第 2 章 Java 编程

南京大学 软件学院 刘钦

2.1 Java 语言基础

本节介绍了编程的基本概念，并对 Java 语言的特性做了简单的介绍，方便大家后续的学习。

2.1.1 编译、执行

现今运行大绝大多数计算机都是遵循图 2-1 所示的冯诺依曼体系结构。在这个结构中，数据和指令都是作为二进制数据存储在存储器中。控制器通过总线从存储器读入二进制指令，然后翻译指令、执行指令。控制运算器从存储器读取数据执行相关算术逻辑操作，再将结果存回存储器。输入输出设备通过存储器存取完成有关的 IO 操作。这其中能够被机器直接执行的二进制指令被称为机器指令。

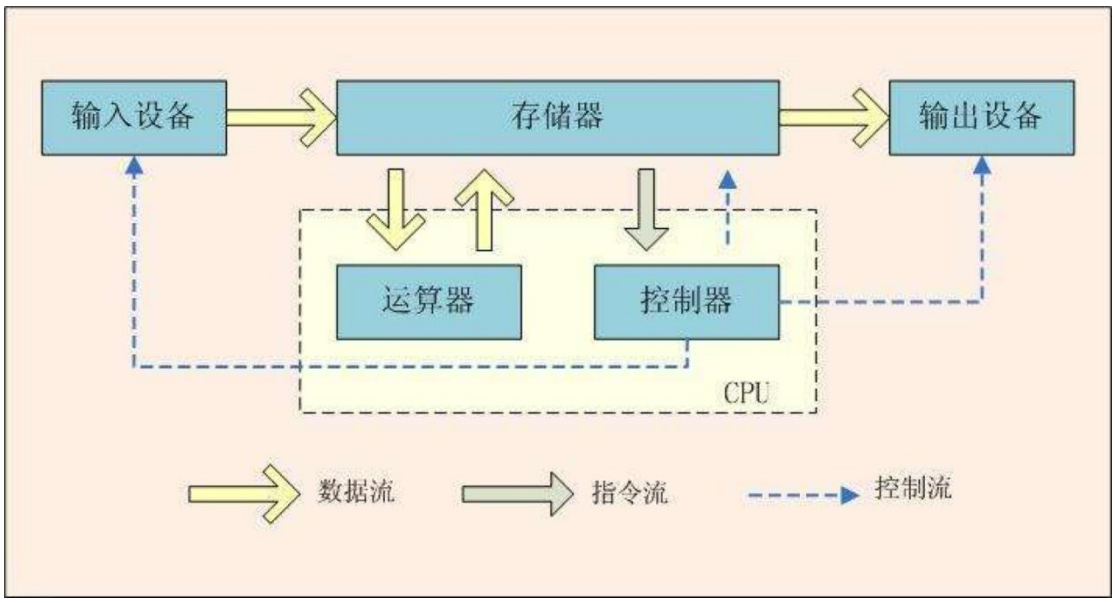

图 2-1 冯诺依曼体系结构

如图 2-2 所示，二进制的机器指令很难被人理解，离现实世界较远。所以，人们发明了中高级语言，比如 C 语言。当我们写出更容易理解的中高级语言，然后通过编译器的编译过程，转化为机器指令，再让计算机执行。执行的时候，结合用户的输入，就可以得到令人期盼的输出了。更细地讲，C 语言程序会先经过预处理，再经过编译（这个阶段应该是生成汇编程序），然后由汇编器将上一步的汇编程序编译成机器可识别的二进制程序，最后再通过链接器将一些库文件内容与当前的二进制程序打包成一个计算机可执行文件（机器指令）。

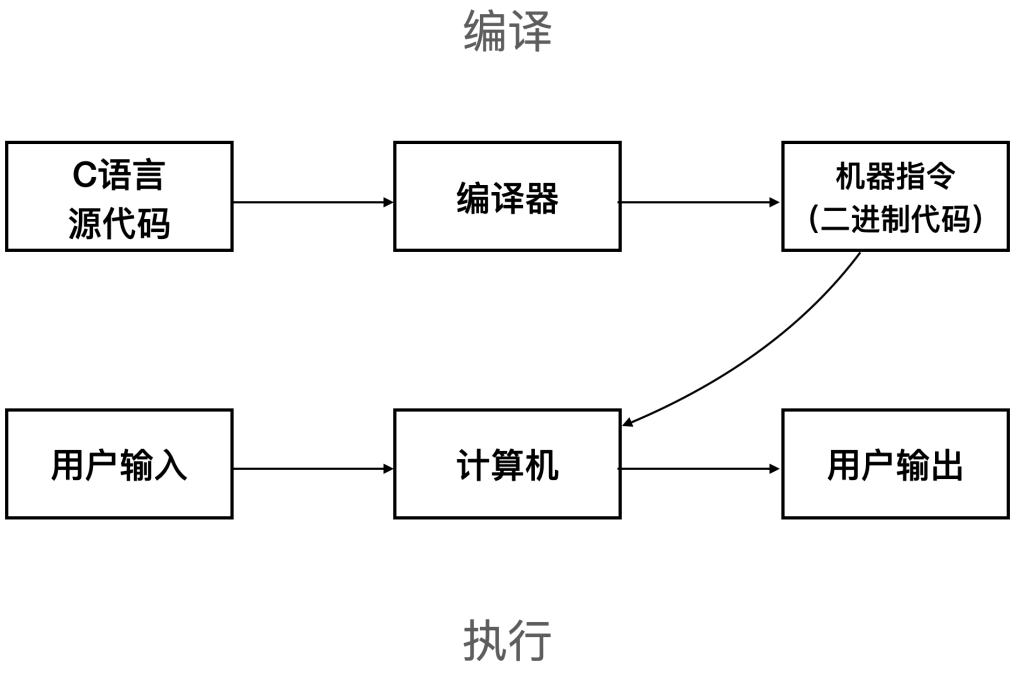

图 2-2 编译与执行

2.1.2 编译与解释

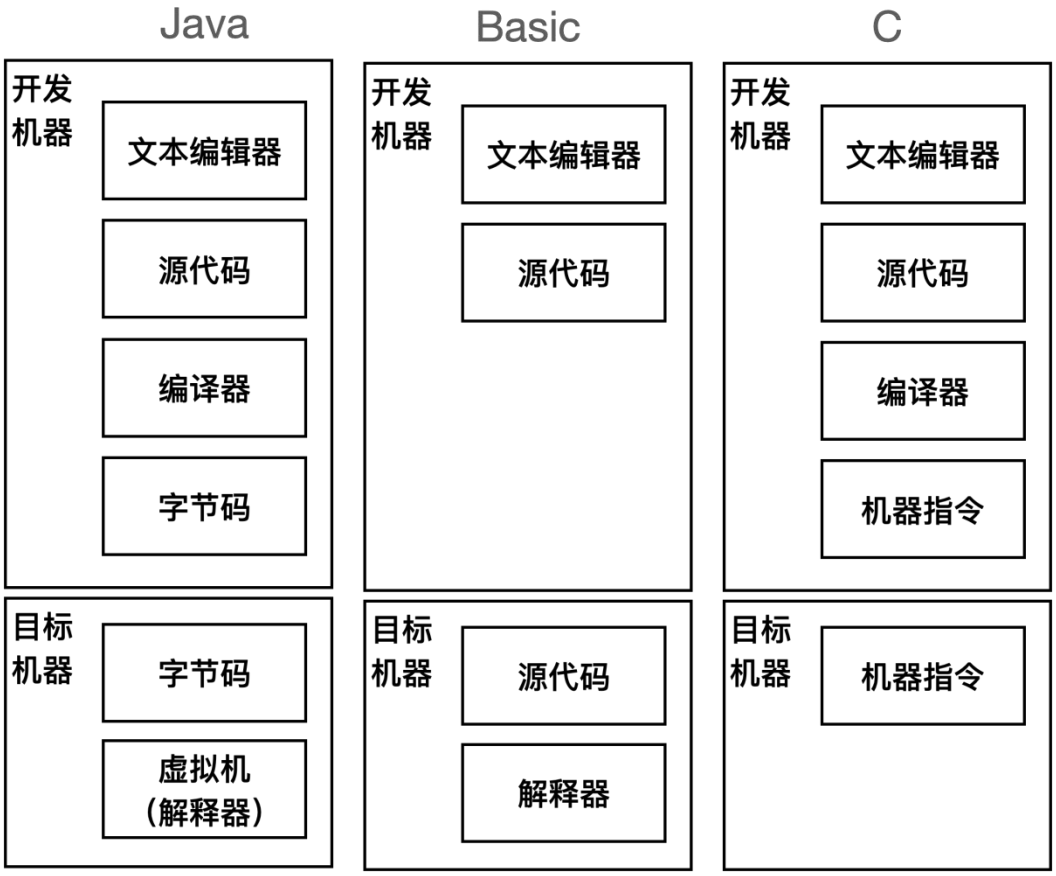

图 2-3 编译与解释

C 语言的程序编译执行过程：在开发者机器上的文字编辑器（Text Editor）或者集成开发环境 IDE 中写成源代码（Source Code），然后通过编译器（Compiler）编译成用户的目标机器的机器指令（Machine Code）。然后机器指令拷贝到目标机器就可以在用户的机器上执行了。注意这里不同的机器能够执行的机器指令是不一样的。那么我们的编译的时候就必须编译成为目标机器上可以执行的指令。这样机器指令是不能随便跨各种硬件平台来执行的。

而 Basic 语言则在写好源代码之后，直接将源代码拷贝到用户机器，在用户机器有一个解释器（Interpreter）。解释器边解释源代码，边执行结果。这样的好处是只要目标机器上有 Basic 的解释器，就可以解释执行 Basic 的源代码。虽然可以做到了 Basic 源代码的跨平台执行，但是这是以为不同硬件平台开发不同的 Basic 解释器为代价的。Basic 解释器本身不跨平台。如果编译执行，有点类似翻译中先笔译再朗读，那么解释执行就类似口译了，直接听英文翻译出中文。

语言再发展后，出现了 Java 语言这类语言。同样写好源代码之后，由编译器进行编译，不过这次并不产生机器指令，而是产生一种我们称之为“字节码（Bytecode）”的中间代码，就是 javac 命令之后产生的 class 文件。然后拷贝这个 class 文件到目标机器，通过目标机器上的 Java 虚拟机（JVM）来执行 class 文件。这样就实现了 Java 语言的跨平台性，但是 JVM 本身是不跨平台的，在安装 JVM 的时候是需要根据硬件平台选择安装的版本的。JVM 针对各种操作系统、平台都进行了定制，并且无论在哪种平台 Java 程序经编译后都可以生成固定格式的字节码供 JVM 使用，从而实现了“一次编译，到处运行”。

最初的时候解释执行启动的效率高，但是相对于编译执行的整体效率会偏低。但随着各种 JIT(just in time)、AOT（ahead of time）等新技术的出现，现在解释执行的效率较以往已经有了长足的进步。

2.1.3 Java 语言特性

如图 2-4 所示，Java 是一个静态、强类型的语言。

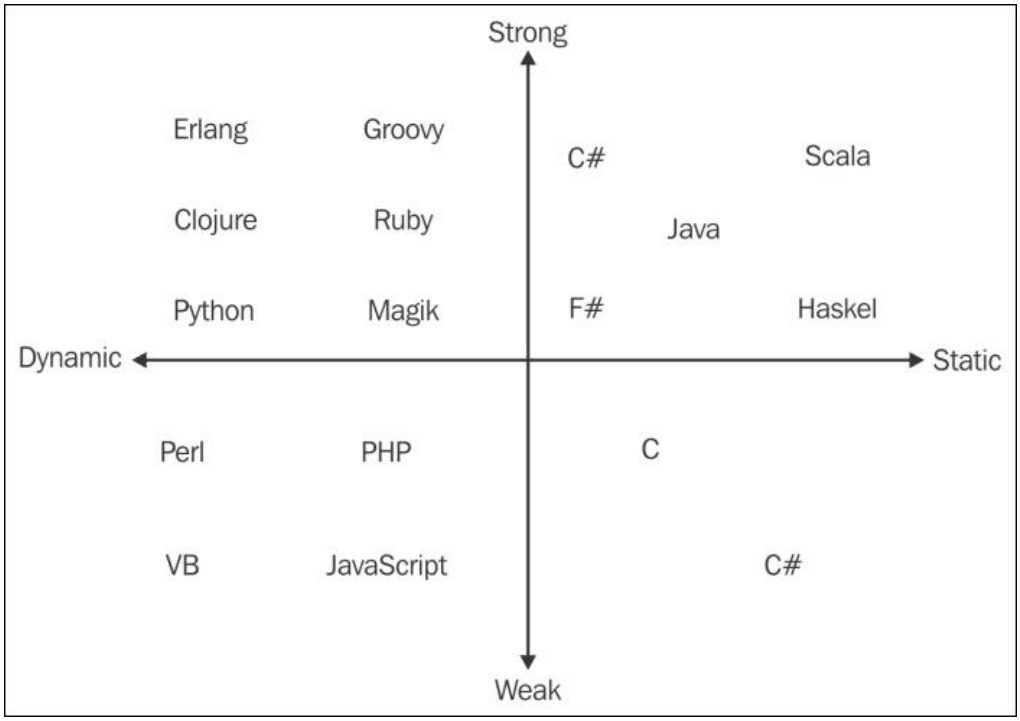

图 2-4 语言特性

所谓的静态语言是相对于动态语言来说的。看编译器/解释器是在什么时间节点怎样处理变量的。静态语言一般是在编译的时候就知道变量的类型。每个变量都对应一个类型。不可以将另外一种类型的值赋给这个变量（可以兼容的类型除外）。有的语言需要你明确声明变量的类型。

而动态语言则是将一个值绑定到一个名字上。名字本身没有类型。这样就可以将不同类型的值绑定与同一个名字之上。所以，一般直到运行的时候，才关心其类型。

下面代码中动态类型的 python 语言支持同时将“12”和 12 赋给 age；但是对于静态类型的 Go 和 Java 就会出现编译的错误。

```
#Python(Dynamic) OK
age = 12
age = "12"
//Go(static) A compile-time assignment error is raised.
age = 12
age = "12"
//Java(static) A compile-time assignment error is raised.
int age = 12;
age = "12"
```

强类型语言和弱类型语言的区别就是当不同类型的值进行运算的时候出现的。强类型语言因为会在运算之前做类型检查，当发现类型不兼容的时候，无论是运行时还是编译时都会直接报错。一般强类型的语言会对类型所占用的空间，运算可以接受的类型，以及运算的意义有明确的规定。而弱类型的语言一般编译时不会检查运算接受的类型，而是会在运行时按某种规则自动做类型的转换，而完成相应的运算。这往往就会产生意料之外的结果。

如下代码中，Java 语言的编译器会区别出“/”运算符（整数除法还是浮点数除法），对不同类型的数据输入产生两种不同的运算结果。而 Javascipt 是运行时，先进行类型转换，再计算的。10 先转换为字符串“10”，再和“K”连接。1 先转换为浮点数 1.0，再和 2.5 进行加法运算。

```
//Java(Strong)
int i =5;
System.out.println(i/2);  // 2
System.out.println(i/2.0);  // 2.5

//Javascript(Weak)
> a = 10
> b = "K"
> a + b
'10K'

> a = 1
> b = 2.5
> a + b
3.5
```

2.1.4 输入、处理、输出

从结构化编程的世界观来说，系统就是输入、处理、输出。然后，每个部分又可以再分为输入、处理、输出。从而不断的细分下去，而完成将一件复杂的事转换为若干简单的事。举一个例子，比如输入一个数，然后做这个数和 100 的加法，输出最后的和。下面是 Java 代码的实现:

```
public class SumDemo{
    public static void main(String[]  args){
        int i = 0;
        int sum = 0;
        String inputString = null;

        // 1 输入
        try {
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("please enter the first integer:”);
        inputString=br.readLine();
        } catch (IOException e) {
            e.printStackTrace();
        }

        // 2.1 处理(转为整数 int)
        i=Integer.parseInt(inputString);
        // 2.2 处理 (求和)
        sum = i + 100;

        // 3、输出
        System.out.println(one+"+"+two+"="+sum);
    }
}
```

可以将从控制台输入数作为输入步骤 1，由输入的字符串转为整数作为处理步骤 2.1，求和作为处理步骤 2.2，最后输出到控制台是输出步骤 3。BufferedReader 类的 readLine 是负责读取控制台输入的一行数据。System.in 代表的就是标准输入，就是控制台输入。try 和 catch 是异常捕获的格式。Integer.parseInt 是完成字符串到整数的转换的语句。“+”是完成加法操作。System.out.println 是负责控制台输出的语句。

2.1.5 变量

1. 变量的类型

物理上, 变量是在计算机内存单元的一块存储空间。有一个“地址”, 里面存着一个“值”。逻辑上，变量是软件模型中的抽象数据单元。比如，上面代码中的 int i，代表一个整数。语

义上。变量有可能表达的是一个类的属性，也有可能是一个方法的临时的状态。比如上面代码中的 sum 就是临时用来存求和方法的累积和的。

Java 的变量分为两大类：基础数据类型和引用变量。不用其他类型来定义的数据类型被称为基本数据类型。几乎所有程序设计语言都提供一组基本数据类型。Java 包括以下几个类型：

(1) 整数：该组包括字节型（byte），短整型（short），整型（int），长整型（long），它们是有符号整数。

(2) 浮点型数：该组包括浮点型（float），双精度型（double），它们代表有小数精度要求的数字。

(3) 字符：这个组包括字符型（char），它代表字符集的符号，例如字母和数字。

(4) 布尔型：这个组包括布尔型（boolean），它是一种特殊的类型，表示真/假值。

基础数据类型是其他类型数据的基础。Java 是完全面向对象的，但简单数据类型不是。因为 Java 可移植性的要求，所有的数据类型都有一个严格的定义的范围。

如下面代码所示，Java 在使用变量的时候首先得声明变量，然后可以赋值，可以运算。给变量赋值的时候，代表着把相应的值存储到变量对应的内存空间去。变量的运算，也是变量的空间存储的值的运算。变量的一个意义就是要暂存相应的状态，方便后续的代码使用。

```
// 计算圆的面积
class AreaDemo {
    public static void main(String args[]) {
        double r, area;  //变量声明
        static final int PI = 3.141593;

        r = 10.8; // 给变量赋值，r 代表圆的半径
        area = PI* r * r; // 变量运算

        System.out.println("Area of circle is " + area);
    }
}
```

在 JAVA 中，变量名可以有不同的长度，但是它必须由字母、下划线（_），或者美元符（$）开头，而其余部分可以是除了 JAVA 运算符（比如+、-或者*）之外的任何字符，但是通常最好只使用字母、数字和下划线字符。Java 是区分大小写的。“pi”和“Pi”代表不同的变量。在名字中间不能包括空格或者制表符。类常量的声明，应该全部大写，单词间用下划线隔开。例如 static final int MIN_WIDTH = 4;

2. 作用域和生命周期

程序变量的作用域是语句的一个范围，在这个范围之内，变量为可见的。如果一个变量在一条语句中可以被引用，这个变量即在这条语句中为可见的。

如果一个变量声明于一个子程序单元或程序块之内，那么它就是这个子程序单元或程序块内的局部变量。程序单元或程序块的非局部变量是指不在这个程序单元或程序块中声明、但又对其可见的变量。

下面代码中 j 被声明了 2 次，这样就无法判断 i+j == 10 那句中 j 到底等于多少，所以，编译就会报错。

```
int num = 0, i, j;
for (i = 1; i <= 10; i++) {
    for(int j = 1; j <= i; j++) {       /* 错误：变量 j 已经声明过 */
        if (i + j == 10)
            num++;
    }
}
```

而在下面代码中，类 ScopeDemo 定义 k 作为成员变量，那么整个类都是 k 的作用域。而在 f()方法中也定义了一个 k。这个 k 的作用域就在这个方法中有效。所以，如果我们在这个方法中想要引用类的成员变量 k。我们就要通过 this.k 这样的方式。

```
class ScopeDemo{
    int k = 0;
    void f() {
        int k = 1;
        int m = k;          /* m 的值为 1 */
        int n = this.k;     /* n 的值为 0 */
    }
}
```

而对于局部变量来说，它的生命期，就是它在时间上的跨度，就到包含他的方法结束。比如，k 就到 f()方法的最后一行代码执行解释，k 的空间就被释放了。

而 this.k 的生命周期是随着创建这个类的对象的生命周期来定的。后续面向对象部分还会补充。

3. 局部变量的语义

作为一个局部变量，一般有以下一些常见的使用场景：收集器、计数、解释、复用、元素。

搜集器作用类似对 1 到 100 求和过程中，保存的 sum 这个临时结果。

计数作用就是 for 循环中的 int i，来记录循环的次数。

top、left、height、width 这四个变量是在表达现实世界中的长方形的左上的坐标和宽和高。所以，是现实世界的解释，表征。

```
int top=0;
int left=0;
int height=100;
int width=100;
Rectangle rect = new Rectangle(top,left,height,width);
```

如果像下面第一段代码那样每次要调用同样一个方法 getCurrentTime()获得同样一个值的时候，可以像第二段代码先用一个值 now 来暂存，作为之后复用的对象。

```
for ( Clock each : getClocks())
   each.setTime ( getCurrentTime());
```

```
long now = getCurrenttime();
for ( Clock each : getClocks())
   each.setTime ( now);
```

下面代码中的 eachSender 和 eachReceiver 就是用来代表我们集合中的一个元素，利于对集合的遍历。

```
broadcast(){
    for(Source eachSender: getSenders())
        for(Source eachReceiver: getReceivers())
            …
}
```

2.1.6 操作符与表达式语句

1. 操作符简介

Java 有如下操作符：

(1) 赋值操作符：“=”
(2) 算术操作符：“+（加法）、-（减法）、*、/、%”
(3) 单操作符：“+（正号）、-（负号）、++、--、！（逻辑非）”
(4) 关系操作符：“==、！=、>、>=、<、<=”
(5) 条件操作符：“&&、||”
(6) 位操作符、位移操作符：“<<、>>、>>>、&、^、|”

操作符和数学运算符类似有优先级。比如，x=7+3*2;这句 x 最后被赋值为 13，而不是 20。因为乘法的优先级，高于加法。

```
class ArithmeticDemo {

    public static void main (String[] args) {

        int result = 1 + 2;
        // result is now 3
        System.out.println("1 + 2 = " + result);
        int originalResult = result;

        result = result - 1;
        // result is now 2
        System.out.println(originalResult + " - 1 = " + result);
        originalResult = result;

        result = result * 2;
        // result is now 4
        System.out.println(originalResult + " * 2 = " + result);
        originalResult = result;

        result = result / 2;
```

```
        // result is now 2
        System.out.println(originalResult + " / 2 = " + result);
        originalResult = result;

        result = result + 8;
        // result is now 10
        System.out.println(originalResult + " + 8 = " + result);
        originalResult = result;

        result = result % 7;
        // result is now 3
        System.out.println(originalResult + " % 7 = " + result);
    }
}
```

上面代码的结果是:

```
1 + 2 = 3
3 - 1 = 2
2 * 2 = 4
4 / 2 = 2
2 + 8 = 10
10 % 7 = 3
```

其中“result = result + 8;”也可以写作“result+=8;”。

对于数值变量来说，由于各个变量的所占的二进制位数是规定的，所以必然会造成越界问题。如下面代码中，1000000*1000000 就超出了 int 的界限。我们必须要用 long 才能存储整个结果，否则会产生截断，输出 -727379968。用 long 之后会输出期望的 1000000000000。由于最后 l-i 等于 0。最后程序会抛出 ArithmeticException 这个异常。

```
class FloatDemo1{
     public static void main(String[] args) {
             int i = 1000000;
             System.out.println(i * i);
             long l = i;
             System.out.println(l * l);
             System.out.println(20296 / (l - i));
     }
}
```

关于浮点数，我们还得关心溢出的问题，包括上溢和下溢。我们得有这样一个认识。计算机表达的浮点数，由于位数的限制，我们表达的所有浮点数并不是占据数轴上所有的点，而是分布在数轴上的一些离散的点。双精度浮点数由于位数多，所以表达精度高，相对于单精度浮点数来说，在数轴上占据的数更多，也更密一点。

另一个浮点数的比较，如果用“==”进行比较，那就是精确的比较，比较数字的每一位，但是由于我们并不能表达数轴上所有的点，所以必然会有精度损失。所以 1.0f/i*i 未必会精确等于 1.0f。而 1.0/i*i 和 1.0 进行比较，因为运算用的是双精度浮点数，所以精度更高，所以损失的概率更低，所以，最后在代码中输出的有损失的数就少。

(int)d 这句其实是强制将浮点数转为整数，就会截断小数部分。

```
class FloatDemo2{
         public static void main(String[] args) {
             for (int i = 0; i < 100; i++) {
                 float z = 1.0f / i;
                 if (z * i != 1.0f)
                     System.out.print(" " + i);
             }
             System.out.println();
             for (int i = 0; i < 100; i++) {
                 double z = 1.0 / i;
                 if (z * i != 1.0)
                     System.out.print(" " + i);
             }
             System.out.println();

             d = 12345.6;
             System.out.println((int)d + " " + (int)(-d));
         }
}
```

上段代码的输出为：

```
0 41 47 55 61 82 83 94 97
0 49 98
12345 -12345
```

```
class ConditionalDemo {

   public static void main(String[] args){
      int value1 = 1;
      int value2 = 2;
      if((value1 == 1) && (value2 == 2))
         System.out.println("value1 is 1 AND value2 is 2");
      if((value1 == 1) || (value2 == 1))
         System.out.println("value1 is 1 OR value2 is 1");
   }
}
```

上例代码显示了如何使用比较操作符和逻辑操作符。

2. 三元操作符和 instanceof

除此之外，Java 中有一个三元操作符，表达式如下所示：

variable x = (expression) ? value if true : value if false

根据 expression 的值来决定最后的取值。真就取第一个值，否则假就取第二个值。

```
public class TernaryOperatorDemo{

   public static void main(String args[]) {
      int a, b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );

      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
}
```

此外，还有一个 instanceof 操作符来判别变量是否是某个类型，是返回 true，否则返回 false。

```
public class InstanceOfDemo{

   public static void main(String args[]) {

      String name = "James";

      boolean result = name instanceof String;
      System.out.println( result );
   }
}
```

3. 表达式语句

Java 中有四种表达式语句：

(1) 赋值语句，例如： “i=100;”。
(2) 自增（++）、自减（--）语句，例如： “i++;”。
(3) 方法调用语句，例如： “Integer.parseInt(“100”);”。
(4) 创建对象语句，例如： “Integer integer = new Integer(100);”。

除了表达式语句，还有声明语句（int i =0;）、控制语句(if else while for 等)，共同构成了我们语句。有了 0 个或者多个语句的顺序排列在{ }中间，就可以组成代码块。代码块就程序的基本组成部分。

2.1.7 决策

当遇到需要进行选择的场景的时候，就需要引入条件控制语句。比如，如果输入的数是偶数，则执行 A 语句，如果输入的数是奇数，则执行 B 语句。再比如，如果要计算一年有多少天，就会发现，每个月有不同的天数，在计算的时候，就会计入不同的分支。

Java 语言用 if 和 else 表达不同的两种情况。if 后面的条件要用“()”括起来，后面执行的语句用“{}”括起来。如果{}中间只有 1 句，则可以省略{}。如果分支有多条，则可以添加

多个 else if 语句，以 else 作为最后一个分支。

对于多分支，可以用 switch 语句，在下面的代码里，根据 c 的值，会进入不同的分支。执行完当前分支，必须调用 break；跳出 switch，否则会延续到下一个分支继续执行。所以当 c 的值为'a'、'e'、'i'、'o'、'u'时，执行的结果是一样。default 是缺省的分支，不在上面列表的分支，执行 default 分支。Java 7 之前 c 的类型只能是整数类型系列（char、short、int、long），但是 Java 7 之后，c 可以是 String 类型的。

```
//如果 i 是偶数，计数加 1
if(i%2==0) {
    count++;
}

//如果 i 是偶数，偶数计数加 1，否则奇数计数加 1
if(i%2==0) countEven++;
else countOdd++;

//i 代表月份。根据 i 不同的，增加不同的天数
if(i==1 || i==3 || i==5 || i==7 || i==8 || i==10 || i=12) sum+=31;
else if(i==2) sum+=28;
else sum+=30;

//c 代表字母，根据字母的不同输出是元音还是辅音
switch(c){
case 'a':
case 'e':
case 'i':
case 'o':
case 'u': System.out.println("vowel");break;
case 'y':
case 'w': System.out.println("vowel");break;
default: System.out.println("consonant");
}
```

2.1.8 方法

方法，有的语言叫函数，是一段代码的集合。那么为什么要有方法？其实有三个原因：

(1) 逻辑的封装：把共同表达一个业务逻辑的一段代码放在一起。
(2) 方便重用：可以在空间维度上，反复利用这一段代码的空间。
(3) 方便修改：方便在时间维度上，对一段代码进行修改。

下列代码是方法的声明定义以及调用。在这个例子中，方法定义时候的参数 int i 是 f 方法的形参（parameter）。而方法调用时的参数 n，则是实参（argument）。对 Java 来说，实参和形参其实是值传递。也就是说 n 传进去赋值给了 i，i 做了改变，但是不会改变原来的 n。所以输出是：

```
n= 5
f(n)= 6
```

```
public class MethodDemo{
    static int f(int i){  //方法的声明和定义
        i++;
        return i;
    }
    public static void main(String[] args) {
        int n = 5;
        int value = f(n);  //方法的调用
        System.out.println("n= "+ n);
        System.out.println("f(n)= "+ value);
    }
}
```

Java 的返回值只能有一个。f()的返回值是 int。规则中以下三者得保持一致：

(1) 返回值本身的类型: return i;这句中 i 的类型 int。
(2) 返回值的类型：f()的返回值 int
(3) 返回之后赋值的类型：赋给变量 value 的类型 int。

就算不一致，至少强制转换之后，不能产生截断造成错误。所以，可以返回 int，赋给 long，但是不能反过来。

Java 支持多参数，多参数时，实参和形参按顺序一一对应。f(int a, int b)对应 f(2,3)。则 a=2，b=3.

2.1.9 重复

当遇到如下典型场景的时候，我们会使用重复这个控制语句：

(1) 计数器: 提示用户输入 5 个数字，计算他们的和。
(2) 直到条件满足: 当用户输入键入无效的输入值，不要退出，而是不断提示用户输入，直到获得有效的输入值。
(3) 递归: 通过递归实现循环。
(4) 嵌套循环: 乘法表.

Java 的循环控制语句有 3 种：while、do while 和 for 语句。值得注意的是 do while 语句最后条件表达式之后有“;”。

```
public class LoopDemo{
    public static void whileLoop(){
        int x = 1;
        while( x < 10 ) {
            System.out.print("value of x : " + x );
            x++;
            System.out.print("\n");
        }
```

```
    }
    public static void doWhileLoop(){
        int x = 1;

        do{
            System.out.print("value of x : " + x );
            x++;
            System.out.print("\n");
        }while( x < 10 );
    }
    public static void forLoop(){
        for(int x = 1; x < 10; x = x+1) {
            System.out.print("value of x : " + x );
            System.out.print("\n");
        }
    }
    public static void main(String args[]) {
        whileLoop();
        doWhileLoop();
        forLoop();
    }
}
```

此外，Java 5 之后引入了用于遍历数组的增强型的 for 循环控制语句。x 和 name 都是每次从数组中取出的元素的临时变量。

```
public class ForDemo{
   public static void main(String args[]){
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ){
         System.out.print( x );
         System.out.print(",");
      }
      System.out.print("\n");
      String [] names ={"James", "Larry", "Tom", "Lacy"};
      for( String name : names ) {
         System.out.print( name );
         System.out.print(",");
      }
   }
}
```

在循环中，可能会希望有跳出的机制。主要有 break 和 continue 两个关键字。break 是跳出上一层循环结构，执行循环语句后面的语句；continue 是结束本次循环，接着再继

续本循环结构的循环，相当于跳过本次循环剩余的部分。

```
public class ContinueBreakDemo{
    public static void main(String args[]) {
        int [] numbers = {10, 20, 30, 40, 50};

        for(int x : numbers ) {
            // x 等于 30 时跳出循环
            if( x == 30 ) {
                break;
            }
            System.out.print( x );
            System.out.print("\n");
        }
        System.out.println("----------------");
        for(int x : numbers ) {
            if( x == 30 ) {
                continue;//x 等于 30 的时候结束此次循环，接着循环
            }
            System.out.println( x );
        }
    }
}
```

该程序输出如下：

```
10
20
----------------
10
20
40
50
```

2.1.10 数据结构

重复的控制结构，通常需要和一批数据结合在一起使用。那么将数据组织在一起的方式，一般语言提供了数组、列表、哈希、映射等多种方式。有了这些方式，可以应付很多典型的场景：

(1) 打印一组姓名中的每一个
(2) 根据随机数，随机从答案数组选一个
(3) 从员工列表中删除一个元素
(4) 抽奖
(5) 计算一组数的统计信息
(6) 过滤值

(7) 排序

1. 数组

数组是在内存中连续排列的一组数据元素。如图 2-5 所示，数据元素的个数就是数组的长度，序号是元素的下标（index）。序号是从 0 开始计数的。长度为 10 的数组的下标集合就是从 0 到 9。我们可以下标来访问数组中的元素。我们在创建数组对象的时候，需要指定数组的长度，这样我们才可以在内容中分配空间。

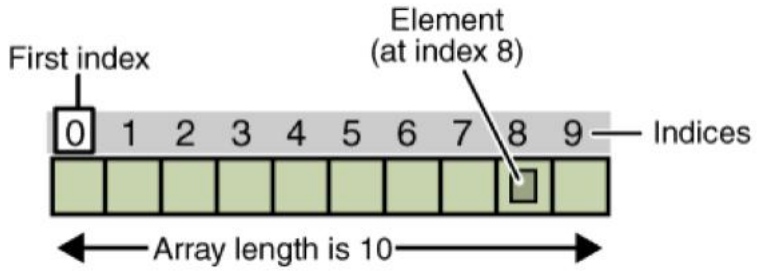

图 2-5 数组的下标

```
//创建数组对象的引用变量
int[] anArray; // 声明一个数组对象的引用变量

//创建数组对象，给数组初始化
anArray = new dataType[10];//创建一个长度为 10 的数组对象
anArray[0] = 100; // 初始化第一个元素

int[] anArray = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000}; //创建一个长度为 10 数组对象，批量初始化元素

//数组的访问
System.out.println("Element 1 at index 0: " + anArray[0]);

//数组的长度
System.out.println(anArray.length);

//拷贝数组
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

//多维数组
String[][] names = {
          {"Mr. ", "Mrs. ", "Ms. "},
          {"Smith", "Jones"}
       };
// 输出 Mr. Smith
System.out.println(names[0][0] + names[1][0]);
```

```
// 输出 Ms. Jones
System.out.println(names[0][2] + names[1][1]);
```

2. ArrayList

Java 中的数组需要提前指定数组的长度，但是如果不能预先知道这个长度的话，可以使用 ArrayList 类。

```
//创建 ArrayList 对象，这个列表中可以放 String 类型的对象。
ArrayList<String> myList = new ArrayList<String>();

//在列表中加元素
String a = new String("foo");
myList.add(a);

//获得列表长度
Int theSize = myList.size();

//访问列表中的元素，get 的参数是下标。
Object o = myList.get(0);

//列表中是否包含某个对象
boolean isIn = myList.contains(a);

//删除某个下标的元素
myList.remove(0);
```

3. HashMap

Java 中 HashMap 是一个哈希表，用来存储键值对(Key-Value)映射。HashMap 实现了 Map 接口，根据键的 HashCode 值存储数据，具有很快的访问速度，最多允许一条记录的键为 null，不支持线程同步。

```
// 引入 HashMap 类
import java.util.HashMap;

public class HashMapDemo{
    public static void main(String[] args) {
        // 创建 HashMap 对象 Sites
        HashMap<Integer, String> sites = new HashMap<Integer, String>();

        // 添加键值对
        sites.put(1, "China");
        sites.put(2, "Japan");
```

```
        sites.put(3, "Korea");

        // 遍历 HashMap，输出 key 和 value
        for (Integer i : sites.keySet()) {
            System.out.println("key: " + i + " value: " + sites.get(i));
        }

        // 返回所有 value 值
        for(String value: sites.values()) {
          // 输出每一个 value
          System.out.print(value + ", ");
        }
    }
}
```

2.1.11 使用 API

对于程序员来说，并不需要所有的功能都自己写，可以借助别人写好的功能。Java 提供相当多的应用程序接口（Application Programming Interface，API）给程序员使用。具体使用细节可以查阅 Java API Specification 文档。

下面是一个 API 的文档。要关注：

(1) 包名+类名
(2) 方法的功能
(3) 方法的参数，返回值
(4) 方法抛出的异常。

其实每个类的完整名称是包含包名和类名的。javax.sound.midi.Sequencer 才是 Sequencer 接口的完整名称。

方法的功能是我们使用 API 的前提。而参数和返回值，则是使用时候的编码细节。

异常是一种错误，有可能是用户输入了非法的数据，有可能是要打开的文件不存在，或者是网络通信的中断等。值得注意的是异常也是 API 的一部分。因为方法抛出的异常，必须 throw 抛出或者用 try catch 捕获。

```
包名：javax.sound.midi
类名：Sequencer
方法：
void setSequence(InputStream stream)
        throws IOException,
               InvalidMidiDataException
功能：Sets the current sequence on which the sequencer operates. The stream must
point to MIDI file data.
This method can be called even if the Sequencer is closed.

参数：
```

```
stream - stream containing MIDI file data.
Throws:
IOException - if an I/O exception occurs during reading of the stream.
InvalidMidiDataException - if invalid data is encountered in the stream, or the
stream is not supported.
```

当想要使用别人的类库或者 API 的时候，需要在代码前面 import 相关的包或者类。这样在书写具体调用的时候，就不用写出完整包含包名的类名。

```
//完整的类名:
graphics.Rectangle myRect = new graphics.Rectangle();
```

```
//导入包的成员:
import graphics.Rectangle;
Rectangle myRectangle = new Rectangle();
```

```
//导入整个包:
import graphics.*;
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

如图 2-6 所示，在 Eclipse IDE 中，可以导入外部包（Add External JARs），来使用别人定义好的 API。

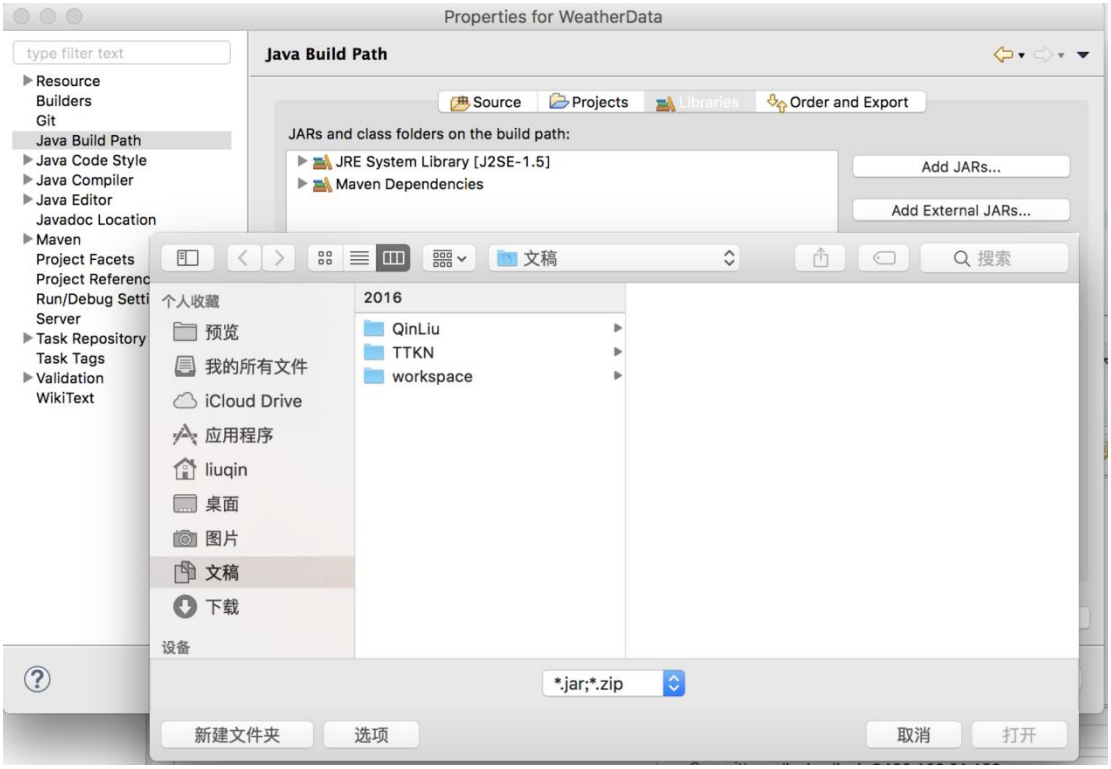

图 2-6 Eclipse 导入外部包

如果使用 Intellij IDEA IDE，可以如图 2-7 所示，在 ProjectSettings 中 Modules 设置中的 Dependency 页面选择 JARs or dictionaries，然后选择所要添加的 JAR 包。

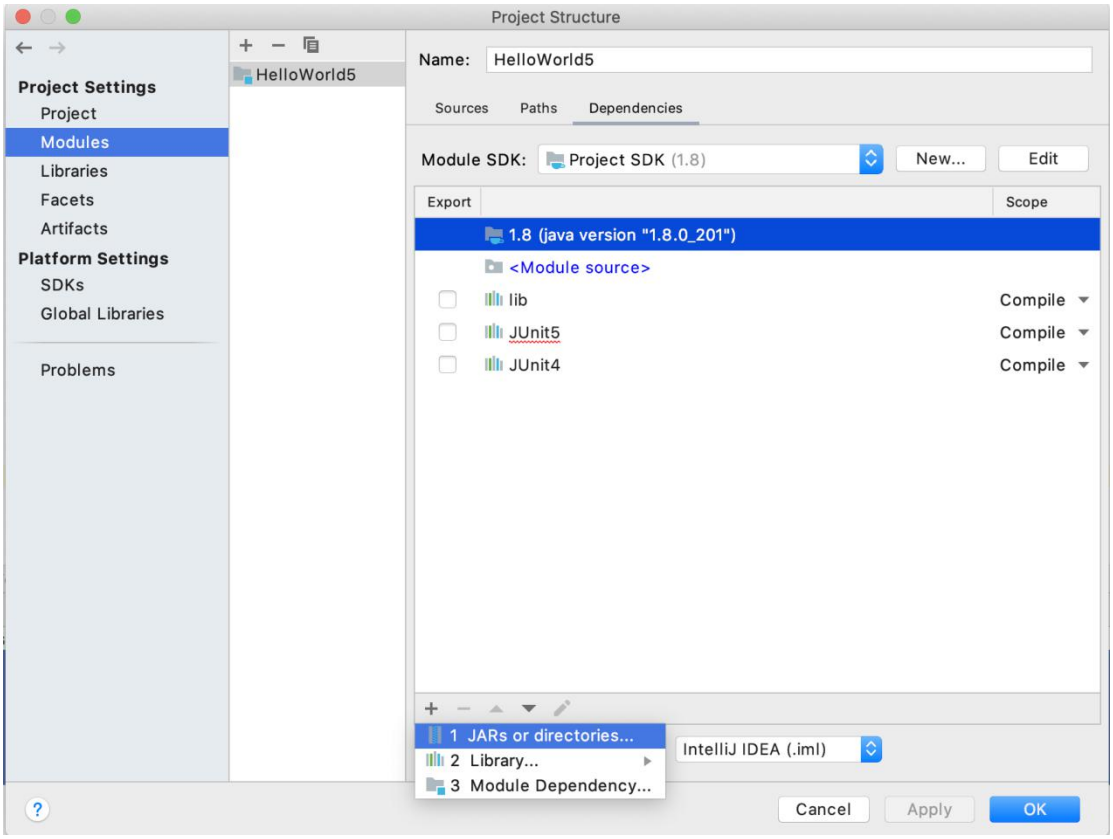

图 2-7 Intellij IDEA 导入外部包

为了更方便的导入外部的包，使用其 API，可以使用 Maven 工具。Maven 需要在项目中添加类似如下 pom.xml 文件。在 pom 文件中，指明依赖的外部包，以及其下载的库（repository）的地址。Maven 有一个全球共享的中央库，只要指定好包的信息，系统同步的时候会直接从中央库去下载包。这样就不用手动下载各个版本的包，再手动导入项目了。当然也可以如下一样指定去自己的 Maven 库中下载对应的依赖包。

```
<repository>
    <id> homework-maven-repo</id>
    <name> Homework Maven Repo</name>
    <url> http://218.94.159.101:8081/repository/homework-central/</url>
     <releases>
         <enabled>true</enabled>
     </releases>
     <snapshots>
          <enabled>true</enabled>
      </snapshots>
</repository>
<dependency>
     <groupId>edu.nju.software</groupId>
     <artifactId>HeWeather</artifactId>
     <version>1.0.0-RELEASE</version>
</dependency>
```

2.1.12 String

String 类其实用来存储字符串，其实现的机制是用的字符串数组。但是其值是不可修改的。创建字符串的方式很多，归纳起来有三类：

(1) 使用 new 关键字创建字符串，比如 String s1 = new String("abc");

(2) 直接指定。比如 String s2 = "abc";

(3) 使用串联生成新的字符串。比如 String s3 = "ab" + "c";

第一种是在内存堆空间创建了一个 String 对象，分配了相应的空间存储“abc”的字符数组。然后将 s1 这个 String 类型的引用变量指向这个对象。而第二种,”abc”这种字符串常量，是在随着类加载存储在代码区。是被整个类共享的。然后将 s2 这个 String 类型的引用变量指向这个字符串常量。而第三种，“ab”和“c”则是 2 条常量池的字符串。然后，系统会在堆区创建 StringBuilder 对象，来合并和存储成为“abc”。然后再让 s3 这个 String 类型的引用变量指向这个对象。

String 的常用方法如下所示。

```
public class StringDemo {
   public static void main(String args[]) {
      //连接两个字符串
      String banner ="Welcome to ".concat("SEECODER");

      //求字符串长度
      int len = banner.length();
      System.out.println( "Length of the banner: " + len );

      //字符串子串
      String substr = banner.substring(0,7);
      System.out.println(substr);

      //以空格分隔字符串，得到一个字符串数组
      String[] subs = banner.split(" ");
      for(String s : subs){
         System.out.println(s);
      }
   }
}
```

上面代码输出的结果为：

```
Length of the banner: 19
Welcome
Welcome
to
SEECODER
```

下面的代码，==比较的是两个引用变量是否指向同一个对象。正如前面提及的“abc”是在常量池中被共享的常量字符串，所以 str1==str2 是 true。str3 和 str4 指向的是在堆中

不同的对象，所以 str3==str4 是 false。而 String 类的 equals()方法比较的是字符串的内容是否相等。所以，str3.equals(str4)是 true。

```
public class StringEqualsDemo{
    public static void main(String[] args) {
        String str1 = "abc";
        String str2 = "abc";
        System.out.println(str1 == str2);

        String str3 = new String("abc");
        String str4 = new String("abc");
        System.out.println(str3==str4);
        System.out.println(str3.equals(str4));
    }
}
```

2.1.13 文件存储

持久化，就是将内存中的数据保存到可以持久访问的硬盘或者网络硬盘上。Java 提供了文件、对象序列化和网络等多种持久化方式。下面主要介绍一下文件存储的类与方法。

```
//创建一个 File 对象，代表一个存在的文件。
File f = new File("MyCode.txt");

//删除一个文件或者一个文件夹
boolean isDeleted = f.delete();
```

```
//创建一个文件夹。
File dir = new File("Chapter2");
dir.mkdir();

//得到文件或者文件夹的绝对地址
System.out.println(dir.getAbsolutePath());

//列举文件夹的内容。
if(dir.isDirectory(){
    String[] dirContents = dir.list();
    for(int i=0;i<dirContents.length;i++){
        System.out.println(dirContents[i]);
    }
}
```

可以通过如下方法写文件。FileWriter 是建立写文件的通道。write()方式是写文件。close()

方法是关闭 IO 通道，将 IO 资源还给操作系统，让其他的进程使用。由于文件可能不存在等原因，所以文件操作存在一定的风险，所以，我们必须用 try 语句捕获异常，并在 catch 语句中处理可能发生的异常。

```
import java.io.FileWriter;
import java.io.IOException;

public class WriteFileDemo{
    public static void main(String[] args){
        try{
            FileWriter writer = new FileWriter(("Foo.txt"));

            writer.write("hello world");
            writer.close();
        }catch (IOException ex){
            ex.printStackTrace();
        }
    }
}
```

可以通过 File、FileReader 类读文件。BufferedReader 类，可以让一行一行的读出文件的内容，起到缓冲的作用。

Java 7 之后，为了简化对资源的使用，可以使用 try-with-resources 语法。try()中的是我们要利用的资源, 这个资源必须是 java.lang.AutoCloseable 对象。这样 try-with-resources 语句会确保在语句执行完毕之后，自动关闭每一个资源。不需要向上段代码那样手动关闭。

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;

public class ReadFileDemo {
    public static void main(String[] args){
        try(BufferedReader reader = new BufferedReader(new FileReader((new
File("Foo.txt"))))
        ){
            String line = null;
            while((line = reader.readLine())!=null){
                System.out.println(line);
            }
        }catch (Exception ex){
            ex.printStackTrace();
        }
    }
}
```

2.2 面向对象编程

本节将会介绍面向对象编程范式的概念和相关 Java 语法。

2.2.1 类和对象

关于类和对象，我们可以从两个角度来理解。一个是现实世界的角度，一个是计算机世界的角度。

现实世界的确会存在抽象和具体两个概念。比如抽象概念的狗，和具体的一条狗。抽象的狗会有一些所有狗都有的属性特征（比如狗的身高），也会有所有狗都有的行为（比如狗会吠）。而具体的狗则会有不同的属性的值（比如，我家的宠物狗身高 20cm），行为的表现也会随着值的不同而略有不同（小的狗一般吠的声音较小）。

而计算机世界的时候，我们会根据类的定义，在计算机中创建狗的对象。这个类的定义的代码空间是被放在代码区，可以重用的的。所以，在计算机世界类是对象创建的蓝图。同一个类，可能会创建不同的对象。不同的对象都是在堆区的，里面存储着不同对象的不同的属性值。

但并不是所有的类都会在现实世界找到对应，有的时候为了可修改性，我们会创建存计算机世界的类和对象。比如，Collection、Event、Controller 等。

Java 中通过 class 关键字来定义 class。new 来创建对象。public Point(int a, int b) 是类 Point 的带参数的构造方法。构造方法没有返回值。

“Point originOne = new Point(23, 94);”这句代码完成 3 个步骤:

(1) 声明 Point 类型的引用变量 originOne。

(2) 在堆区通过带参数的构造方法创建 Point 类型的对象。

(3) 将引用变量 originOne 指向创建的对象。

最后，通过 originOne.getX()调用 getX()方法。

```
public class Point {
    public int x = 0;
    public int y = 0;
    //带参数的构造方法
    public Point(int a, int b) {
        x = a;
        y = b;
    }
    public int getX(){
        return x;
    }
}
```

```
Point originOne = new Point(23, 94);
int x = originOne.getX();
```

2.2.2 封装

封装代表了数据职责和行为职责的在一起。数据是行为计算时所需要用到的数据；行为

是拥有这些数据必然需要提供出来的行为。如果用结构化编程思想，我们会设计这样的“BadMatrix matrixC= plus(matrixA, matrixB);”的 plus 方法，这是一种把操作和数据分开看的思想。对两个矩阵数据做加法操作，得到一个新的矩阵。而在面向对象编程思想下，MyMatrix 中既有属性又有方法。plus 方法的参数和返回值都是同样 MyMatrix 类型，而且参数只有一个。通过“MyMatrix matrixC= matrixA.plus(matrixB);”来调用，代表矩阵 A 自身与另一个矩阵 B 相加得到矩阵 C。每个矩阵对象代表一个既有数据又有行为的个体，相互协作完成更大的职责。

```
/**
 * 矩阵类，实现矩阵的加法，矩阵乘法，点乘以及转置方法
 */
public class MyMatrix {
    private int[][] data;
    private int rows;
    private int columns;

    /**
     * 构造函数，参数为 2 维 int 数组
     * a[i][j]是矩阵中的第 i+1 行，第 j+1 列数据
     * @param a
     */
    public MyMatrix(int[][] a){
        this.data = a;
        this.rows = a.length;
        if (a != null) {
            this.columns = a[0].length;
        }
    }

    public int getRows() {
        return rows;
    }

    public int getColumns() {
        return columns;
    }

    public int[][] getData() {
        return data;
    }

    /**
     * 实现矩阵加法，返回一个新的矩阵
     * @param B
     * @return
```

```
     */
    public MyMatrix plus(MyMatrix B){
        int[][] dataOfB = B.getData();
        int[][] result = new int[rows][columns];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                result[i][j] = data[i][j] + dataOfB[i][j];
            }
        }
        //创建新的矩阵对象，不可以复用之前对象的空间。
        MyMatrix resultMatrix = new MyMatrix(result);
        return  resultMatrix;
    }
    /**
     * 打印出该矩阵的数据
     * 起始一个空行，结束一个空行
     * 矩阵中每一行数据呈一行，数据间以空格隔开
     * example:
     *
     * 1 2 3
     * 1 2 3
     * 1 2 3
     * 1 2 3
     *
     */
    public void print(){
        System.out.println();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                System.out.print(data[i][j]);
                if (j < columns - 1) {
                    System.out.print(" ");
                }
            }
            System.out.println();
        }
        System.out.println();
    }
    public static void main(String[] args){
        int[][] data = {
                {1 , 1 , 1},
                {1 , 1 , 1},
                {1 , 1 , 1}
        };
```

```
        MyMatrix matrix = new MyMatrix(data);
        MyMatrix result = matrix.plus(matrix);
        result.print();
    }
}
```

上面的代码的输出结果是：

```
2 2 2
2 2 2
2 2 2
```

2.2.3 继承与多态

继承是 Java 面向对象编程技术的一块基石，因为它允许创建分等级层次的类。继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。注意，这里子类会继承父类所有的属性和行为。包括私有变量。如果子类定义了和父类同样的方法，则会覆盖父类对这个方法的具体实现。这个叫做 Overriding。Java 中通过 extends 关键字来实现继承。Java 中可以存在多重继承（狗类继承犬科动物类、犬科动物类继承动物类）；但是不允许多继承。一个类只能继承一个父类（狗类继承了犬科动物类，就不能继承宠物类）。

多态（Polymorphism）从英语字根的角度理解，就是多种形态的意思。是一个对象可以当做多种类型来看待。在 Java 中特指，对于任何一个子类对象都可以当做父类对象来看待。反之不可以。多态的本质是将做什么和怎么做分开。因为所有子类型对象都可以当做父类型对象，所以，可以用父类引用变量来写做什么，而在每个子类的 draw 方法里写怎么做。Java 编译器编译的时候，只说明调用父类 Shape 的 draw 方法；当在 JVM 运行时，系统会进行动态绑定，自动运行各个引用变量实际指向的具体子类对象的 draw 方法，从而实现 Overriding。

```
public class Shape {
    void draw() {}

    public static void main(String[] args){
        Shape[] shapes = new Shape[5];

//子类创建对象，赋给父类的引用变量
        shapes[0] = new Circle();
        shapes[1] = new Square();
        shapes[2] = new Triangle();
        shapes[3] = new Square();
        shapes[4] = new Triangle();

//通过父类引用变量调用
        for(Shape s : shapes)
```

```
            s.draw();
        }
}

class Circle extends Shape {
    void draw() {
        System.out.println("Circle.draw()");
    }
}

class Square extends Shape {
    void draw() {
        System.out.println("Square.draw()");
    }
}

class Triangle extends Shape {
    void draw() {
        System.out.println("Triangle.draw()");
    }
}
```

2.2.4 接口

1. 接口的定义

抽象方法是没有实现，只定义了接口的方法。抽象类是不能实例化的类。都是通过 abstract 关键字来实现。而 Java 8 之前，接口就是 100%的纯抽象类（Java 8 引入了 default 方法可以在接口中添加缺省实现，以解决修改接口之后不影响以前的接口的实现类）。类里面的所有方法都是抽象方法，没有方法的实现。一个类只能继承一个类，但是可以实现多个接口。实现接口也是除了继承之外，实现多态的方式。

下面代码是一个典型的使用继承和接口的案例。狗是一种动物，且是一种宠物。Dog 类继承了 Animal 类，实现了 Pet 接口。main 方法中是 Animal 类型的引用变量来调用 eat 和 sleep 方法，但由于实际是 Dog 对象，所以运行时还是调用 Dog 类的 eat 和 sleep 方法。由于 Dog 类没有写自己的 eat 方法，所以 Dog 的 eat 方法复用的是 Animal 的 eat 方法。如果调用 animal.play()方法，由于 Java 是强类型语言，编译器会发现 Animal 类没有 play 方法，所以会直接报错。必须通过类型强制转换，转成 Pet 类型的引用变量，才能调用 play 方法。

```
public abstract class Animal {
    int height;
```

```
    public void eat(){
        System.out.println("Animal eat!");
    }
    public  abstract void sleep();
}

public interface Pet {
    public void play();
}

public class Dog extends Animal implements Pet {

    public void sleep() {
        System.out.println("Dog sleep");
    }
    public void play(){
        System.out.println("Dog play");
    }
    public static void main(String[] args){
        Animal animal = new Dog();
        animal.eat();
        animal.sleep();

        Pet pet = (Pet) animal;
        pet.play();
    }
}
```

以上代码输出为：

```
Animal eat!
Dog sleep
Dog play
```

2. Lambda 表达式

Lambda 表达式，也可称为闭包，它是推动 Java 8 发布的最重要新特性。Lambda 允许把函数作为一个方法的参数（函数作为参数传递进方法中）。使用 Lambda 表达式可以使代码变的更加简洁紧凑，可以提供更好的算法的抽象。

Lambda 表达式的语法格式如下：

(1) (parameters) -> expression

(2) (parameters) ->{ statements; }

以下是一些示例代码：

```
// 1. 不需要参数,返回值为 5
() -> 5

// 2. 接收一个参数(数字类型),返回其 2 倍的值
x -> 2 * x

// 3. 接受 2 个参数(数字),并返回他们的差值
(x, y) -> x - y

// 4. 接收 2 个 int 型整数,返回他们的和
(int x, int y) -> x + y

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回 void)
(String s) -> System.out.print(s)
```

下面是 Lambda 表达式的一个应用案例。旧方法是传入一个匿名类。新方法是直接一个 Lambda 表达式。

```
//旧方法:
button.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent e) {
   System.out.println("The button was clicked using old fashion code!");
}
});

//新方法
button.addActionListener( (e) -> {
    System.out.println("The button was clicked. From Lambda expressions !");
});
```

2.3 设计模式

本节介绍了 3 种常见的设计模式。这些模式在 Android 和 HarmonyOS 的开发中都会遇到。

2.3.1 Observer 模式

Observer 模式是观察者（Observer）对被观察者（Subject）的一种订阅。当关注的事件发生的时候，观察者会被通知，从而做出适当的回应。一般通知的方式通过事件机制，从而将 Observer 和 Subject 进一步解耦。

图 2-8 所示，观察者模式允许文本编辑器对象将自身的状态改变通知给其他服务对象。

Editor 是被观察者，EventManager 管理了所有的观察者（诸多 Listener）。当 Editor 状态改变之后，EventManager 会根据之前的订阅（subscribe()方法），通知所有观察者，调用其 update()方法。

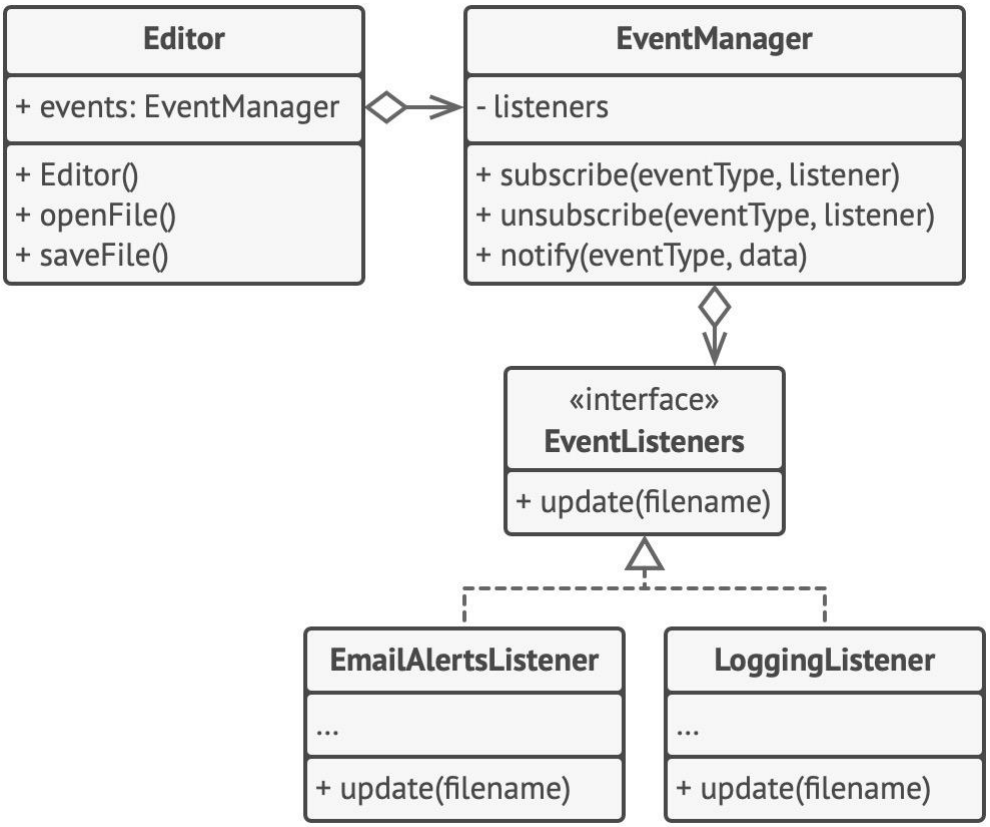

图 2-8 Observer 模式案例类图

```
import java.util.List;
import java.util.ArrayList;
import java.util.Scanner;

class EventSource {
    public interface Observer {
        void update(String event);
    }
    //观察者 List
    private final List<Observer> observers = new ArrayList<>();

    //遍历观察者 List，通知观察者，调用回调方法 update
    private void notifyObservers(String event) {
        observers.forEach(observer -> observer.update(event));
    }

    //增加观察者，主要是设置其回调方法
```

```
    public void addObserver(Observer observer) {
        observers.add(observer);
}

//通过输入字符串，模拟状态改变，通知观察者
    public void scanSystemIn() {
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            notifyObservers(line);
        }
    }
}
public class ObserverDemo {
    public static void main(String[] args) {
        System.out.println("Enter Text: ");
        EventSource eventSource = new EventSource();

        //增加观察者
        //Lambda 表达式实现了回调方法
        eventSource.addObserver(event -> {
            System.out.println("Received response: " + event);
        });

        eventSource.scanSystemIn();
    }
}
```

2.3.2 Adapter 模式

Adapter 模式往往是在一个类的接口和用户期望的不一致，或者不希望暴露过多的接口给用户时使用的。我们会利用一个适配器类来解决接口不兼容的问题。

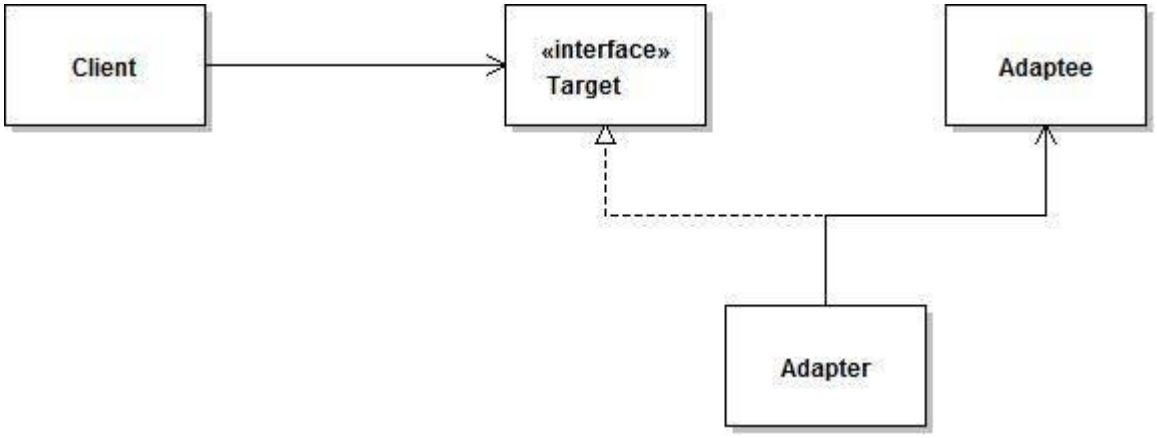

图 2-9 Adapter 模式类图

如图 2-9 所示，Client 类想调用 Target 接口的方法，用 Adapter 实现 Target 接口。而这个方法的具体实现其实是调用了 Adaptee 类的方法。这样就解决了 Target 接口和 Adaptee 接口不兼容的问题。

比如下面代码，目标是新的日志操作的 Api 接口 LogDbOpeApi。其新增日志的方法是 createLog 方法。但是现有的日志功能 Adaptee 类是 LogFileOperate，其接口是 LogFileOperateApi。提供读写日志文件的方法。两个接口不兼容。所以，创建 LogAdapter 类，实现了 LogDbOpeApi 接口。createLog 方法的实现是通过 LogFileOperate 类的读写日志文件的方法实现。其中的具体业务逻辑，我们用桩代码替换了。

```
import java.io.File;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

 interface LogDbOpeApi {
    /*
     * 新增日志
     * @param 需要新增的日志对象
     */
    public void createLog(LogBean logbean);
}



/*
 * 适配器对象，将记录日志到文件的功能适配成数据库功能
 */
 class LogAdapter implements LogDbOpeApi{
    private LogFileOperateApi adaptee;
    public LogAdapter(LogFileOperateApi adaptee){
        this.adaptee = adaptee;
    }
    @Override
    public void createLog(LogBean logbean) {
        List<LogBean> list = adaptee.readLogFile();
        list.add(logbean);
        adaptee.writeLogFile(list);
    }
}

/*
 * 读取日志文件，从文件里面获取存储的日志列表对象
 * @return 存储的日志列表对象
```

```
 */
 interface LogFileOperateApi {
    public List<LogBean> readLogFile();
    /**
     * 写日志文件，把日志列表写出到日志文件中去
     * @param list 要写到日志文件的日志列表
     */
    public void writeLogFile(List<LogBean> list);
}

/*
 * 日志数据对象
 */
 class LogBean {
    private String logId;//日志编号
    private String opeUserId;//操作人员

    public String getLogId(){
        return logId;
    }
    public void setLogId(String logId){
        this.logId = logId;
    }

    public String getOpeUserId(){
        return opeUserId;
    }
    public void setOpeUserId(String opeUserId){
        this.opeUserId = opeUserId;
    }
    public String toString(){
        return "logId="+logId+",opeUserId="+opeUserId;
    }
}

/*
 * 实现对日志文件的操作
 */
 class LogFileOperate implements LogFileOperateApi{
    /*
     * 设置日志文件的路径和文件名称
     */
    private String logFileName = "file.log";
    /*
```

```
     * 构造方法，传入文件的路径和名称
     */
    public LogFileOperate(String logFilename){
        if(logFilename!=null){
            this.logFileName = logFilename;
        }
    }

    @Override
    public List<LogBean> readLogFile() {
        List<LogBean> list = null;
        ObjectInputStream oin =null;

        //业务代码,这里写的桩代码，创建了一个长度为 1 的 List<LogBean>对象，返回了
        LogBean logbean = new LogBean();
        logbean.setLogId("2");
        logbean.setOpeUserId("jordan");
        list = new ArrayList<LogBean>();
        list.add(logbean);
        return list;
    }

    @Override
    public void writeLogFile(List<LogBean> list) {

        File file = new File(logFileName);
        ObjectOutputStream oout = null;

        //业务代码,这里直接打印输入 list 的 size，代表写日志到文件中去。运行结果显示 2.
        System.out.println(list.size());
    }

}
public class AdapterClient {

    public static void main(String[] args){
        LogBean logbean = new LogBean();
        logbean.setLogId("1");
        logbean.setOpeUserId("michael");

        LogFileOperateApi logFileApi = new LogFileOperate("");
        //创建操作日志的接口对象
        LogDbOpeApi api = new LogAdapter(logFileApi);
        api.createLog(logbean);
```

```
    }
}
```

Android 源码中，Spinner 控件使用的 ArrayAdapter、Array 就是利用了适配器模式。

2.3.3 Composite 模式

Composite 模式主要用在创建复合对象上，将对象组合成树形结构以表示“部分-整体”的层次结构，同时使得对单一对象和组合对象的操作具有一致性。如果球队和联赛都是一种团体。联赛可以由球队和联赛混合组成。如图 2-10 所示，队伍一和队伍二构成本地联赛，再和队伍三、队伍四构成顶级联赛。这时候就要用上 Composite 模式。

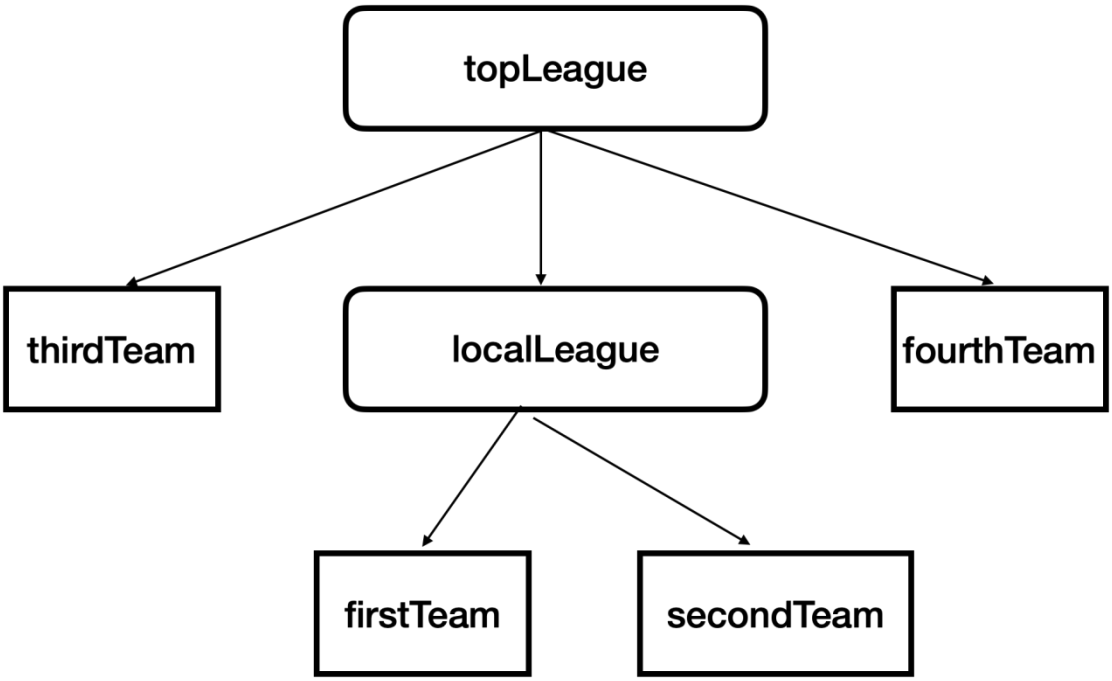

图 2-10 Composite 模式案例

如图 2-11 所示，Composite 模式中，队伍就是 Leaf 类，联赛就是 Composite 类。两者都是 Component 类的子类。Composite 中持有一个 Component 类型的 List。

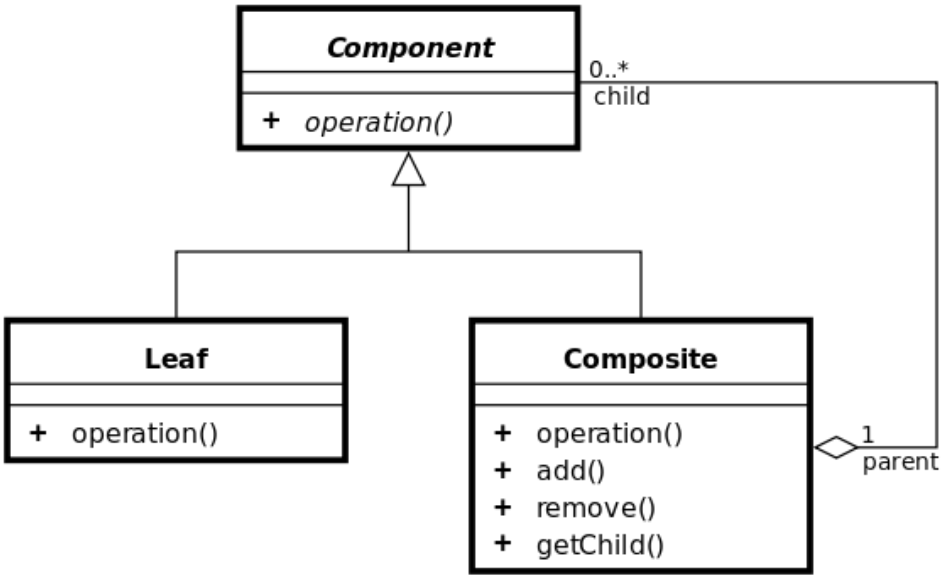

图 2-10 Composite 模式类图

```
import java.util.ArrayList;

// Component
abstract class Group {
    private String name;
    Group(String name) {
        this.name = name;
    }
    String getName() {
        return name;
    }
    public abstract void print();

}

// Leaf
class Team extends Group {
    Team(String name) {
        super(name);
    }
    @Override
    public void print() {
        System.out.println(getName()+"(Leaf)");
    }
}

// Composite
```

```
class League extends Group {
    private ArrayList<Group> groupList = new ArrayList<>();
    League(String name) {
        super(name);
    }
    @Override
    public void print() {
        System.out.println(getName()+"(Composite):");
        for(Group group:groupList) {
            System.out.print("----");
            group.print();
        }
    }
    public void addGroup(Group group) {
        groupList.add(group);
    }
}


public class CompositeClient {
    public static void main(String[] args) {
        League topLeague = new League("Top League");
        League localLeague = new League("Local League");
        Group firstTeam = new Team("First Team");
        Group secondTeam = new Team("Second Team");
        Group thirdTeam = new Team("Third Team");
        Group fourthTeam = new Team("Fourth Team");

        localLeague.addGroup(firstTeam);
        localLeague.addGroup(secondTeam);

        topLeague.addGroup(thirdTeam);
        topLeague.addGroup(localLeague);
        topLeague.addGroup(fourthTeam);
        topLeague.print();
    }
}
```

代码输出的结果为：

```
Top League(Composite):
----Third Team(Leaf)
----Local League(Composite):
----First Team(Leaf)
```

```
----Second Team(Leaf)
----Fourth Team(Leaf)
```

在 Android 源码中，View 和 ViewGroup 就是运用的 Composite 模式。

练习 2

1.【编程题】编写一个货币兑换程序，将欧亚兑换成美元。输入手中的欧元数，以及欧元的当前汇率。打印可以兑换的美元数。
示例输出：
How many euros are you exchanging? 81
What is the exchange rate? 137.51
81 euros at an exchange rate of 137.51 is
111.38 U.S. dollars.

2.【编程题】编写一个程序，让用户输入 3 个数。首先确认所有数字各不相同，如果存在相同的数，退出程序，否则显示其中最大的。
示例输出
Enter the first number: 1
Enter the second number: 51
Enter the third number: 2
The largest number is 51.

3.【编程题】计算卡蒙内心率
TargetHeartRate =(((220-age)-RestingHR)*intensity)+RestingHR
示例输出
Resting Pulse: 65
Age: 22 //提示输入
Intensity |Rate
55% |138bpm

60%　　|145bpm

。。。

95%　　|191bpm

4.【编程题】验证输入

写一个程序验证输入的数据(First Name, Last Name, EmployeeID, Zip code). The first name 和 last name 必须填，至少两个字母. EmployeeID 必须如下格式 t:"AA-1234". Zip code 必须是数字.

示例输出：

Enter the fisrt name: J

Enter the last name:

Enter the ZIP code: ABCDE

Enter an employee ID: A12-1234

"J" is not a valid first name. It is too short.

The last name must be filled in.

The ZIP code must be numeric.

A12-1234 is not a valid ID.

5.【编程题】计算退休计划

输入输出：

What is your current age? 25

At what age would you like to retire? 65

You have 40 years left until you can retire.

It's 2021, so you can retire in 2061.

使用系统当前时间来计算。

6.【编程题】编写一个程序，读入以下数据文件

Ling,Mai,55900

Johnson,Jim,56500

…

Zarnecki,Sabrina,51500

处理改记录，并以格式化的表格形式显示结果，间隔均匀（4 个空格）。

示例输出。

Last　　Fisrt　　Salary

Ling　　Mai　　55900

Johnson　　Jim　　56500

…

Zarnecki　　Sabrina　　51500