第一章 绪论

第二章 基本数据类型、数组、枚举

标识符

命名规则：

- 字母、下划线_、美元符号$，阿拉伯数字，长度不限
- 第一个字符非数字
- 不能是关键字、true、false、null
- 区分大小写

基本数据类型

逻辑类型：boolean

可取值：true、false

```
boolean condition = true, x = false;
```

整数类型：byte、short、int、long

int：4个字节，范围$-2^{31}\sim2^{31}-1$

```
int dec = 114; // 十进制
int bin = 0b1110010; // 二进制使用0b前缀
int oct = 0162; // 八进制使用0前缀
int dex = 0x72; // 十六进制使用0x前缀
```

byte：1个字节，范围$-2^{7}\sim2^{7}-1$

```
byte x = -12;
```

short：2个字节，范围$-2^{15}\sim2^{15}-1$

```
short x = 12;
```

long：8个字节，范围$-2^{63}\sim2^{63}-1$，需要加后缀L

```
long width = 12L;
```

若不加L，编译器会默认把一串数字认为是int类型，若其超过int的表示范围就会报错

字符类型：char

java使用unicode编码，兼容ascii码。 char类型分配2个字节，范围$0\sim65525$，在内存中存储的是字符对应的unicode（utf）码

```
char ch1 = 'A', ch2 = '国', ch3 = '\\', ch4 = 'の';
```

可以用int显式转换char类型以查看字符对应的unicode码：

```
public class Main {
    public static void main(String[] args) {
        char ch1 = 'ω';
        int ch2 = 32831;
        System.out.println("\""+ch1+"\""+"的utf码是"+(int)ch1);
        System.out.println("utf码为"+ch2+"的字符是"+"\""+(char)ch2+"\"");
    }
}
```

浮点类型：float、double

float：4个字节，保留8位有效数字，需要加后缀F或f

```
float x = 22.76f, y = 1e-12F;
```

double：8个字节，保留16位有效数字，后缀D或d可以省略

```
double x = 1e12D, y = 12.4444444;
```

基本数据类型的转换

数据类型精度的排序（从低到高）：

```
byte short char int long float double
```

低级别的值赋给高级别的变量时会自动转换位高级别的类型，成为“隐式转换”即“自动类型提升”，如：

```
int a = 10;
long b = a;
```

高等级的值不允许直接赋值给低等级的变量，因为可能导致数据丢失或数值不精确。若要强制赋值需要进行“显式转换”即“强制类型转换”，如：

```
long a = 10;
int b = (int)a;
```

需要注意，若高等级的值超过了低等级变量能表示的范围则会导致数值溢出或截断，如long类型的3000000000转换为int会变为-1294967296，double类型的123456789.123456789转换为float会变为1.23456792E8。

常见的一个用法是将double或float强制转换为int来取其整数部分。

从命令行输入输出数据

输入基本类型数据

可以使用`java.util`包中的Scanner类来从命令行输入基本类型数据。

使用以下方法从缓冲区中读取从命令行输入的基本类型数据：

```
nextInt(), nextFloat(), nextDouble(),
next(),  // 字符串，跳过空白（空格、制表符、换行符），到空白结束
nextLine(),  // 一行文本，到换行符结束
nextShort(), nextLong(), nextByte(), nextBoolean()
```

使用以下方法判断缓冲区中的下一个token是否符合某种类型（has方法不会从缓冲区中删除token）：

```
hasNext(), // 缓冲区是否还有下一个标记，返回值为true或false
hasNextInt(), hasNextFloat(), hasNextDouble(), hasNextShort(), hasNextLong(),
hasNextByte(), hasNextBoolean(), hasNextLine() // 检查缓冲区中下个token是否为指定类型，返回值为true或false
```

示例：

```java
import java.util.Scanner; // 导入Scanner类

public class InputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // 创建Scanner对象来读取输入

        // 检查是否有下一个整数并读取
        System.out.print("请输入一个整数：");
        if (scanner.hasNextInt()) {  // 使用 hasNextInt() 进行检查
            int intValue = scanner.nextInt();
            System.out.println("你输入的整数是：" + intValue);
        } else {
            System.out.println("输入的不是一个有效的整数！");
            scanner.next(); // 清除无效输入
        }

        // 检查是否有下一个
        System.out.print("请输入一个字符串：");
        if (scanner.hasNext()) {  // 使用 hasNext() 进行检查
            String stringValue = scanner.next();
            System.out.println("你输入的字符串是：" + stringValue);
        } else {
            System.out.println("没有输入有效的字符串！");
        }

        // 读取一整行文本
        scanner.nextLine(); // 清除缓冲区
        System.out.print("请输入一整行文本：");
        String line = scanner.nextLine();
        System.out.println("你输入的文本是：" + line);

        scanner.close(); // 关闭Scanner以释放资源
    }
}
```

输出基本类型数据

使用`System.out.println()`或者`System.out.print()`输出表达式或串的值，前者输出后自动换行，后者不自动换行。

在括号中，可以使用并置符号`+`将多个表达式或值合并起来输出，如：

```java
System.out.println("你输入的文本是："+"\""+input+"\"");
```

若要输出的字符串较长需要拆分为几部分输出，可以使用并置符号加回车实现，如：

```
System.out.println("这是第一部分"+
                   "这是第二部分"+
                   "这是第三部分");
```

也可以使用c语言中printf函数类似的格式控制输出，可用的格式控制符号如下：

格式控制符号	输出类型
%d	int
%f	float，保留6位小数
%c	char
%s	string
%md	int占m列
%m.nf	float占m列，小数保留n位

示例：

```
System.out.printf("你输入的数字是%d", input_int);
```

数组

java中的数组需要先声明后创建，也可在声明的同时创建。

数组的声明

声明数组的格式如下：

```
// 一维数组
类型[] 数组名;
类型 数组名[];

// 二维数组
类型[][] 数组名;
类型[] 数组名[];
```

java采用"数组的数组"来声明多维数组，如数组`[3][4]`是由3个长度为4的数组组成。

java在声明数组时不允许指定数组的大小，数组的大小只能在创建的时候指定。

数组的创建

数组的声明只是指定了数组的名字和元素类型，要向其中存放数据还要为其分配内存空间，即创建数组。

创建数组的格式如下：

```
数组名 = new 元素类型[元素个数];
```

与c语言不同，java允许使用int类型变量来指定数组的长度，但是无法改变已创建的数组的长度。

数组在声明和创建时的元素类型必须相同，否则会报错。

数组的声明和创建同时完成

```
int num[] = new int[5];
// 等号左边：声明一个数组，名字叫num，储存int类型的值
// 等号右边：为这个数组分配5个int类型元素的空间
```

数组的初始化和赋值

数组在创建时可以直接为其赋初值，元素的数量可以代替指定的数组长度，但是 **java不允许在数组声明和初始化时混用指定大小和直接初始化列表** ，如：

```
int[] num = new int[]{1,2,3};

// 错误示范：
// int num[] = new int[3]{1, 2, 3}; // 不能同时指定大小和初始化列表
```

也可以使用初始化列表直接赋值，省略new关键字，简化代码，如：

```
int[] num = {1, 2, 3}; // 数组的长度是3
```

也可以逐个为数组元素赋值，如：

```
int[] num = new int[5];
num[0] = 1;
num[1] = 10;
// 没赋值的部分为默认值0

//二维数组的初始化
int a[][] = {{1},{1,1},{1,2,1},{1,3,3,1},{1,4,6,4,1}};
```

使用var类型简化变量的创建

在声明和创建数组时，指定两次数组元素的类型虽然安全但是十分繁琐，因此可以使用var类型，让编译器自动判断变量的类型，简化代码，如：

```
int num[] = new int[5];
// 可以等价替换为
var num = new int[5];
// 再使用初始化列表直接赋值
var num2 = new int[]{1,2,3};

int a[][] = {{1},{1,1},{1,2,1},{1,3,3,1},{1,4,6,4,1}};
// 可以等价替换为
var a = new int[][]{{1},{1,1},{1,2,1},{1,3,3,1},{1,4,6,4,1}};

// 同理，其他类型也可以使用var
var input1 = 123; // int
var input2 = 3.14; // double
var input3 = 'A';  //char
var input4 = "你好"; // String
```

数组在使用var时不可以省略new和类型[长度]。

使用var类型必须立即对其初始化，否则无法判断其类型。

var的类型推断由赋值表达式的类型决定，使用时应确保表达式的类型是明确的。

var类型仅可用于局部变量，不可用于成员变量、方法参数或返回类型。

数组元素的使用和length的使用

java数组元素的索引从0开始，若访问的索引超过数组长度虽然可以编译通过，但是运行时会报错。

使用`数组名.length`可以返回数组的长度，如var num = new int[5]，则new.length的值为5。

数组的复制

数组的复制分为浅拷贝和深拷贝。浅拷贝只是复制了值的引用，两个变量都指向了同一个内存中的数组，复制后对两个变量的操作是共享的；后者将值又复制了一份（占用双倍的内存空间），复制后对两个变量的操作是独立的。

数组复制的前置要求是两个数组的类型必须相同。

浅拷贝

java的数组是引用型变量，即数组变量存储的是数组对象在内存中的地址，而不是数组元素本身。数组的浅拷贝可以让两个变量指向同一个数组，如：

```
var num1 = new int[]{1,2,3};
var num2 = new int[]{4,5};
num2 = num1;
// 系统将释放分配给{4,5}的内存（因为没有任何变量指向{4,5}），同时让num2指向{1,2,3}
// 由于num1和num2都指向{1,2,3}，对num1或num2做任何修改都会影响另一个变量所指的数组
```

深拷贝

深拷贝将数组中的每一个元素都复制一份，创建了一个新的数组对象。这个新数组与原数组独立，占用新的内存空间，对新数组的修改不会影响原数组。
深拷贝需要手动实现，通过遍历数组元素将其逐一复制到新数组中，这意味着两个数组的长度必须相等。

```
// 第一种方法，手动拷贝
var num1 = new int[]{1, 2, 3};
var num2 = new int[num1.length];
for (int i = 0; i < num1.length; i++) {
    num2[i] = num1[i];
}

// 第二种方法，使用Arrays.copyOf方法
import java.util.Arrays;
var num3 = new int[]{1, 2, 3};
// 使用 Arrays.copyOf 进行深拷贝
var num4 = Arrays.copyOf(num3, num3.length);
// 第一个参数是原数组，第二个参数是新的数组长度，若大于原数组长度会用0或false或null填充，若小于原数组长度会截断

// 第三种方法，使用System.arraycopy方法
int[] src = {1, 2, 3, 4, 5};
int[] dest = new int[7]; // 目标数组有更大的空间
System.arraycopy(src, 1, dest, 2, 3);
// arraycopy的参数为：原数组，原数组索引起始位置，目标数组，目标数组起始位置，复制的元素数量
```

数组的字符串输出

使用Arrays类中的toString方法将数组转换为字符串输出，如：

```
import java.util.Arrays;

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

```
        int[] numbers = {10, 20, 30, 40, 50};
        String[] names = {"Alice", "Bob", "Charlie"};

        System.out.println(Arrays.toString(numbers));
        // 输出: [10, 20, 30, 40, 50]
        System.out.println(Arrays.toString(names));
        // 输出: [Alice, Bob, Charlie]
    }
}
```

数组的排序和二分法查找

使用Arrays类中的sort方法对数组排序，如：

```
import java.util.Arrays;

public class SortExample {
    public static void main(String[] args) {
        int[] numbers = {5, 2, 8, 3, 1};
        Arrays.sort(numbers);
        // 此时numbers为{1, 2, 3, 5, 8}
        System.out.println(Arrays.toString(numbers));
    }
}
```

sort方法支持仅对数组中某个部分进行排序，如：

```
int[] arr = {5, 2, 8, 3, 1};
Arrays.sort(arr, 1, 4); // 对索引1到3的元素排序
// 输出: 5, 2, 3, 8, 1
```

使用Arrays类中的binarySearch方法对 **已排序** 的数组进行二分搜索，能在$O(\log n)$的时间复杂度内完成元素的查找，返回目标元素的索引，若查找失败则返回负值，其绝对值为插入的位置，如：

```
import java.util.Arrays;

public class BinarySearchExample {
    public static void main(String[] args) {
        int[] numbers = {1, 3, 5, 7, 9, 11};
        int index = Arrays.binarySearch(numbers, 7);
        System.out.println("元素 7 的索引: " + index); // 输出: 元素 7 的索引: 3

        index = Arrays.binarySearch(numbers, 4);
        System.out.println("元素 4 的索引: " + index); // 输出: 元素 4 的索引: -3
```

```
    }
}
```

枚举

java的枚举类型用于定义一组常量，如四季、一周的七天，如：

```
// 定义枚举类型 Day
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class EnumExample {
    public static void main(String[] args) {
        // 赋值
        Day today = Day.MONDAY;  // 直接赋值为枚举常量
        System.out.println("今天是: " + today);
        // 今天是: MONDAY

        // 比较
        if (today == Day.MONDAY) {
            System.out.println("星期一开始工作! ");
            // 输出: 星期一开始工作!
        }

        // 使用name()方法输出枚举类型变量的值
        System.out.println("今天是: " + today.name());
        // 输出: 今天是: MONDAY

        // 使用ordinal()方法输出某个常量的索引值，从0开始
        System.out.println("今天是第 " + (today.ordinal() + 1) + " 天"); // 输出:
今天是第 1 天
    }
}
```

第三章 运算符、表达式、语句

运算符和表达式

算术运算符和算术表达式

- 加减乘除 + - * /
- 取余 %

- 自增自减 ++ --，x++表示先使用x后自增，++x表示先自增后使用

算数混合运算的精度

数据类型精度的排序（从低到高）：

```
byte short char int long float double
```

精度计算规则（从上到下）：

- byte，short，char类型参与计算时先转化为int
- 若有double则按double计算
- 若有float则按float计算
- 若最高精度为long则按long计算
- 若最高精度低于int则按int计算
- 不允许把超过byte的int赋值给byte

```
byte x = 97 + 1, y = 1;
// 这个不会报错，因为编译器可以直接计算出x是98，y是1，不超过byte的表示范围

byte z = x + y;
// 这个会报错，因为byte类型在参与计算时会先转为int，x+y得到的int结果不能自动转为byte类型，因为可能导致数据溢
```

关系运算符和关系表达式

运算返回值为boolean型的true或false。

运算符	含义
>	大于
<	小于
>=	大于等于
<=	小于等于
==	等于
!=	不等于

逻辑运算符和逻辑表达式

运算符	含义
&&	与
||	或
!	非

对“与”和“或”运算，存在“短路原则”，若根据前一个表达式的值已经可以判断出整个表达式的真假，则不再计算后边表达式的值。

赋值运算符和赋值表达式

- 赋值 =

```
int a = b = 100;
// 会报错，因为编译器不知道b的类型

int c;
int d = c =100;
// 可以，因为c和d的类型都是明确的，他两个都会被赋值为100

int m = 100;
int n = 100;
// 最好这么写，代码易读
```

位运算符

java的位运算用于对整数类型的位进行直接操作，性能高效，适用于性能和资源有限的环境。

位运算符	含义	解释
&	按位与	都为1则为1
|	按位或	其中一个为1则为1
^	按位异或	不同则为1
~	按位取反	取反
<<	左移	位左移指定的位数，右侧用0填充 相当于乘2
>>	右移	位右移指定的位数，右侧用符号位填充（正0负1） 相当于除2
>>>	无符号右移	右移指定的位数，左侧用0填充

举例：

- a = 0b0101，b = 0b0011
- a & b = 0b0001
- a | b = 0b0111
- a ^ b = 0b0110
- ~a = 0b1010, ~b = 0b1100

设 c = -24 = 0b 1110 1000，对其作左右移的位运算：

左移2位 c << 2

> 0b 1010 0000
> 左边两个1被丢弃，右边补两个0

右移2位 c >> 2

> 0b 1111 1010 右边两个0被丢弃，左边补符号位1（负数）

无符号右移3位 c >>> 3

> 0b 0001 1101 右边3位被丢弃，左边补0

instanceof 运算符

`instanceof`用于判断对象是否是某个类的实例，用来检查对象是否是指定类或其子类、实现类的实例，返回值为boolean类型的true或false。

语法：

```
object instanceof ClassName
```

- 若 object 是 ClassName 类型或其子类、实现类的实例，则返回 true。
- 若 object 不是 ClassName 类型，则返回 false。

运算符的优先级

优先级由高到低排序：

运算符类型	运算符
括号	()
一元运算符	++ -- + -(正负) ! ~
类型转换	(type)
乘除、取余	* / %
加减	+ -

运算符类型	运算符
移位	<< >> >>>
关系	> < >= <= instanceof
相等性	== !=
按位与	&
按位异或	^
按位或	|
逻辑与	&&
逻辑或	||
三元运算符	?:
赋值	= += -+ *= /=

语句概述

java中的语句分为：

- 方法调用语句 `System.out.println("hello");`
- 表达式语句 `x = 23`
- 复合语句

```
{
    x = 123;
    System.out.println("%d", x);
}
```

- 空语句（只有一个分号）
- 控制语句（条件分支、开关、循环）
- package和import语句

条件分支语句 if

if - else if - else：

```
if (表达式) {
    语句
}
else if (表达式){
    语句
}
```

```
else {
    语句
}
```

开关语句 switch

`switch`是单条件多分支的开关语句，语法如下：

```
switch (表达式) {
    case 常量1: {
        语句
        break; // 可选
    }
    case 常量2: {
        语句
    }
    ...
    default: {
        //可选
        语句
    }
}
```

switch会先计算表达式的值，若其与某个case的常量值相等，则从这个case的语句开始依次执行，直到遇到break

switch语句适合与枚举类型搭配使用，如：

```
public class main {
    // 定义枚举类型
    enum color {
        red, green, blue
    }
    public static void main(String[] args) {
        for (color a: color.values()){
            for (color b: color.values()){
                for (color c: color.values()){
                    if(a != b && a != c && b != c){
                        // 输出所有可能的颜色组合
                        System.out.println("a=" + a + ", b=" + b + ", c=" + c);
                    }
                }
            }
        }

    }
}
```

输出：

```
a=red, b=green, c=blue
a=red, b=blue, c=green
a=green, b=red, c=blue
a=green, b=blue, c=red
a=blue, b=red, c=green
a=blue, b=green, c=red
```

循环语句

for循环

```
for (初始化; 条件; 每次循环最后执行){
    语句
}
```

在JDK1.5中新增了“增强型for循环”或“for-each循环”，用于简化对数组或集合的遍历。语法如下：

```
for(声明循环变量: 数组名字){
    语句
}
```

比如：

```
int[] numbers = {1, 2, 3, 4, 5};

for (int number : numbers) {
    System.out.println(number);
}
```

for循环创建了一个int变量`number`，每次循环都会将数组`numbers`中的下一个元素赋值给`number`，直接体现了遍历的对象，简化了代码，有效防止数组越界，但是仅适用于顺序遍历的情况

while循环

```
while (表达式){
    语句
```

```
}
```

do-while循环

```
do {
    语句
} while (表达式)
```

先执行一次循环体内的程序，然后根据表达式的值决定是否重复执行

break、continue

break直接跳出当前一层的循环，continue让当前循环直接进行下一次循环

第四章 类与对象

类

类，class，是java程序的基本要素。一个java程序由若干个类组成。

```
class 类名 {
    类体
}
```

类体的组成：

- 变量的声明：储存对象的属性
- 方法的定义：对类中的属性进行操作

类的变量

类的变量分为两种：

- 成员变量：在变量声明部分声明的变量，在整个类中都有效
  - 用`static`修饰的成员变量称为类变量（静态变量）
  - 否则为实例变量
- 方法变量：在方法部分声明的变量，在整个方法中有效

> 静态变量是类级别的，只在类被第一次创建对象时初始化，对这个类的所有对象都可见且保持一致，在任意一个对象中修改静态变量都会同步到其他对象中，在程序终止时才被释放。适用于定义常量，如

> MAX_VALUE。

若方法中局部变量的名字与成员变量相同，则成员变量会被隐藏。若要引用成员变量需要加this关键字。

```
class Square {
    // 变量声明->成员变量
    double length;

    // 方法定义
    void EditLength {
        length = scanner.nextDouble();
    }

    double CalcArea {
        // 方法变量
        double area = length*length;
        return area;
    }
}
```

注意，声明类的成员变量时可以直接为其赋初值，但是不可以声明成员变量后再为其赋值，因为赋值操作只能出现在方法中，如：

```
class example {
    int a = 1;
    // 这样是可以的

    int b;
    b = 1;
    // 这样不可以
}
```

类的方法

```
方法声明部分 {
    方法体的内容
}
```

基本的方法声明包括方法名和方法的返回类型，可以是任意java的数据类型或void（不返回数据）。

方法的重载 Overload

java允许在一个类中定义多个方法，他们的方法名相同但是 **参数列表不同**，根据传入参数的不同执行不同的方法，称为 **重载**

重载的返回类型可以不同，但是不用于区分重载

```
class MathOperations {
    int add(int a, int b) {
        return a + b;
    }

    // 重载的add方法，参数列表不同
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperations math = new MathOperations();
        System.out.println(math.add(5, 3));
        // 输出: 8
        System.out.println(math.add(5.5, 3.2));
        // 输出: 8.7
    }
}
```

构造方法

构造方法是创建对象时初始化对象的特殊方法，在创建对象时由new关键字自动调用。
构造方法可以用于设置对象的初始状态，执行必要的初始化操作。
构造方法在一个对象被创建时被调用，它 **与类同名**，没有返回类型，可以被重载。

```
class Person {
    // 成员变量
    String name;
    int age;

    // 无参构造方法
    public Person() {
        this.name = "Unknown";
        this.age = 0;
    }

    // 有参构造方法
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
    // 打印信息方法
    void PrintInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        // 使用无参构造方法创建对象
        Person person1 = new Person();
        // 使用new关键字时自动运行构造方法
        person1.PrintInfo();  // 输出: Name: Unknown, Age: 0

        // 使用有参构造方法创建对象
        Person person2 = new Person("Alice", 30);
        person2.PrintInfo();  // 输出: Name: Alice, Age: 30
    }
}
```

类方法和实例方法

类方法是用`static`修饰的方法，有以下特性：

- 类方法是类级别的，属于类本身，而不是类的某个实例
- 可以直接通过`ClassName.ClassMethod`调用而不需要实例化
- 不能访问非静态的成员变量和实例方法，只能访问静态变量和静态方法

实例方法没有`static`修饰，有以下特性：

- 实例方法是属于对象的方法，需要通过创建类的实例来调用，如`ObjectName.InstanceMethods`
- 实例方法可以访问实例变量和实例方法，也可以访问类变量和类方法

```
class Calculator {
    // 类方法：不依赖于具体的对象
    public static int add(int a, int b) {
        return a + b;
    }

    // 实例方法：依赖于具体的对象
    public int multiply(int a, int b) {
        return a * b;
    }
}

public class Main {
    public static void main(String[] args) {
        // 调用类方法：直接通过类名调用，不需要创建对象
```

```
        int sum = Calculator.add(5, 10);
        System.out.println("Sum: " + sum);  // 输出: Sum: 15

        // 创建Calculator对象
        Calculator calculator = new Calculator();

        // 调用实例方法：需要通过对象调用
        int product = calculator.multiply(5, 10);
        System.out.println("Product: " + product);  // 输出: Product: 50
    }
}
```

实例方法能对类变量和实例变量操作，但是类方法只能操作类变量

实例方法可以调用类中的其他方法，类方法只能调用类中的类方法

对象

在java中，用类来声明变量，然后用类来给出这个类的一个实例，也就是创建一个对象

构造方法

当程序创建对象时需要使用类的构造方法

若没有编写构造方法，则编译器会默认这个类只有一个构造方法，他没有参数，方法体中没有语句

创建对象

创建对象前需要先声明对象：`类名 对象名;`
此时没有为这个对象分配空间

然后使用new运算符为对象分配变量，即创建对象：`对象名 = new 类名(参数)`
实例化对象后会为成员变量分配空间并赋初值，然后根据成员变量的地址计算出一个称为“引用”的值作为`new 类名(参数)`的运算结果，赋值给对象名

分配给对象的变量习惯性称为对象的实体

使用`类名 对象名 = new 类名(参数);`同时声明并创建变量：

```
class Person {
    // 成员变量
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Person person2 = new Person("Alice", 30);
    }
}
```

若两个对象有相同的引用，则二者有相同的实体，如：

```
class Point {
    int x, y;
    Point (int x, int y){
        this.x = x;
        this.y = y;
    }
    void PrintPos(){
        System.out.printf("x = %d, y = %d\n", this.x, this.y);
    }
}

public class Main{
    public static void main(String[] args){
        Point p1 = new Point(1, 2);
        Point p2 = new Point(3, 5);
        p1.PrintPos();
        p2.PrintPos();

        p1 = p2;
        // 此后对p1、p2任意一者的更改都会同步到对方身上，因为p1和p2都指向了同一个对象
        p1.PrintPos();
        p2.PrintPos();
    }
}
```

输出：

```
x = 1, y = 2
x = 3, y = 5
x = 3, y = 5
x = 3, y = 5
```

当程序发现内存中的某个实体不被任何一个对象引用，则会被自动释放内存，称为“垃圾收集机制”。比如上述程序中，p1指向的对象被p2覆盖，没有任何一个对象指向这个实体，那他就会被释放内存。

使用对象

使用点运算符/引用运算符/访问运算符：`.`来访问对象的变量或方法

```
class example {
    int num1;
    double num2;

    public example(int num1, double num2){
        this.num1 = num1;
        this.num2 = num2;
    }

    void print2(){
        System.out.printf("num2 = %f\n", num2);
    }
}

public class Main {
    public static void main(String[] args) {
        example exam = new example(10, 3.14);
        System.out.printf("num1 = %d\n", exam.num1);
        exam.print2();
    }
}
```

输出：

```
num1 = 10
num2 = 3.140000
```

可变参数

可变参数允许方法接收多个相同类型的参数而无需明确指定参数的数量，语法如下，其中...为省略号语法

```
public void methodName(Type... parameterName) {
    // 方法体
}
```

也可以明确前几个参数的类型和数量，最后输入不确定数量的参数，如：

```
public static int sum(double sum, double... numbers){
    // 方法体
}
```

实例：

```
public class VarargsExample {
    public static int sum(int... numbers) {
        int total = 0;
        for (int number : numbers) {
            total += number;
        }
        return total;
    }

    public static void main(String[] args) {
        System.out.println(sum(1, 2, 3));
        // 输出: 6
        System.out.println(sum(10, 20));
        // 输出: 30
        System.out.println(sum(5, 5, 5, 5, 5));
        // 输出: 25
    }
}
```

编译器实际上为所有传入的输入创建了一个数组，也就是说可以使用数组的方式来访问这些参数，如：

```
public class Main {
    // 定义一个接受可变参数的 sum 方法
    public static int sum(int... numbers) {
        int total = 0;
        for (int i = 0; i < numbers.length; i++) {
            total += numbers[i]; // 通过索引访问数组元素
        }
        return total;
    }

    public static void main(String[] args) {
        System.out.println(sum(1, 2, 3));
        // 输出: 6
        System.out.println(sum(10, 20));
        // 输出: 30
        System.out.println(sum(5, 5, 5, 5, 5));
        // 输出: 25
    }
}
```

对象的组合

java中"对象的组合"允许一个类包含其他类的对象作为其成员变量，表示某个类是由其他类组成的，即一个对象拥有另一个对象的实例。

示例：

```
class Address {
    private String city;

    public Address(String city) {
        this.city = city;
    }

    public String getCity() {
        return city;
    }
}

class Person {
    private String name;
    private Address address; // Person 包含 Address 对象

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
        // 这里保存的是address对象的引用而不是副本，因此后续直接修改address对象时会同步到person中
    }

    public void printInfo() {
        System.out.println("Name: " + name + ", City: " + address.getCity());
    }
}

public class Main {
    public static void main(String[] args) {
        Address address = new Address("New York");
        Person person = new Person("John", address);
        person.printInfo();  // 输出: Name: John, City: New York
    }
}
```

static 关键字

static 关键字可以用来修饰变量（类变量-实例变量）和方法（类方法-实例方法），表示它们属于类而不是某个具体的对象实例。

实例方法可以调用实例变量和实例方法，也可以调用类变量和类方法，但是类方法只能调用类变量和类方法，不能调用实例变量和实例方法。

一个类的不同对象实例共享同一个类变量，使用相同的内存空间，直到程序结束运行才释放内存。

示例：

```
class MyClass {
    private static int count = 0; // 静态变量，属于类而不是对象实例
```

```
    public MyClass() {
        count++; // 每次创建对象时，count 变量加 1
    }

    public static int getCount() {
        return count; // 静态方法，可以通过类名直接调用
    }
}

public class Main {
    public static void main(String[] args) {
        // 创建 MyClass 的两个实例
        MyClass obj1 = new MyClass();
        MyClass obj2 = new MyClass();
        MyClass obj3 = new MyClass();

        // 输出 MyClass 的实例数量
        System.out.println("MyClass 的实例数量: " + MyClass.getCount());
    }
}
```

this 关键字

`this`关键字在构造方法和实例方法中用于引用当前对象的实例变量和方法，但是静态方法中不能使用`this`。

在构造方法中，用于区分构造方法的参数名与实例变量名：

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

在实例方法中，用于区分局部变量和实例变量：

```
public class Person {
    private String name;

    public void setName(String name) {
        this.name = name;
        // this.name 是实例变量，name 是方法参数
    }

    public void printName() {
```

```
        System.out.println(this.name);
        // 通过 this 访问实例变量
    }
}
```

在构造方法中，调用当前类的其他构造方法，即重载：

```
public class Person {
    private String name;
    private int age;

    public Person() {
        this("Unknown", 0);  // 调用带有参数的构造方法
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

包 package