Java

极夜酱

目录

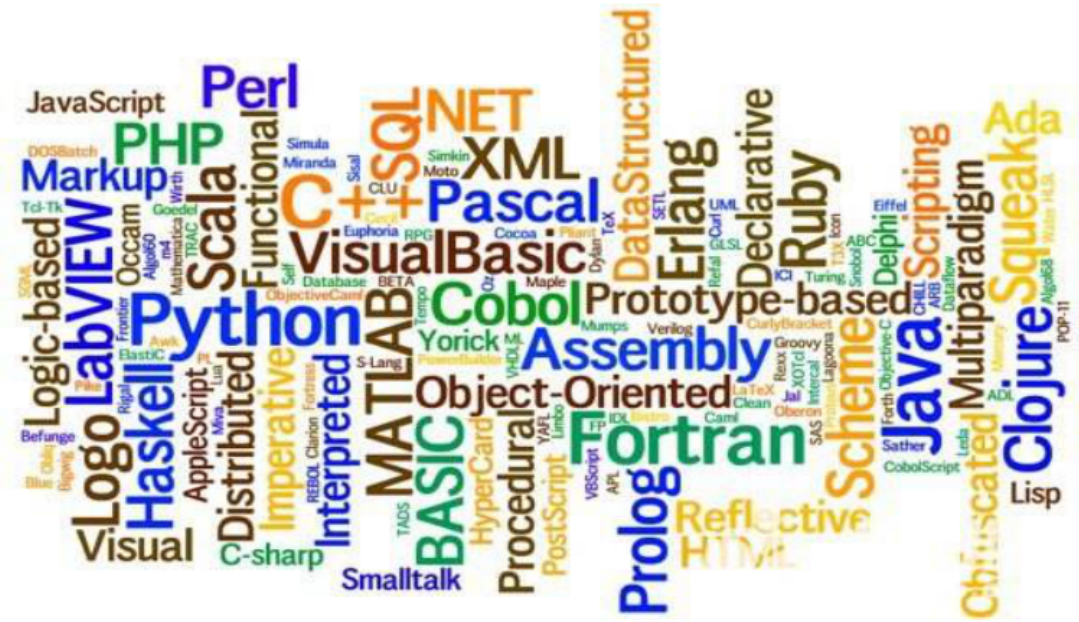
Chapter 1 Hello World!

1.1 Hello World!

1.1.1 编程语言（Programming Language）

程序是为了让计算机去解决某些问题，它由一系列指令构成。但是计算机并不能理解人类的语言，即使是最简单的，例如“计算一下 1+2 是多少”。

计算机采用的是二进制（binary），也就是只能够理解 0 和 1，因此编程语言用于作为人类与计算机之间沟通的桥梁。



通过使用编程语言来描述解决问题的步骤，从而让计算机一步一步去执行。流程图（flow chat）成为了一种程序的图形化表示方式。

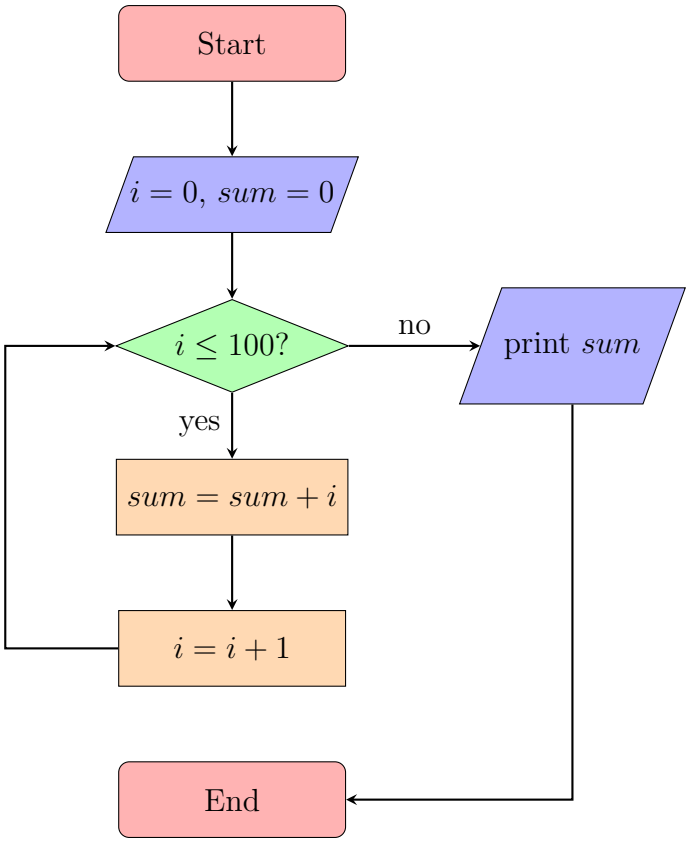


图 1.1: 计算 $\sum_{i=1}^{100} i$ 的流程图

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

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

C++

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "Hello World!" << endl;
    return 0;
}
```

Python

```
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

```
/*
* Author: Terry
* Date: 2022/11/16
*/

public class Comment {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // display "Hello World!"
    }
}
```

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	$-2^{31} \sim 2^{31} - 1$
float	4 字节	$1.2 \times 10^{-38} \sim 3.4 \times 10^{38}$
double	8 字节	$2.3 \times 10^{-308} \sim 1.7 \times 10^{308}$
char	1 字节	$-128 \sim 127$

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 wage = 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

```java
public class Rectangle {
    public static void main(String[] args) {
        int length = 10;
        int width = 5;
        double area = length * width;
        System.out.println(
            "Area = " + length + " * " + width + " = " + area
        );
    }
}
```

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

```java
import java.util.Scanner;

public class Circle {
    public static void main(String[] args) {
        final double PI = 3.14159;

        Scanner scanner = new Scanner(System.in);

        System.out.print("Radius: ");
        double r = scanner.nextDouble();

        double area = PI * Math.pow(r, 2);
        System.out.printf("Area = %.2f\n", area);

        scanner.close();
    }
}
```

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

```java
import java.util.Scanner;

public class Reverse {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a 3-digit integer: ");
        int num = scanner.nextInt();

        int a = num / 100;
        int b = num / 10 % 10;
        int c = num % 10;

        System.out.println("Reversed: " + (c * 100 + b * 10 + a));

        scanner.close();
    }
}
```

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

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

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
$<$	<
$>$	>
$\leq$	<=
$\geq$	>=
$=$	==
$\neq$	!=

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

```java
import java.util.Scanner;

public class Leap {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a year: ");
        int year = scanner.nextInt();

        /*
            * A year is a leap year if it is
            * 1. exactly divisible by 4, and not divisible by 100;
            * 2. or is exactly divisible by 400
            */
        if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0) {
            System.out.println("Leap year");
        } else {
            System.out.println("Common year");
        }

        scanner.close();
    }
}
```

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

```java
import java.util.Scanner;

public class Character {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a character: ");
        char c = scanner.next().charAt(0);

        if (c >= 'a' && c <= 'z') {
            System.out.println("Lowercase");
        } else if (c >= 'A' && c <= 'Z') {
            System.out.println("Uppercase");
        } else if (c >= '0' && c <= '9') {
            System.out.println("Digit");
        } else {
            System.out.println("Special character");
        }

        scanner.close();
    }
}
```

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
```

```
            );
            break;
        case '/':
            System.out.printf(
                    "%d / %d = %d\n", num1, num2, num1 / num2
            );
            break;
        default:
            System.out.println("Error! Operator is not supported");
            break;
        }
    }
}
```

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。

```
do {
    // code
} while(condition);
```

整数位数

```
import java.util.Scanner;

public class Digits {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter an integer: ");
        int num = scanner.nextInt();
        scanner.close();

        int n = 0;

        do {
            num /= 10;
            n++;
        } while (num != 0);

        System.out.println("Digits: " + n);
    }
}
```

运行结果

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

猜数字

```java
import java.util.Random;
import java.util.Scanner;

public class Guess {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();

        // generate random number between 1 and 100
        int answer = random.nextInt(100) + 1;

        int num = 0;
        int cnt = 0;

        do {
            System.out.print("Guess a number: ");
            num = scanner.nextInt();
            cnt++;

            if (num > answer) {
                System.out.println("Too high");
            } else if (num < answer) {
                System.out.println("Too low");
            }
        } while (num != answer);

        System.out.println("Correct! You guessed " + cnt + " times.");
        scanner.close();
    }
}
```

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。

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

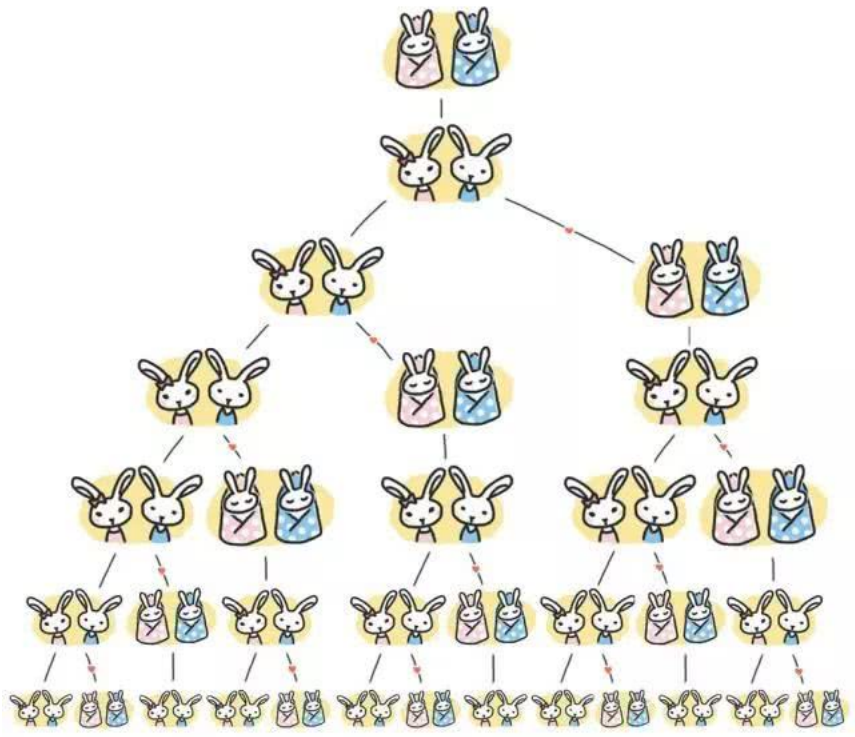
累加

```
public class Sum {
    public static void main(String[] args) {
        int sum = 0;
        for (int i = 1; i <= 100; i++) {
            sum += i;
        }
        System.out.println("Sum = " + sum);
    }
}
```

运行结果

```
Sum = 5050
```

斐波那契数列



```java
import java.util.Scanner;

public class Fibonacci {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of terms: ");
        int n = scanner.nextInt();
        scanner.close();

        if (n == 1) {
            System.out.println("1");
        } else if (n == 2) {
            System.out.println("1, 1");
        } else {
            int num1, num2, val;
            num1 = 1;
            num2 = 1;
            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

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

打印图案

```
*
**
***
****
*****
```

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

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");
```

```
        }
    }
}
```

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

```java
import java.util.Scanner;

public class Multiples {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int n = 10;
        System.out.print("Enter " + n + " integers: ");

        int sum_square = 0;
        for (int i = 0; i < n; i++) {
            int num = scanner.nextInt();
            if (num < 0) {
                continue;
            }
            sum_square += num * num;
        }

```

```
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 }
```

查找数据

```java
import java.util.Scanner;

public class Search {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of elements: ");
        int n = scanner.nextInt();

        int[] arr = new int[n];
        System.out.print("Enter the elements: ");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }

        System.out.print("Enter the key: ");
        int key = scanner.nextInt();
        scanner.close();

        boolean found = false;

        for (int i = 0; i < n; i++) {
            if (arr[i] == key) {
                found = true;
                break;
            }
        }

        if (found) {
            System.out.println(key + " exists.");
        } else {
            System.out.println(key + " not found.");
        }
    }
}
```

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

```java
public class MaxMin {
    public static void main(String[] args) {
        int[] num = {7, 6, 2, 9, 3, 1, 4, 0, 5, 8};
        int n = num.length;
        int max = num[0];
        int min = num[0];

        for (int i = 1; i < n; i++) {
            if (num[i] > max) {
                max = num[i];
            }
            if (num[i] < min) {
                min = num[i];
            }
        }

        System.out.println("Max = " + max);
        System.out.println("Min = " + min);
    }
}
```

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。

```java
int[][] arr = new int[3][4];
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。

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

对于容量较大的二维数组，可以通过两层循环进行初始化。

```java
int[][] arr = new int[3][4];

for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        arr[i][j] = 0;
    }
}
```

矩阵运算

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

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

```java
public class Matrix {
    public static void main(String[] args) {
        int[][] A = {
                {1, 3},
                {1, 0},
                {1, 2}
        };
        int[][] B = {
                {0, 0},
                {7, 5},
                {2, 1}
        };
        int[][] C = new int[3][2];

        System.out.println("Matrix Addition");
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 2; j++) {
                C[i][j] = A[i][j] + B[i][j];
                System.out.printf("%3d", C[i][j]);
            }
            System.out.println();
        }

        System.out.println("Matrix Subtraction");
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 2; j++) {
                C[i][j] = A[i][j] - B[i][j];
                System.out.printf("%3d", C[i][j]);
            }
            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	NUT	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 = {'h', 'e', 'l', 'l', 'o'};
```

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 = "hello";
```

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

```java
import java.util.Scanner;

public class StringIO {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter string 1: ");
        String str1 = scanner.nextLine();
        System.out.println(str1);

        System.out.print("Enter string 2: ");
        String str2 = scanner.next();
        System.out.printf("%s", str2);

        scanner.close();
    }
}
```

运行结果

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

字符统计

```java
import java.util.Scanner;

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。

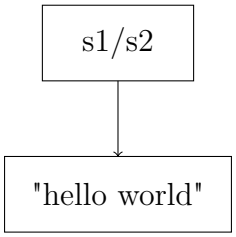
```
1 Integer num1 = Integer.valueOf("10");
2 System.out.println(num1);
3
4 int num2 = Integer.parseInt("10");
5 System.out.println(num2);
```

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。

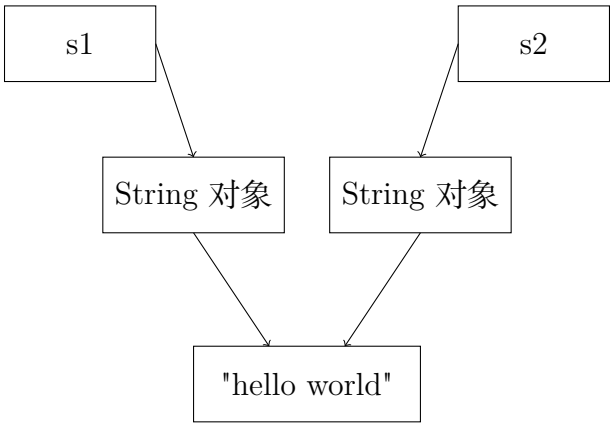
```
1 String s1 = "hello world";
2 String s2 = "hello world";
```



常量池

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");
```



常量池

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()	获取字符首次出现位置
lastIndexOf()	获取字符最后一次出现位置
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()	比较两个字符串的大小

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。

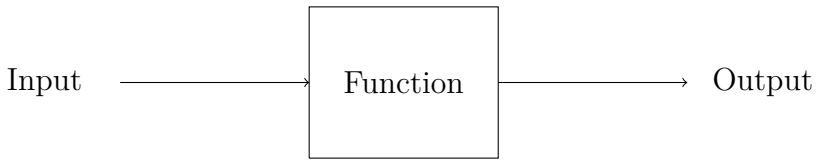


图 5.1: 函数

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

```java
public class Max {
    public static void main(String[] args) {
        System.out.println(max(4, 12));
        System.out.println(max(54, 33));
        System.out.println(max(-999, -774));
    }

    public static int max(int num1, int num2) {
        // if(num1 > num2) {
        //     return num1;
        // } else {
        //     return num2;
        // }

        return num1 > num2 ? num1 : num2;
    }
}
```

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          print_board(3, 3);
4      }
5
6      public static void print_board(int row, int col) {
7          for (int i = 0; i < row; i++) {
8              for (int j = 0; j < col - 1; j++) {
9                  System.out.print("   |");
10             }
11             System.out.println();
12
13             if (i < row - 1) {
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。

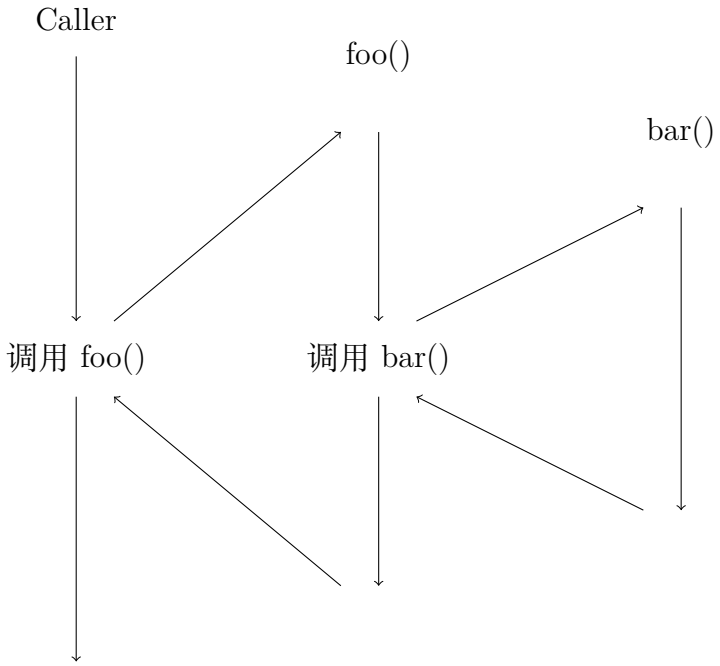


图 5.2: 函数调用

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，局部变量的生存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

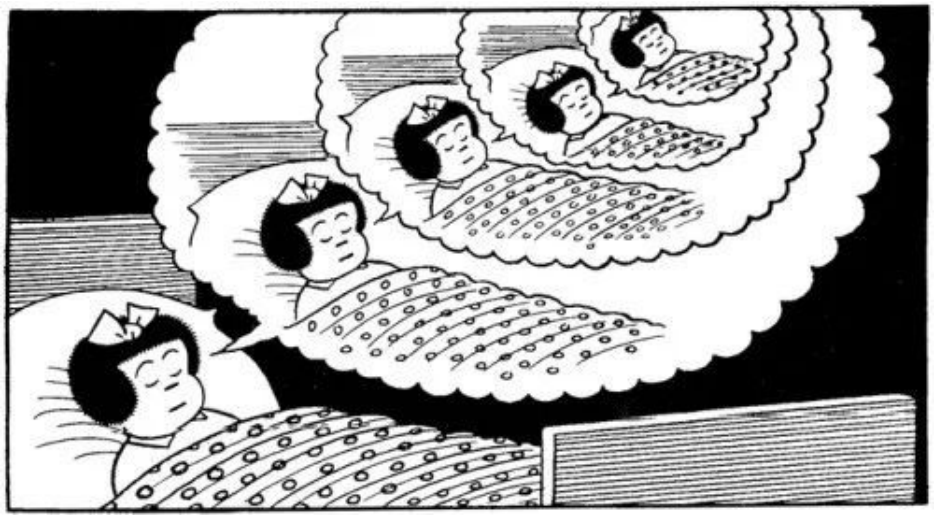
```
Before: a = 1, b = 2
swap(): a = 2, b = 1
After: a = 1, b = 2
```

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();
```

```
    }
}
```

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

```java
public class Factorial {
    public static void main(String[] args) {
        System.out.println("5! = " + factorial(5));
    }

    public static int factorial(int n) {
        if (n == 0 || n == 1) {
            return 1;
        }
        return n * factorial(n - 1);
    }
}
```

运行结果

```
5! = 120
```

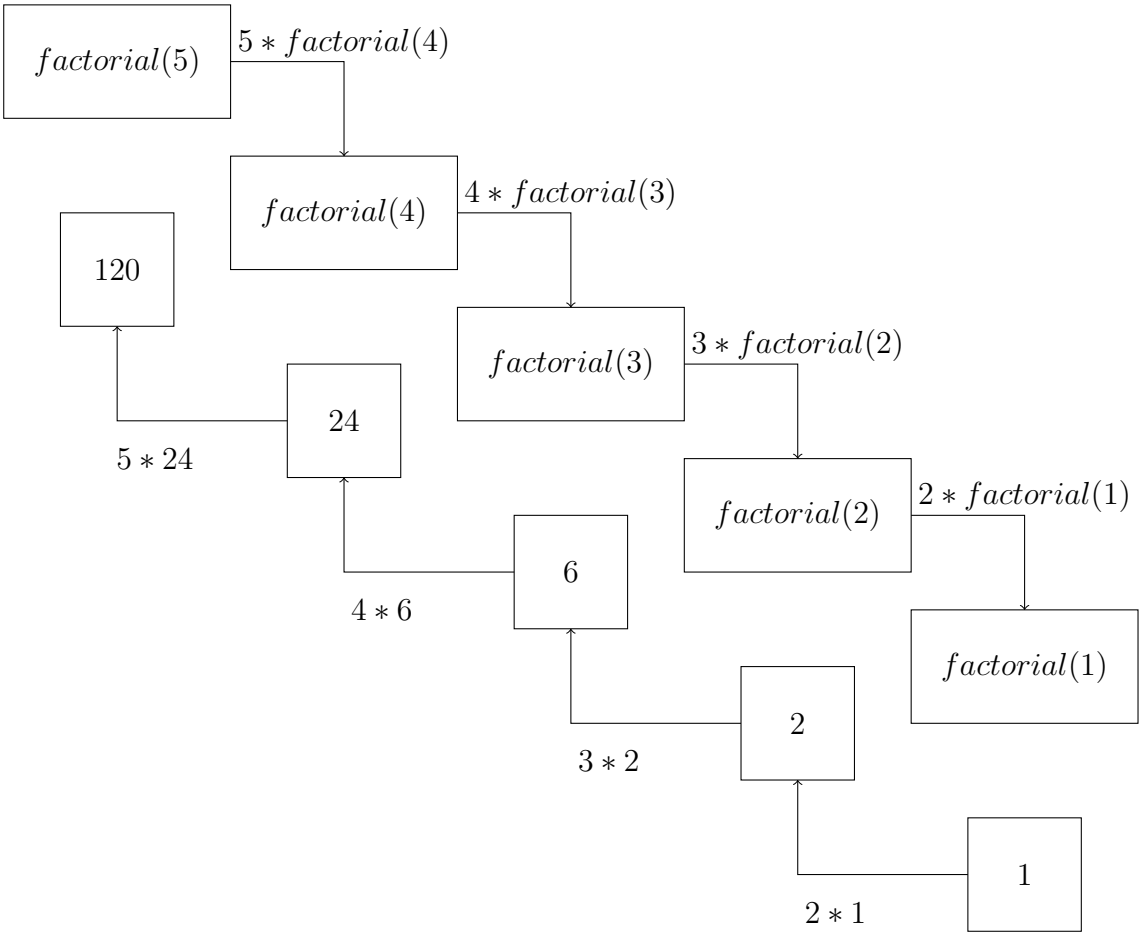


图 5.3: 阶乘

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
```

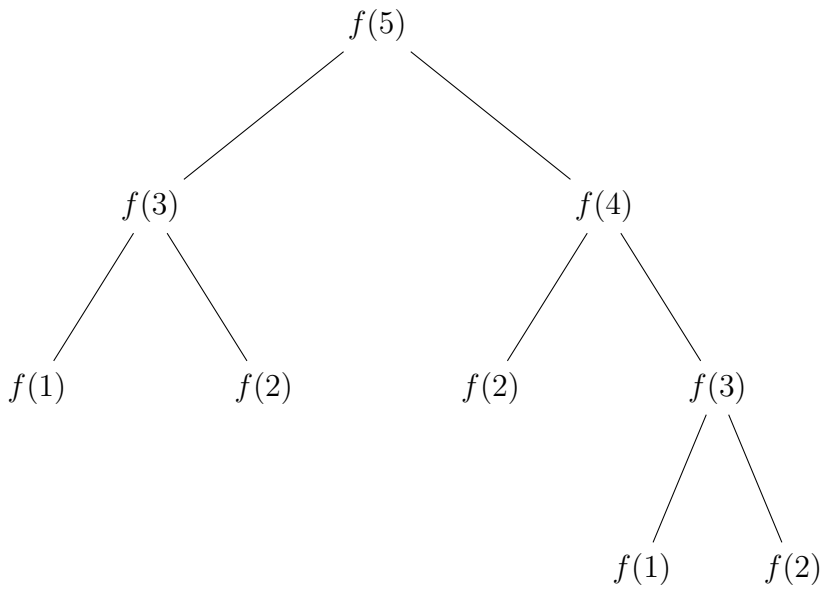


图 5.4: 递归树

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

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

```java
public class Ackermann {
    public static void main(String[] args) {
        System.out.println(A(3, 4));
    }

    public static int A(int m, int n) {
        if (m == 0) {
            return n + 1;
        } else if (m > 0 && n == 0) {
            return A(m - 1, 1);
        } else {
            return A(m - 1, A(m, n - 1));
        }
    }
}
```

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。

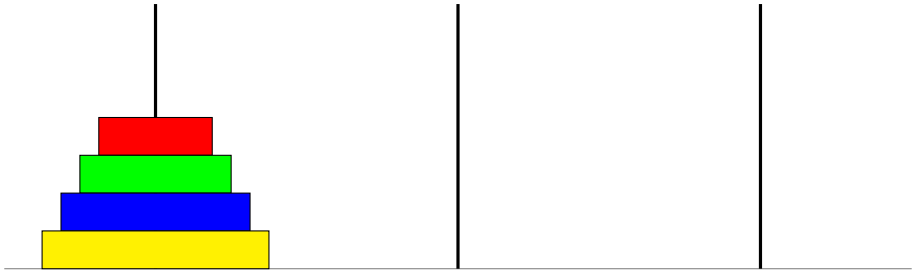


图 5.5: 汉诺塔

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  }
```

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。

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 }
```

```java
public class Bank {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        account.owner = "Terry";
        account.account = "6250941006528599";
        account.balance = 50;

        System.out.println("Owner: " + account.owner);
        System.out.println("Account: " + account.account);
        System.out.println("Balance: " + account.balance);

        account.deposit(100);
        System.out.println("Balance: " + account.balance);

        account.withdraw(70);
        System.out.println("Balance: " + account.balance);
    }
}
```

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) {
```

```
            this.account = account;
        }
    }

    public String getAccount() {
        return account;
    }

    public void setBalance(double balance) {
        if (balance >= 0) {
            this.balance = balance;
        }
    }

    public double getBalance() {
        return balance;
    }

    public boolean deposit(double amount) {
        if (amount <= 0) {
            return false;
        }
        balance += amount;
        return true;
    }

    public boolean withdraw(double amount) {
        if (amount <= 0 || amount > balance) {
            return false;
        }
        balance -= amount;
        return true;
    }
}
```

```
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。

```
public BankAccount() {
    owner = "admin";
    account = "0000000000000000";
    balance = 0;
}
```

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。

```
public BankAccount(String owner, String account, double balance) {
    if (!owner.isEmpty()) {
        this.owner = owner;
    }

    if (account.length() == ACCOUNT_DIGITS) {
        this.account = account;
    }

    if (balance >= 0) {
        this.balance = balance;
    }
}
```

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

```
public class BankAccount {
    private final int ACCOUNT_DIGITS = 16;

    private String owner;
    private String account;
    private double balance;

    public BankAccount() {
        owner = "admin";
        account = "0000000000000000";
        balance = 0;
    }

    public BankAccount(String owner, String account, double balance) {
        if (!owner.isEmpty()) {
            this.owner = owner;
        }

        if (account.length() == ACCOUNT_DIGITS) {
            this.account = account;
        }

        if (balance >= 0) {
            this.balance = balance;
        }
    }

    public void setOwner(String owner) {
        if (!owner.isEmpty()) {
            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) {
```

```
            return false;
        }
        balance -= amount;
        return true;
    }

    public boolean withdraw(double amount, double fee) {
        if(amount <= 0 || amount + fee > balance) {
            return false;
        }

        balance -= amount + fee;
        return true;
    }
}
```

```
public class Bank {
    public static void main(String[] args) {
        BankAccount account1 = new BankAccount();
        System.out.println(
                "Account 1 Owner: " + account1.getOwner()
        );
        System.out.println(
                "Account 1 Account: " + account1.getAccount()
        );
        System.out.println(
                "Account 1 Balance: " + account1.getBalance()
        );

        BankAccount account2 = new BankAccount(
                "Terry", "6250941006528599", 50
        );
        System.out.println(
                "Account 2 Balance: " + account2.getBalance()
        );

```

```
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: 0000000000000000
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。

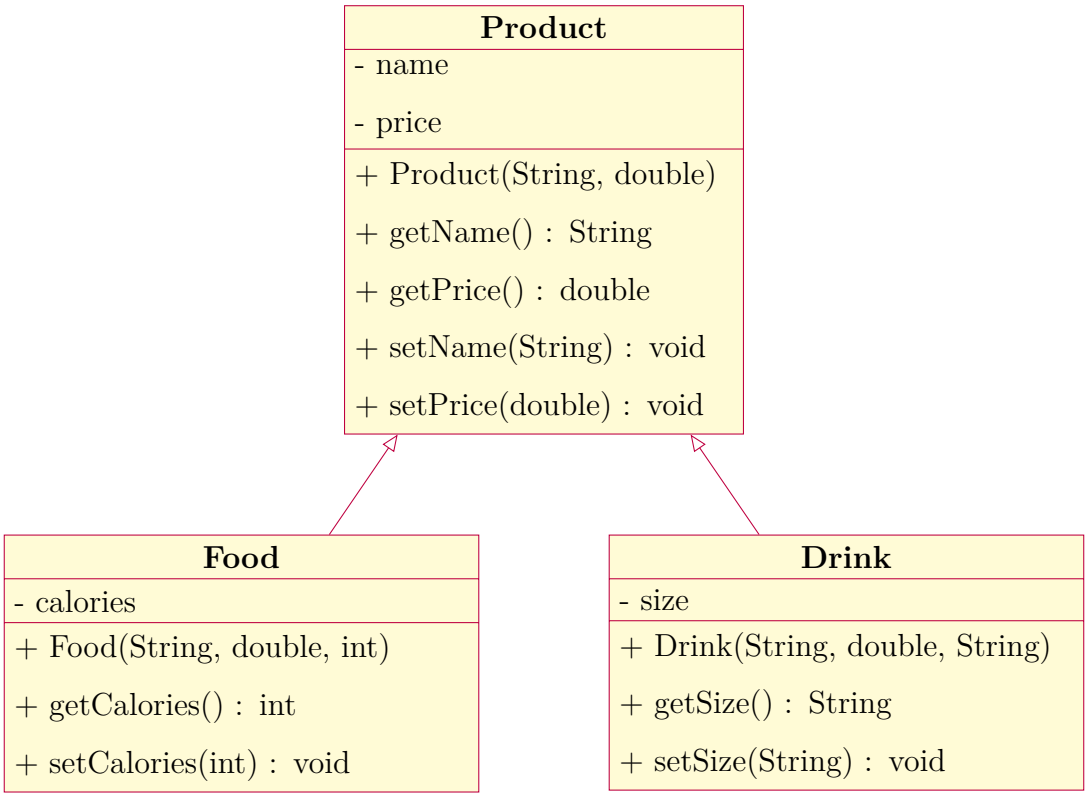


图 6.1: 继承

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 {
```

```
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }
}
```

```
public class Food extends Product {
    int calories;

    public Food(String name, double price, int calories) {
        super(name, price);
        this.calories = calories;
    }

    public int getCalories() {
        return calories;
    }
```

```
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。

麦当劳

```
1  public class Product {
2      private String name;
3      private double price;
4
```

```
    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return String.format("%s ($%.2f)", name, price);
    }
}
```

```
public class Food extends Product {
    int calories;

    public Food(String name, double price, int calories) {
        super(name, price);
        this.calories = calories;
    }

    public int getCalories() {
```

```
        return calories;
    }

    public void setCalories(int calories) {
        this.calories = calories;
    }

    @Override
    public String toString() {
        return "Food: " + super.toString() + " " + calories + " Kcal";
    }
}
```

```
public class Drink extends Product {
    private String size;

    public Drink(String name, double price, String size) {
        super(name, price);
        this.size = size;
    }

    public String getSize() {
        return size;
    }

    public void setSize(String size) {
        this.size = size;
    }

    @Override
    public String toString() {
        return "Drink: " + super.toString() + " " + size;
    }
}
```

```java
public class McDonalds {
    public static void main(String[] args) {
        Food food = new Food("Cheeseburger", 5.45, 302);
        Drink drink = new Drink("Coke", 3.7, "Large");

        System.out.println(food);
        System.out.println(drink);
    }
}
```

运行结果

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

6.3.3 equals()

【==】运算符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。

equals() 方法默认返回地址比较，通过重写 equals() 方法，可以自定义两个对象的等值比较规则。

重写 equals()

Dog.java

```java
public class Dog {
    private String name;
    private int age;
    private String type;

    public Dog(String name, int age, String type) {
        this.name = name;
        this.age = age;
        this.type = type;
```

```
    }

    /**
     * 自定义规则：实现两个对象的等值比较
     * @param obj - 需要比较的对象
     * @return 比较的结果：相同true，不同false
     */
    @Override
    public boolean equals(Object obj) {
        // 1. 如果两个对象地址相同，返回true
        if(this == obj) {
            return true;
        }

        // 2. 如果obj是null，返回false
        if(obj == null) {
            return false;
        }

        // 3. 如果两个对象类型不同，返回false
        if(this.getClass() != obj.getClass()) {
            return false;
        }

        // 4. 如果两个对象中的属性全部相同，返回true，否则返回false
        Dog dog = (Dog)obj;
        return this.name.equals(dog.name)
                && this.age == dog.age
                && this.type.equals(dog.type);
    }
}
```

TestDog.java

```
public class TestDog {
    public static void main(String[] args) {
        Dog dog1 = new Dog("狗子", 3, "哈士奇");
        Dog dog2 = new Dog("狗子", 3, "哈士奇");
```

```
5
6           System.out.println(dog1 == dog2);
7           System.out.println(dog1.equals(dog2));
8       }
9   }
```

运行结果

```
false
true
```

Chapter 7 多态

7.1 抽象类

7.1.1 多态（Polymorphism）

多态是同一个行为具有多个不同表现形式或形态的能力。例如可以把一只哈士奇，当成它的父类去看待，因此哈士奇是一只狗、一个动物或一个生物。

通过父类引用指向子类对象，例如 Animal animal = new Dog()，从而产生多种形态。父类引用仅能访问父类所声明的属性和方法，不能访问子类独有的属性和方法。

在一对有继承关系的类中都有一个方法，其方法名、参数列表、返回值均相同，通过调用方法实现不同类对象完成不同的事件。

7.1.2 抽象类（Abstract Class）

抽象类不能被用于实例化对象，只是提供了所有的子类共有的部分。例如在动物园中，存在的都是“动物”具体的子类对象，并不存在“动物”对象，所以动物类不应该被独立创建成对象。

抽象类的作用是可以被子类继承，提供共性的属性和方法。

7.1.3 抽象方法（Abstract Method）

父类提供的方法很难满足子类不同的需求，如果不定义该方法，则表示所有的子类都不具有该行为。如果定义该方法，所有的子类都在重写，那么这个方法在父类中是没有必要实现的，显得多余。

被 abstract 关键字修饰的方法称为抽象方法。抽象方法只有声明，没有实现。抽象方法只能包含在抽象类中。产生继承关系后，子类必须重写父类中所有的抽象方法，否则子类还是抽象类。

非抽象类在继承自一个抽象父类的同时，必须重写实现父类中所有的抽象方法。因此，抽象类可以用来做一些简单的规则制定。在抽象类中制定一些规则，要求所有的子类必须实现，约束所有子类的行为。

但是，类是单继承的，一个类有且只能有一个父类，所以如果一个类需要受到多种规则的约束，无法再继承其它父类。此时可以使用接口进行这样的复杂的规则制定。

7.2 对象转型

7.2.1 对象转型

对象由子类类型转型为父类类型，即是向上转型。向上转型是一种隐式转换，一定会转型成功。向上转型后的对象，只能访问父类中定义的成员。

由父类类型转型转型为子类类型，即是向下转型。向下转型存在失败的可能性，会出现 ClassCastException 异常。向下转型需要进行强制类型转换，是一个显式转换。向下转型后的对象，将可以访问子类中独有的成员。

向下转型存在失败的可能性。如果引用实际指向的对象，不是要转型的类型，此时强制转换会出现 ClassCastException 异常。所以，在向下转型之前，最好使用 instanceof 关键字进行类型检查。

对象转型

Animal.java

```java
public abstract class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }

    public abstract void makeSound();
}
```

Dog.java

```java
public class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    @Override
```

```
7       public void makeSound() {
8           System.out.println("汪汪~");
9       }
10  }
```

TestDog.java

```
1   public class TestDog {
2       public static void main(String[] args) {
3           Animal animal = new Dog("狗子");
4           if(animal instanceof Dog) {
5               Dog dog = (Dog)animal;
6               dog.makeSound();
7           }
8       }
9   }
```

运行结果

汪 汪~

7.3 接口

7.3.1 接口

在面向对象中会使用抽象类为外部提供一个通用的、标准化的接口。

宏观上来讲，接口是一种标准。例如常见的 USB 接口，电脑通过 USB 接口连接各种外设设备，每一个接口不用关心连接的外设设备是什么，只要这个外设设备实现了 USB 的标准，就可以连接到电脑上。

从程序上来讲，接口代表了某种能力和约定。当父类的方法无法满足子类需求时，可实现接口扩充子类的能力，接口中方法的定义代表能力的具体要求。

定义接口需要使用关键字 interface，接口中可以定义：

1. 属性：默认都是静态常量，访问权限都是 public。
2. 方法：默认都是抽象方法，访问权限都是 public。

接口和抽象类的相同点有：

1. 都不能创建对象。
2. 都具备 Object 类中所定义的方法。
3. 都可以写抽象方法。

接口和抽象类的不同点有：

1. 接口中所有的属性都是公开静态常量，缺省用 public static final 修饰。
2. 接口中所有的方法都是公开抽象方法，缺省用 public abstract 修饰。
3. 接口中没有构造方法、构造代码段和静态代码段。

因为接口中有很多抽象方法，因此非抽象类在实现接口的时候必须重写实现接口中所有的抽象方法。

使用接口可以进行对行为的约束和规则的制定，接口表示一组能力，那么一个类可以接受多种能力的约束。因此一个类可以实现多个接口，实现多个接口的时候，必须要把每一个接口中的方法都实现。如果一个类实现的多个接口中有相同的方法，实现类只需实现一次即可。

接口

USB.java

```java
public interface USB {
    /**
     * USB接口返回当前连接设备的类型
     * @return 当前连接设备
     */
    String getDeviceInfo();
}
```

Mouse.java

```java
public class Mouse implements USB {
    @Override
    public String getDeviceInfo() {
        return "mouse";
    }
}
```

Keyboard.java

```java
public class Keyboard implements USB {
    @Override
    public String getDeviceInfo() {
        return "keyboard";
    }
}
```

Computer.java

```java
public class Computer {
    // 电脑有2个USB接口
    private USB usb1;
```

```
4       private USB usb2;
5
6       public void setUsb1(USB usb1) {
7           this.usb1 = usb1;
8       }
9
10      public void setUsb2(USB usb2) {
11          this.usb2 = usb2;
12      }
13
14      /**
15       * 获取USB接口连接设备的信息
16       */
17      public String getUsbInfo() {
18          return "USB 1: " + this.usb1.getDeviceInfo() + "\n"
19                  + "USB 2: " + this.usb2.getDeviceInfo();
20      }
21  }
```

TestUsb.java

```
1   public class TestUsb {
2       public static void main(String[] args) {
3           Computer computer = new Computer();
4
5           // 外设设备连接到电脑上
6           computer.setUsb1(new Mouse());
7           computer.setUsb2(new Keyboard());
8
9           System.out.println(computer.getUsbInfo());
10      }
11  }
```

运行结果

```
USB 1: mouse
USB 2: keyboard
```

Chapter 8 异常

8.1 异常

8.1.1 异常（Exception）

异常就是程序在运行过程中出现的非正常的情况。异常本身是一个类，产生异常就是创建异常对象并抛出一个异常对象。Java 处理异常的方法是中断处理。

如果程序遇到了未经处理的异常，会导致这个程序无法进行编译或者运行。

Throwable 类用来描述所有的不正常的情况。Throwable 有两个子类：

1. Error：描述发生在 JVM 虚拟机级别的错误信息，这些错误无法被处理。
2. Exception：描述程序遇到的异常，异常是可以被捕获处理的。

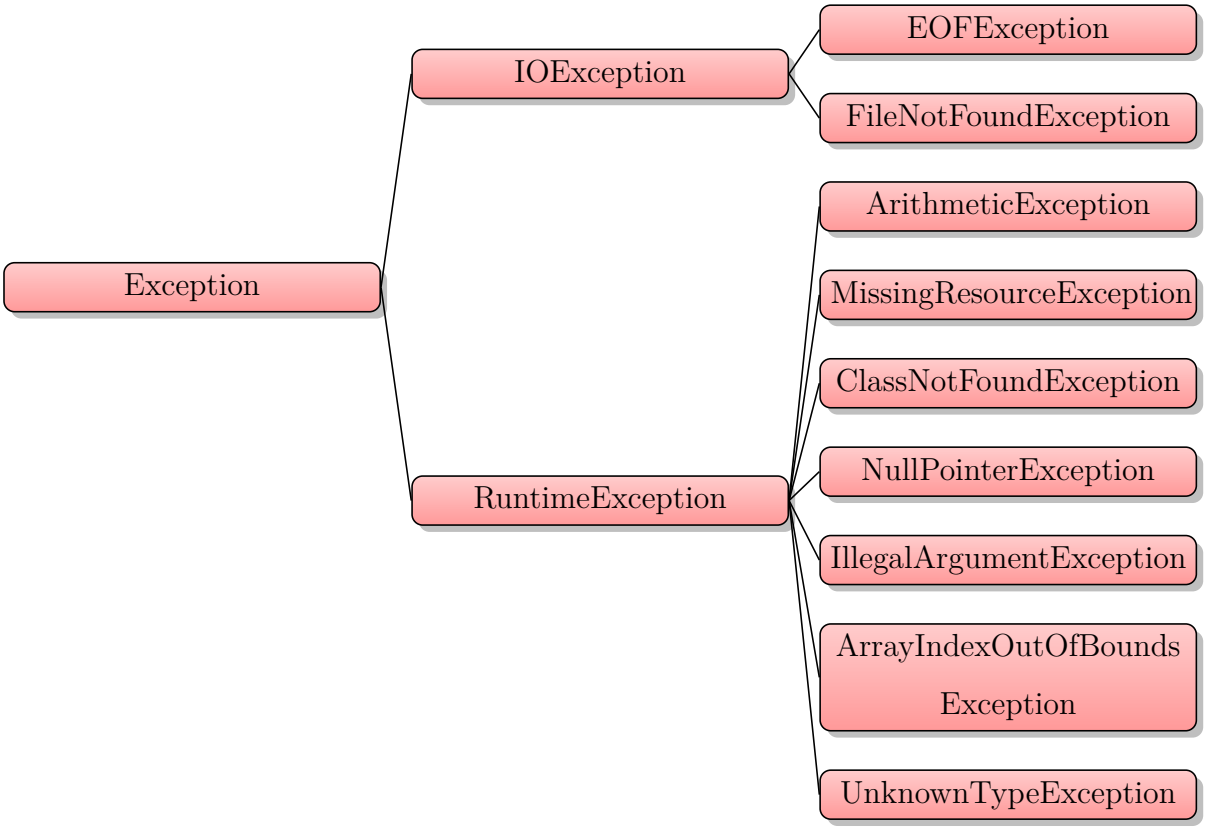


图 8.1: 异常分类

数组越界异常

```
1 public class ArrayIndexException {
2     public static void main(String[] args) {
3         int[] arr = {0, 1, 2, 3, 4};
4         System.out.println(arr[5]);
5     }
6 }
```

运行结果

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException:
    Index 5 out of bounds for length 5
```

普通的异常会导致程序无法完成编译，这样的异常被称为非运行时异常（non-runtime exception），但是由于异常是发生在编译时期的，因此常常称为编译时异常。

Exception 类有一个子类 RuntimeException 类对异常进了自动的处理，这种异常不会影响程序的编译，但是在运行中如果遇到了这种异常，会导致程序执行的强制停止。这样的异常被称为运行时异常。

8.2 异常的捕获处理

8.2.1 try-catch

如果一个异常不去处理，会导致程序无法编译或者运行。使用 try-catch 语句可以捕获并处理异常。

```
try {
    // 可能出现异常的代码
} catch(exceptionType e) {
    // 异常的类型和catch的异常的类型匹配，执行此处逻辑
}
```

一个异常如果被捕获处理了，那么将不再影响程序的执行。

如果在 try 结构中出现了异常，那么从异常出现的位置开始，try 结构中往后的代码将不再执行。

捕获数组越界异常

```
public class TryCatch {
    public static void main(String[] args) {
        int[] arr = {0, 1, 2, 3, 4};
        try {
            int elem = arr[5];
            System.out.println("elem = " + elem);
```

```
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("数组下标越界异常被捕获处理了");
        }
    }
}
```

运行结果

数组下标越界异常被捕获处理了

8.2.2 finally

finally 出现在 try-catch 结构的结尾，无论 try 代码段中有没有异常、无论 try 里面出现的异常有没有被捕获处理，finally 中的代码始终会执行。基于这个特点，常常在 finally 中进行资源释放、流的关闭等操作。

finally

```
import java.util.Scanner;

public class Finally {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int[] arr = {0, 1, 2, 3, 4};
        try {
            System.out.print("输入新数据：");
            arr[5] = scanner.nextInt();
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("数组下标越界异常被捕获处理了");
        } finally {
            System.out.println("关闭输入流...");
            scanner.close();
        }
    }
```

```
17 }
```

运行结果

```
输入新数据：5
数组下标越界异常被捕获处理了
关闭输入流...
```

8.2.3 catch

如果多个 catch 捕获的异常类型之间没有继承关系存在，此时先后顺序无所谓。

如果多个 catch 捕获的异常类型之间存在继承关系，则必须保证父类异常在后，子类异常在前。

catch

```
1  import java.util.Scanner;
2  import java.util.InputMismatchException;
3
4  public class Catch {
5      public static void main(String[] args) {
6          Scanner scanner = new Scanner(System.in);
7          try {
8              System.out.println("【除法运算】");
9              System.out.print("输入被除数：");
10             int num1 = scanner.nextInt();
11             System.out.print("输入除数：");
12             int num2 = scanner.nextInt();
13             int result = num1 / num2;
14             System.out.println("结果：" + result);
15         } catch(ArithmeticException e) {
16             System.err.println("算术异常");
17         } catch(InputMismatchException e) {
```

```
18              System.err.println("输入类型异常");
19          } finally {
20              scanner.close();
21          }
22      }
23  }
```

运行结果

【除法运算】
输入被除数：10
输入除数：0
算术异常

运行结果

【除法运算】
输入被除数：hey
输入类型异常

8.3 throw 与 throws

8.3.1 throw / throws

throw 关键字用于抛出一个异常，一般用于程序出现某种逻辑时程序员主动抛出某种特定类型的异常。

```
throw e;
```

throws 关键字用在声明方法的时候，表示该方法可能要抛出异常。定义了 throws 异常抛出类型的方法，在当前的方法中可以不处理这个异常，由调用方处理。

```
accessModifier returnType func([paramList]) throws exceptionType {
    // code
}
```

抛出异常

```
public class Person {
    private int age;

    public void setAge(int age) throws Exception {
        if(age < 0 || age > 130) {
            throw new Exception();
        }
```

```
8           this.age = age;
9       }
10
11      public static void main(String[] args) {
12          try {
13              Person person = new Person();
14              person.setAge(-1);
15          } catch(Exception e) {
16              System.out.println("年龄异常");
17          }
18      }
19  }
```

运行结果

年龄异常

8.4 自定义异常

8.4.1 自定义异常

使用异常是为了处理一些重大的逻辑 bug，这些逻辑 bug 可能会导致程序的崩溃。此时，可以使用异常机制，强迫修改这个 bug。

系统中提供了很多的异常类型，但是异常类型提供地再多，也无法满足所有的需求。当需要的异常类型系统没有提供的时候，此时就需要自定义异常了。

系统提供的每一种异常都是一个类，所以自定义异常其实就是写一个自定义的异常类。自定义的异常类，理论上来讲，类名可以任意定义，但是出于规范，一般都会以 Exception 作为结尾，例如 ArrayIndexOutOfBoundsException、NullPointerException、ArithmeticException 等。

如果要自定义一个编译时异常，需要继承自 Exception 类，如果要自定义一个运行时异常，需要继承自 RuntimeException 类。

自定义异常

AgeException.java

```
1 public class AgeException extends RuntimeException {
2     public AgeException() {
3         super("年龄异常");
4     }
5 
6     public AgeException(int age) {
7         super("年龄异常：" + age);
8     }
9 }
```

Person.java

```
1 public class Person {
2     private int age;
3 
```

```
4       public void setAge(int age) throws AgeException {
5           if(age < 0 || age > 130) {
6               throw new AgeException(age);
7           }
8           this.age = age;
9       }
10
11      public static void main(String[] args) {
12          try {
13              Person person = new Person();
14              person.setAge(-1);
15          } catch(AgeException e) {
16              e.printStackTrace();
17          }
18      }
19  }
```

运行结果

```
AgeException: 年龄异常: -1
at Person.setAge(Person.java:6)
at Person.main(Person.java:15)
```