

# Java

## 1. Java基础

### 1. *Java* 区分大小写

### 2. *Java* 的编译加解释运行方式

编译器类似翻译家

- 方式：一次将源代码直接翻译为与平台强相关的可直接执行的机器码
- 产物：会生成可执行独立文件

| 比如在 *Windows* 平台上生成可直接执行的 *.exe* 文件

- 速度：运行速度快
- 跨平台性：跨平台性弱，但运行时不要求编译器在场

解释器类似同声传译

- 方式：在需要运行时才开始逐行执行
- 产物：并不生成独立的任何文件
- 速度：会产生运行时翻译的开销，较慢
- 跨平台性：跨平台性很强，但每次运行都要求解释器在场

而 *Java* 采用了先编译后解释的方式

通过引入一个中间代码 *.class* 字节码文件实现了极强的跨平台性

先使用 *JDK* 中的 *javac* 工具将源代码编译成 *.class* 字节码文件

字节码是专门为 *JVM* 虚拟机设计的平台无关的指令集

在 *JVM* 中

- 首先执行类加载

*JVM* 内的类加载器会找到并加载 文件 *.class* 文件，将字节码读入内存

- 然后执行**字节码校验**

为了安全，校验器会检查字节码是否符合 *JVM* 规范，有没有潜在的恶意操作

- 最后逐行对其解释执行

为了提升性能，引入了 *JIT* 即时编译

*JVM* 会在程序运行时进行监控，识别出那些被频繁执行的**热点代码**

然后，*JIT* 编译器会将这些热点字节码编译成**高度优化的本地机器码**，并缓存起来

下次再执行这些代码时，就直接运行编译好的机器码，速度大幅提升

路径如下所示

源文件(.java) -> [Java编译器javac] -> 字节码文件(.class) -> [JVM] -> [解释器 + JIT编译器] -> 操作系统

3. 在 *Java* 编程范式中，所有可执行代码必须封装于类 (*class*)

方法 (*method*) 是指类的一种成员，类似于其他语言中的函数 (*function*)，但有本质区别

用户可在类中定义自定义方法，并在 *main* 方法内调用这些方法来实现复杂的功能

4. *Java* 虚拟机 (*JVM*) 将严格从指定类的 *main* 方法开始执行程序代码

因此为了代码能够执行，在类的源文件中**必须有且仅有一个** *main* 方法

**但是每个类都可以有一个** *main* 方法

*main* 方法**必须**按照如下方式编写

代码块

```
1 public static void main(String[] args) {  
2     // 程序逻辑  
3 }
```

其中 *static* 表示这是一个静态方法，即不需要实例化类就可以调用的方法

且在静态方法内部只能调用其他静态方法和静态变量

*void* 表示不返回值，如果是 *int* 则返回 *int* 类型的值，其他类型亦然

在 *main* 中如果涉及到其他类下方法的调用，则会转向另一类执行,执行完毕后转回 *main* 方法

当 *main* 方法执行完毕，运行结束

如果 *main* 方法正常退出，那么 *Java* 应用程序的退出代码（不是返回值）为 0，表示成功地运行了程序

如果希望在终止程序时返回其他的代码，那就需要调用 `System.exit()` 方法

5. 源代码文件名必须与文件中声明的公共类 (`public class`) 的名称**完全匹配**，**包括大小写**，并用 `.java` 作为扩展名

且同时只能有一个公共类，可以同时存在多个非公共类

6. 在 *Java* 中，用大括号划分程序的各个部分，被大括号包围的一段代码称为块

*Java* 中任何方法的代码都用 { 开始，用 } 结束

使用大括号有两种风格，一种是**次行风格**，一种是**行尾风格**

```
1  public class MyClass
2  {
3      public static void
4      main(String[] args)
5      {
6          if (condition)
7              // 代码块
8          }
9          else
10         {
11             // 代码块
12         }
13     }
14 }
```

```
1  public class MyClass {
2      public static void
3      main(String[] args) {
4          if (condition) {
5              // 代码块
6          } else {
7              // 代码块
8          }
9      }
10
11
12
13
14
```

次行风格将花括号垂直对齐，因而使程序容易阅读

而行尾风格更节省空间，并有助于避免犯一些细小的程序设计错误

*coder* 可按自己的喜好选择风格，不过行尾风格是目前**绝大多数人**的选择

**但无论选择哪种风格，整个项目应保持一致**

---

7. *javac* 编译器将 `.java` 源文件转换为**平台无关**的字节码文件 `.class`

- 编译命令: `javac SourceFile.java`
- 执行命令: `java ClassName` (注意执行的是**类名**而非**文件名**)

不过在 *Java11* 以后可以使用

`java A.java` 这样的命令

同时编译并运行

---

## 8. java 主要采用三种命名方法

**大驼峰** (*UpperCamelCase*) 所有单词首字母均大写

**小驼峰** (*lowerCamelCase*) 第一个单词首字母不大写，其余单词首字母均大写

**全部大写并用下划线分割单词**

通常约定，类一般采用大驼峰命名，方法和局部变量使用小驼峰命名，而大写下划线命名通常是常量和枚举中使用

eg: *class HashMap, class ArrayList*

*getUserName()*

*MAX\_VALUE*

除此之外，还有特殊的对包命名的方法

包名**统一使用小写且为单数形式**

**如果类名有复数含义，则可以使用复数形式**

使用点分隔符进行分隔

如 *org.apache.commons*

对于常见的复合词和组合词不进行分隔

如 *springframework, deepspace*

包名的构成是 【前缀】 【作者名】 【项目名】 【模块名】

常见的前缀可以分为以下几种

前缀	例	含义
indi 或 onem	indi.发起者名.项目名.模块名.....	个体项目 个人发起，但非自己独自完成 可公开或私有项目， copyright主要属于发起者。
pers	pers.个人名.项目名.模块名.....	个人项目 指个人发起，独自完成， 可分享的项目 copyright主要属于个人
priv	priv.个人名.项目名.模块名.....	私有项目，指个人发起，独自完成 非公开的私人使用的项目， copyright属于个人。
team	team.团队名.项目名.模块名.....	团队项目，指由团队发起 并由该团队开发的项目 copyright属于该团队所有
顶级域名	com.公司名.项目名.模块名.....	公司项目 copyright由项目发起的公司所有

更加详细的介绍可以参考下文

<https://zhuanlan.zhihu.com/p/138429217>

## 9. Java 中的注释不会出现在可执行程序中

因此，可以在源程序中根据需要添加任意多的注释，而不必担心可执行代码会膨胀

java 中有三种标记注释的方法

- 单行注释 //
- 多行注释 以/\* 开始      \*/结束
- 文档注释 以/\*\* 开始      \*/结束

**在 Java 中， /\*\*/ 注释不能嵌套**

也就是说，不能简单地把代码用 /\* 和 \*/ 括起来作为注释，因为这段代码本身可能也包含一个 \*/

## 10. Java 是一种强类型语言

在 Java 中，一共有 8 种基本类型 (*primitive type*)

这些基本类型存储所需的字节数是**固定**的，这使得 *Java* 程序的可移植性增强

也就是说在 *Java* 中，所有的数值类型所占据的字节数量与平台无关

4 种整型, *byte short int long*

*byte* 采用 1 个字节存储，范围为  $-128 \sim 127$ ，即  $-2^7 \sim 2^7 - 1$

*short* 采用 2 个字节存储，范围为  $-32768 \sim 32767$ ，即  $-2^{15} \sim 2^{15} - 1$

*int* 采用 4 个字节存储，范围为  $-2147483648 \sim 2147483647$ ，即  $-2^{31} \sim 2^{31} - 1$

*long* 采用 8 个字节存储，范围为  $-9223372036854775808 \sim 9223372036854775807$ ，即  $-2^{63} \sim 2^{63} - 1$

同时注意，*Java* **没有任何**无符号 (*unsigned*) 形式的 *int*、*long*、*short* 或 *byte* 类型

*long* 类型的字面量**需要**在后加上 *L* 或者 *l*

*Long num = 120L;*

因为在 *Java* 中没有后缀的整型**字面量**默认类型为 *int*

```
1  long num1 = 123;      // 合法: int字面量隐式转为long
2  long num2 = 0x7FFF;   // 合法: int字面量隐式转为long
```

当这个字面量超过 *int* 上限或下限的时候，就会报 *error: integer number too large*

```
1  // 合法写法
2  long valid1 = 2147483648L;  // 显式声明为long
3  long valid2 = 3000000000L;  // 超出int范围必须加L
4
5  // 非法写法 (编译错误)
6  long invalid1 = 2147483648; // 错误: 超出int范围且未加L转换为long类型
7  long invalid2 = 3000000000; // 错误: 同上
```

我们需要显式得给这些超出 *int* 类型上下限的字面量加上后缀转换为 *long* 类型的防止报错

所以当字面量在 *int* 范围内时，可以不写 *L* 或 *l* 这个后缀，但是**并不推荐**

因为对于 *int* 范围内的字面量，如果不写 *L* 或 *l* 这个后缀的话

是在将 *int* 类型的字面量进行自动转换为范围更大的 *long* 类型

当超出时，必须写 *L* 或 *l* 这个后缀

所以为了防止混淆，我们要求 *long* 类型的数字**需要**在后加上 *L* 或者 *l*

---

2 种浮点类型，*float double*

*float* 采用 4 个字节存储，范围为  $\pm 1.4E - 45 \sim \pm 3.4E38$ (6 - 7位有效数字)

*double* 采用 8 个字节存储，范围为  $\pm 4.9E - 324 \sim \pm 1.7E308$ (15位有效数字)

*float* 类型的数字**需要**在后加上 *F* 或者 *f*

原因同上，*Java* 中的浮点型字面量默认类型为 *double*，如果直接赋给 *float* 类型可能会导致精度丢失

我们这里采用 *IEEE 754* 浮点数标准来进行讲解，这也是所有语言通用的浮点数表示标准

*IEEE 754* 中使用**规格化数** (*Normalized Number*) 表示大部分浮点数

使用**非规格化数** (*Denormal Number*) 表示非常接近于零的极小数值的一种特殊形式

**浮点类型由符号位 + 指数位 + 尾数位组成**

*float* 类型 4 个字节中的 32 位比特被分为了

1 位符号位

8 位指数位

23 位尾数位

**规格化** *float* 的位运算公式如下

$$(-1)^s \times 2^{e-127} \times (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i})$$

**非规格化** *float* 的计算公式

$$(-1)^s \times 2^{-126} \times (0 + \sum_{i=1}^{23} b_{23-i} 2^{-i})$$

其中 *s* 是指 32 位中的最高位，第 31 位，我们称其**符号位**

*s* = 0 就说明该数字为正数

*s* = 1 就说明该数字为负数

$e$  是指 32 位中的第 30 位至第 23 位，共 8 位，表示 8 位无符号整数，我们称其**指数位**

理论上  $e \in [0, 255]$ ，但是

$e = 0$  用于表示零

$e = 255$  用于表示无穷大或  $NaN$

所以可用的  $e \in [1, 254]$

则实际指数范围  $e - 127 \in [-126, 127]$

$i$  是指第 22 位至第 0 位（共 23 位）中的第  $i$  位，我们**统称**后 23 位和对于隐式补的一位为**尾数位**

所以**尾数位**共有 24 位，**额外补的一位也就是运算公式中尾数中额外加的1或是0，不显式存储**

原本补的这一位是打算占用 23 位中的 1 位的

但是因为对于规格化数最高有效位（ $MSB$ ）总是 1

非规格化数的最高有效位（ $MSB$ ）总是 0

由于这个 1 或者是 0 是固定的，*IEEE* 754 标准规定可以省略存储这一位，从而将节省的 1 位用于增加尾数的精度

---

举个例子

01000000110100000000000000000000

可被分为

0 10000001 101000000000000000000000

$s = 0$  说明该数字为正数

$e = 2^7 + 2^0 = 128 + 1 = 129$ ，则实际指数为  $e - 127 = 2$

我们提取后 23 位中为 1 的有效位

尾数值  $= 1 + (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}) = 1 + 0.5 + 0.125 = 1.625$

则

值  $= (+1) \times 4 \times 1.625 = 6.5$

---

*double* 类型 8 个字节中的 64 位比特被分为了



1 位符号位

11 位指数位

52 位尾数位

**规格化double**的位运算公式如下

$$(-1)^s \times 2^{e-1023} \times (1 + \sum_{i=1}^{52} b_{52-i} 2^{-i})$$

为了防止出现一些数学上的错误导致正常的 *double* 和 *float* 无法表示

比如用正整数除以 0 会产生正无穷大，0 除以 0 和计算负数的平方根不符合数学原理

我们准备了两种特殊的浮点数

- **无穷大** (*Infinity*) 属于规格化数

分为**正无穷大**：符号位  $s = 0$ ，指数位  $E = 255$ （全1），显式尾数位  $M = 0$ （全0）

在 *java* 中为 *Float.POSITIVE\_INFINITY* 或 *Double.POSITIVE\_INFINITY*

- 0 11111111 000000000000000000000000
- 输出：+*Infinity/Inf*

和**负无穷大**：符号位  $s = 1$ ，指数位  $E = 255$ （全1），显式尾数位  $M = 0$ （全0）

在 *java* 中为 *Float.NEGATIVE\_INFINITY* 或 *Double.NEGATIVE\_INFINITY*

- 1 11111111 000000000000000000000000
- 输出：-*Infinity/ - Inf*

相较于 *NaN*，无穷大是**有序**的

*Double.NEGATIVE\_INFINITY == Double.NEGATIVE\_INFINITY* 结果是 *true*

- **NaN** (*Not a Number*) 是一种特殊的数值，用于表示未定义或无效的运算结果

比如  $0/0$  或者  $\sqrt{-1}$

在 *java* 中为 *Float.NaN* 或 *Double.NaN*

**注意**

*NaN* 是**无序**的，任何与 *NaN* 的比较（包括与自身比较）都会返回 *false*

*Double.NaN == Double.NaN* 的结果是 *false*

不过我们可以使用 `Float.isNaN()/Double.isNaN()` 方法来判断一个数是不是 `NaN`

## 1 种用于表示 *Unicode* 编码的字符单元的字符类型 *char*

*char* 使用两个字节进行存储

*char* 类型的字面量值要用单引号括起来

*char* 类型的值可以表示为十六进制值，其范围从 `\u0000` 到 `\Uffff`

*Java* 中，转义序列 `\u` 用于将十六进制数转为对应的 *Unicode* 字符

且在 `\u` 后不需按照书写十六进制字面量的规则加上前缀 `0x` 或 `0X`

例如：`\u2122` 表示注册商标 (*TM*)，`\u03C0` 表示希腊字母  $\pi$

正是由于 *char* 类型的值可以表示为十六进制值，*Java* 有一个非常有趣的特性

因为转义序列 `\u` 还可以出现在加引号的字符常量或字符串之外（而其他所有转义序列不可以）

又由于 *Java* 中我们编写的的所有字符都是 *char* 的一部分

所以我们可以用转义序列 `\u` 写出任意的 *Java* 程序

比如 `public static void main(String[ ] args)`

可以被写作 `public static void main(String\u005B\u005D args)`

`\u005B\u005D` 分别是 `[` 和 `]` 的编码

当然我们也可以更加极端一点

```
1 public \u0073\u0074\u0061\u0074\u0069\u0063 \u0076\u0066\u0069\u0064
   \u0064\u0061\u0069\u0065(\u0053\u0074\u0072\u0069\u0065\u0067\u005B\u005D
   \u0061\u0072\u0067\u0073){}
```

这种特性在使用需要十分小心，我们需要注意以下两点

### 1. *Unicode* 转义序列会在解析代码之前,也就是词法分析阶段得到处理

例如，`"\u0022 + \u0022"` 并不是一个由引号 (`" + 0022`) 包围加号构成的字符串

实际上，`\u0022` 会在解析之前转换为 `"`，这会得到 `" + "`，也就是一个空串

## 2. 一定要注意注释中的 `\u`

因为 *Java* 规定转义序列 `\u` 可以出现在加引号的字符常量或字符串之外  
注释也不例外，这就会导致很多不必要的错误

```
//\u00A0是一个换行符
```

会产生一个语法错误，因为读程序时 `\u00A0` 会替换为一个换行符

类似地，下面这个注释

```
//详情见c:\users
```

也会产生一个语法错误，因为 `\u` 后面并未跟着十六进制数

---

除此之外，*Java* 的 *char* 类型存在一定的局限性

我们先说说 *Unicode*

*Unicode* 通过为全球字符分配**唯一码点** (*Code Point*) 实现统一编码

**码点范围**最初设计为 16 位比特可存储的范围 ( $U + 0000$ 到 $U + FFFF$ )

后扩展至 21 位比特可存储的范围 ( $U + 000000$ 到 $U + 10FFFF$ )，覆盖约 110 万字符

分为 17 个平面，其中基本多语言平面 (*BMP*,  $U + 0000$ – $U + FFFF$ ) 包含常用字符

其余 16 个辅助平面 (如 $U + 10000$ – $U + 10FFFF$ ) 用于扩展 (如中日韩表意文字、表情符号)

因为 *Java* 在设计之初基于 16 位 *char* 类型存储 *Unicode* 字符

但是随着 *Unicode* 字符的拓展

***BMP* 外的字符无法单 *char* 表示**

因此，*Java* 采用了 *UTF-16* 对 *Unicode* 字符进行处理

*UTF-16* **映射**规则如下

对于 *BMP* 中的字符 ( $U + 0000$ – $U + FFFF$ ) 内字符使用**单 16 位代码单元**直接编码，**数值与码点一致**

例如拉丁字母 'A' ( $U + 0041$ ) 的 *UTF-16* 编码为 `0X0041`，在 *java* 中则是 `\u0041`

汉字“汉” ( $U + 6C49$ ) 编码为 `0X6C49`，在 *java* 中则是 `\u6C49`

超出 *BMP* 的字符 ( $U + 0000$ – $U + FFFF$ ) 需通过**代理对** (*Surrogate Pair*) 编码为两个 16 位代码单元

分别是

- **前导代理** (*Lead Surrogate*) : 范围  $U + D800 - U + DBFF$  , 对应码点的高 10 位  
前导代理也叫**高代理区**
- **后尾代理** (*Trail Surrogate*) : 范围  $U + DC00 - U + DFFF$  , 对应码点的低 10 位  
后尾代理也叫**低代理区**

**代理对必须由一个高代理区接着低代理区组成, 不能两个高代理区或者两个低代理区**

```
1 System.out.println("\uD83D\uDE00"); // 输出: 😊
```

保留的  $U + D800 - U + DFFF$  代理区域不映射实际字符

专门用于代理对编码, 确保编码自同步性 (*Self - Synchronizing*)

例如符号  $U + 1D546$  的编码过程为

- 码点减去  $0x10000$  得  $0xD546$  (二进制 0000 1101 0101 0100 0110)
- 前 10 位 0000 1101 01 加  $0xD800$  得前导代理  $0xD835$
- 后 10 位 01 0100 0110 加  $0xDC00$  得后尾代理  $0xDD46$

所以根据上文, *Java* 的 *char* 类型仅表示单个 *UTF - 16* 代码单元, 无法直接处理辅助平面字符

对于在辅助平面的字符, 单独访问任一 *char* 会得到无效结果

这也导致了很多的 *bug* 存在

所以在实际中除了你可以保证只处理 *UTF - 16* 代码单元, 不包含辅助平面字符

或者是你确实需要把辅助平面的字符分开处理之外

我们**强烈建议**不要在程序中使用 *char* 类型和其方法

诸如 *String* 中的方法 *charAt()*

可以使用

*String* 类的 *codePointAt()* 和 *codePointCount()* 方法处理完整码点

```
1 String s = " ";  
2 int codePoint = s.codePointAt(0); // 返回 0x1D546
```

```
3    int charCount = Character.charCount(codePoint); // 返回2
```

1 种用于表示真值的 *boolean* 类型

*boolean* 类型使用 1 字节进行存储

在 *C++* 中，数值甚至指针可以代替 *boolean* 值

值 0 和空指针相当于布尔值 *false*，非 0 值和非空指针相当于布尔值 *true*

而 *Java* 的 *boolean* 是独立的基本类型，仅允许 *true* 和 *false* 两种值

**不支持与其他类型的隐式转换**

除此之外，如果基本的整数和浮点数精度满足不了需求

或者在某些不允许浮点误差的场景下

*java.math* 包中还有两种可以表示任意精度的类，即 *BigInteger* 和 *BigDecimal*

*BigInteger* 实现了任意精度的整型运算

*BigDecimal* 实现了任意精度的浮点型运算

**注意不是新的 *Java* 基础类型**

- 使用静态的 *valueOf()* 方法可以将普通的数值转换为大数值

```
1    import java.math.*;
2    public class Main {
3        public static void main(String[] args){
4            BigInteger bignumber = BigInteger.valueOf(100);
5        }
6    }
```

- 大数类型不支持使用 *+*、*\** 运算符

只能使用 *add()*, *subtract()*, *multiply()*, *divide()*, *mod()* 计算加减乘除和取余

```

1  import java.math.*;
2  public class Main {
3      public static void main(String[] args){
4          BigInteger a = BigInteger.valueOf(100);
5          BigInteger b = BigInteger.valueOf(200);
6          BigInteger c = BigInteger.valueOf(300);
7          a = a.add(b);
8          System.out.println(a == c); //false
9          System.out.println(a.equals(c)); //true
10         System.out.println(a.compareTo(c)); //0
11         a = a.subtract(b);
12         a = a.multiply(b);
13         a = a.divide(BigInteger.valueOf(7));
14         System.out.println(a); //2857
15     }
16 }

```

其中，*BigInteger* 的 *divide()* 如果有小数直接截断

而 *BigDecimal* 的 *divide()* 较为复杂

```

1  // 方法一：指定精度和舍入模式（最常用）
2  BigDecimal divide(BigDecimal divisor, int scale, RoundingMode roundingMode);
3
4  BigDecimal a = new BigDecimal("10");
5  BigDecimal b = new BigDecimal("3");
6  // 保留2位小数，四舍五入
7  BigDecimal result = a.divide(b, 2, RoundingMode.HALF_UP); // 3.33
8
9  // 方法二：使用除数的精度，但需指定舍入模式
10 BigDecimal divide(BigDecimal divisor, RoundingMode roundingMode);
11
12 BigDecimal a = new BigDecimal("10.000");
13 BigDecimal b = new BigDecimal("3");
14 // 结果保留3位小数（与除数精度一致）
15 BigDecimal result = a.divide(b, RoundingMode.HALF_UP); // 3.333
16
17 // 方法三：完全精确除法（仅在结果为有限小数时可用，否则抛异常）
18 BigDecimal divide(BigDecimal divisor);
19
20 BigDecimal a = new BigDecimal("10");
21 BigDecimal b = new BigDecimal("2");
22 BigDecimal result = a.divide(b); // 5.0（精确结果）

```

额外补充三种舍入模式

*RoundingMode.HALF\_DOWN* 五舍六入，负数先取绝对值再五舍六入再负数

*RoundingMode.HALF\_UP* 四舍五入，负数原理同上

*RoundingMode.HALF\_EVEN* 四舍六入五成双，负数原理同上

- 由于大数是对象的原因同样不能用 `==` 直接进行比较内容，需要使用 *equals()* 或者 *compareTo()*
- 

## 10. Java 中字面量的书写技巧

- 第一点，从 *Java 7* 开始，可以为数字字面量加下划线

如用 `1_000_00` (`0b1111_0100_0010_0100_0000`) 表示一百万

这些下划线只是为了让人更易读，*Java* 编译器会去除这些下划线

但不能出现在以下位置

- 数字的开头或结尾 (如 `_100` 或 `100_`)
- 小数点的前后 (如 `3_.14` 或 `3._14`)
- 与符号 (如 `0x`、`0b`) 相邻的位置 (如 `0x_AB`)

感觉是西方的惯用写法，在写数字时使用逗号每三位进行一次分隔，在编程中也沿用了这种设计

没有使用惯用的逗号是因为避免与已有符号冲突，逗号已用于参数列表

且下划线 `_` 在多数键盘布局中易输入

总体是为了 *coders* 更加易读

- 第二点，从 *Java 7* 开始，加上前缀 `0b` 或 `0B` 就可以写二进制数

例如，`0b1001` 就是 9

- 第三点，十六进制数值有一个前缀 `0x` 或 `0X` (如 `0xCAFE`)

八进制有一个前缀 `0`，`010` 对应十进制中的 8

- 第四点，数字字面量可以使用科学计数法来表示以增强可读性

科学计数法使用  $e$  或  $E$  表示 10 的幂次

`double largeNumber = 1.234_567E4; //`等价于 $12.34567 \times 10^4$

---

## 11. 变量名必须是一个以字母开头并由字母或数字构成的序列

需要注意，与大多数程序设计语言相比，*Java* **字母**和**数字**的范围更大

字母包括 ' $A \sim Z$ '、' $a \sim z$ '、'\_'、'\$' 或在某种语言中表示字母的**任何** *Unicode* 字符

同样，数字包括 ' $0 \sim 9$ ' 和在某种语言中表示数字的**任何** *Unicode* 字符

例如，德国的用户可以在变量名中使用字母  $\ddot{a}$ ，希腊人可以用  $\pi$

中国的用户可以使用 `int 变量 = 1;` 这样的初始化语句

- 但 '+' 和 '©' 这样的符号不能出现在变量名中，空格也不行
- 而且尽管 \$ 是一个合法的 *Java* 字符，但不要在你自己的代码中使用这个字符  
它只用在 *Java* 编译器或其他工具生成的名字中
- 另外，不能使用 *Java* 保留字和关键字作为变量名

要确定哪些 *Unicode* 字符属于 *Java* 中的字母

可以使用 *Character* 类的 `isJavaIdentifierStart` 和 `isJavaIdentifierPart` 方法

这两个方法可以帮助你检查一个字符是否可以作为 *Java* 标识符的起始字符或组成部分

```
1 public class UnicodeLetterChecker {
2     public static void main(String[] args) {
3         for (int i = 0; i < 0x10FFFF; i++) {
4             if (Character.isJavaIdentifierStart(i)) {
5                 System.out.println("Character: " + new
String(Character.toChars(i)) + " (U+" + String.format("%04X", i) + ")");
6             }
7         }
8     }
9 }
```

变量名的长度基本没有限制



---

## 12. 在 *Java* 中，变量的声明和定义是合并在一起的，不做区分

声明基础类型变量的同时，不论其是否初始化，都会为其分配内存空间

因为 *Java* 中对于基础数据类型设定了默认值，只要声明就必须分配空间去存储这些默认值

在 *C/C++* 中声明不分配内存，只有定义时才分配

---

## 13. *final* 用于声明一个常量

常量是一种特殊的变量

*final* 表示这个变量只能在初始化时被赋值一次，之后不可对该变量赋值

而且要求这个变量必须在声明的同时就被初始化或者声明后不初始化，但必须在静态或普通构造器中被初始化

可以在方法中声明，也可以在类中方法之外的地方声明

*static final* 用于声明一个类常量

表示在常量的基础上，将该变量静态化

要求这个变量必须在声明的同时就被初始化或者声明后不初始化，但必须在静态构造器中被初始化

且必须在类中方法之外的地方声明

普通常量和类常量均可以在类中的所有方法中使用

一个区别在于

- 普通常量对于每个实例都有自己的一份副本，不能通过类名直接访问，需要创建对象后再访问
  - 类常量对于所有实例共享同一个副本，可以通过类名直接访问，也可创建对象后再访问
- 

## 14. 复合赋值运算符

复合赋值表达式  $E1 \text{ op} = E2$ （例如  $s+ = 1$ ）在语义上等同于  $E1 = (T)((E1)\text{op}(E2))$

其中  $T$  是  $E1$  的类型

---

```
1 代码块 short s = 1;
2    s = s + 1; //错误
3    s = (short)(s + 1); //正确
4    s += 1; // 这行代码竟然可以正常编译和运行!
```

## 15. 对于运算符的一些拓展

对于浮点数的算术运算

虽然 *double* 类型使用 64 位存储一个数值，而有些处理器使用 80 位浮点寄存器

这些寄存器增加了中间过程的计算精度

比如 *double w = x \* y / z;*

很多 *Intel* 处理器计算  $x * y$ ，并且将结果存储在 80 位的寄存器中，再除以  $z$  并将结果截断为 64 位

这样可以得到一个更加精确的计算结果，并且还能够避免产生指数溢出

但是，这个结果可能与始终在 64 位机器上计算的结果不一样

为了 *Java* 的设计哲学：跨平台性良好

一开始 *Java* 虚拟机的**最初规范**规定所有的中间计算都必须进行截断，任何计算都不能提高位数

但是截断计算不仅可能导致溢出，而且由于截断操作需要消耗时间，所以在计算速度上实际上要比精确计算慢

于是，*Java* 虚拟机给出了两套方案

在默认情况下，允许中间计算结果采用扩展的精度并在之后截断

但是，如果一个方法或一个类用了 *strictfp* 关键字进行修饰

```
public static strictfp void main(String[ ] args){}
```

那么所有的浮点运算和**api调用**都会严格按照指定的精度（比如 *float* 的 32 位或 *double* 的 64 位）执行截断

避免不同平台上的硬件优化导致结果细微差异

*strictfp* 确保了跨平台性，但是可能会造成许多麻烦，比如溢出，精度损失和截断带来的时间损耗

默认情况下确保了精度和不会溢出，但是可能会导致跨平台性损失

在正常使用中，如果不需要极强的跨平台性，不建议使用 *strictfp*

而且真的到了需要使用 *strictfp* 的地步，完全可以使用 *BigInteger* 或者 *BigDecimal*

比用 *strict fp* 去修补浮点数的短板要省心得多

---

## 16. *sqrt()* 的深挖

*Math.sqrt(double a)* 接受一个 *double* 非负类型的参数，返回它的 *double* 类型非负平方根

- 三种特殊情况
  - 如果是传参为负数，返回 *NaN*
  - 如果传参为  $-0.0$  返回  $-0.0$ （虽然在大多数场景下和  $0.0$  等价）
  - *Math.sqrt(Double.POSITIVE\_INFINITY)* 返回 *POSITIVE\_INFINITY*
- 一种特殊版本

*StrictMath.sqrt(double a)* 类似 *strict fp* 保证跨平台一致性

如果在被 *strict fp* 修饰的类或方法中使用 *Math.sqrt(double a)* 也是一样的

---

- 具体实现

### 牛顿迭代法

具体来说，假设我们要计算某个正数  $a$  的平方根，等价于求  $x$  使得  $x^2 = a$

牛顿迭代法的核心思想是基于函数  $f(x) = x^2 - a$

找到  $f(x) = 0$  的根，此时  $x^2 = a$

利用牛顿迭代公式：

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

其中  $f(x) = x^2 - a$ ，导数  $f'(x) = 2x$

代入后，公式简化为：

$$x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{x_n^2 + a}{2x_n}$$

最终使用的公式是  $x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2}$

这个公式可以看作是对  $x_n$  的更新，通过不断迭代， $x_n$  会逐渐趋近于  $\sqrt{a}$

有两种推导出牛顿迭代公式的方法

### 泰勒公式--代数法

假设  $x^*$  是  $f(x) = 0$  的根，而  $x_n$  是当前估计值， $x_{n+1}$  是我们希望得到的更接近  $x^*$  的值

可以将函数  $f(x)$  在  $x_n$  附近进行泰勒展开

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{f''(x_n)}{2}(x - x_n)^2 + \dots$$

设  $x = x^*$

将泰勒展开截断到一阶（线性近似），忽略高阶项（当  $x$  很接近  $x_n$  时，高阶项很小）

$$f(x^*) \approx f(x_n) + f'(x_n)(x^* - x_n)$$

由于  $f(x^*) = 0$ ，代入得：

$$0 = f(x_n) + f'(x_n)(x^* - x_n)$$

解出  $x^* - x_n$  为

$$x^* - x_n = -\frac{f(x_n)}{f'(x_n)}$$

两边同时加上  $x_n$

$$x^* = x_n - \frac{f(x_n)}{f'(x_n)}$$

这里  $x^*$  是真正的根，但我们不知道它的确切位置

于是，我们用  $x_{n+1}$  代替  $x^*$  作为下一次迭代的估计值，得到

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

### 切线--几何法

$f(x_n)$  是函数在  $x_n$  处的值，表示当前点与  $x$  轴（ $y = 0$ ）的垂直距离

$f'(x_n)$  是函数在  $x_n$  处的导数，也就是切线的斜率

切线方程为  $y - f(x_n) = f'(x_n)(x - x_n)$

我们希望找到切线与  $x$  轴的交点（即  $y = 0$  的位置）

设  $x = x_{n+1}$ ，得  $0 - f(x_n) = f'(x_n)(x_{n+1} - x_n)$

$$\text{化简得 } x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

这个交点  $x_{n+1}$  通常比  $x_n$  更接近真实的根，因为切线是对函数局部行为的线性近似

## 具体实现

```
1  public class Main {
2      static final double PRECISION = 1E-14;
3      public static void main(String[] args) {
4          int temp = 2;
5          System.out.println(Math.abs(MyMath.sqrt(temp) - Math.sqrt(temp)) <
6      PRECISION);
7      }
8  }
9  class MyMath {
10     public static double sqrt(double x) {
11         if (Double.isNaN(x)) return Double.NaN;
12         if (x == -0.0) return -0.0;
13         if (x == Double.POSITIVE_INFINITY) return Double.POSITIVE_INFINITY;
14         if (x < 0) return Double.NaN;
15
16         double res = (x >= 1) ? x / 2 : 1.0;
17         double precision = Main.PRECISION;
18
19         while (true) {
20             double nextX = (res + (x / res)) / 2;
21             if (Math.abs(nextX - res) < precision) {
22                 break;
23             }
24             res = nextX;
25         }
26         return res;
27     }
28 }
```

### 17. `pow()` 的深挖

*Math.pow(double a, double b)*

返回  $a$  的  $b$  次幂，即  $a^b$ ，结果为 *double* 类型

- 特殊情况（一下特殊情况分先后，即代码实现时要先实现靠前的特殊情况判断）
  - 如果底数或指数是 *NaN*，则结果为 *NaN*

- 指数为 0.0 或底数为 1.0 时直接返回 1.0
- 底数为 0 如果指数大于 0 返回 0.0，如果指数小于 0 返回
- 如果底数是 *Infinity* 且指数大于 0，结果为 *Infinity*，如果指数小于 0，结果为 0.0
  - 如果底数是负无穷且指数为整数，结果取决于指数的奇偶性（奇数返回负无穷，偶数返回正无穷）
- 如果指数是负数，先计算指数的绝对值的结果，再取反
- 如果指数是分数，使用  $a^b = e^{b \ln a}$ ，但这要求  $a > 0$ 
  - 如果  $a < 0$  且  $b$  不是整数，结果为 *NaN*，因为负数的非整数次幂在实数范围内未定义
  - 如果  $a = 0$  且  $b > 0$ ，结果为 0.0

```

1  import static java.lang.Math.*;
2
3  public class MyMath {
4      public static double myPow(double a, double b) {
5          // 处理 NaN
6          if (Double.isNaN(a) || Double.isNaN(b)) {
7              return Double.NaN;
8          }
9
10         // 处理指数 b = 0
11         if (b == 0) {
12             return 1.0;
13         }
14
15         // 处理底数 a = 0
16         if (a == 0) {
17             if (b > 0) {
18                 return 0.0;
19             } else if (b < 0) {
20                 return Double.POSITIVE_INFINITY;
21             } else {
22                 return 1.0; // b = 0, 已在上面处理
23             }
24         }
25
26         // 处理无穷大
27         if (Double.isInfinite(a)) {
28             if (a > 0) { // 正无穷
29                 if (b > 0) {
30                     return Double.POSITIVE_INFINITY;
31                 } else if (b < 0) {
32                     return 0.0;

```

```

33         } else {
34             return 1.0; // b = 0
35         }
36     } else { // 负无穷
37         if (b % 1 == 0) { // 指数是整数
38             if ((long) b % 2 == 0) { // 偶数次幂
39                 return myPow(-a, b); // 返回正无穷
40             } else { // 奇数次幂
41                 return -myPow(-a, b); // 返回负无穷
42             }
43         } else { // 指数不是整数
44             return Double.NaN;
45         }
46     }
47 }
48
49 // 处理负底数且指数不是整数
50 if (a < 0 && b % 1 != 0) {
51     return Double.NaN; // 实数范围内未定义
52 }
53
54 // 处理整数指数（快速幂算法）
55 if (b % 1 == 0) {
56     long exponent = (long) b;
57     double result = 1.0;
58     double base = a;
59     boolean negativeExponent = exponent < 0;
60     if (negativeExponent) {
61         exponent = -exponent;
62     }
63     while (exponent > 0) {
64         if (exponent % 2 == 1) {
65             result *= base;
66         }
67         base *= base;
68         exponent /= 2;
69     }
70     return negativeExponent ? 1.0 / result : result;
71 }
72
73 // 处理分数指数
74 if (a > 0) {
75     return exp(b * log(a)); // 使用  $e^{(b * \ln(a))}$  计算
76 } else {
77     return Double.NaN; // a < 0 且 b 不是整数
78 }
79 }

```

```

80
81     public static void main(String[] args) {
82         // 测试用例
83         System.out.println(myPow(2, 3));    // 8.0
84         System.out.println(myPow(2, -1));   // 0.5
85         System.out.println(myPow(0, 0));    // 1.0
86         System.out.println(myPow(-2, 2));   // 4.0
87         System.out.println(myPow(-2, 3));   // -8.0
88         System.out.println(myPow(4, 0.5));  // 2.0
89         System.out.println(myPow(-4, 0.5)); // NaN
90         System.out.println(myPow(0, 5));    // 0.0
91         System.out.println(myPow(0, -1));   // Infinity
92         System.out.println(myPow(Double.POSITIVE_INFINITY, 2)); // Infinity
93         System.out.println(myPow(Double.POSITIVE_INFINITY, -1)); // 0.0
94         System.out.println(myPow(Double.NEGATIVE_INFINITY, 3)); // -Infinity
95     }
96 }

```

## 18. 一些其他常用的数学方法

- *floorMod()* 方法

在大多数编程语言（C/C++、Java、JavaScript）中

% 运算符对负数的处理遵循

**余数的符号与被除数（第一个操作数）一致**

但是不满足欧几里得除法定义的余数非负性，即余数应满足  $0 \leq r < |b|$

欧几里得取模的核心是**向下取整除法**

其公式为

$$\text{floorMod}(a, b) = a - b \times \lfloor a/b \rfloor$$

$\lfloor a/b \rfloor$  是商向下取整的结果

```

1     public static int floorMod(int a, int b) {
2         int mod = a % b;
3         return (mod >= 0) ? mod : mod + Math.abs(b);
4     }

```

- *Math.sin()*, *Math.cos()*, *Math.tan()*, *Math.atan2()*



常用的三角函数

其中  $\text{Math.atan2}()$  是反正切函数，即  $\arctan$

一般我们只使用  $\text{Math.atan2}()$ ，因为  $\text{Math.atan}()$  容易出错

- $\text{Math.exp}(), \text{Math.log}(), \text{Math.log10}()$

$\text{Math.exp}()$  是  $e^x$

$\text{Math.log}()$  是  $\ln x$

$\text{Math.log10}()$  是  $\log_{10} x$

- 两个数学常量  $\text{Math.PI}$ ， $\text{Math.E}$

- 互质函数  $\text{Math}$  中没有给出实现

我们这里给出两种实现方法

### 1. 递归法

```
1 public static int gcdRecursive(int a, int b) {
2     if (b == 0) {
3         return Math.abs(a); // 处理负数
4     }
5     return gcdRecursive(b, a % b);
6 }
```

### 2. 迭代法（更优）

```
1 public static int gcdIterative(int a, int b) {
2     a = Math.abs(a);
3     b = Math.abs(b);
4     while (b != 0) {
5         int temp = b;
6         b = a % b;
7         a = temp;
8     }
9     return a;
10 }
```

- 反转函数 *Math* 中没有给出实现

```
1  public class NumberReverser {
2      public static int reverse(int x) {
3          int reversed = 0;
4          while (x != 0) {
5              int pop = x % 10; // 获取当前最后一位数字
6              x /= 10;          // 移除已处理的最后一位
7
8              // 检查反转过程中是否会导致溢出
9              if (reversed > Integer.MAX_VALUE / 10 ||
10                 (reversed == Integer.MAX_VALUE / 10 && pop > 7)) {
11                  return 0; // 正数溢出
12              }
13              if (reversed < Integer.MIN_VALUE / 10 ||
14                 (reversed == Integer.MIN_VALUE / 10 && pop < -8)) {
15                  return 0; // 负数溢出
16              }
17
18              reversed = reversed * 10 + pop; // 构建反转后的数字
19          }
20          return reversed;
21      }
```

### 正数溢出

若当前反转值 *reversed* 超过 *Integer.MAX\_VALUE*/10

或等于该值但新加入的数字 *pop* > 7（因 *Integer.MAX\_VALUE* = 2147483647 最后一位为 7），则返回 0

### 负数溢出

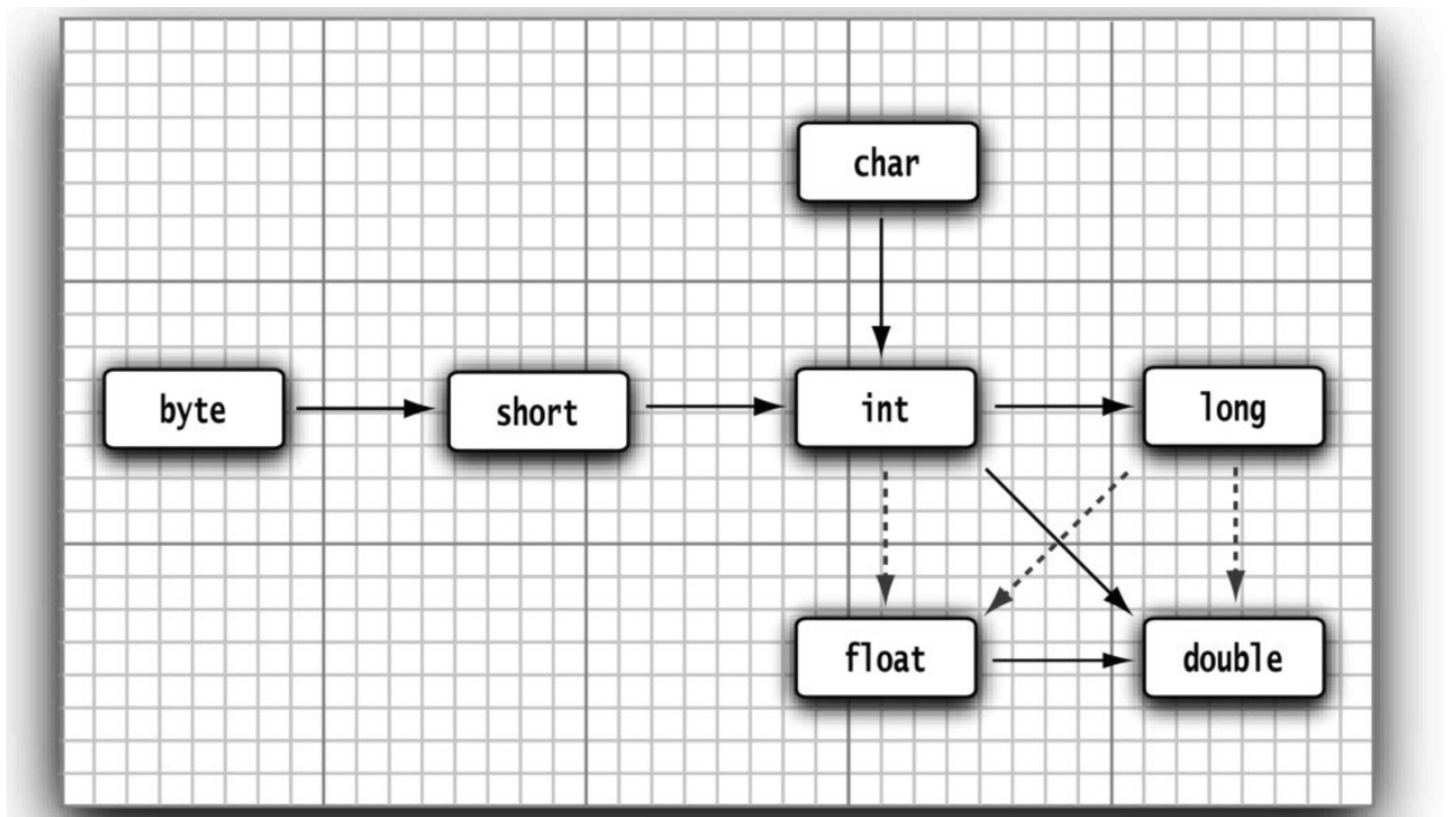
若 *reversed* 低于 *Integer.MIN\_VALUE*/10

或等于该值但 *pop* < -8（因 *Integer.MIN\_VALUE* = -2147483648，最后一位为 -8），则返回 0

## 18. 所有的 *Math* 类方法都可以替换为 *StrictMath* 类方法

其作用与加上 *strictfp* 关键字一致

## 19. 数据类型转换



有 6 个实心箭头，表示无信息丢失的转换

有 3 个虚箭头，表示可能有精度损失的转换

当两个不同类型的操作数进行计算  $(n + f)$  时

系统会按照以下优先级隐式地将较低优先级的类型转换为较高优先级的类型

1. 最高优先级 *double*

若任意一个操作数为 *double*，另一个操作数必须转换为 *double*

2. 次优先级 *float*

若没有 *double*，但有一个操作数为 *float*，另一个操作数必须转换为 *float*

3. 第三优先级 *long*

若没有 *double* 或 *float*，但有一个操作数为 *long*，另一个操作数必须转换为 *long*

4. 最低优先级 *int*

若所有操作数均为整型 (*byte/short/int/char*)，则统一转换为 *int* 类型

除了这些隐式转换

我们也可以进行显式转换

- 如果试图将一个数值从一种类型强制转换为另一种类型，而又超出了目标类型的表示范围结果就会截断成一个完全不同的值

例如，`(byte) 300` 的实际值为 44

- 而且不要在 `boolean` 类型与任何数值类型之间进行强制类型转换，这样可以防止发生错误
- 只有极少数的情况才需要将布尔类型转换为数值类型，这时可以使用条件表达式 `b?1:0`

如果想对浮点数进行舍入运算，以便得到最接近的整数

可以不用强制转换

可以使用 `Math.round()` 进行**四舍六入五留双**

对于 `Math.round()` 需要注意

其返回值是 `long` 类型

所以我们一般要对它的返回值进行 `long -> int` 的强制转换

```
1 float num1 = 9.997;
2 int intnum1 = (int)Math.round(num1);
```

而且对于中间值 0.5 的情况

*Java* 采用“银行家舍入法”（向偶取整）

即如果小数值为 0.5 时

整数位为偶数转换为整数位，整数位为奇数转换为整数位 +1

## 20. >> 和 >>> 的区别

对于 >>，普通右移来说

正数左边空出来的位置填 0（比如 `8 >> 1` 变成 4）

负数左边空出来的位置填 1（比如 `-8 >> 1` 变成 -4）

但对于 >>>，无符号右移来说

不论正负，左边空出的位全部填 0

但是注意这并不直接相当于对普通右移后的结果加绝对值

对于正数来说，两种右移完全一致

对于负数来说，无符号右移会得到完全不同的一个正数

举个例子

-8 的二进制是 11111111 11111111 11111111 11111000

$-8 \gg 1$  变成 11111111 11111111 11111111 11111100（还是负数，值是  $-4$ ）

但是  $-8 \ggg 1$  变成 01111111 11111111 11111111 11111100，这时值变为 2147483644

## 21. Java 没有内置的字符串类型

而是在标准 Java 类库中提供了一个预定义类，很自然地叫做 *String*

每个用双引号括起来的字符串都是 *String* 类的一个实例

- Java 语言允许使用  $+$  号连接（拼接）两个字符串

当将一个字符串与一个非字符串的值进行拼接时，后者被转换成字符串

**任何一个 Java 对象都可以转换成字符串**

- 如果需要把多个字符串放在一起，用一个定界符分隔，可以使用静态 *String.join()* 方法

```
1  String joinedString = String.join(定界符, 元素列表);
2
3  //第一种用法, 连接数组中的字符串
4  String[] fruits = {"苹果", "香蕉", "橘子"};
5  String result = String.join(" - ", fruits);
6  // 输出: 苹果 - 香蕉 - 橘子
7
8  //第二种用法, 连接集合中的字符串
9  List<String> colors = Arrays.asList("红", "绿", "蓝");
10 String result = String.join(" | ", colors);
11 // 输出: 红 | 绿 | 蓝
12
13 //第三种用法, 直接连接
14 String result = String.join(",", "中国", "美国", "日本");
15 // 输出: 中国、美国、日本
```

- Java 中的 *String* 对象是不可变的，字符串变量相当于一个指向 *String* 对象的引用

变量可以任意指向别的对象，但是不可以直接对 *String* 对象进行修改

任何对字符串的操作，如 *concat()*、*substring()*、*toUpperCase()* 都会创建新对象，原对象保持不变

不可变 *String* 的核心优点是**编译器可以让字符串共享**

缺点也很明显，无法便捷快速地修改

具体的实现方式是

1. 当通过字面量 `String s = "abc"`; 创建字符串变量时

可以想象将各种 `String` 对象存放在**字符串常量池**中



**字符串常量池**位于堆，支持动态扩展和垃圾回收

当我们创建一个字符串变量的时候

现在字符串常量池中进行查询是否有要赋给字符串变量的 `String` 对象

如果有，直接将这个变量指向这个 `String` 对象

如果没有，先在字符串常量池中创建一个 `String` 对象，再引用

如果复制一个字符串变量，直接将新的变量指向原来变量指向的 `String` 对象即可

```
1 String s1 = "hello";           // 第一次创建，存入常量池
2 String s2 = "hello";           // 直接引用池中已有的 "hello"
3 System.out.println(s1 == s2); // true (内存地址相同)
```

2. `new String("abc")` 时

当通过 `new` 创建字符串 `String s = new String("abc")`; 时

`String` 对象 `"abc"` 会先进入常量池（如果尚未存在）

然后 `new` 会在堆内存中创建一个新对象，而不会复用常量池中的对象（除非显式调用 `intern()`）

```
1 String s1 = new String("abc"); // 常量池已有"abc", s1指向堆中的新对象
2 String s2 = "abc";             // s2直接指向常量池中的"abc"
3 System.out.println(s1 == s2);  // false (引用不同对象)
```

3. `String a = "Hello" + "World"`

这涉及到 `java` 中的**常量折叠**，常量折叠是编译期间的优化

编译前: `"Hello" + "World"`

编译后: `"HelloWorld"`（字符串直接合并）

那么我们就只检测字符串常量池中是否存在 `"HelloWorld"`，而不是分开查找

这样只创建一个对象或不创建新的对象

- 如果想要检测两个字符串变量所指向的 *String* 对象**内容一样**

使用 *equals()*

*equalsIgnoreCase()* 忽略大小写

```
1  String s = "Hello";
2  String t = new String("Hello");
3
4  System.out.println(s.equals(t));           // true (内容相同)
5  System.out.println(s.equalsIgnoreCase("HELLO")); // true (忽略大小写)
```

如果想要检测两个字符串变量所指向的**地址一样**

使用 *==*

```
1  String s1 = "Hello";
2  String s2 = "Hello"; // 复用常量池中的对象
3  System.out.println(s1 == s2); // true
```

- 空串是一个 *Java* 对象，有自己的串长度 0 和内容（空）

*String* 变量还可以存放一个特殊的值，名为 *null*，这表示目前没有任何对象与该变量关联

```
1  if(str != null || str.length() != 0)
2  //用于判断一个String变量是不是空串或null
```

## 22. 一些经典 *String* 类 *api*

1. *char charAt (int index)* 返回给定位置的代码单元

2. *int compareTo(String other)*

从索引 0 开始，逐个比较对应位置的字符 *Unicode* 值，直到找到第一个不同的字符

若发现差异，返回当前字符 *Unicode* 值  $-$  *other* 字符 *Unicode* 值

若所有字符相同，但长度不同，返回当前长度  $-$  *other* 长度

""（空字符串）的字典序小于任何非空字符串

*s1.compareTo(s2) == 0* 等价于 *s1.equals(s2)*，但前者更高效，可提前终止比较

### 3. *boolean startsWith(String prefix)* 和 *boolean endsWith(String suffix)*

如果字符串以 *suffix* 开头或结尾，则返回 *true*

### 4. *int indexOf(String str, int fromIndex)* 和 *int indexOf(int cp, int fromIndex)*

用于从 *fromIndex* 开始查找并返回与字符串 *str* 或代码点 *cp* 匹配的第一个子串的开始位置

如果未找到返回  $-1$

可以不提供 *fromIndex* 参数，此时从字符串开头开始寻找

*int lastIndexOf(String str, int fromIndex)* 和

*int lastIndexOf(int cp, int fromIndex)*

用于从 *fromIndex* 开始查找并返回与字符串 *str* 或代码点 *cp* 匹配的最后一个子串的开始位置

如果未找到返回  $-1$

可以不提供 *fromIndex* 参数，此时从字符串末尾开始寻找

### 5. *int length()*

返回 *UTF-16* 代码单元数，非实际字符数

### 6. *String replace(CharSequence oldString, CharSequence newString)*

返回一个用 *newString* 代替原始字符串中所有的 *oldString* 后的新字符串

```
1 String s = "apple, apple pie";
2 String sNew = s.replace("apple", "banana"); // "banana, banana pie"
```

### 7. *String substring(int beginIndex, int endIndex)*

提取子串

注意是左闭右开区间



## 8. *String toLowerCase()* 和 *String toUpperCase()*

这个字符串将原始字符串中的大写字母改为小写，或者将原始字符串中的所有小写字母改成了大写字母

## 9. *String trim()* 返回一个删除了原始字符串头部和尾部空格的新字符串

## 23. 由于 *String* 对象的不可变性

在 *JDK5.0* 中引入 *StringBuilder* 类，其对象可变

这个类的前身是 *StringBuffer*，其效率低，但允许多线程

*StringBuilder* 效率高，不允许多线程

这两个类的 *API* 是**完全相同**的

一些经典 *StringBuilder* 类 *api*

- 创建一个空的 *StringBuilder* 对象

```
1  StringBuilder builder = new StringBuilder();
```

- *String toString()*

返回一个与构建器内容相同的字符串

- *StringBuilder append(String str)* 追加一个字符串并返回 *this*

*StringBuilder append(char c)* 追加一个代码单元并返回 *this*

- *StringBuilder insert(int offset, String str)* 在 *offset* 位置插入一个字符串并返回 *this*

*StringBuilder insert(int offset, Char c)* 在 *offset* 位置插入一个代码单元并返回 *this*

- *StringBuilder delete(int startIndex, int endIndex)*

删除偏移量从 *startIndex* 到 *endIndex - 1* 的代码单元并返回 *this*

左闭右开

24. *Java* 中打印输出到“标准输出流”（即控制台窗口）是一件非常容易的事情，只需调用 *System.out.println()*

而读取输入则需要使用到 *Scanner* 类，*Scanner* 类定义在 *java.util* 包中

当使用的类不是定义在基本 *java.lang* 包中时，一定要使用 *import* 指示字将相应的包加载进来

首先需要构造一个 *Scanner* 对象，并与“标准输入流” *System.in* 关联

*Scanner(InputStream in)* 用给定的输入流创建一个 *Scanner* 对象

```
1  import Java.util.Scanner;
2  Scanner scanner = new Scanner(System.in);
3  int temp = scanner.nextInt();
4  double num = scanner.nextDoule();
5  String word = scanner.next();
6  String line = scanner.nextLine();
```

常用方法如下

*String nextLine()* 读取输入的下一行内容

*String next()* 读取输入的下一个单词，以空格作为分隔符

*int nextInt()*

*double nextDouble()*

一些检测方法

*boolean hasNext()* 检测输入中是否还有其他单词

*boolean hasNextInt()* 和 *boolean hasNextDouble()* 检测是否还有表示整数或浮点数的下一个字符序列

25. *System.out.println()* 输出内容后自动换行

*System.out.print()* 输出内容后不换行

以上是 *Java* 中输出的简单方法

*Java* 也提供了复杂地格式化输出的方法 *System.out.printf()*

格式说明符如下

%[参数索引\$][标志][宽度][.精度]转换符

- 参数索引语法

格式说明符中通过 %n\$ 指定参数索引，其中 n 是参数的序号（从 1 开始）

%1\$s 表示第一个参数按字符串格式处理

%2\$d 表示第二个参数按十进制整数处理

- 常用标志

标 志	目 的	举 例
+	打印正数和负数的符号	+3333.33
空格	在正数之前添加空格	3333.33
0	数字前面补 0	003333.33
-	左对齐	3333.33
(	将负数括在括号内	( 3333.33 )
,	添加分组分隔符	3,333.33
# (对于 f 格式)	包含小数点	3,333.
# (对于 x 或 0 格式)	添加前缀 0x 或 0	0xcafe
\$	给定被格式化的参数索引。例如，%1\$d，%1\$x 将以十进制和十六进制格式打印第 1 个参数	159 9F
<	格式化前面说明的数值。例如，%d%<x 以十进制和十六进制打印同一个数值	159 9F

- 宽度是指输出的最小字符数，不足时填充
- 精度是指浮点数的小数位数或字符串最大长度
- 常用转换符

转换符	类 型	举 例	转换符	类 型	举 例
d	十进制整数	159	s	字符串	Hello
x	十六进制整数	9f	c	字符	H
o	八进制整数	237	b	布尔	True
f	定点浮点数	15.9	h	散列码	42628b2
e	指数浮点数	1.59e+01	tx 或 Tx	日 期 时 间 (T 强制大写)	已经过时，应当改为使用 java.time 类，参见卷 II 第 6 章
g	通用浮点数	—	%	百分号	%
a	十六进制浮点数	0x1.fccdp3	n	与平台有关的 行分隔符	—

26. 除了进行标准输出和标准输入

我们还可以进行文件输出和文件输入

我们需要一个用 File 对象构造一个 Scanner 对象

- 读取操作

```
1 import java.util.Scanner;
2 Scanner file = new Scanner(Paths.get("myfile.txt") , "UTF-8");
```

如果文件名中包含反斜杠符号，就要记住在每个反斜杠之前再加一个额外的反斜杠

“c : \\mydirectory\\myfile.txt”

- 写入操作

要想写入文件，就需要构造一个 *PrintWriter* 对象

注意这里我们需要导入的是 *java.io* 包中的 *PrintWriter* 对象

```
1 import java.io.PrintWriter;
2 PrintWriter out = new PrintWriter("myfile.txt" , "UTF-8");
3 // 写入内容
4 out.println("Hello, World!");
5 out.printf("数字示例: %d, 浮点数示例: %.2f", 42, 3.14159);
```

如果文件不存在，创建该文件

可以像输出到 *System.out* 一样使用 *print*、*println* 以及 *printf* 命令

- 在输入文件名的时候

分为相对路径和绝对路径

相对路径是指相对于 *Java* 虚拟机 (*JVM*) 启动时的当前工作目录

```
1 // 同级目录下的文件
2 new PrintWriter("myfile.txt");
3 // 子目录中的文件
4 new PrintWriter("mydirectory/myfile.txt");
5 // 上级目录中的文件
6 new PrintWriter("../myfile.txt");
```

如果在命令行方式下用下列命令启动程序

相对路径就是 *cmd* 的当前路径

如果使用 *IDE* 启动

相对路径则需要使用 *System.getProperty("user.dir")*; 获取

```
1 String currentDir = System.getProperty("user.dir");
2 System.out.println("当前工作目录: " + currentDir);
```

绝对路径就是从盘符开始写路径

比如 “*c : \\mydirectory\\myfile.txt*”

- 在实践中我们推荐直接使用 *Paths.get* (文件名) 方法获取路径

27. 块（即复合语句）是指由一对大括号括起来的若干条简单的 *Java* 语句

- 块确定了变量的作用域
- 块可以嵌套在另一个块中

但是，不能在嵌套的两个块中声明同名的变量

这与 *C++* 不同，*C++* 支持覆盖

28. *java* 中的 *switch* 语句需要注意直通现象

即如果没有写 *break* 语句，则会执行所有的 *case*

```
1 int code = 2;
2 switch (code) {
3     case 1:
4         System.out.println("执行操作1");
5     case 2:
6         System.out.println("执行操作2"); // 无break, 继续执行case 3
7     case 3:
8         System.out.println("执行操作3"); // 无break, 继续执行default
9     default:
```

```

10         System.out.println("默认操作");
11     }
12     // 输出:
13     // 执行操作2
14     // 执行操作3
15     // 默认操作

```

所以如果需要使用 *switch* 语句，一定记得写 *break*

如果确实需要使用直通

可以使用 `@SuppressWarnings("fallthrough")` 抑制编译器的警告

```

1  @SuppressWarnings("fallthrough")
2  public void process(int code) {
3      switch (code) {
4          case 1:
5              prepare();
6              // 故意fallthrough
7          case 2:
8              execute();
9              break;
10     }
11 }

```

*switch* 中的 *case* 支持多种类型

1. 类型为 *char*、*byte*、*short* 或 *short* 的常量表达式
2. *Java5* 后增加支持枚举类型

直接使用枚举常量，无需枚举名前缀

```

1  enum Season { SPRING, SUMMER, AUTUMN, WINTER }
2
3  Season season = Season.WINTER;
4  switch (season) {
5      case WINTER: // 直接写WINTER, 而非Season.WINTER
6          System.out.println("冬季");
7          break;
8      // 其他case...
9  }

```

### 3. Java7 后增加支持字符串

字符串字面量作为 *case* 标签，区分大小写

```
1 String fruit = "Apple";
2 switch (fruit) {
3     case "apple":
4         System.out.println("小写苹果");
5         break;
6     case "Apple": // 匹配成功
7         System.out.println("首字母大写苹果");
8         break;
9 }
```

### 29. *break* 和 *continue* 语句都支持 *label* 跳转

如果不使用 *label* , *break* 和 *continue* 默认作用于最近的循环

若要使用 *label* , 则在在块前用 标签名 : 标记代码块

这里的标签名和变量标识符类似, 只要遵循标识符规则可以任意起名

*break* 标签名;

*continue* 标签名;

会跳转到指定标签的循环进行操作

### 30. 数组

- 在 *Java* 中, 允许数组长度为 0 , 但是长度为 0 的数组和 *null* 不一样
- 声明数组的两种写法

```
1 int[] nums = new int[10];
2 //也可以写成下面的, 不过及其不常用
3 int nums[] = new int[10];
```

- 创建数组的三种写法

```
1  int[] nums = new int[10]; //动态创建不初始化
2  int[] nums_1 = new int[]{1,23,4}; //静态初始化, 直接在声明时赋值
3  int[] nums_2 = {1,1,1,1}; //静态初始化, 直接在声明时赋值
```

- 数组的默认值

创建一个数字数组时, 所有元素都初始化为 0 或者 0.0

*boolean* 数组的元素会初始化为 *false*

*char* 数组默认值 `\u0000` (不可见空字符)

对象数组的元素则初始化为一个特殊值 *null*, 这表示这些元素未存放任何对象

- 数组的复制与扩容

使用 *static type copyOf(type[] a, int length)*

注意先导入 *import java.util.Arrays;*

如果需要复制

```
1  import java.util.Arrays;
2  int[] oldnums = {1,2,3};
3  int newnums[] = Arrays.copyOf(oldnums , oldnums.length);
```

如果需要扩容

```
1  import java.util.Arrays;
2  int[] oldnums = {1,2,3};
3  oldnums = Arrays.copyOf(oldnums , 2 * oldnums.length);
```

如果需要更细化的操作, 可以使用进阶版

*static type copyOfRange(type[] a, int start, int end)*

返回一个新数组, 包含从 *start* 到 *end* - 1 的元素, 左闭右开

如果 *end* 大于源数组长度, 超出部分会用默认值填充



- 数组的排序，查找与填充

`static void sort(type[] a)` 使用快速排序，默认进行升序排序

`static int binarySearch(type[] a, int start, int end, type target)` 对数组进行二分查找

`static void fill(type[] a, type v)` 将数组的所有数据元素值设置为  $v$

- 命令行参数

之前在写 `main` 方法之前都会固定传入一个 `String[] args` 参数

这个参数表明 `main` 方法将接收一个字符串数组，也就是命令行参数

- 传入命令行参数

如果使用 `cmd`，直接在执行命令的文件名后传入

```
1 javac CommandLineArgsDemo.java
2 java CommandLineArgsDemo arg1 arg2 arg3
```

在 `IDE` 中需要根据平台自行配置

- 参数以空格分隔，若参数本身包含空格，需用引号包裹

```
1 java CommandLineArgsDemo "Hello World" 123
```

- 使用前需检查参数数量，避免 `ArrayIndexOutOfBoundsException`

若未传入参数，`args` 为空数组 (`length = 0`)，而非 `null`

```
1 if (args.length < 1) {
2     System.err.println("请提供至少一个参数!");
3     return;
4 }
```

- 两个辅助函数

```
static String toString(type[] a)
```

除了将一维数组转化为字符串，还可以用来输出一个一维数组

```
1 System.out.println(Arrays.toString(matrix));
```

```
static boolean equals(type[] a,type[] b)
```

- 多维数组

严格意义上

Java 实际上没有多维数组，只有一维数组

多维数组被解释为 **数组的数组**

这也就导致我们可以创建不规则的多维数组，即数组的每一行有不同的长度

多维数组的声明方式也类似一维数组

```
1 //二维数组
2 int[][] temp = new int[5][6];
3 int[] temp1[] = new int[5][6];
4 int temp2[][] = new int[5][6];
5
6 //三维数组
7 int[][][] temp = new int[5][6][7];
8 int[][] temp1[] = new int[5][6][7];
9 int[] temp2[][] = new int[5][6][7];
10 int temp3[][][] = new int[5][6][7];
```

初始化方式也类似

```
1 int[][] temp0 ={
2     {1,2,3,4},
3     {1,2,3,4},
4     {1,2,3,4},
5     {1,2,3,4}
6 };//静态初始化
7 int[][] temp1 = new int[5][5];//动态初始化
```

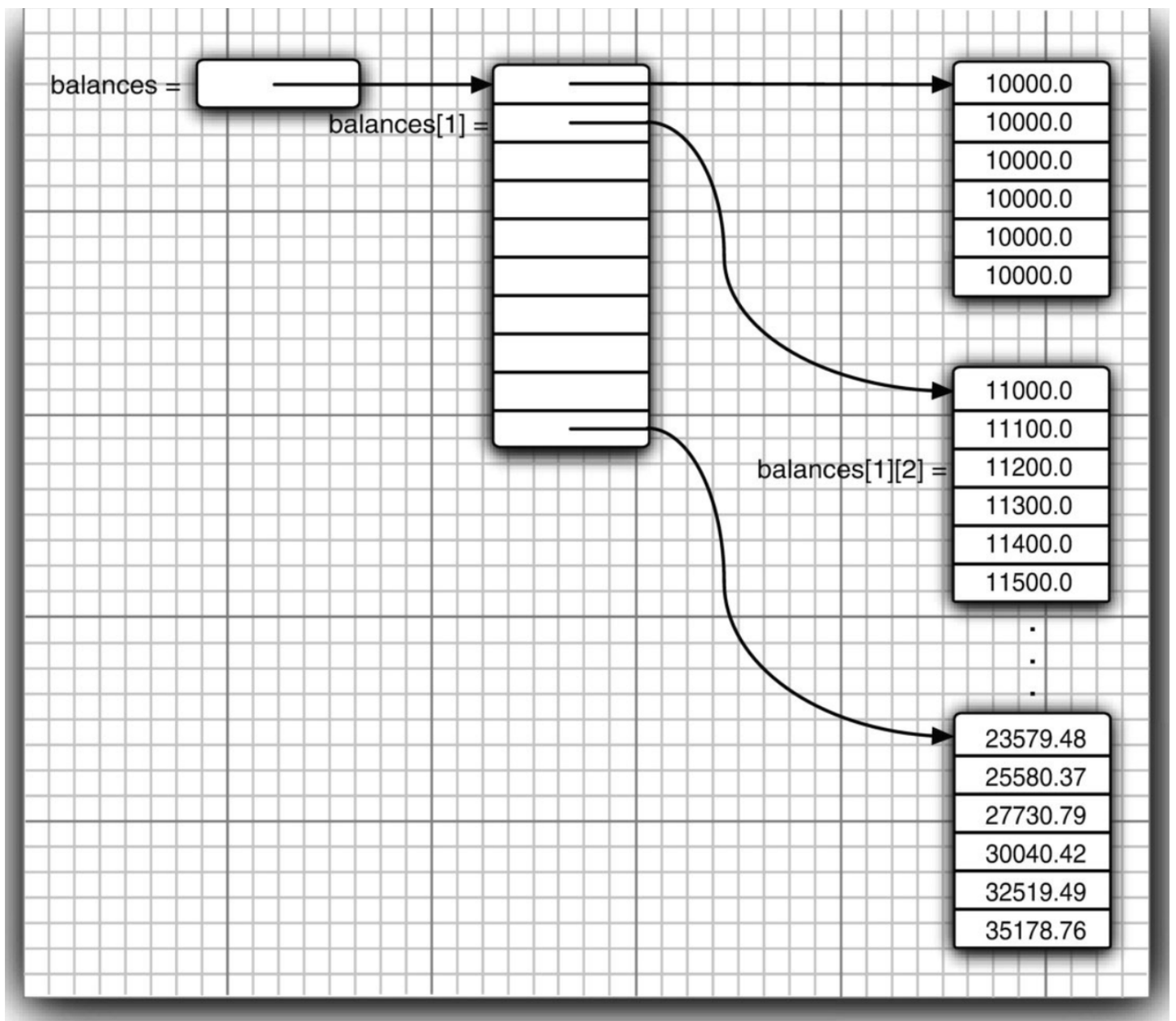
```
8  int[][] temp2 = new int[][]{
9      {1,2,3,4},
10     {1,2,3,4},
11     {1,2,3,4},
12     {1,2,3,4}
13 };//动态初始化
```

*Arrays.deepToString(Object[] a)* 本身是将任意维数组转化为字符串

如果想要快速输出一个多维数组

也可以调用 *Arrays.deepToString(Object[] a)*

```
1  import java.util.Arrays;
2
3  public class DeepToStringDemo {
4      public static void main(String[] args) {
5          int[][][] cube = {
6              {
7                  {1, 2},
8                  {3, 4}
9              },
10             {
11                 {5, 6},
12                 {7, 8}
13             }
14         };
15
16         // 使用 Arrays.deepToString() 打印三维数组
17         System.out.println(Arrays.deepToString(cube));
18         //输出[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
19     }
20 }
```



多维数组本质上就是对低维数组的引用

用 *C* 的说法就是高维数组存着低维数组的指针

### 31. *io* 流

先区分字节流和字符流

- **字节流**

处理一切类型的数据（图片、视频、文本、程序等），以 *byte* 为单位进行读写  
是 *io* 的底层基础

- **字符流**

专门用于处理**纯文本数据**，以 *char* 为单位进行读写  
它们在内部会自动处理字符编码（如 *UTF - 8*, *GBK*）的转换

## 字节流的输入和输出方法

### *FileInputStream* 和 *FileOutputStream*

直接操作文件的原始字节数据，不进行任何处理和转换

#### ◦ *FileInputStream*

##### ■ *int read()*

读取**一个字节**的数据，返回该字节的十进制数据

如果到达文件末尾，返回  $-1$

返回 *int* 是因为 *Java* 中的 *byte* 是有符号的，范围为  $-128 - 127$

而读取的数据是无符号的，范围为  $0 - 255$

为了保证可以正确读取选择了 *int*

##### ■ *int read(byte[] b)*

读取多个字节到字节数组 *b* 中

返回实际读取的字节数，如果到达文件末尾，返回  $-1$

如果文件剩余内容不足以填满整个缓冲区，这个值可能小于 *b.length*

##### ■ *int read(byte[] b, int off, int len)*

从文件的当前位置读取最多 *len* 个字节，从字节数组 *b* 的 *off* 索引位置开始存放

##### ■ 构造方法

###### • *FileInputStream(String name)*

###### • *FileInputStream(File file)*

#### 代码块

```
1  import java.io.FileInputStream;
2  import java.io.FileOutputStream;
3  import java.io.IOException;
4
5  public class ByteStreamExample {
6      public static void main(String[] args) {
7          // 使用 try-with-resources 语句，可以自动关闭流
8          try (FileInputStream fis = new
9              FileInputStream("source_image.jpg");
10              FileOutputStream fos = new
11                  FileOutputStream("destination_image.jpg")) {
12
13              byte[] buffer = new byte[1024]; // 创建一个1KB的缓冲区
14              int bytesRead;
```

```

15          // fis.read(buffer) 返回读取的字节数, -1 表示结束
16          while ((bytesRead = fis.read(buffer)) != -1) {
17              // 将读取到的内容写入输出流
18              fos.write(buffer, 0, bytesRead);
19          }
20          System.out.println("文件复制成功! ");
21
22      } catch (IOException e) {
23          e.printStackTrace();
24      }
25  }
26  }

```

## ◦ *FileOutputStream*

### ■ 核心方法

- *void write(int b)*

写入**一个字节**

- *void write(byte[] b)*

将字节数组 *b* 中的所有字节写入文件

- *void write(byte[] b, int off, int len)*

将字节数组 *b* 中从 *off* 索引开始的 *len* 个字节写入文件

### ■ 构造方法

- *FileOutputStream(String name)*

如果文件存在, 会**覆盖**原有内容

- *FileOutputStream(String name, boolean append)*

如果 *append* 为 *true*, 则在文件末尾**追加**内容

如果为 *false*, 则覆盖

- *FileOutputStream(File file)*
- *FileOutputStream(File file, boolean append)*

字符流的输入和输出方法

*FileReader* 和 *FileWriter*

基于字节流构建, 并在其上增加了**字符编码**的处理功能

使用的是**平台默认的字符集**

*FileReader*

### ■ 核心方法

- `int read()`

读取一个字符

如果到达文件末尾，返回 -1

- `int read(char[] cbuf)`

读取多个字符到字符数组 `cbuf` 中

返回实际读取的字符数，如果到达文件末尾，返回 -1

### ■ 构造方法

- `FileReader(String fileName)`
- `FileReader(File file)`

*FileWriter*

### ■ 核心方法

- `void write(int c)`: 写入一个字符
- `void write(char[] cbuf)`: 将字符数组 `cbuf` 写入文件
- `void write(String str)`: 将一个**字符串**写入文件（这是字符流特有的便捷方法）

### ■ 构造方法

- 与 `FileOutputStream` 类似，也支持覆盖和追加模式
- `FileWriter(String fileName)` 或 `FileWriter(File file)`
- `FileWriter(String fileName, boolean append)` 或 `FileWriter(File file, boolean append)`

## 2. 对象与类



核心

**面向过程是先决定逻辑后决定结构**

**面向对象是先决定结构后决定逻辑**

1. **类** (`class`) 是构造对象的模板或蓝图，定义了**实例域（也叫字段）**和**方法（也叫行为）**

由**类**构造 (`construct`) **对象**的过程称为创建类的**实例** (`instance`)

对于每个特定的**对象**都有一组特定的**实例域值**

这些值的集合就是这个对象的当前**状态** (`state`)

```

1  public class Person {
2      // 实例域 (数据)
3      private String name;
4      private int age;
5
6      // 构造方法 (用于初始化对象状态)
7      public Person(String name, int age) {
8          this.name = name;
9          this.age = age;
10     }
11
12     // 方法 (行为)
13     public void speak() {
14         System.out.println(name + " says: Hello!");
15     }
16
17     // Getter/Setter (操作数据)
18     public String getName() { return name; }
19     public void setName(String name) { this.name = name; }
20     public int getAge() { return age; }
21     public void setAge(int age) { this.age = age; }
22 }
23
24 public class Main {
25     public static void main(String[] args) {
26         // 创建对象 (实例化)
27         Person alice = new Person("Alice", 30);
28         Person bob = new Person("Bob", 25);
29
30         // 操作对象状态
31         alice.speak(); // 输出: Alice says: Hello!
32         bob.setAge(26); // 修改Bob的年龄 (状态变化)
33     }
34 }

```

## 2. 封装 (encapsulation) ，有时称为数据隐藏

核心在于**隐藏对象的内部状态，仅通过公共方法与外界交互**

展开说就两点

- 禁止直接访问其他类的实例域（字段），必须通过方法操作数据
- 对象内部的数据只能由自身的方法修改，外部代码无法绕过方法直接修改数据



封装给对象赋予了“黑盒”特征，这是提高重用性和可靠性的关键

3. 在 *Java* 中，所有的类都源自于一个**超类**，它就是 *Object* 类

4. 对象的三个主要特性

- 行为 (*behavior*)

**对象的行为是用可调用的方法定义的**

同一个类的所有对象实例，由于支持相同的行为而具有家族式的相似性

- 状态 (*state*)

对象的状态可能会随着时间而发生改变，但这种改变不会是自发的

**对象状态的改变必须通过调用方法实现**

**如果不经方法调用就可以改变对象状态，只能说明封装性遭到了破坏**

- 标识 (*identity*)

但是，对象的状态并不能完全描述一个对象

每个对象都有一个唯一的身份 (*identity*)

例如，在一个订单处理系统中，任何两个订单都存在着不同之处，即使所订购的货物完全相同也是如此

需要注意，作为一个类的实例，每个对象的标识永远是不同的，状态常常也存在着差异







对象的这些关键特性在彼此之间相互影响着

例如，对象的状态影响它的行为

（如果一个订单“已送货”或“已付款”，就应该拒绝调用具有增删订单中条目的方法

反过来，如果订单是“空的”，即还没有加入预订的物品，这个订单就不应该进入“已送货”状态）

5. 类之间的关系

关 系	UML 连接符
继承	
接口实现	
依赖	
聚合	
关联	
直接关联	

常见的三种关系是**依赖**，**聚合**，**继承**

- **依赖** (*dependence*) ，即 “*uses - a*” 关系
- **聚合** (*aggregation*) ，即 “*has - a*” 关系
- **继承** (*inheritance*) ，即 “*is - a*” 关系

- 依赖表示一个类**临时使用**另一个类，但并不拥有它

这种关系的特点是耦合度较低，依赖的类可以被替换或改变，而不一定会影响使用它的类

一个 *Driver* 类可能有一个方法 *drive(Car car)*

其中 *Car* 是作为参数传递进来的

*Driver* 使用 *Car* 来完成驾驶行为，但 *Driver* 并不包含或拥有 *Car*

- 聚合表示一个类包含或**拥有**另一个类的对象，形成一种 “整体-部分” 的关系

但被包含的对象可以独立于整体存在

也就是说，整体和部分的生命周期不一定绑定

一个 *Car* 类可能包含一个 *Engine* 对象

*Car* 有 *Engine* ，但如果 *Car* 被销毁，*Engine* 可能仍然可以被取出并用于其他地方

代码中，这通常通过类的**字段**来实现

```

1  public class Car {
2      private Engine engine;
3      public Car(Engine engine) {
4          this.engine = engine;
5      }
6  }
```

- 继承表示一个类是另一个类的子类型，子类继承父类的属性和行为

一个 *SportsCar* 类继承自 *Car* 类

*SportsCar* 是一个 *Car*，因此它拥有 *Car* 的所有特性，并可以添加自己的特有功能或重写父类的行为

代码中，这通过关键字 *extends* 实现

```
1 public class SportsCar extends Car {  
2     // 额外的功能  
3 }
```

这三种关系的耦合度是递进的

### 依赖 < 聚合 < 继承

- 低耦合更灵活，易于修改和扩展，但可能需要更多的代码来协调对象之间的关系
- 高耦合可以简化代码复用，但会导致系统僵化，难以应对变化

因此，常见的建议是**优先使用组合（包括聚合）而非继承**，以在复用性和灵活性之间找到平衡

依赖和聚合通常比继承更符合“松耦合、高内聚”的设计原则

## 6. 类的修饰符

- *public* 对所有类可见

使用对象：类、接口、变量、方法

- *protected* 对同一包内的类和所有子类可见

使用对象：变量、方法

- *default* 或 *friendly*: 在同一包内可见，不使用任何修饰符

使用对象：类、接口、变量、方法

- *private* 在同一类内可见

使用对象：变量、方法

注意，*protected* 和 *private* 不能修饰类和接口

访问级别	访问修饰符	同类	同包	子类	不同包
公开	public	√	√	√	√
受保护的	protected	√	√	√	×
默认	default	√	√	×	×
私有的	private	√	×	×	×

7. 在 *Java* 程序设计语言中，使用**构造器** (*constructor*) 构造新对象

我们这里以 *Java* 官方 *api* 中的 *Date* 类作为例子

**构造器**是一种特殊的方法，用来构造并初始化对象

- **构造器的名字必须与类名相同**，因此 *Date* 类的构造器名为 *Date*
- 与普通方法不同，构造器**没有返回值类型**，这与 *void* 返回类型完全不同，*void* 只是不返回而不是没有返回值

**但是构造器实际上会返回一个新创建的对象**的引用

- 只能通过在构造器前面加上 *new* 操作符来调用构造器，用于在内存中分配空间并返回新对象的引用
- 构造器可以选择传入参数，也可以不传入，具体根据构造器的实现而选择

*new Date()*; 用于调用 *Date* 类的构造器创建一个新 *Date* 对象，并返回该对象的**引用**

**所有的Java对象都存储在堆中，引用是堆中对象的地址**

**注意对于构造器来说，返回的都是新对象的引用，而不是新对象**

如果需要的话，也可以将这个对象传递给一个方法

```
System.out.println(new Date())
```

*Date* 类中有一个 *toString()* 方法，这个方法将返回日期的字符串描述

```
String s = new Date().toString();
```

通常，希望构造的对象可以多次使用

所以我们通常将对象存放在一个变量中以便多次复用

```
Date birthday = new Date();
```

一定要记住**一个对象变量并没有实际包含一个对象，而仅仅引用一个对象**

可以显式地将对象变量设置为 *null*，表明这个对象变量目前没有引用任何对象

## 8. 如果在编写一个类时没有编写构造器，那么系统就会提供一个**无参构造器**

这个构造器将所有的实例域设置为默认值

于是，实例域中的数值型数据设置为 0、布尔型数据设置为 *false*、所有对象变量将设置为 *null*

如果类中提供了至少一个构造器，但是没有提供无参数的构造器，那么系统自动提供的无参构造器就会消失

**则在构造对象时如果没有提供参数就会被视为不合法**

## 9. *Date* 类的实例表示一个当前的**时间点**，单位是**毫秒**

当前时间点是相对于一个固定时间点计算的

这个固定的时间点叫做**纪元** (*epoch*)，是 UTC1970 年 1 月 1 日 00:00:00

当前时间点就是距离纪元有多少毫秒，正数代表当前在纪元之后，负数代表当前在纪元之前

```
1 Date date = new Date(0); // 表示 1970-01-01 00:00:00 UTC
2 System.out.println(date); // 输出可能是 "Thu Jan 01 08:00:00 CST 1970" (取决于时区)
```

但是时间点和日历表示是两个不同的概念

时间点是绝对的、与文化无关的

而日历表示法（如公历、农历、希伯来历）是人类文化的产物，依赖于具体的历法规则

因此，*Java* 将这两个概念分离到不同的类中

- *Date* 类表示时间点（毫秒数），不关心具体的日历表示
- *LocalDate* 类（*Java SE 8* 引入，位于 *java.time* 包）

表示日历上的日期（如 2025 — 03 — 09），不包含时间或时区信息

将时间与日历分开是一种优秀的面向对象设计原则

**单一职责原则** (*Single Responsibility Principle*)

每个类负责一个明确的概念，避免将无关的功能混杂在一起

不过 *LocalDate* 类的构造器是私有的，我们不能通过构造器去新建一个 *LocalDate* 类的对象应当使用静态工厂方法（*factory method*）代表你调用 *LocalDate* 构造器

- *LocalDate.now()* 会构造一个新对象，表示构造这个对象时的日期
- *LocalDate date = LocalDate.of(2025, 3, 9)* 提供年、月和日来构造对应一个特定日期的对象

当创建了一个 *LocalDate* 类的对象后

可以用方法 *getYear()*、*getMonthValue()*、*getDayOfMonth()* 得到年、月和日

这三个方法返回值是 *int*

而 *getDayOfWeek()* 返回值是 *DayOfWeek* 类的一个实例

需要调用 *getValue* 来得到 1~7 之间的一个数

也可以使用 *plusDays()* 方法或 *minusDays()* 在原来的日期上加上或减去一些天数，得到一个新对象

注意并不是在原来的对象上进行修改，而是直接返回一个新对象

这与我们之前讲过的 *String* 类的所有操作方法比如 *toUpperCase()* 类似

由于对象的不可变性，不修改原来的对象，只通过访问得到对象此时的状态

并根据得到的状态和传入的参数创建一个新对象进行返回

只**访问**对象而**不修改**对象的方法称为**访问器方法**（*accessor method*）

**直接修改**对象的方法称为**更改器方法**（*mutator method*）

10. 在这一点，我们将通过 *LocalDate* 的各种方法输出一个日历

目的在于展示 *OOP* 的好处

编写程序者并不需要知道 *LocalDate* 类如何计算月和星期几，只需要使用这个类的接口即可实现复杂的任务

```
1  import java.time.*;
2  import java.time.format.TextStyle;
3  import java.util.Locale;
4  import java.util.Scanner;
5
6  public class Main {
```

```

7      public static void main(String[] args) {
8          Scanner sc = new Scanner(System.in);
9          while (true) {
10             System.out.println("请输入想要查看的年份 (输入0退出):");
11             int year = sc.nextInt();
12             if (year == 0) break;
13
14             System.out.println("请输入想要查看的月份 (1-12):");
15             int month = sc.nextInt();
16             if (month < 1 || month > 12) {
17                 System.out.println("无效的月份, 请输入1到12之间的数字。");
18                 continue;
19             }
20
21             LocalDate date = LocalDate.of(year, month, 1);
22             LocalDate now = LocalDate.now();
23             int today = (year == now.getYear() && month ==
now.getMonthValue()) ? now.getDayOfMonth() : -1;
24
25             // 打印月份和年份 (居中)
26             String monthName = date.getMonth().getDisplayName(TextStyle.FULL,
Locale.CHINA);
27             System.out.printf("%" + (11 - monthName.length()) + "s%s %d\n",
"", monthName, year);
28
29             // 打印星期标题 (固定宽度)
30             System.out.println(" 周一 周二 周三 周四 周五 周六 周日");
31
32             // 计算第一天是星期几并打印空格
33             int value = date.getDayOfWeek().getValue() % 7; // 周一=1, 周日=0
34             for (int i = 0; i < value; i++) {
35                 System.out.print("    "); // 每个格子占4个空格宽度
36             }
37
38             // 打印该月的每一天
39             int dayInMonth = date.lengthOfMonth();
40             for (int i = 1; i <= dayInMonth; i++) {
41                 if (i == 12) { // 特殊标记3月12日
42                     System.out.printf("%2d* ", i);
43                 } else if (today == i) {
44                     System.out.printf("%2d* ", i); // 当前日期加星号
45                 } else {
46                     System.out.printf("%3d ", i); // 普通日期
47                 }
48                 if ((i + value) % 7 == 0 || i == dayInMonth) {
49                     System.out.println(); // 每七天换行或月末换行
50                 }

```

```

51         }
52         System.out.println(); // 打印空行分隔
53     }
54     sc.close();
55 }
56 }

```

## 11. 之前我们都只在 *public* 修饰的主类中进行编写程序

现在开始学习如何设计复杂应用程序所需要的各种**主力类** (*workhorse class*)

通常，这些类没有 *main* 方法，却有自己的实例域和实例方法

要想创建一个完整的程序，应该将若干类组合在一起，其中只有一个类有 *main* 方法

一个最简单的主力类应包括

- 至少一个字段
- 至少一个构造器
- 至少一个方法

下面我们构造一个非常简单的 *Employee* 类

```

1  import java.time.LocalDate;
2
3  public class EmployeeTest {
4      public static void main(String[] args) {
5          Employee[] staff = new Employee[3];
6          staff[0] = new Employee("Tony", 40000, LocalDate.of(2020, 3, 15));
7          staff[1] = new Employee("Alice", 50000, LocalDate.of(2019, 7, 1));
8          staff[2] = new Employee("Bob", 45000, LocalDate.of(2021, 9, 23));
9
10         // 涨薪5%
11         for (Employee e : staff) {
12             e.raiseSalary(5);
13         }
14
15         // 打印员工信息
16         for (Employee e : staff) {
17             System.out.printf("Name: %s, Salary: %.2f, Hire Date: %s\n",
18                 e.getName(), e.getSalary(), e.getHireDay());
19         }
20     }
21 }

```



```

22
23 class Employee {
24     private String name;
25     private double salary;
26     private LocalDate hireDay;
27
28     public Employee(String name, double salary, LocalDate hireDay) {
29         this.name = name;
30         this.salary = salary;
31         this.hireDay = hireDay;
32     }
33
34     public String getName() { return name; }
35     public double getSalary() { return salary; }
36     public LocalDate getHireDay() { return hireDay; }
37
38     public void raiseSalary(double percent) {
39         salary *= (1 + percent / 100);
40     }
41 }

```

- *raiseSalary* 方法有两个参数

第一个参数称为隐式 (*implicit*) 参数, 是出现在方法名前的 *Employee* 类对象

隐式参数也称为方法调用的目标或接收者

第二个参数位于方法名后面括中的数值, 这是一个显式 (*explicit*) 参数

- 由于许多程序员习惯于将每一个类存在一个单独的源文件中

比如, 只将 *Employee* 类存放在文件 *Employee.java* 中

只将 *EmployeeTest* 类存放在文件 *EmployeeTest.java* 中

如果喜欢这样组织文件, 可以有两种编译源程序的方法

一种是使用通配符调用 *Java* 编译器

```
javac Employee *.java
```

这样所有与通配符匹配的源文件都将被编译成类文件

或者只编译 *EmployeeTest.java*

```
javac EmployeeTest.java
```

第二种方式，并没有显式地编译 *Employee.java*

然而，当 *Java* 编译器发现 *EmployeeTest.java* 使用了

*Employee* 类时会查找名为 *Employee.class* 的文件

如果没有找到这个文件，就会自动地搜索 *Employee.java*，然后，对它进行编译

更重要的是：如果 *Employee.java* 版本较已有的 *Employee.class* 文件版本新

*Java* 编译器就会自动地重新编译这个文件

## 12. 对于一个可以运用于实际生产的类来说

应该提供下面三项内容

- **私有**的数据域
- 一个**公有**的域访问器方法 (*get*)

注意不要写直接返回引用可变对象的访问器方法，*Employee* 类中的 *getHireDay()* 就违反了这一点

如果需要返回一个可变对象的引用，应该首先对它进行克隆 (*clone*)

```
1 public LocalDate getHireDay() { return (Date)hireDay.clone(); }
```

- 一个**公有**的域更改器方法 (*set*)

## 13. 静态汇总

被 *static* 修饰的成员

核心特点

- *static* 成员属于类，而非对象实例
- 直接通过类名访问（如 *ClassName.staticMember*），无需创建对象
- 静态成员在内存中仅存在一个副本，所有实例共享同一份数据
- 静态成员随类的加载而初始化，随类的卸载而销毁，生命周期贯穿程序运行期间
- 类字段

```
1 class Employee{  
2     private static int nextId;
```

```
3     private id;
4 }
```

现在，每一个雇员对象都有一个自己的 *id* 域，但这个类的所有实例将共享一个 *nextId* 域  
换句话说，如果有 1000 个 *Employee* 类的对象，则有 1000 个实例字段 *id*  
但是，只有一个类字段 *nextId*

- 类变量/常量

多次使用的类常量是 *System.out*

```
1 public class System{
2     public static final PrintScream out = ...;
3 }
```

不过也有可以修改 *final* 的方法

如果查看一下 *System* 类，就会发现有一个 *setOut* 方法，它可以将 *System.out* 设置为不同的流  
为什么这个方法可以修改 *final* 变量的值原因在于，*setOut* 方法是一个本地方法，而不是用 *Java* 语言实现的

本地方法可以绕过 *Java* 语言的存取控制机制

- 类方法

类方法**没有隐式的参数**，可以认为类方法是没有 *this* 参数的方法

而在一个非静态的方法中，*this* 参数表示这个方法的隐式参数

类方法不能直接访问非静态字段，必须通过创建一个实例后通过这个实例访问该对象的非静态字段  
同样也不能使用非静态方法，必须通过创建实例后通过这个实例访问

- 静态方法的一个应用是静态工厂方法，是用来替代构造器的
  - 构造器的名字强制绑定了类名，只能通过传不同的参数来表示区别  
无法通过名称区分不同场景的构造逻辑
  - 构造器只能返回当前类的实例，无法灵活返回其子类对象  
而静态方法可以自由命名，清晰表达对象的创建意图

```

1 // 无法通过构造器名称区分“货币格式”和“百分比格式”
2 NumberFormat currency = new NumberFormat(...); // 需要额外参数区分用途
3 NumberFormat percent = new NumberFormat(...); // 但参数设计可能不直观
4
5 // 通过方法名直接表明生成的格式类型
6 NumberFormat currencyFmt = NumberFormat.getCurrencyInstance(); // 货币格式
7 NumberFormat percentFmt = NumberFormat.getPercentInstance(); // 百分比格式

```

而且静态方法可以返回当前类的子类实例，对外屏蔽具体实现

```

1 // 无法让构造器直接返回子类 DecimalFormat (隐藏实现细节)
2 NumberFormat fmt = new DecimalFormat(...); // 暴露了具体子类，违背封装性
3
4 public abstract class NumberFormat {
5     // 静态工厂方法返回子类 DecimalFormat
6     public static NumberFormat getCurrencyInstance() {
7         return new DecimalFormat("$#,##0.00");
8     }
9 }

```

#### 14. 每个类都可以有一个 *main* 方法

#### 15. *java* 中的方法参数传递只使用值传递，不存在引用传递

按值调用 (*call by value*) 表示方法接收的是调用者提供的值

而按引用调用 (*call by reference*) 表示方法接收的是调用者提供的变量地址

这个对于基本类型很好理解

*Java* 将传入的值拷贝到参数  $x$  中， $x$  是传入参数的副本，二者独立存在

当函数使用完之后，该副本自动被销毁

对于引用类型

传递的只是传递的是对象地址的拷贝，方法内通过地址可以修改指向对象的状态

但无法修改原引用变量的指向

```

1 public static void changeReference(Employee e) {

```

```

2     e = new Employee(10000); // 修改的是副本的指向
3 }
4
5 public static void main(String[] args) {
6     Employee emp = new Employee(5000);
7     changeReference(emp);
8     System.out.println(emp.getSalary()); // 输出 5000 (原对象未变)
9 }

```

总之，不论如何是不会改变传入参数的值的

但是对于引用类型可以改变指向参数的值

## 16. 在同一个类中，方法名字相同，但是传入参数列表**必须不同**

比如 *String* 类中的 *indexOf()* 方法实现了重载

*int indexOf(int i)*

*int indexOf(int i, int i)*

*int indexOf(String s)*

*int indexOf(String s, int i)*

**仅返回值类型不同而参数列表完全相同的方法**会引发编译错误

返回值类型不同不能单独作为重载的依据

```

1  int add(int a, int b) { ... }
2  double add(int a, int b) { ... } // 编译错误!

```

## 17. 如果构造器的第一个语句形如 *this(...)*，且如果使用这种语法，*this(...)* 必须作为构造器的第一条语句

这个构造器将调用同一个类的另一个构造器

下面是一个典型的例子

```

1  public Employee(double s){
2      this("Employee" + nextId, s);
3      nextid;

```

当调用 `new Employee(60000)` 时，`Employee(double)` 构造器将调用 `Employee(String, double)` 构造器

这样我们可以只写一次 `public` 的构造器，其他的构造器可以修饰为私有

通过对公有构造器的重载以及 `this()` 的使用即可以实现

## 18. 初始化块和静态初始化块

**如果一个常量字段没有在声明的时候被初始化，那么就必须在对应的构造器中进行初始化**

静态初始化块专用于初始化类常量和类变量，不可以初始化实例字段

初始化块用于初始化普通实例字段，也可以修改类变量，但是不能初始化类常量，不过可以对其进行访问

## 19. 导入的本质是一种语法糖，让代码更简洁

为了保证包名的绝对唯一性，*Sun* 公司建议将公司的因特网域名（这显然是独一无二的）以逆序的形式作为包名

并且对于不同的项目使用不同的子包

比如 `zzemu.top` 是我使用的域名

那么我的包就应该命名为 `top.zzemu`

如果要使用其他类的成员

有两种方式

- 一种是使用全限定的类名

即在代码中直接写出完整的包路径和类名

一般在类名冲突时（如多个包中有同名类），或临时使用某个类

```

1  public class MyClass {
2      public static void main(String[] args) {
3          // 使用全限定类名创建对象
4          java.util.List<String> list = new java.util.ArrayList<>();
5      }
6  }
```

- 一种是使用 *import* 语句直接将这个类都导入

*import* 必须位于 *package* 声明之后、类定义之前

```
1  package com.example.mypackage;
2
3  // 导入单个类
4  import java.util.ArrayList;
5  // 导入整个包的所有类 (不推荐, 可能引起命名冲突)
6  import java.util.*;
7
8  public class MyClass {
9      public static void main(String[] args) {
10         // 直接使用类名
11         List<String> list = new ArrayList<>();
12     }
13 }
```

也包含一种静态导入 (*Static Import*)

只导入静态成员

```
1  import static java.lang.Math.PI; // 导入静态字段
2  import static java.lang.Math.sqrt; // 导入静态方法
3
4  public class Test {
5      public static void main(String[] args) {
6          double radius = 5.0;
7          double area = PI * radius * radius; // 直接使用PI
8          double root = sqrt(area);           // 直接使用sqrt()
9      }
10 }
```

如果使用 *import* 语句同时导入了两个标识符相同的成员

只能使用全限定

```
1  import java.util.*;
2  import java.sql.*;
3
```

```

4  public class Test {
5      public static void main(String[] args) {
6          // 编译错误: Date类存在歧义
7          Date date = new Date();
8
9          // 正确: 使用全限定类名
10         java.util.Date utilDate = new java.util.Date();
11         java.sql.Date sqlDate = new java.sql.Date(0);
12     }
13 }

```

20. 若要将一个类放入包中，必须在源文件的开头使用 *package* 语句声明想要放入的包名中

```

1  package com.horstmann.corejava; // 包声明必须位于第一行（注释除外）
2
3  public class Employee {
4      // 类定义
5  }

```

如果源文件中没有 *package* 声明，则该类会被放在默认包 (*default package*) 中

默认包没有名称，通常用于简单测试代码

因其无法被其他包直接访问，在正式项目中应避免使用

- *Java* 要求包名**必须**与文件所在的目录结构完全一致

比如包名是 *com.horstmann.corejava*

包名中的每个 . 分隔符对应一级子目录

那么文件的路径就必须是

```

1  基目录/
2      └─ com/
3          └─ horstmann/
4              └─ corejava/
5                  └─ Employee.java // 包声明为 package com.horstmann.corejava

```

基目录是包含包层级起点（如 *com* 目录）的父目录

所有编译和运行操作都基于此目录



- 在古早代码中，程序员封装的意识不足，有时不会对字段使用 *private* 修饰

这就导致所有同一个包的类都可以对其进行修改

这就有可能被有心人进行攻击

只要将其编写好的类导入想要攻击的包中，就可以对所有默认标识符的非常量字段进行修改

为了解决这个问题

*Java9* 引入模块系统，通过模块描述符 (*module-info.java*) 封闭包，彻底解决此问题

- 模块声明：明确导出哪些包可被外部访问
- 强封装性：未导出的包或未开放的类，其他模块无法反射访问

```
1  // 模块描述符示例
2  module com.example.myapp {
3      exports com.example.publicapi; // 仅导出 publicapi 包
4  }
```

## 21. 类文件除了存储在文件系统的子目录中

也可以存储在 *JAR* (*Java* 归档)文件中

在一个 *JAR* 文件中，可以包含多个压缩形式的类文件和子目录

*JAR* 文件使用 *ZIP* 格式组织文件和子目录

可以使用所有 *ZIP* 实用程序查看内部的 *rt.jar* 以及其他的 *JAR* 文件

将 *class* 后缀的编译后的类文件放入一个基目录，例如 *c:\classdir*

将打包好的 *JAR* 后缀的类放入另一个指定目录，例如 *c:\archives*

然后**设置类路径** (*class path*)

类路径是包含所有类文件路径的集合

*Windows* 用分号；分隔，例如 *c:\classdir;c:\archives\archive.jar;*

句点. 表示当前目录

从 *Java SE 6* 起，支持 *JAR* 文件目录中的通配符，例如 *\archives\\**

注意：运行时库（如 *rt.jar* 和 *jre\lib* 下的文件）会自动加载，无需手动加入类路径

类路径列出的目录和归档文件（如 *.jar* 文件）是 *JVM* 和编译器搜寻类的起始点

*JVM* 和编译器会按照类路径的顺序查找所需的类文件

查找顺序如下

#### 系统类文件：

- 首先检查 *JVM* 自带的运行时库（*jre/lib* 和 *jre/lib/ext* 下的 *.jar* 文件，如 *rt.jar*）
- 如果在系统类中找不到，则继续查找类路径

#### 类路径中的位置：

- *c:\classdir\com\horstmann\corejava\Employee.class*（基目录下的类文件）
- *c:\archives\archive.jar* 内的 *com\horstmann\corejava\Employee.class*（归档文件中的类文件）
- *.\com\horstmann\corejava\Employee.class*（当前目录下的类文件）

那么如何设置类路径呢

#### 推荐方法：使用 *-classpath* 或 *-cp* 选项

- 在运行 *java* 或 *javac* 命令时，通过 *-classpath*（简称 *-cp*）选项指定类路径是首选方法

```
1 java -cp c:\classdir;c:\archives\archive.jar;. com.example.MyProgram
```

当然也可以

在系统环境变量中设置 *CLASSPATH*

## 22. 文档注释 *javadoc*

它可以由源文件生成一个 *HTML* 格式的 *API* 文档

*Java* 标准库的联机 *API* 文档就是通过 *javadoc* 从源代码生成的

核心优点在于**将文档注释嵌入源代码，避免代码与独立文档不一致的问题**

修改代码后，重新运行 *javadoc* 即可保持同步

格式在之前提供，以 */\*\** 开始，以 *\*/* 结束，每行开头 *\** 可选（IDE通常自动添加）

其使用**自由格式文本**

第一句应为概要性句子（会被提取到概要页）

**标记**以 @ 开头，用于指定特定信息（如 @author、@param）

**标记示例：**

- @param 描述方法参数
- @return 描述返回值
- @author 作者信息

```
1  /**
2   * This is a utility class for employee management.
3   * It provides methods to handle employee data efficiently.
4   * @author John Doe
5   * @version 1.0
6   */
7  public class Employee {
8      /** The name of the employee. */
9      public String name;
10
11     /**
12      * Constructs an Employee with a given name.
13      * @param name the name of the employee
14      */
15     public Employee(String name) {
16         this.name = name;
17     }
18
19     /**
20      * Gets the employee's name.
21      * @return the employee's name as a {@code String}
22      */
23     public String getName() {
24         return name;
25     }
26 }
```

- 支持 *HTML* 修饰符：
  - `<em> ... </em>` 强调
  - `<strong> ... </strong>` 着重强调
  - `<img...>` 嵌入图像

不过要**避免** `<h1>`、`<hr>` 等会干扰文档格式的标签

若要输入**等宽代码**，可以使用 `{@code...}`（推荐），无需转义 `<` 等字符  
避免使用 `< code > ... < /code >`，因为需要手动转义

如果注释中包含链接（如图像文件），需将文件放入 `doc - files` 子目录

- 文件路径： `src/doc - files/uml.png`
  - *HTML* 标签： `< img src = "doc - files/uml.png" alt = "UMLdiagram" >`
- javadoc* 会自动将 `doc - files` 目录及其内容拷贝到生成的文档目录中

在 *cmd* 中输入以下命令来生成文档注释

```
1 javadoc -d doc src/Employee.java
```

## 3. 继承

继承最需要注意的一点是

在运行时，*JVM* 只会检查实际类型

**继承** (*inheritance*) 的本质就是复用（继承）这些类的方法和域

在此基础上，还可以添加一些新的方法和域，以满足新的需求

这是 *Java* 程序设计中的一项核心技术

### 1. 关键字 *extends* 表示继承

在 *Java* 中，所有的继承都是公有继承，而没有 *C++* 中的私有继承和保护继承

```
1 public class Manger extends Employee{  
2  
3 }
```

关键字 *extends* 表明正在构造的新类派生于一个已存在的类

已存在的类称为超类 (*super class*)、基类 (*base class*) 或父类 (*parent class*)

新类称为子类 (*sub class*)、派生类 (*derived class*) 或孩子类 (*child class*)

与听起来恰恰相反，**子类比超类拥有的功能更加丰富**

## 2. 继承拥有单向性

子类可以使用超类中被访问修饰符允许访问的成员

比如 *protected*，*public*，还有同包时的默认

如果要使用不被允许的，比如不同包时的默认，*private*

就只能像其他类一样使用超类提供的公有接口

但从超类的视角来看，子类与其他类在访问权限上没有任何区别

## 3. 重写 (*override*)

方法重写是指子类重新实现父类中已存在的非私有方法

要求子类方法与父类方法**具有相同的名称、参数列表完全一致**

如果不完全一致，则会被视为新方法而非方法重写

**返回类型可以为父类或父类的子类，其他均不可**

通过重写，子类可以根据自身需求提供特定的功能实现，而非直接继承父类的默认行为

```
1  class Animal {
2      public void sound() { System.out.println("Animal sound"); }
3  }
4
5  class Dog extends Animal {
6      @Override
7      public void sound() { System.out.println("Dog barks"); } // 重写父类方法
8  }
```

- 子类方法的访问修饰符（如 *public*、*protected*）不能比父类方法更严格

例如，父类方法为 *public*，子类方法不能为 *protected* 或 *private*

- 子类方法抛出的异常必须与父类方法抛出的异常相同或更具体，且不能抛出父类未声明的更宽泛异常

- 子类方法的返回类型可以是父类方法返回类型的子类型（协变返回类型）

例如，父类返回 *Animal*，子类可返回 *Dog*

- 当父类引用指向子类对象时，*JVM* 在运行时根据对象的**实际类型**动态调用方法

这一机制称为**动态绑定**

而且动态绑定时如果要调用方法**最好要使用被重写的方法，也就是父子类均有**

如果子类没有而父类有，会直接调用父类中的方法

如果父类没有，会报错

因为编译器只根据引用类型（*Parent*）检查方法是否存在

而根据实际类型类型（*Son*）去调用方法决定调用哪个方法实现

（如果方法被重写，则调用子类的版本，否则调用父类的版本）

```
1  Animal animal = new Dog();
2  animal.sound(); // 输出"Dog barks" (调用子类重写的方法)
```

而且补充一点

动态绑定只适用于实例方法（非 *static*、*final* 或 *private* 方法）

静态方法使用静态绑定，编译时就确定调用哪个版本

*static* 是类级别的，不参与多态

#### 4. *super* 关键字

当我们已经重写了父类的方法之后，如果还想要访问父类的方法

可以使用 *super* 关键字

*super* 并不像 *this* 一样指向某个具体对象的内存地址

而是一个编译器指令，用于明确指示调用父类的成员（方法、变量或构造器）

例如，当子类覆盖父类方法时，*super.method()* 会绕过动态绑定，直接调用父类实现

```

1  class Parent {
2      void show() { System.out.println("Parent"); }
3  }
4
5  class Child extends Parent {
6      void show() {
7          super.show(); // 直接调用父类方法，输出"Parent"
8          System.out.println("Child");
9      }
10
11     void test() {
12         Parent p = this; // 合法: this是对象引用
13         // Parent p2 = super; // 编译错误: super不是对象
14     }
15 }

```

子类构造器需通过 *super* 显式调用超类的构造器，以初始化从超类继承的私有字段

若未显式调用，编译器会自动插入 *super()*（即调用超类的无参构造器）

*super* 调用必须为子类构造器的第一条语句

代码块

```

1  // 超类Employee的构造器
2  public Employee(String name, double salary, int year, int month, int day) {
3      this.name = name;
4      this.salary = salary;
5      this.hireDay = LocalDate.of(year, month, day);
6  }
7
8  // 子类Manager的构造器
9  public Manager(String name, double salary, int year, int month, int day, double
    bonus) {
10     super(name, salary, year, month, day); // 必须为首行
11     this.bonus = bonus;
12 }

```

## 5. 继承层级

由一个公共超类派生出来的所有类的集合被称为**继承层次** (*inheritance hierarchy*)

```
2  |—— Manager (子类)
3  |   |—— Executive (更具体的子类)
4  |—— Programmer (子类)
5  |—— Secretary (子类)
```

在继承层次中，从某个特定的类到其祖先的路径被称为该类的继承链 (*inheritance chain*)

例如，*Executive* 类的继承链为

*Executive* → *Manager* → *Employee* → *Object*

继承链决定了类成员的访问顺序：**子类优先调用自身方法，若未定义则沿链向上查找父类实现**

*Java* 中只支持**单继承**

即每个类只能直接继承一个父类，避免多重继承的“菱形问题”（如父类方法冲突）

```
1  class Animal {}
2  class Mammal extends Animal {}
3  class Dog extends Mammal {}
```

虽然类不直接支持多继承，但接口 (*Interface*) 允许一个类实现多个接口，从而间接实现多继承

例如

代码块

```
1  interface Flyable { void fly(); }
2  interface Swimmable { void swim(); }
3  class Duck implements Flyable, Swimmable {
4      public void fly() { /* 实现飞行 */ }
5      public void swim() { /* 实现游泳 */ }
6  }
```

## 6. 多态

**置换法则** (*Liskov Substitution Principle*) 也就是 *is - a* 关系表明**子类的每个对象也是超类的对象**

在程序中可以用子类对象替换超类对象，而不会破坏程序的正确性

但绝对不能将超类引用赋给子类变量

这在 *Java* 中通过**多态性**得以实现



```
1 Employee e = new Manager();
```

这里  $e$  是 *Employee* 类型的引用，但实际指向一个 *Manager* 对象  
这是合法的

除了这个多态还引起了**数组的协变性**

这意味着**子类数组的引用可以赋值给超类数组的引用**

```
1 Manager[] managers = new Manager[10];
2 Employee[] staff = managers; // 合法
3 staff[0] = new Employee(); // 编译器允许，但运行时抛出 ArrayStoreException
```

因为 *staff* 的声明类型是 *Employee*[]，而 *Employee* 对象显然可以存储在 *Employee*[] 中  
但在运行时，JVM 会检查数组的**实际类型**（这里是 *Manager*[]）  
并发现 *Employee* 不是 *Manager* 的子类型，因此抛出 *ArrayStoreException*  
这是 Java 数组类型安全的一种保护机制

## 7. 下面我们对一次多态方法调用进行解析

假设我们要调用  $x.f(args)$ ，其中隐式参数  $x$  被声明为类  $C$  的对象

下面是整个过程的步骤

- 编译器检查声明类型和方法名

**输入：** $x.f(param)$ ， $x$  的**声明类型**是  $C$ ，**注意是声明类型而不是运行类型**

**任务：**编译器列出所有可能的方法

- 检查  $C$  类中所有名为  $f$  的方法
- 检查  $C$  的**超类（注意是声明类型的超类而不是运行类型的超类）**中所有名为  $f$  且访问权限为 *public* 的方法（私有方法不可访问）

**结果：**得到一组候选方法

可能是  $f(int)$ 、 $f(String)$  等

- 重载解析 (*Overloading Resolution*)

**输入：**调用时提供的参数类型，例如  $x.f("Hello")$

**任务：**编译器从候选方法中选择与参数类型最匹配的一个

- 如果有**完全匹配**的方法，如  $f(String)$ ，直接选中
- 如果没有完全匹配的，允许类型转换（如  $int$  到  $double$ ， $Manager$  到  $Employee$ ），**但必须唯一匹配**

**可能的结果：**

- 找到唯一匹配的方法：继续下一步
- 未找到匹配方法：编译错误
- 找到多个匹配方法（歧义）：编译错误

- 绑定类型确定

**静态绑定：**

- 如果方法是 *private*、*static*、*final* 或**构造器**，编译器在编译时就能确定具体调用哪个方法
- 原因：这些方法不能被子类覆盖，因此无需运行时决定

被 *private* 修饰的子类就算按照规则重写了也会被认为是一个独立方法，不会绑定它

```
1  class Parent {
2      private void sayHello() {
3          System.out.println("Hello from Parent");
4      }
5      void callPrivate() {
6          sayHello(); // 静态绑定到 Parent.sayHello()
7      }
8  }
9  class Child extends Parent {
10     private void sayHello() { // 不是重写，只是新方法
11         System.out.println("Hello from Child");
12     }
13 }
14 public class Main {
15     public static void main(String[] args) {
16         Parent p = new Child();
17         p.callPrivate(); // 输出: Hello from Parent
18     }
19 }
```

**动态绑定：**

- 如果方法是普通的实例方法非 *private*，*static*，*final* 或构造器

编译器生成动态调用指令，具体方法在运行时根据对象的实际类型决定

- 运行时的动态绑定

**输入：**运行时  $x$  的实际对象类型（假设是  $D$ ， $D$  是  $C$  的子类）

**过程：**

- $JVM$  获取  $x$  的实际类型（例如  $D$ ）
- 在  $D$  的方法表中查找与匹配的方法。
  - 如果  $D$  定义了  $f(String)$ ，调用它
  - 否则，向上查找  $D$  的超类，直到找到为止

**优化：** $JVM$  为每个类预先创建**方法表**（*vtable*）

记录所有方法的签名和实际实现地址，避免每次调用都搜索继承链

## 8. *final* 类和方法--**阻止继承与内联优化**

如果一个类被声明为 *final*，则它不能被继承，即不能有子类

如果一个方法被声明为 *final*，则子类不能重写（*override*）该方法

域（字段）被声明为 *final* 时，构造对象后其值不可更改

在 *final* 类中，所有方法自动成为 *final*，因为类本身不可继承，无需显式声明

与 *final* 方法不同，*final* 类的域不会自动成为 *final*，需要显式声明

早期 *Java* 中，*final* 方法被用来提示编译器进行内联（*inlining*）优化

即将方法调用替换为方法体的直接代码，减少调用开销

```
1  class Employee {
2      final String getName() {
3          return "John";
4      }
5  }
6  // 内联后: String name = "John"; (而不是调用方法)
```

现在的即时编译器（*JIT Compiler*）更智能：

- 它分析类层次关系，检测方法是否被重写

- 如果方法未被重写且适合内联，*JIT* 会自动内联，无需依赖 *final*
- 如果后续加载的子类重写了方法，*JIT* 会取消内联（去优化），确保正确性

现代 *Java* 中，*final* 的性能优化作用已不显著，主要用于设计控制

## 9. 对象引用类型的转换

**向上转型**是指将子类引用转换为超类引用（如 *Manager* 转为 *Employee*）

这是多态的核心，是隐式完成的，是继承中很重要的语法

**向下转型**将超类引用转换为子类引用（如 *Employee* 转为 *Manager*）

这需要显式类型转换，因为编译器无法确定运行时对象的实际类型

```
1  Employee emp = new Manager("Bob", 60000, 10000);
2  Manager mgr = (Manager) emp; // 显式向下转换，成功
3  mgr.setBonus(20000);
```

类型转换的根本原因是

**某些情况下，我们需要暂时忽略对象的实际类型**

**以便访问其全部功能，或者将对象放入某种通用容器（如数组或集合）中**

```
1  class Employee {
2      private String name;
3      private double salary;
4
5      public Employee(String name, double salary) {
6          this.name = name;
7          this.salary = salary;
8      }
9
10     public double getSalary() {
11         return salary;
12     }
13 }
14
15 class Manager extends Employee {
16     private double bonus;
17
18     public Manager(String name, double salary, double bonus) {
19         super(name, salary);
```

```

20         this.bonus = bonus;
21     }
22
23     public void setBonus(double bonus) {
24         this.bonus = bonus;
25     }
26
27     @Override
28     public double getSalary() {
29         return super.getSalary() + bonus; // 重写方法，包含奖金
30     }
31 }

```

现在，假设我们有一个 *Employee* 数组，其中既包含普通员工，也包含经理：

```

1  Employee[] staff = new Employee[2];
2  staff[0] = new Employee("Alice", 50000);
3  staff[1] = new Manager("Bob", 60000, 10000);

```

我们在数组的协变性里之前说了

在运行时，*JVM* 只会检查数组的**实际类型**（这里是 *Manager[]*）

```

1  Employee[] staff = new Employee[2];
2  staff[0] = new Employee("Alice", 50000);
3  staff[1] = new Manager("Bob", 60000, 10000);

```

*staff* 数组的类型是 *Employee[]*，因此每个元素都被视为 *Employee* 类型

对于 *staff[1]*，它实际引用的是一个 *Manager* 对象

但由于运行时的数组类型限制，我们只能通过 *Employee* 类型访问它

如果想调用 *Manager* 特有的 *setBonus* 方法，必须进行类型转换

```

1  ((Manager) staff[1]).setBonus(15000); // 向下转换为 Manager 类型

```

向下转型很容易报错

向下转型必须要求被转换的对象的运行时类型（实际类型）是要转换到的目标类型或其子类型

```
1 Employee emp = new Employee("Alice", 50000);
2 Manager mgr = (Manager) emp; // 运行时抛出 ClassCastException
```

当向下转换失败时，*Java* 运行时抛出 *ClassCastException*

这是运行时错误，未捕获会导致程序终止

为了避免 *ClassCastException*，*Java* 提供了 *instanceof* 操作符

用于在向下转换前检查对象是否属于目标类型

```
1 if (staff[1] instanceof Manager) {
2     Manager mgr = (Manager) staff[1]; // 安全转换
3     mgr.setBonus(15000);
4     System.out.println("Bonus set for " + mgr.getSalary());
5 } else {
6     System.out.println("Not a Manager!");
7 }
```

10. *abstract* 关键字可以修饰方法或类

被 *abstract* 关键字修饰的方法**强制子类实现**，且 *abstract* 方法不能有主体

包含抽象方法的类必须被声明为 *abstract*

```
1 abstract class Person {
2     private String name;
3
4     public Person(String name) {
5         this.name = name;
6     }
7
8     public String getName() {
9         return name;
10    }
11
12    // 抽象方法，子类必须实现
13    public abstract String getDescription();
14 }
15
16 class Employee extends Person {
17     private String jobTitle;
18
19     public Employee(String name, String jobTitle) {
```

```

20         super(name);
21         this.jobTitle = jobTitle;
22     }
23
24     @Override
25     public String getDescription() {
26         return "Employee: " + jobTitle;
27     }
28 }
29
30 class Student extends Person {
31     private String major;
32
33     public Student(String name, String major) {
34         super(name);
35         this.major = major;
36     }
37
38     @Override
39     public String getDescription() {
40         return "Student majoring in " + major;
41     }
42 }

```

- 抽象类可以包含具体方法和抽象方法

如 *Person* 中的 *getName()* 是具体方法，而 *getDescription()* 是抽象方法

这种设计允许共享通用逻辑，同时强制子类实现特定行为

抽象类不必全是抽象方法，也可以没有抽象方法，但仍可声明为 *abstract*（例如，防止实例化）

- 抽象类不能实例化

*new Person()* 是非法的，因为抽象类是 **未完成** 的模板

但可以用它声明变量

```

1  Person p = new Student("Alice", "CS"); // 合法，引用具体子类对象

```

- 子类的选择

- 如果子类没有实现所有抽象方法，它必须也被声明为 *abstract*
- 如果子类实现了所有抽象方法，它就成为具体类，可以实例化

- *abstract* 不可以与 *static*, *final*, *private* 同时出现

因为 *abstract* 要保证其修饰的方法可以被子类重写

## 11. 谨慎使用 *protected* 字段，建议使用 *protected* 方法

**优先使用 *private* 加 *public* 的 *getter/setter***：将域保持私有，通过公共或受保护的方法控制访问

**使用 *protected* 方法**：相比域，*protected* 方法更安全，因为它只暴露行为，不暴露数据

例如 *Object.clone()* 是一个 *protected* 方法，子类可以重写它，但不能直接访问内部状态

## 12. *Object* 类详解

是 *Java* 中所有类的始祖，在 *Java* 中所有类都直接或间接继承自它

八种基本类型也可以通过自动装箱机制转为包装类被 *Object* 对象引用

- 如果没有明确地指出超类，*Object* 就被认为是这个类的超类

所以我们不需要显式地去 *extends Object* 这个超类

这种设计确保了 *Java* 中所有对象都共享一组由 *Object* 提供的基本方法，并支持多态性

- 可以使用 *Object* 类型的变量引用任何类型的对象

```
1  Object obj1 = new String("Hello");    // String 对象
2  Object obj2 = new Integer(42);        // Integer 对象
3  Object obj3 = new ArrayList<String>(); // ArrayList 对象
```

但是，*Object* 类型变量只能调用 *Object* 类中声明的方法

如果需要调用具体类型的方法，必须进行类型转换

```
1  Object obj = "Hello";                // 合法，String 是 Object 的子类
2  if (obj instanceof String) {
3      String str = (String) obj;
4      System.out.println(str.length());
5  } // 类型转换后才能调用 String 的方法
```

- *toString()*



源码如下

代码块

```
1 public String toString() {  
2     return getClass().getName() + "@" + Integer.toHexString(hashCode());  
3 }
```

返回对象的字符串表示，默认实现是 类名@哈希码

示例

```
1 Object obj = new Object();  
2 System.out.println(obj.toString());
```

输出类似 *java.lang.Object@1b6d3586*

**子类通常重写此方法以提供更有意义的描述**

- *equals(Object obj)*

源码如下

代码块

```
1 public boolean equals(Object obj) {  
2     return (this == obj);  
3 }
```

比较两个对象是否相等，默认实现**比较引用**（`==`）

子类可以重写以实现内容比较（如 *String* 类）

因为 *Java* 对于 *equals()* 方法有以下要求

**自反性**：对于任何非空引用 *x*，*x.equals(x)* 应该返回 *true*

**对称性**：对于任何引用 *x* 和 *y*，当且仅当 *y.equals(x)* 返回 *true*，*x.equals(y)* 也应该返回 *true*

**传递性**：对于任何引用 *x*、*y* 和 *z*，如果 *x.equals(y)* 返回 *true*，*y.equals(z)* 返回 *true*，*x.equals(z)* 也应该返回 *true*

**一致性**：如果 *x* 和 *y* 引用的对象没有发生变化，反复调用 *x.equals(y)* 应该返回同样的结果

**非空性**：对于任意非空引用 *x*，*x.equals(null)* 应该返回 *false*

所以根据以上五条特性

如果要重写，可以遵循以下代码

而且注意如果要重写 *equals()* 方法，一定要重写 *hashCode()*

```
1  class Employee {
2      private String name;
3      private LocalDate hireDay;
4
5      @Override
6      public boolean equals(Object otherObject) {
7          if (this == otherObject) return true; // 引用相同
8          if (otherObject == null) return false; // null 检查
9          if (getClass() != otherObject.getClass()) return false; // 类不同
10         Employee other = (Employee) otherObject;
11         return Objects.equals(name, other.name) && Objects.equals(hireDay,
12             other.hireDay);
13     }
14
15     @Override
16     public int hashCode() {
17         return Objects.hash(name, hireDay);
18     }
19 }
```

首先一定要注意传入的参数一定要是 *Object* 类型的

如果不是，则不是一个对于 *equals()* 方法的重写，而是完全独立的一个方法

所以我们建议重写时始终在前面添加 *@Override*

*if(this == otherObject) return true;* 如果引用是相同的，说明内容绝对相同，没有比较的必要

*if(otherObject == null) return false;* 如果要比较的对象是 *null*，直接返回 *false*

避免后续操作因空指针异常中断

*if(getClass() != otherObject.getClass()) return false;* 这一步严格限制了同类比较，不认为子类与超类等

如果需要认为子类和超类相等需要额外的判断

*Employee other = (Employee) otherObject;*

因为前一步已经确定了 *otherObject* 的实际类型是 *Employee*

这一步可以安全向下转型

*return Objects.equals(name, other.name) && Objects.equals(hireDay, other.hireDay);*

是重写的核心部分

先说一下 *Objects.equals()* 的机制

代码块

```
1 public static boolean equals(Object a, Object b) {  
2     return (a == b) || (a != null && a.equals(b));  
3 }
```

如果 *a* 和 *b* 都是 *null*，返回 *true*

如果其中一个是 *null*，另一个不是，返回 *false*

如果都不为 *null*，调用对应类型的 *a.equals(b)*，如果 *a* 和 *b* 是 *String* 那么就调用 *String* 的 *equals()*

这只是不涉及继承时的 *equals()* 设计

如果涉及，那么**超类不能用** *getClass()* 会导致子类调用 *super.equals()* 时总是失败，破坏继承协作

超类用 *instanceof*，允许子类对象通过

子类根据需求选择 *instanceof*（共享超类语义）或 *getClass()*（独立语义）

- *hashCode()*

**散列码** (*hashCode*) 是由**对象**导出的一个 *int* 类型的值

*null* 的 *hashCode* 是不存在的，但在一些工具类比如 *Objects* 里把 *null* 视为 0

由于 *hashCode* 方法定义在 *Object* 类中，因此每个对象都有一个默认的散列码

**并不一定是对象的地址**

散列码是没有规律的，如果 *x* 和 *y* 是两个不同的对象，*x.hashCode()* 与 *y.hashCode()* 基本上不会相同

⚽ 如果两个对象 *equals()* 为 *true*，它们的 *hashCode()* 必须相同

但是如果 *x.equals(y)* 返回 *false* , *x.hashCode()* 和 *y.hashCode()* 也可以相同 (哈希冲突)

但应尽量避免过多冲突以提高性能

### 而基础类型没有散列码, 是因为它们不是对象

而许多类型都会重写 *hashCode()* 比如 *String* 类型

这时该对象的 *hashCode()* 就根据类的 *equals()* 方法而定

比如 *String* 类型的 *hashCode()* 方法是根据该对象存储的内容决定的

就是为了配合 *String* 类的 *equals()* 重写

因为 *String* 类的 *equals()* 比较的是内容而非地址, 所以 *hashCode()* 方法就必须根据该对象存储的内容而决定

- *getClass()*

返回对象的**运行时类** (*Class* 对象) , 不可重写 (*final* 方法)

*obj.getClass().getName()* 返回类名

*obj.getClass().getSuperClass()* 返回超类的信息

- *clone()*

创建对象的副本, 默认是 *protected* , 需要实现 *Cloneable* 接口并重写

**线程相关方法** (如 *wait()*、*notify()*、*notifyAll()*)

- 用于多线程同步, 在第 14 章会详细介绍

## 13. 泛型数组列表 *List* 接口

其有两个实现类 *ArrayList* 和 *LinkedList*

都是采用**类型参数**的**泛型类**

*JavaSE7* 中, 可以省去右边的类型参数

代码块

```
1 List<Employer> linkedlist = new LinkedList<Employer>();
2 List<Employer> linkedlist = new LinkedList<>();
3 List<Employer> arraylist = new ArrayList<>(100);
```

分为长度和容量，长度是指当前存储的元素个数，容量是在不移动到新的内存块前所能存储的最多元素个数

*ArrayList* 默认容量为 10，可以在构造器中指定容量

代码块

```
1 private static final int DEFAULT_CAPACITY = 10;
```

如果调用 *add* 且长度已经等于容量

*List* 就将自动地创建一个大的数组，并将所有的对象从较小的数组中拷贝到较大的数组中

此外，*CPP* 向量是值拷贝

如果 *a* 和 *b* 是两个向量，赋值操作 *a = b* 将会创建一个与 *b* 长度相同的新向量 *a*，并将所有的元素由 *b* 拷贝到 *a*

而 *Java* 中只是让 *a* 和 *b* 指向同一个数组

代码块

```
1 List<Employer> a = new ArrayList<>();
2 List<Employer> b = new ArrayList<>();
3 System.out.println(a == b); //false
4 a = b;
5 System.out.println(a == b); //true
```

*ensureCapacity()* 和 *trimToSize()*

代码块

```
1 /**
2  * 将 ArrayList 的底层数组 elementData 的长度调整为当前元素数量 (size)，避免内存浪费。
3  * 触发条件：仅当 elementData.length > size 时执行扩容操作。
4  * */
5 public void trimToSize() {
6     modCount++;
7     if (size < elementData.length) {
8         //检查是否需要扩容
9         elementData = (size == 0)
10             ? EMPTY_ELEMENTDATA
```

```

11          //若当前列表为空 (size == 0) , 直接将 elementData 指向预定义的
    EMPTY_ELEMENTDATA (静态常量空数组)
12          : Arrays.copyOf(elementData, size);
13          //调用 Arrays.copyOf() 创建一个长度为 size 的新数组, 并将原数组浅拷贝
    地指向新数组
14      }
15  }
16
17
18
19  public void ensureCapacity(int minCapacity) {
20      // 检查是否需要扩容 (当前容量不足且不满足快速路径条件)
21      if (minCapacity > elementData.length
22          && !(elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA
23              && minCapacity <= DEFAULT_CAPACITY)) {
24          modCount++;          // 结构性修改计数
25          grow(minCapacity);   // 执行扩容
26      }
27  }

```

在软件开发中, 我们经常需要让新编写的、使用了泛型以增强类型安全性的代码

去调用那些在 *Java* 泛型出现之前编写的遗留代码

以前版本的遗留代码是没有泛型的, *ArrayList* 底层就是 *Object* 数组, 没有检查

现在加入泛型以后, 虽然底层也是 *Object* 数组, 但是加入了编译时检查内部是否为我们泛型指定的类型

但是我们可以通过遗留的无泛型 *ArrayList* 直接绕过有泛型 *ArrayList* 检查, 加入非指定对象

代码块

```

1  import java.util.ArrayList;
2
3  class Employee {
4      private String name;
5      public Employee(String name) { this.name = name; }
6      @Override
7      public String toString() { return "Employee: " + name; }
8  }
9  /**
10   * 一个遗留的类, 它在泛型出现前编写, 因此使用原始类型 ArrayList。
11   */
12  class LegacyManager {
13      public void update(ArrayList staff) {
14          System.out.println("遗留代码: 向列表中添加一个非 Employee 对象...");

```

```

15         // 这是一个危险的操作，因为它向列表中添加了一个不兼容的类型
16         staff.add("I am not an Employee object!");
17     }
18 }
19 public class Main {
20     public static void main(String[] args) {
21         // 1. 创建一个泛型化的列表，保证了类型安全
22         ArrayList<Employee> staff = new ArrayList<>();
23         staff.add(new Employee("Alice"));
24
25         // 2. 将泛型列表传递给遗留方法
26         LegacyManager legacyManager = new LegacyManager();
27         legacyManager.update(staff); // 编译器通常会在此处给出一个“未经检查的调用”警告
28
29         // 3. 尝试从被“污染”的列表中检索元素
30         System.out.println("尝试从列表中读取数据...");
31         try {
32             // 当代码尝试将 "I am not an Employee object!" 强制转换为 Employee 类型时，
33             // for-each 循环会隐式进行类型转换，从而抛出 ClassCastException。
34             for (Employee e : staff) {
35                 System.out.println(e);
36             }
37         } catch (ClassCastException e) {
38             System.err.println("出错了！列表中包含了非 Employee 类型的对象。");
39             e.printStackTrace();
40         }
41     }
42 }

```

#### 代码块

```

1 // 下面这行代码同样会产生 "unchecked cast" (未经检查的转换) 警告
2 ArrayList<Employee> staff = (ArrayList<Employee>) rawList;

```

这是因为 *Java* 泛型的**类型擦除**机制

在编译后，*ArrayList* < *Employee* > 会被擦除为原始的 *ArrayList*

这样，不需要为每个泛型类型生成新的类

*ArrayList* < *Employer* > 和 *ArrayList* < *Employee* > 编译后都是 *ArrayList*

只是编译时执行了泛型的检查，避免了运行时类型检查的开销

因此，在运行时，*JVM* 无法区分 (*ArrayList* < *Employee* >) 和 (*ArrayList*) 这两种转换

它执行的是相同的运行时检查

所以这个转换本质上是未经编译时检查的，只在运行时检查

当你确认与遗留代码的交互是你需要的，并且你已经理解并愿意承担潜在的 *ClassCastException* 风险时

你可以使用 `@SuppressWarnings("unchecked")` 注解来抑制编译器的警告

## 14. 自动装箱与拆箱

因为八大基础类型并不是引用类型，它们的变量直接存储**值**，通常存放在速度更快的**栈内存**中，性能很高

但 *Java* 中的泛型只能接受引用类型，它们的变量存储的是对象的**内存地址引用**，而对象本身存放在**堆内存**中

所以需要通过包装类来把基础类型包装为类

*Integer*、*Long*、*Float*、*Double*、*Short*、*Byte*、*Character*、*Void*、*Boolean*

**包装类** *final* 且是**不可变**的

代码块

```
1 ArrayList<Integer> arrayList = new ArrayList<>();
```

由于每个值分别包装在对象中，所以 *ArrayList < Integer >* 的效率远远低于 *int[]*

其他基础类型也同理

*Java5* 提供的两个语法糖

- **自动装箱**，当需要一个包装类对象时，可以直接提供一个基础类型的值
- **自动拆箱**，当需要一个基础类型的值时，可以直接使用一个包装类对象

代码块

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class WrapperDemo {
5     public static void main(String[] args) {
6         // 使用泛型时，必须用包装类 Integer
7         List<Integer> list = new ArrayList<>();
8         // 这个地方不能写成 List<int> list = new ArrayList<>();
```



```

9      // 因为是自动装箱和拆箱是对于一个对象而言的，而不是类
10
11     // 自动装箱：编译器自动将 int 100 转换为 Integer.valueOf(100)
12     list.add(100);
13     // 等价于
14     list.add(Integer.valueOf(100));
15     // 自动拆箱：编译器自动将 list.get(0) 对象转换为 int 类型
16     int value = list.get(0);
17     // 等价于
18     int value = list.get(0).intValue();
19
20     System.out.println("从列表中取出的值: " + value);
21
22     // 在计算中也同样适用
23     Integer a = 5; // 自动装箱
24     int b = 10;
25
26     // a 会自动拆箱为 int, 然后进行加法运算
27     int sum = a + b;
28     System.out.println("总和: " + sum);
29 }
30 }

```

#### 代码块

```

1  Integer a = 1000;
2  Integer b = 1000;
3  System.out.println(a == b); // 输出: false
4
5  Integer c = 100;
6  Integer d = 100;
7  System.out.println(c == d); // 输出: true

```

在比较两个包装类对象的值是否相等时始终使用 `.equals()` 方法

#### 这是因为自动装箱的规范要求

对于 `boolean`、`byte`、`char`（范围 0 – 127）

以及 `short`、`int`（–128 – 127）

当进行自动装箱时，会从一个内部的缓存池中获取对象，而不是每次都创建新对象

- 当 `Integer c = 100;` 和 `Integer d = 100;` 被执行时，因为 100 在 [–128, 127] 范围内

它们都从缓存池中获取了**同一个** `Integer` 对象

所以 `c` 和 `d` 指向相同的内存地址，`c == d` 为 `true`

- 而 1000 超出了这个范围，所以 `Integer a = 1000;` 和 `Integer b = 1000;` 会创建两个全新的、独立的 `Integer` 对象，因此 `a == b` 为 `false`

## 15. 参数数量可变的方法

**可变参数** (`...`) 允许方法接收不定数量的参数

它本质上是一个**语法糖**，在底层被编译器转换为一个**数组**

调用时，基础类型的参数会被**自动装箱**

自定义可变参数时，必须是**最后一个参数**，并且**只能有一个**

代码块

```
1 public static void main(String... args) {
2     double[] numbers = { 3.1, 40.4, -5 };
3     double maxVal = MathUtils.max(numbers); // 完全合法，直接传递数组
4 }
```

## 16. Enum 语法糖

枚举 `Enum` 是一种**特殊的 `final` 类**，用于定义一组固定的常量对象

但远不止是常量的集合，它可以像一个常规的 `final` 类一样拥有自己的**构造器、字段和方法**

枚举的构造器只能在内部使用，用于在创建枚举常量时初始化其字段

因此，它通常被声明为 `private`，也可以被声明为 `package-private`，即默认

如果枚举定义的任何成员（字段或方法），枚举常量列表的末尾必须以**分号 ( ; )** 结束

代码块

```
1 public enum Size {
2     SMALL,
3     MEDIUM,
4     LARGE,
5     EXTRA_LARGE
6     //无需分号
7 }
8
9 public enum Size {
10     // 在定义常量时，调用构造器
11     SMALL("S"),
12     MEDIUM("M"),
```

```

13     LARGE("L"),
14     EXTRA_LARGE("XL");
15
16     private String abbreviation; // 实例域
17
18     // 构造器必须是 private 或 package-private (默认)
19     // 它只在JVM构造枚举常量时被调用
20     private Size(String abbreviation) {
21         this.abbreviation = abbreviation;
22     }
23
24     // 普通方法
25     public String getAbbreviation() {
26         return abbreviation;
27     }
28 }
29 /*
30 类似的等价写法
31 不过实际上Enum<T>是final的，不能被我们直接继承，只能使用Enum关键字声明
32 */
33 public final class Color extends Enum<Color> {
34     // 枚举常量是静态不可变的实例
35     public static final Color RED = new Color("RED", 0);
36     public static final Color GREEN = new Color("GREEN", 1);
37     public static final Color BLUE = new Color("BLUE", 2);
38
39     // 私有构造器
40     private Color(String name, int ordinal) {
41         super(name, ordinal);
42     }
43
44     // 静态代码块初始化
45     static {
46         // 注册所有枚举常量
47     }
48 }

```

实际上定义了一个名为 *Size* 的类

它有且仅有四个**唯一的** *public static final* 的实例

*Size.SMALL*, *Size.MEDIUM*, *Size.LARGE*, *Size.EXTRA\_LARGE*

- **实例固定：** 不能使用 *new* 关键字来创建枚举的新实例
- **类型安全：** 不能将一个 *Size* 类型的变量赋值为除了这四个实例之外的任何值，包括 *null*
- **可以直接用 == 比较：** 每个枚举常量在 *JVM* 中都是唯一的单例

所有的枚举类型都隐式地继承自 *java.lang.Enum* 类，**也不允许显式继承**

并自动获得

*toString()*：返回枚举常量的名字

*name()*：返回枚举常量的名字

输出完全一致

区别在于

*toString()* 来自于 *java.lang.Object* 类，只是 *Enum* 重写使其调用 *name()*，且不为 *final* 可以继续重写

而 *name()* 直接由 *java.lang.Enum* 类定义，是 *final* 的，不可重写

*ordinal()*：返回枚举常量的序号，第一个为 0，第二个为 1，以此类推

*compareTo()*：如果前枚举常量序号减去后枚举常量序号的值

且编译器会为每个枚举添加 *values()* 和 *valueOf(String name)* 静态方法，非常方便

*values()*：**返回一个包含所有枚举常量的数组**，顺序与声明时一致

*valueOf(String name)*：**根据给定的名称字符串返回对应的枚举常量**

如果名称不匹配，则抛出 *IllegalArgumentException*

代码块

```
1  package org.Shu;
2
3  import java.util.Arrays;
4
5  public class Main {
6      public static void main(String... args) {
7          var temp = MyEnum.values();
8          for (MyEnum myEnum : temp) {
9              System.out.println("序号为" + myEnum.ordinal() + ", 名字为" +
myEnum.name() + ", 内容为" + MyEnum.valueOf(myEnum.name()));
10         }
11     }
12 }
13 enum MyEnum{
14     SMALL("S" , "小号"),
15     MEDIUM("M" , "中号"),
16     LARGE("L" , "大号");
17     private final String[] string;
18     @Override
19     public final String toString() {
20         return Arrays.toString(string);
21     }
22 }
```

```
21     }
22     MyEnum(String... string){
23         this.string = string;
24     }
25 }
```

## 17. 反射

Java 提供了一种**运行时机制**，它允许一个正在运行的程序**检查和操纵**自身

- **在运行时分析类的能力**：获取一个类的所有构造器、方法、字段（成员变量）、父类、接口等信息
- **在运行时检查和操作对象**：可以获取任意对象的字段值，也可以设置新的值，**即使它们是 *private* 的**

它的核心入口是 `java.lang.Class` 对象，它代表了 *JVM* 中一个类的所有元信息

获取 `Class` 对象有三种主要方式

`object.getClass()`：通过对象实例获取

`ClassName.class`：知道具体的类可以直接使用 `.class` 语法，这是**最安全、性能也最好**的一种方式

`Class.forName()`：通过类的**全限定名**获取

代码块

```
1  String str = "Hello";
2  Class<?> c1 = str.getClass(); // 获取 String 类的 Class 对象
3
4  Employee emp = new Employee();
5  Class<?> c2 = emp.getClass(); // 获取 Employee 类的 Class 对象
6
7  Class<String> c1 = String.class;
8  Class<Employee> c2 = Employee.class;
9  Class<Integer> c3 = int.class; // 也可以获取基本类型的 Class 对象
10
11 try {
12     // 必须提供完整的包名 + 类名
13     Class<?> c = Class.forName("java.util.Scanner");
14 } catch (ClassNotFoundException e) {
15     // 如果找不到这个类，会抛出异常
16     e.printStackTrace();
17 }
```

由于一个类只有唯一的一个 `Class` 对象，所以可以使用 `==` 进行比较

也可以对一个 *Class* 对象调用 *newInstance()* 来创建一个对应的实例

所以可以用这种方式绕过无法直接 *new* 一个 *Enum* 的编译时检查，但运行时仍会抛出异常

代码块

```
1  package org.Shu;
2
3  import java.util.Arrays;
4
5  public class Main {
6      public static void main(String... args) {
7          System.out.println(MyEnum.SMALL.getClass() ==
MyEnum.LARGE.getClass()); //输出:true
8      }
9  }
10 enum MyEnum{
11     SMALL("S" , "小号"),
12     MEDIUM("M" , "中号"),
13     LARGE("L" , "大号");
14     private final String[] string;
15     @Override
16     public final String toString() {
17         return Arrays.toString(string);
18     }
19     MyEnum(String... string){
20         this.string = string;
21     }
22 }
```

---

*Java* 的反射库 (*java.lang.reflect*) 提供了一套强大的工具

核心的反射类包括：

- *Class*: 程序的入口，代表一个类/接口的元数据
  - *Field*: 代表类的一个字段（成员变量）
  - *Method*: 代表类的一个方法
  - *Constructor*: 代表类的一个构造器
  - *Modifier*: 一个工具类，用于解码由 *getModifiers()* 返回的整数

*Class* 类提供了两组方法来获取其成员

非 *Declared*

仅限 *public* 成员，均包含父类成员

*getFields()*

*getMethods()*

*getConstructors()*

## Declared

获取当前类中**所有声明的成员**，无视访问权限，均不包含父类成员

*getDeclaredFields()*

*getDeclaredMethods()*

*getDeclaredConstructors()*

## 1. 获取名称

这三个类都有 *getName()* 方法，返回成员的名字

- *field.getSimpleName()*: 返回字段名
- *method.getSimpleName()*: 返回方法名
- *constructor.getSimpleName()*: 返回类名

如果是 *getName()* 则返回全限定名

## 2. 获取类型

- **字段类型**: *field.getType()* 返回一个 *Class* 对象，代表该字段的类型
- **方法返回类型**: *method.getReturnType()* 返回一个 *Class* 对象，代表方法的返回类型
- **参数类型**: *method.getParameters()* 或 *constructor.getParameters()* 返回一个 *Parameter[]* 数组，其中每个对象代表一个参数

## 3. 获取修饰符

*getModifiers()* 方法返回一个 *int* 值

这个整数的每一位代表一个修饰符（如 *public*, *static*, *final*）

直接解读这个整数很麻烦，因此我们使用 *java.lang.reflect.Modifier* 工具类来解码它

一般只使用 *Modifier.toString(int mod)*:

**最方便的方法**，直接将修饰符整数转换为我们熟悉的字符串形式，例如 "*public static final*"

- `Modifier.isPublic(int mod)`: 判断是否为 *public*
  - `Modifier.isPrivate(int mod)`: 判断是否为 *private*
  - `Modifier.isStatic(int mod)`: 判断是否为 *static*
  - `Modifier.isFinal(int mod)`: 判断是否为 *final*
-

如何利用反射来读取和修改一个对象的域（成员变量）的值，即使这个域是私有的

*Field.get(Object obj), Field.set(Object obj, Object newValue)*

获取或修改指定对象 *obj* 中，由当前 *Field* 对象所代表的那个域的值

如果 *f* 是代表 *name* 域的 *Field* 对象

*employee* 是一个具体的员工对象

那么 *f.get(employee)* 就会返回该员工的姓名（一个 *String* 对象）

### 默认情况下，反射不能破坏类的封装性

*Java* 的安全机制默认会进行访问权限检查

如果尝试通过反射去 *get/set* 一个 *private* 的域

程序会直接抛出 *IllegalAccessException*

解决方案 *AccessibleObject.setAccessible(boolean flag)*

*Field*、*Method* 和 *Constructor* 类都继承自 *AccessibleObject*，因此都有这个方法

这用于**关闭访问权限检查**

在调用 *get* 或 *set* 方法之前，先对 *Field* 对象调用 *field.setAccessible(true)*

执行之后，即使是 *private* 的域，也可以被成功地读取和修改

### 通过反射实现动态拓展数组，也就是实现 *Arrays.copyOf()*

一个直观但错误的想法是创建一个 *Object[]* 数组来接收任何类型的数组，然后向下转型成实际要用的数组

代码块

```
1  /**
2   * 一个错误的数组拷贝方法。
3   * @param a 需要拷贝的原始数组（必须是对象数组）
4   * @param newLength 新数组的长度
5   * @return 一个新的 Object[] 数组，其中包含原始数组的元素
6   */
7  public static Object[] badCopyOf(Object[] a, int newLength) {
8      // 问题所在：这里创建的数组类型是固定的 Object[]
9      Object[] newArray = new Object[newLength];
10     System.arraycopy(a, 0, newArray, 0, Math.min(a.length, newLength));
11     return newArray;
12 }
13
14 Employee[] staff = new Employee[3];
```



```

15 // ... 初始化 staff ...
16
17 // badCopyOf 返回的是一个 Object[]
18 Object[] result = badCopyOf(staff, 10);
19
20 // 这行代码在运行时会抛出 ClassCastException!
21 // java.lang.ClassCastException: [Ljava.lang.Object; cannot be cast to
    [Lcom.example.Employee;
22 Employee[] newStaff = (Employee[]) result;

```

但向下转型的前提是需要父类引用指向子类对象，也就是实际类型本身就是我们要向下转换的类型  
但是如果本身就是父类引用指向父类对象，则无法向下转换，会报 *ClassCastException*

为了解决这个问题，新数组的运行类型必须与原始数组的运行类型完全一致

例如，如果原始数组是 *Employee[]*，新数组也必须是 *Employee[]*

*java.lang.reflect.Array* 类提供了实现这一目标的关键方法：*newInstance()*

同时，我们将 *Object[]* 改为了 *Object*

因为基本类型数组（如 *int[]*）可以被转换为 *Object*

但不能被转换为 *Object[]*

这使得我们的方法真正通用，可以处理任何类型的数组

代码块

```

1  import java.lang.reflect.Array;
2
3  /**
4   * 一个通用的、正确的数组拷贝方法。
5   * @param a 需要拷贝的原始数组（可以是任何对象数组或基本类型数组）
6   * @param newLength 新数组的长度
7   * @return 一个与原始数组类型相同的新数组，长度为 newLength
8   */
9  public static Object goodCopyOf(Object a, int newLength) {
10     // 1. 获取原始数组的 Class 对象
11     Class<?> cl = a.getClass();
12
13     // 2. 确认它是一个数组
14     if (!cl.isArray()) {
15         return null; // 或者抛出异常
16     }
17
18     // 3. 获取数组的组件类型（例如，对于 Employee[] 数组，组件类型就是
        Employee.class)

```

```

19     Class<?> componentType = cl.getComponentType();
20
21     // 4. 获取原始数组的长度
22     int length = Array.getLength(a);
23
24     // 5. 使用 Array.newInstance 创建一个与原始数组类型相同的新数组
25     Object newArray = Array.newInstance(componentType, newLength);
26
27     // 6. 拷贝元素
28     System.arraycopy(a, 0, newArray, 0, Math.min(length, newLength));
29
30     return newArray;
31 }

```

- `static Object get(Object array, int index)`: 获取指定数组在指定索引处的值。
- `static xxx getXxx(...)`: 针对每种基本类型 (`boolean`, `byte`, `int` 等) 的 `get` 版本。
- `static void set(Object array, int index, Object newValue)`: 设置指定数组在指定索引处的值。
- `static void setXxx(...)`: 针对每种基本类型的 `set` 版本。
- `static int getLength(Object array)`: 返回任意数组的长度 (比 `array.length` 更通用, 因为它适用于 `Object` 类型的参数)。
- `static Object newInstance(Class<?> componentType, int length)`: 创建一个具有指定组件类型和长度的一维数组。
- `static Object newInstance(Class<?> componentType, int... dimensions)`: 创建一个多维数组。

## java 中实现方法指针

我们可以类比之前通过 *Field* 类的 *get* 方法来访问对象域

与此类似, *java.lang.reflect.Method* 类中有一个核心的 *invoke* 方法

它允许我们调用封装在 *Method* 对象中的具体方法

### 代码块

```

1  Object invoke(Object obj, Object... args)
2  /*
3   obj: 隐式参数。即调用该方法的对象实例。
4   如果方法是实例方法 (非static) , obj就是调用该方法的对象。
5   如果方法是静态方法 (static) , obj可以被忽略, 传入null即可。

```

```
6  args: 显式参数。一个可变参数列表，包含了需要传递给被调用方法的所有参数。如果没有参数，可以不传或传入null。  
7  */
```

反射代码通常比直接调用更冗长、更难理解

`invoke` 方法的参数和返回值都是 `Object` 类型，这绕过了编译器的类型检查

如果传入了错误的参数类型，编译器不会报错，错误只会在运行时以异常的形式出现，这增加了调试的难度