s1.equals(s2));  
7     }  
8 }
```

### 运行结果

```
false  
true
```

## 4.3 字符串方法

### 4.3.1 字符串匹配

方法	功能
indexOf()	获取字符/字符串首次出现位置
lastIndexOf()	获取字符/字符串最后一次出现位置
contains()	判断字符串是否包含子串
startsWith()	判断字符串是否以指定子串开头
endsWith()	判断字符串是否以指定子串结尾
equals()	判断字符串是否相等
equalsIgnoreCase()	判断字符串忽略大小写是否相等
compareTo()	字符串比较

#### 字符串匹配

```
1 public class StringMatch {
2     public static void main(String[] args) {
3         String str = "Hello World!";
4
5         System.out.println("IndexOf");
6         System.out.println(str.indexOf('l'));
7         System.out.println(str.indexOf('l', 5));
8         System.out.println(str.lastIndexOf('l'));
9         System.out.println(str.lastIndexOf('l', 5));
10
11        System.out.println("Contains");
12        System.out.println(str.contains("llo"));
13
14        System.out.println("Starts/Ends With");
15        System.out.println(str.startsWith("Hell"));
16        System.out.println(str.endsWith("ld"));
17    }
```

```
18     System.out.println("Equals");
19     System.out.println(str.equals("hello"));
20     System.out.println(str.equalsIgnoreCase("hello world!"));
21
22     System.out.println("CompareTo");
23     System.out.println(str.compareTo("Hall"));
24 }
25 }
```

### 运行结果

```
[IndexOf]
2
9
9
3
[Contains]
true
[Starts/Ends With]
true
false
[Equals]
false
true
[CompareTo]
4
```

### 4.3.2 字符串修改

方法	功能
concat()	字符串拼接
toLowerCase()	转换小写
toUpperCase()	转换大写
trim()	去除首尾空白字符
replace()	字符串替换

#### 字符串修改

```
1 import java.util.Locale;
2
3 public class StringModify {
4     public static void main(String[] args) {
5         String str = "Hello World!";
6
7         System.out.println("[Concat]");
8         System.out.println(str.concat("!!"));
9
10        System.out.println("[To Lower/Upper Case]");
11        System.out.println(str.toLowerCase());
12        System.out.println(str.toUpperCase());
13
14        System.out.println("[Trim]");
15        System.out.println("  Hello World!\n \t".trim());
16
17        System.out.println("[Replace]");
18        System.out.println(str.replace('l', 'L'));
19        System.out.println(str.replace("Hello", "Bye"));
20    }
21 }
```

### 运行结果

```
[Concat]
Hello World!!!

[To Lower/Upper Case]
hello world!
HELLO WORLD!

[Trim]
Hello World!

[Replace]
HeLlO WorLd!

Bye World!
```

### 4.3.3 字符串分割

方法	功能
substring()	字符串截取子串
split()	字符串分割

### 字符串分割

```
1 public class StringSlice {
2     public static void main(String[] args) {
3         String str = "Hello World!";
4
5         System.out.println("[Substring]");
6         System.out.println(str.substring(4));
7         System.out.println(str.substring(4, 8));
8
9         System.out.println("[Split]");
10        String[] items = str.split(" ");
```

```
11         for (String item : items) {  
12             System.out.println(item);  
13         }  
14     }  
15 }
```

#### 运行结果

[Substring]

o World!

o Wo

[Split]

Hello

World!



# Chapter 5 函数

## 5.1 函数

### 5.1.1 函数 (Function)

数学中的函数  $y = f(x)$ ，通过输入  $x$  的值，经过计算可以得到  $y$  的值。计算机中的函数也是如此，将输入传给函数，经过处理后，会得到输出。

函数是一段可重复使用的代码，做了一个特定的任务。例如 `println()` 和 `length()` 就是函数，其中 `println()` 的功能是输出字符串，`length()` 的功能是计算字符串的长度。

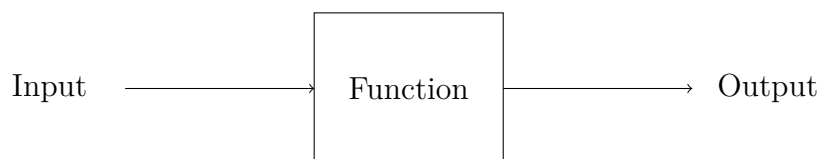


图 5.1: 函数

除了这些内置的函数以外，开发者还可以自定义函数，将程序中会被多次使用的代码或做了一件特定的任务的代码写成一个函数，这样就能避免重复写相同的代码，提高开发效率，也利于维护。

在编写函数时需要：

1. 确定函数的功能
  - 函数名
  - 确保一个函数只做一件事
2. 确定函数的输入（参数）
  - 是否需要参数

- 参数个数
- 参数类型

### 3. 确定函数的输出（返回值）

- 是否需要返回值
- 返回值类型

#### 最大值

```
1 public class Max {  
2     public static void main(String[] args) {  
3         System.out.println(max(4, 12));  
4         System.out.println(max(54, 33));  
5         System.out.println(max(-999, -774));  
6     }  
7  
8     public static int max(int num1, int num2) {  
9         // if(num1 > num2) {  
10            //     return num1;  
11            // } else {  
12            //     return num2;  
13            // }  
14  
15         return num1 > num2 ? num1 : num2;  
16     }  
17 }
```

#### 运行结果

```
12  
54  
-774
```

函数也可以没有返回值，因为它执行完函数中的代码，并不需要将结果返回给调

用者，此时函数的返回值类型为 void。

### 棋盘

```
1 public class Board {
2     public static void main(String[] args) {
3         printBoard();
4     }
5
6     public static void printBoard() {
7         for (int i = 0; i < 3; i++) {
8             for (int j = 0; j < 2; j++) {
9                 System.out.print("  |");
10            }
11            System.out.println();
12
13            if (i < 2) {
14                System.out.println("----+----+----");
15            }
16        }
17    }
18 }
```

### 运行结果

```
  |  |
----+----+----
  |  |
----+----+----
  |  |
```

### 5.1.2 函数调用

当调用函数时，程序会记录下当前的执行位置，并跳转到被调用的函数处执行。当被调用的函数执行结束后，程序会回到之前的位置继续执行。

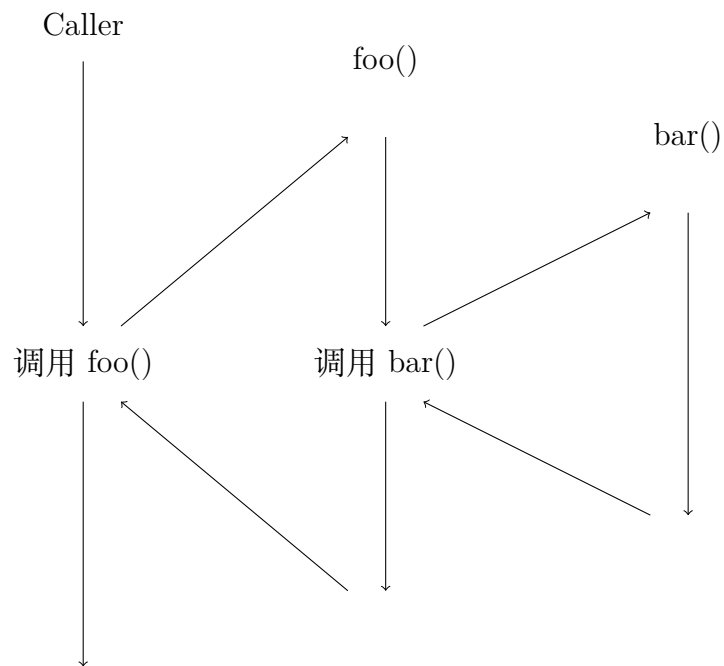


图 5.2: 函数调用

#### 两点间距离

```
1 import java.util.Scanner;
2
3 public class Distance {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Enter (x1, y1): ");
8         double x1 = scanner.nextDouble();
9         double y1 = scanner.nextDouble();
10        System.out.print("Enter (x2, y2): ");
11        double x2 = scanner.nextDouble();
12        double y2 = scanner.nextDouble();
```

```
13 scanner.close();
14
15 System.out.printf(
16     "Distance: %.2f\n", distance(x1, y1, x2, y2)
17 );
18 }
19
20 public static double square(double x) {
21     return x * x;
22 }
23
24 public static double distance(double x1, double y1,
25     double x2, double y2) {
26     return Math.sqrt(square(x1 - x2) + square(y1 - y2));
27 }
28 }
```

#### 运行结果

Enter (x1, y1): 0 0

Enter (x2, y2): 3 4

Distance: 5.00

## 5.2 作用域

### 5.2.1 局部变量 (Local Variable)

定义在块中的变量称为局部变量，在进入块时变量才会被创建，当离开块时变量就会被销毁。因此，局部变量的生命周期为从声明时开始到所在块结束。

例如有些变量只在程序的某一段代码中使用，而在其它地方不会被使用。这时就可以将这些变量定义在一个块（if、for、函数等）中，这样可以避免变量名冲突的问题。最典型的一个例子就是在 for 循环中，循环变量 i 被定义被块中，因为 i 的作用仅用于控制循环次数，在离开循环后就没有存在的必要了。

```
1 for (int i = 0; i < 5; i++)
```

块与块之间的局部变量是互相独立的，即使变量名相同，它们也不是同一个变量。

例如在函数调用中，函数的参数也是局部变量，它们的作用域仅限于函数内。

例如一个用于交换两个变量的函数 swap()，在 main() 中的变量 a 和 b 与 swap() 中的 a 和 b 并不是同一个变量。在调用 swap() 时，是将 main() 中的 a 和 b 的值复制给 swap() 中的 a 和 b。swap() 交换的是其内部的局部变量，并不会对 main() 中的 a 和 b 产生任何影响。

#### 局部变量

```
1 public class Swap {
2     public static void main(String[] args) {
3         int a = 1;
4         int b = 2;
5
6         System.out.println("Before: a = " + a + ", b = " + b);
7         swap(a, b);
8         System.out.println("After: a = " + a + ", b = " + b);
9     }
```

```
10
11     public static void swap(int a, int b) {
12         int temp = a;
13         a = b;
14         b = temp;
15         System.out.println("swap(): a = " + a + ", b = " + b);
16     }
17 }
```

#### 运行结果

Before: a = 1, b = 2

swap(): a = 2, b = 1

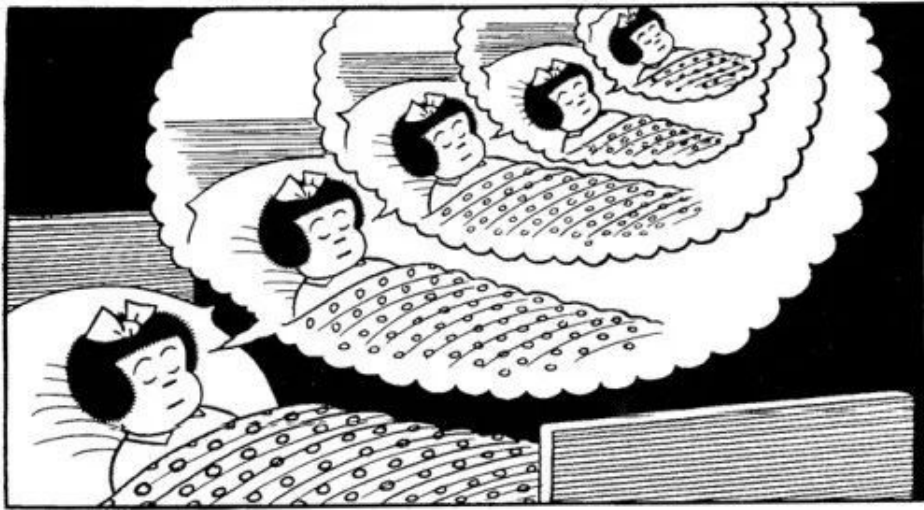
After: a = 1, b = 2

## 5.3 递归

### 5.3.1 递归 (Recursion)

要理解递归，得先理解递归（见5.3章节）。

一个函数调用自己的过程被称为递归。递归可以轻松地解决一些复杂的问题，很多著名的算法都利用了递归的思想。



#### 讲故事

```
1 public class TellStory {  
2     public static void main(String[] args) {  
3         tellStory();  
4     }  
5  
6     public static void tellStory() {  
7         String story;  
8         story += "从前有座山，山里有座庙\n";  
9         story += "庙里有个老和尚\n";  
10        story += "老和尚在对小和尚讲故事： ";  
11        System.out.println(story);  
12  
13        tellStory();  
}
```



```
14     }  
15 }
```

### 运行结果

从前有座山，山里有座庙  
庙里有个老和尚  
老和尚在对小和尚讲故事：  
从前有座山，山里有座庙  
庙里有个老和尚  
老和尚在对小和尚讲故事：  
从前有座山，山里有座庙  
庙里有个老和尚  
老和尚在对小和尚讲故事：  
...

一个永远无法结束的递归函数最终会导致栈溢出。因此递归函数需要确定一个结束条件，确保在递归过程中能在合适的地方停止并返回。

### 阶乘

```
1 public class Factorial {  
2     public static void main(String[] args) {  
3         System.out.println("5! = " + factorial(5));  
4     }  
5  
6     public static int factorial(int n) {  
7         if (n == 0 || n == 1) {  
8             return 1;  
9         }  
10        return n * factorial(n - 1);  
11    }  
12 }
```

### 运行结果

5! = 120

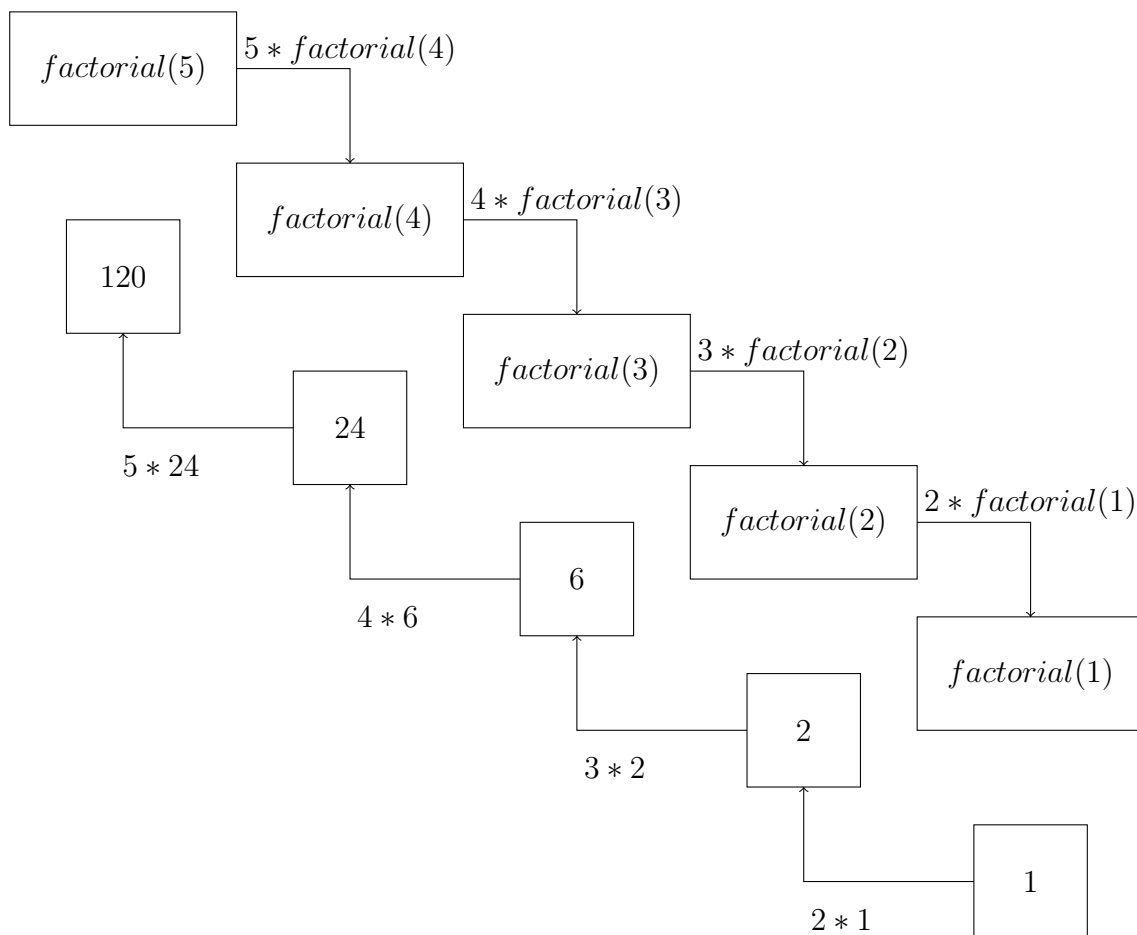


图 5.3: 阶乘

### 斐波那契数列

```
1 public class Fibonacci {
2     public static void main(String[] args) {
3         int n = 7;
4         System.out.println(fibonacci(n));
5     }
6
7     public static int fibonacci(int n) {
8         if (n == 1 || n == 2) {
```

```

9         return n;
10    }
11    return fibonacci(n - 2) + fibonacci(n - 1);
12 }
13 }

```

#### 运行结果

21

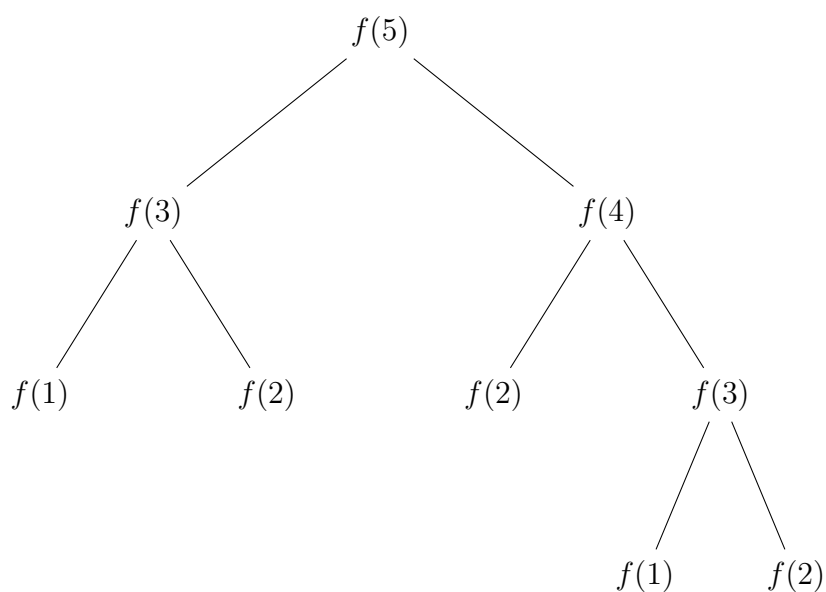


图 5.4: 递归树

递归的特点就是将一个复杂的大问题逐步简化为一个可以解决的小问题，然后再逐步计算出大问题的解。

递归的优点在于代码简洁易懂，但是缺点也很明显，就是效率很低。每次递归都会产生函数调用，而函数调用的开销是很大的，不适合用来解决大规模的问题。

例如在计算斐波那契数列的第 40 项时，递归需要花费大量时间，因为其中包含了大量的重复计算。相比而言，使用循环的方式能够节省大量的时间。因此像阶乘和斐波那契数列这样的情况，通常会采用循环，而不是递归进行计算。

然而还存在很多问题不得不使用递归的思想才能解决。

### 阿克曼函数

$$A(m, n) = \begin{cases} n + 1 & m = 0 \\ A(m - 1, 1) & m > 0, n = 0 \\ A(m - 1, A(m, n - 1)) & m > 0, n > 0 \end{cases}$$

```
1 public class Ackermann {
2     public static void main(String[] args) {
3         System.out.println(A(3, 4));
4     }
5
6     public static int A(int m, int n) {
7         if (m == 0) {
8             return n + 1;
9         } else if (m > 0 && n == 0) {
10            return A(m - 1, 1);
11        } else {
12            return A(m - 1, A(m, n - 1));
13        }
14    }
15 }
```

### 运行结果

125

### 汉诺塔

有三根柱子 A、B、C，A 柱子上从下到上套有 n 个圆盘，要求将 A 柱子上的圆盘移动到 C 柱子上。每次只能移动一个圆盘，且大圆盘始终不能叠在小圆盘上面。

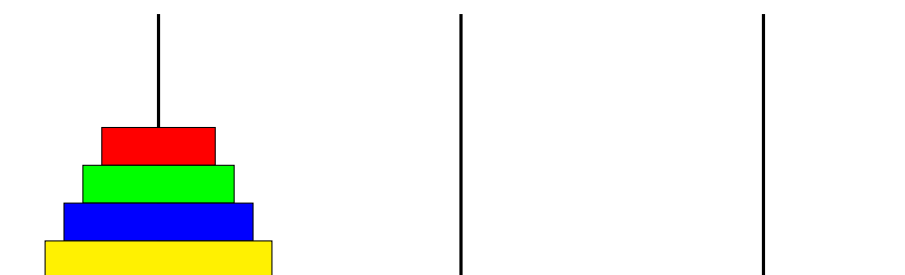


图 5.5: 汉诺塔

递归算法求解汉诺塔问题：

1. 将  $n-1$  个圆盘从 A 借助 C 移到 B。
2. 将第  $n$  个圆盘从 A 移到 C。
3. 将  $n-1$  个圆盘从 B 借助 A 移到 C。

```
1 public class Hanoi {
2     static int move = 0;
3
4     public static void main(String[] args) {
5         hanoi(3, 'A', 'B', 'C');
6         System.out.println("Moves: " + move);
7     }
8
9     public static void hanoi(int n, char src, char mid, char dst) {
10        if (n == 1) {
11            System.out.println(src + " -> " + dst);
12            move++;
13        } else {
14            // move top n-1 disks from src to mid
15            hanoi(n - 1, src, dst, mid);
16            System.out.println(src + " -> " + dst);
17            move++;
18            // move top n-1 disks from mid to dst
19            hanoi(n - 1, mid, src, dst);
20        }
21    }
22 }
```

```
21     }  
22 }
```

### 运行结果

```
A -> C  
A -> B  
C -> B  
A -> C  
B -> A  
B -> C  
A -> C  
Moves: 7
```

假设每次移动花费 1 秒，解决一个 64 层的汉诺塔问题大约需要 5800 亿年。

吓得我抱起了

抱着抱着抱着我的小鲤鱼的我的我的我



# Chapter 6 面向对象

## 6.1 封装

### 6.1.1 类与对象

在面向对象编程中，把构成问题的事物分解成各个对象，每个对象都有自己的数据和行为，程序通过对象之间的交互来实现功能。

类（class）是一个模板，定义了对象的属性和方法，用来描述同一类对象的共同特征和行为。对象（object）是类的实例，它具有类定义的属性和方法。

关键字 `new` 可以实例化一个类对象，之后就可以通过访问对象的属性和方法来操作对象。

#### 银行账户

```
1 public class BankAccount {  
2     String owner;  
3     String account;  
4     double balance;  
5  
6     public void deposit(double amount) {  
7         balance += amount;  
8     }  
9  
10    public void withdraw(double amount) {  
11        balance -= amount;  
12    }  
13 }
```

```
1 public class Bank {  
2     public static void main(String[] args) {  
3         BankAccount account = new BankAccount();  
4         account.owner = "Terry";  
5         account.account = "6250941006528599";  
6         account.balance = 50;  
7  
8         System.out.println("Owner: " + account.owner);  
9         System.out.println("Account: " + account.account);  
10        System.out.println("Balance: " + account.balance);  
11  
12        account.deposit(100);  
13        System.out.println("Balance: " + account.balance);  
14  
15        account.withdraw(70);  
16        System.out.println("Balance: " + account.balance);  
17    }  
18 }
```

#### 运行结果

```
Owner: Terry  
Account: 6250941006528599  
Balance: 50.0  
Balance: 150.0  
Balance: 80.0
```

### 6.1.2 封装 (Encapsulation)

封装是面向对象的重要原则，尽可能隐藏对象的内部实现细节。封装可以认为是一个保护屏障，防止该类的数据被外部随意访问。当要访问该类的数据时，必须通过指定的接口。合适的封装可以让代码更容易理解和维护，也加强了程序的安全性。



为了实现封装，需要对类的属性和方法进行访问权限的控制：

1. public：允许任何地方访问。
2. private：只允许在类的内部访问。
3. protected：只允许在类的内部和子类中访问。
4. default：只允许在同一个包中访问。

通常会将类的属性设置为 private，然后对外提供一对 setter/getter 方法来访问该属性。

为了避免方法的参数与类的属性重名造成歧义，可以使用 this 关键字用来指代当前对象。

### 银行账户

```
1 public class BankAccount {
2     private final int ACCOUNT_DIGITS = 16;
3
4     private String owner;
5     private String account;
6     private double balance;
7
8     public void setOwner(String owner) {
9         if (!owner.isEmpty()) {
10             this.owner = owner;
11         }
12     }
13
14     public String getOwner() {
15         return owner;
16     }
17
18     public void setaccount(String account) {
19         if (account.length() == ACCOUNT_DIGITS) {
```

```

20         this.account = account;
21     }
22 }
23
24 public String getAccount() {
25     return account;
26 }
27
28 public void setBalance(double balance) {
29     if (balance >= 0) {
30         this.balance = balance;
31     }
32 }
33
34 public double getBalance() {
35     return balance;
36 }
37
38 public boolean deposit(double amount) {
39     if (amount <= 0) {
40         return false;
41     }
42     balance += amount;
43     return true;
44 }
45
46 public boolean withdraw(double amount) {
47     if (amount <= 0 || amount > balance) {
48         return false;
49     }
50     balance -= amount;
51     return true;
52 }
53 }

```

```

1 public class Bank {

```

```
2     public static void main(String[] args) {
3         BankAccount account = new BankAccount();
4         account.setOwner("Terry");
5         account.setAccount("6250941006528599");
6         account.setBalance(50);
7
8         System.out.println("Owner: " + account.getOwner());
9         System.out.println("Account: " + account.getAccount());
10        System.out.println("Balance: " + account.getBalance());
11
12        account.deposit(100);
13        System.out.println("Balance: " + account.getBalance());
14
15        account.withdraw(70);
16        System.out.println("Balance: " + account.getBalance());
17    }
18 }
```

### 运行结果

Owner: Terry  
Account: 6250941006528599  
Balance: 50.0  
Balance: 150.0  
Balance: 80.0

## 6.2 构造方法

### 6.2.1 构造方法 (Constructor)

构造方法是一种特殊的方法，会在创建对象时自动调用，用于创建并初始化对象。每个类可以有一个或多个构造方法，构造方法的名字必须和类名一致。构造方法没有返回值，返回值类型部分不写。

```
1 public BankAccount() {  
2     owner = "admin";  
3     account = "0000000000000000";  
4     balance = 0;  
5 }
```

如果一个类中没有写构造方法，系统会自动提供一个 public 的无参构造方法，以便实例化对象。如果一个类中已经写了构造方法，系统将不会再提供默认的无参构造方法。

```
1 public BankAccount(String owner, String account, double balance) {  
2     if (!owner.isEmpty()) {  
3         this.owner = owner;  
4     }  
5  
6     if (account.length() == ACCOUNT_DIGITS) {  
7         this.account = account;  
8     }  
9  
10    if (balance >= 0) {  
11        this.balance = balance;  
12    }  
13 }
```

### 6.2.2 重载 (Overload)

重载用于在同一个类定义多个同名方法，但是这些方法的参数列表不同。重载的主要用途是提供方法的多种版本，以便满足不同的需求。

重载还可以使代码更具可读性，因为它使得方法名更具描述性，而不必考虑特定的参数列表。

## 银行账户

```
1 public class BankAccount {
2     private final int ACCOUNT_DIGITS = 16;
3
4     private String owner;
5     private String account;
6     private double balance;
7
8     public BankAccount() {
9         owner = "admin";
10        account = "0000000000000000";
11        balance = 0;
12    }
13
14    public BankAccount(String owner, String account, double balance) {
15        if (!owner.isEmpty()) {
16            this.owner = owner;
17        }
18
19        if (account.length() == ACCOUNT_DIGITS) {
20            this.account = account;
21        }
22
23        if (balance >= 0) {
24            this.balance = balance;
25        }
26    }
27
28    public void setOwner(String owner) {
29        if (!owner.isEmpty()) {
30            this.owner = owner;
```

```
31     }
32 }
33
34 public String getOwner() {
35     return owner;
36 }
37
38 public void setAccount(String account) {
39     if (account.length() == ACCOUNT_DIGITS) {
40         this.account = account;
41     }
42 }
43
44 public String getAccount() {
45     return account;
46 }
47
48 public void setBalance(double balance) {
49     if (balance >= 0) {
50         this.balance = balance;
51     }
52 }
53
54 public double getBalance() {
55     return balance;
56 }
57
58 public boolean deposit(double amount) {
59     if (amount <= 0) {
60         return false;
61     }
62     balance += amount;
63     return true;
64 }
65
66 public boolean withdraw(double amount) {
67     if (amount <= 0 || amount > balance) {
```

```

68         return false;
69     }
70     balance -= amount;
71     return true;
72 }
73
74 public boolean withdraw(double amount, double fee) {
75     if(amount <= 0 || amount + fee > balance) {
76         return false;
77     }
78
79     balance -= amount + fee;
80     return true;
81 }
82 }

```

```

1 public class Bank {
2     public static void main(String[] args) {
3         BankAccount account1 = new BankAccount();
4         System.out.println(
5             "Account 1 Owner: " + account1.getOwner()
6         );
7         System.out.println(
8             "Account 1 Account: " + account1.getAccount()
9         );
10        System.out.println(
11            "Account 1 Balance: " + account1.getBalance()
12        );
13
14        BankAccount account2 = new BankAccount(
15            "Terry", "6250941006528599", 50
16        );
17        System.out.println(
18            "Account 2 Balance: " + account2.getBalance()
19        );
20    }

```

```
21         account2.withdraw(20);
22         System.out.println(
23             "Account 2 Balance: " + account2.getBalance()
24         );
25
26         account2.withdraw(10, 1);
27         System.out.println(
28             "Account 2 Balance: " + account2.getBalance()
29         );
30     }
31 }
```

#### 运行结果

```
Account 1 Owner: admin
Account 1 Account: 000000000000000000
Account 1 Balance: 0.0
Account 2 Balance: 50.0
Account 2 Balance: 30.0
Account 2 Balance: 19.0
```



## 6.3 继承

### 6.3.1 继承 (Inheritance)

继承指一个类可以继承另一个类的特征和行为，并可以对其进行扩展。这样就可以避免在多个类中重复定义相同的特征和行为。

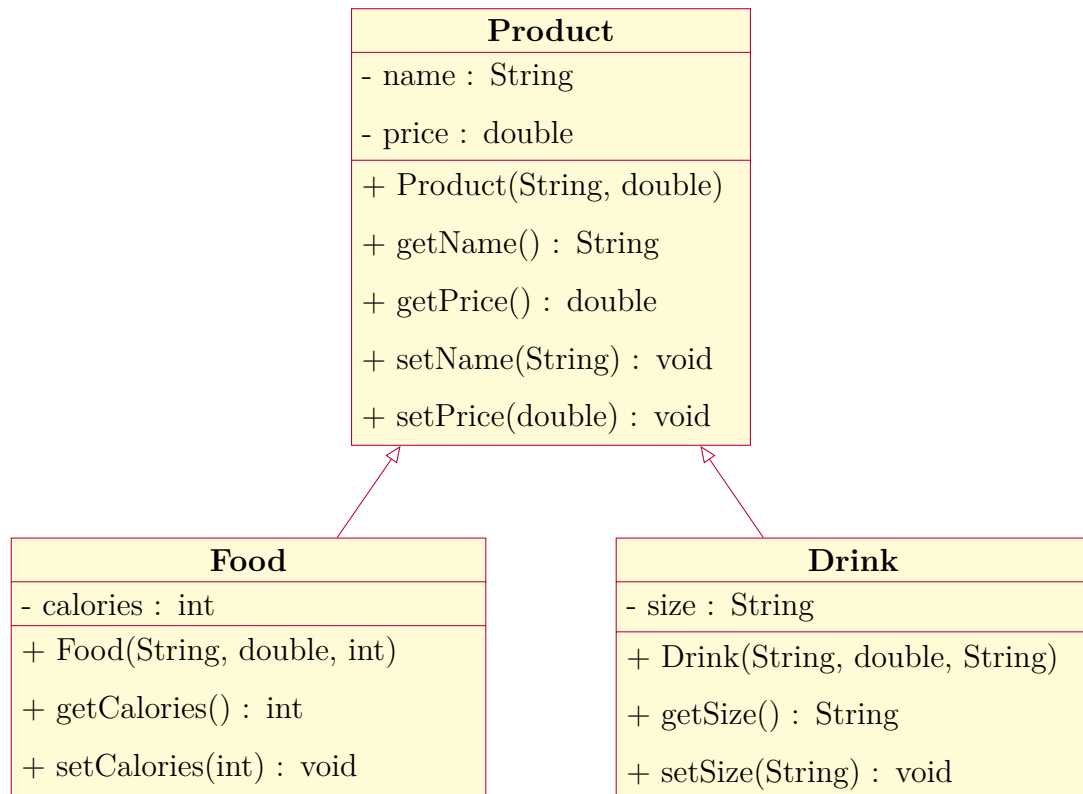


图 6.1: 继承

`extends` 关键字用于指定一个类继承于另一个类，产生继承关系后，子类可以通过 `super` 关键字调用父类中的属性和方法，也可以定义子类独有的属性和方法。

在创建子类对象时，会先调用父类的构造方法，然后再调用子类的构造方法。因此父类中必须存在一个构造方法，否则将无法创建子类对象。

麦当劳

```
1 public class Product {
```

```
2     private String name;
3     private double price;
4
5     public Product(String name, double price) {
6         this.name = name;
7         this.price = price;
8     }
9
10    public String getName() {
11        return name;
12    }
13
14    public void setName(String name) {
15        this.name = name;
16    }
17
18    public double getPrice() {
19        return price;
20    }
21
22    public void setPrice(double price) {
23        this.price = price;
24    }
25 }
```

```
1 public class Food extends Product {
2     int calories;
3
4     public Food(String name, double price, int calories) {
5         super(name, price);
6         this.calories = calories;
7     }
8
9     public int getCalories() {
10        return calories;
11    }
```

```
12
13     public void setCalories(int calories) {
14         this.calories = calories;
15     }
16 }
```

```
1 public class Drink extends Product {
2     private String size;
3
4     public Drink(String name, double price, String size) {
5         super(name, price);
6         this.size = size;
7     }
8
9     public String getSize() {
10         return size;
11     }
12
13     public void setSize(String size) {
14         this.size = size;
15     }
16 }
```

```
1 public class McDonalds {
2     public static void main(String[] args) {
3         Food food = new Food("Cheeseburger", 5.45, 302);
4         Drink drink = new Drink("Coke", 3.7, "Large");
5
6         System.out.printf(
7             "Food: %s ($%.2f) %d Kcal\n",
8             food.getName(), food.getPrice(), food.getCalories()
9         );
10        System.out.printf(
11            "Drink: %s ($%.2f) %s\n",
12            drink.getName(), drink.getPrice(), drink.getSize()
```

```
13         );  
14     }  
15 }
```

#### 运行结果

```
Food: Cheeseburger ($5.45) 302 Kcal  
Drink: Coke ($3.70) Large
```

### 6.3.2 重写 (Override)

Object 类是所有类的根类，所有的类都直接或者间接地继承自 Object 类。Object 类中包含的方法在其它所有类中都可以使用，例如 getClass()、hashCode()、toString()、clone()、equals() 等。

当直接输出一个对象时，会自动调用该对象的 toString() 方法，将其以字符串的形式输出。

```
1 System.out.println(food); // Food@41629346
```

在没有重写 toString() 方法的情况下，输出的内容是对象的类名及其哈希码 (hash code)，但这并不是预期想要的结果。因此，可以重写从父类继承的 toString()，以满足程序的需求。

在重写方法时，需要使用 @Override 注解，以便编译器检查该方法是否真的是从父类继承的。

Object 类的 equals() 方法默认比较的是两个对象的地址，如果地址相同则为 true，否则为 false。而 String 类的 equals() 方法就重写了 Object 类的 equals() 方法，以便比较两个字符串的内容是否相同。

#### 麦当劳

```

1 public class Product {
2     private String name;
3     private double price;
4
5     public Product(String name, double price) {
6         this.name = name;
7         this.price = price;
8     }
9
10    public String getName() {
11        return name;
12    }
13
14    public void setName(String name) {
15        this.name = name;
16    }
17
18    public double getPrice() {
19        return price;
20    }
21
22    public void setPrice(double price) {
23        this.price = price;
24    }
25
26    @Override
27    public String toString() {
28        return String.format("%s (%.2f)", name, price);
29    }
30 }

```

```

1 public class Food extends Product {
2     int calories;
3
4     public Food(String name, double price, int calories) {

```

```

5         super(name, price);
6         this.calories = calories;
7     }
8
9     public int getCalories() {
10         return calories;
11     }
12
13     public void setCalories(int calories) {
14         this.calories = calories;
15     }
16
17     @Override
18     public String toString() {
19         return "Food: " + super.toString() + " " + calories + " Kcal";
20     }
21 }

```

```

1 public class Drink extends Product {
2     private String size;
3
4     public Drink(String name, double price, String size) {
5         super(name, price);
6         this.size = size;
7     }
8
9     public String getSize() {
10         return size;
11     }
12
13     public void setSize(String size) {
14         this.size = size;
15     }
16
17     @Override
18     public String toString() {

```

```
19         return "Drink: " + super.toString() + " " + size;
20     }
21 }
```

```
1 public class McDonalds {
2     public static void main(String[] args) {
3         Food food = new Food("Cheeseburger", 5.45, 302);
4         Drink drink = new Drink("Coke", 3.7, "Large");
5
6         System.out.println(food);
7         System.out.println(drink);
8     }
9 }
```

#### 运行结果

Food: Cheeseburger (\$5.45) 302 Kcal

Drink: Coke (\$3.70) Large

## 6.4 抽象类

### 6.4.1 抽象类 (Abstract Class)

有些类只能用来做继承，不能用于创建对象。例如在动物园中并不存在“动物”这个对象，只有动物的子类对象，因此动物类不应该被实例化。

抽象类是一种不能被实例化的类，它用于定义接口或公共实现，供其它类继承并实现。

```
1 public abstract class Animal {}
```

### 6.4.2 抽象方法

有时候父类提供的方法无法满足子类不同的需求，但是如果不定义该方法，就表示该类具有该行为。

这种情况就可以将这个父类的方法定义为抽象方法，这样所有的子类都必须重写该方法，否则子类仍然为抽象类。

抽象方法只需声明，而不用实现。包含抽象方法的类必须声明为抽象类。

动物

```
1 public abstract class Animal {  
2     public abstract String sound();  
3 }
```

```
1 public class Dog extends Animal {  
2     @Override  
3     public String sound() {  
4         return "Woof";  
    }
```



```
5     }  
6 }
```

```
1 public class Cat extends Animal {  
2     @Override  
3     public String sound() {  
4         return "Meow";  
5     }  
6 }
```

```
1 public class AnimalSound {  
2     public static void main(String[] args) {  
3         Dog dog = new Dog();  
4         Cat cat = new Cat();  
5  
6         System.out.println("Dog's sound: " + dog.sound());  
7         System.out.println("Cat's sound: " + cat.sound());  
8     }  
9 }
```

#### 运行结果

Dog's sound: Woof

Cat's sound: Meow

## 6.5 多态

### 6.5.1 多态 (Polymorphism)

多态是指对象可以具有多种形态，即同一个对象在不同时刻表现出不同的行为。例如 Dog 和 Cat 都是 Animal 的子类，因此可以将子类对象赋值给父类引用，从而产生多种形态。

```
1 Animal animal = new Dog();
```

由子类类型转型为父类类型，称为向上转型。向上转型是一种隐式转换，向上转型后的对象，只能访问父类中定义的成员。

由父类类型转型为子类类型，称为向下转型。向下转型存在失败的可能，因为父类对象并不一定是子类对象。

向下转型需要显式地强制转换，在转换时需要使用 instanceof 关键字进行类型检查。

#### 员工工资

```
1 public abstract class Employee {  
2     private String name;  
3  
4     public Employee(String name) {  
5         this.name = name;  
6     }  
7  
8     public String getName() {  
9         return name;  
10    }  
11  
12    public abstract double getSalary();  
13 }
```

```

1 public class FullTimeEmployee extends Employee {
2     private double basicSalary;
3     private double bonus;
4
5     public FullTimeEmployee(
6         String name, double basicSalary, double bonus
7     ) {
8         super(name);
9         this.basicSalary = basicSalary;
10        this.bonus = bonus;
11    }
12
13    @Override
14    public double getSalary() {
15        return basicSalary + bonus;
16    }
17 }

```

```

1 public class PartTimeEmployee extends Employee {
2     private double dailyWage;
3     private int workingDays;
4
5     public PartTimeEmployee(
6         String name, double dailyWage, int workingDays
7     ) {
8         super(name);
9         this.dailyWage = dailyWage;
10        this.workingDays = workingDays;
11    }
12
13    @Override
14    public double getSalary() {
15        return dailyWage * workingDays;
16    }
17 }

```

```
1 public class Salary {
2     public static void main(String[] args) {
3         Employee[] employees = new Employee[]{
4             new FullTimeEmployee("Alice", 5783, 173),
5             new PartTimeEmployee("Bob", 150, 15)
6         };
7
8         for (Employee employee : employees) {
9             System.out.println(
10                 employee.getName() + ": $" + employee.getSalary()
11             );
12         }
13     }
14 }
```

#### 运行结果

Alice: \$5956.0

Bob: \$2250.0

## 6.6 接口

### 6.6.1 接口 (Interface)

接口是一种特殊的抽象类，接口中的所有方法都是抽象方法，接口中的成员变量必须是常量。

接口用于定义一组标准，代表了某种能力和约定。例如 USB 接口代表了电脑和外设设备之间的连接标准。USB 接口不用关心连接的外设设备是什么，只要这个外设设备满足 USB 的标准，就可以连接到电脑上。

#### USB

```
1 public interface USB {  
2     String getDeviceInfo();  
3 }
```

```
1 public class Mouse implements USB {  
2     @Override  
3     public String getDeviceInfo() {  
4         return "Mouse";  
5     }  
6 }
```

```
1 public class Keyboard implements USB {  
2     @Override  
3     public String getDeviceInfo() {  
4         return "Keyboard";  
5     }  
6 }
```

```
1 public class Computer {
2     private USB usb1;
3     private USB usb2;
4
5     public void setUSB1(USB usb) {
6         usb1 = usb;
7     }
8
9     public void setUSB2(USB usb) {
10        usb2 = usb;
11    }
12
13    @Override
14    public String toString() {
15        return "USB 1: " + usb1.getDeviceInfo() + "\n"
16            + "USB 2: " + usb2.getDeviceInfo();
17    }
18
19    public static void main(String[] args) {
20        Computer computer = new Computer();
21
22        computer.setUSB1(new Mouse());
23        computer.setUSB2(new Keyboard());
24
25        System.out.println(computer);
26    }
27 }
```

#### 运行结果

USB 1: Mouse

USB 2: Keyboard

# Chapter 7 异常

## 7.1 异常

### 7.1.1 异常 (Exception)

异常就是程序在运行过程中出现的非正常的情况，它可以被捕获并处理，以防止程序崩溃。

Exception 是一个异常类，发生异常的时候会抛出一个异常对象。如果不处理异常，程序就会被中断。

异常	描述
ArithmeticException	异常的运算条件
ArrayIndexOutOfBoundsException	非法索引访问数组
ClassCastException	不匹配的类型转换
ClassNotFoundException	找不到相应类
FileNotFoundException	文件无法找到
IllegalArgumentException	非法参数
IndexOutOfBoundsException	索引超出范围
InputMismatchException	输入类型不匹配
NullPointerException	空指针异常
NumberFormatException	无法将字符串转换为数值类型

表 7.1: 常用异常

例如当数组访问越界时，会抛出一个 `ArrayIndexOutOfBoundsException` 异常；当访问一个不存在的对象时，会抛出一个 `NullPointerException` 异常。

### 7.1.2 捕获异常

try-catch-finally 结构可以用于捕获并处理异常，将可能出现异常的代码放在 try 结构中，将异常处理的代码放在 catch 结构中，finally 结构中的代码无论是否出现异常都会执行。

当在 try 结构中出现异常时，程序会跳转到 catch 结构中，执行 catch 结构中的代码。一个异常被处理后，将不再影响程序的执行。

对方接住你  
抛出的异常并完美解决



整除

```
1 import java.util.InputMismatchException;
2 import java.util.Scanner;
3
4 public class Division {
5     public static void main(String[] args) {
6         Scanner scanner = new Scanner(System.in);
7
8         while (true) {
9             try {
10                 System.out.print("Enter an integer for dividend: ");
11                 int dividend = scanner.nextInt();
12                 System.out.print("Enter an integer for divisor: ");
13                 int divisor = scanner.nextInt();
14                 int quotient = dividend / divisor;
```



```

15         System.out.println(
16             dividend + " / " + divisor + " = " + quotient
17         );
18     } catch (InputMismatchException e) {
19         System.out.println("Only integers supported.");
20     } catch (ArithmeticException e) {
21         System.out.println("Divisor cannot be 0.");
22     } finally {
23         scanner.nextLine();
24     }
25 }
26 }
27 }

```

### 运行结果

```

Enter an integer for dividend: 21
Enter an integer for divisor: 4
21 / 4 = 5
Enter an integer for dividend: 5
Enter an integer for divisor: 0
Divisor cannot be 0.
Enter an integer for dividend: 3.6
Only integers supported.

```

### 7.1.3 throw

throw 关键字用于在方法内部抛出一个异常，throws 关键字用于方法签名中，表示该方法可能会抛出异常。

抛出异常后，需要由方法的调用者来处理异常。如果调用者没有处理异常，那么异常就继续向上抛出，直到被处理。

Exception in thread "main" java.lang.NullPointerException  
at Ex62.Demo.main(Demo.java:24)



对方不想你和说话  
并向你抛出了一个异常

### 阶乘

```
1 import java.util.Scanner;
2
3 public class Factorial {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Enter n: ");
8         int n = scanner.nextInt();
9
10        try {
11            System.out.println(n + "! = " + factorial(n));
12        } catch (IllegalArgumentException e) {
13            e.printStackTrace();
14        } finally {
15            scanner.close();
16        }
17    }
18
19    public static int factorial(int n)
20        throws IllegalArgumentException {
21        if (n < 0) {
22            throw new IllegalArgumentException(
23                "Factorial of negative numbers is not defined."
```

```
24         );
25     }
26
27     if (n == 0 || n == 1) {
28         return 1;
29     }
30     return n * factorial(n - 1);
31 }
32 }
```

#### 运行结果

```
Enter n: -1
java.lang.IllegalArgumentException: Factorial of negative
numbers is not defined.
    at Factorial.factorial(Factorial.java:21)
    at Factorial.main(Factorial.java:11)
```

## 7.2 自定义异常

### 7.2.1 自定义异常

为了满足某些特定的需求，用户可以自定义异常，自定义异常继承于 `Exception` 类或其子类。自定义异常的目的是为了提供更具体和有意义的错误处理。

#### 库存

```
1 public class OutOfStockException extends Exception {
2     public OutOfStockException(String msg) {
3         super(msg);
4     }
5 }
```

```
1 public class Product {
2     private String name;
3     private int stock;
4
5     public Product(String name, int stock) {
6         this.name = name;
7         this.stock = stock;
8     }
9
10    public void purchase() throws OutOfStockException {
11        if (stock <= 0) {
12            throw new OutOfStockException(name + " is out of stock.");
13        }
14        stock--;
15    }
16 }
```

```
1 public class Purchase {
```

```
2     public static void main(String[] args) {  
3         Product product = new Product("Cheeseburger", 50);  
4  
5         try {  
6             for (int i = 0; i < 60; i++) {  
7                 product.purchase();  
8             }  
9         } catch (OutOfStockException e) {  
10             e.printStackTrace();  
11         }  
12     }  
13 }
```

#### 运行结果

```
OutOfStockException: Cheeseburger is out of stock.  
    at Product.purchase(Product.java:12)  
    at Purchase.main(Purchase.java:7)
```