

20180526

## 1 Java 简介

Java 流行的原因是许多大公司都用 Java 开发其核心业务。

Java 由 SUN 公司开发，SUN 公司后被 Oracle 收购。

### ✧ 1.1 Java 技术 3 个发展方向：

- ① Java SE，实现 Java 基础支持，可以进行普通的单机版程序开发；
- ② Java EE，进行企业平台的搭建，现在主要是互联网平台开发；
- ③ Java ME，嵌入式开发，基本被 Android 开发(利用 Java 封装底层 Linux 操作)取代；后来由于 Oracle 和 Google 的撕逼，Google 推出了自己的专属语言 [Kotlin](#) 来进行 Android 开发。

### ✧ 1.2 Java 特点

- ① 面向对象编程；
- ② 自动内存回收机制；
- ③ 避免复杂的指针，使用简单的引用代替；
- ④ 为数不多支持多线程的语言；
- ⑤ 高效的网络处理能力，基于 NIO 实现更高效的数据传输处理；
- ⑥ 跨平台，良好的可移植性。

### ✧ 1.3 Java 虚拟机(Java Virtual Machine)

JVM 是一个由软件和硬件模拟出来的计算机，不同操作系统使用不同的 JVM。Java 源文件\*.java 经编译得到[字节码文件](#)\*.class，再经过解释得到[机器码指令](#)，解释都要求放在 JVM 上处理。

Java 编译器针对 JVM 产生的.class 文件是独立于平台的；Java 解释器负责将 JVM 的代码在特定平台上运行。

### ✧ 1.4 JDK 简介

Java Development Kit: Java 开发工具包。

JDK 历史：

- JDK 1.0，只提供基础环境，功能不完善；
- JDK 1.2，更名为 Java2，增加了 GUI 改进包、类集框架；
- JDK 1.4 是一个使用较广泛的版本，定义了许多重要的组件，如 NIO；
- JDK 1.5 是 Java 发展的重要里程碑，公布了新的结构化特点，极大简化开发难度；
- JDK 1.7，Oracle 收购 SUN 之后推出的正式版本，修复了很多漏洞；
- JDK 1.8，第一次正式提出 lambda 表达式的使用，支持函数式编程；
- JDK 1.9，提出交互式命令行工具、模块化设计。

然而现在 [JDK 1.10](#) 都已经发布了...

JDK1.8 是经过长期测试稳定的版本，项目中建议使用 JDK1.8。

JRE (Java Runtime Environment)只提供运行环境，不提供开发环境。

如果客户端要运行 Java，只需使用 JRE 就可以。

JDK 包含了 JRE，安装 JDK 也会安装 JRE。

两个重要的命令：

① 编译命令：javac.exe;

② 解释命令：java.exe

都在 JDK 安装目录的 bin 目录下，将该 bin 目录添加到环境变量。

```
$ javac -version
javac 10.0.1

$ java -version
java version "10.0.1" 2018-04-17
Java(TM) SE Runtime Environment 18.3 (build 10.0.1+10)
Java HotSpot(TM) 64-Bit Server VM 18.3 (build 10.0.1+10, mixed mode)
```

## 2 Java 编程起步

### ✧ 2.1 第一个程序

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("hello world!");
    }
}
```

1) 对源代码进行编译

```
$ javac Hello.java
```

利用 JVM 编译出与平台无关的字节码文件 Hello.class

2) 在 JVM 上进行程序的解释执行：

```
$ java Hello
hello world!
```

① Java 最基础的单元是类，所有程序必须封装在类中执行。

1) **public class** 定义的类名必须与文件名一致，一个.java 文件只能有一个 public class 定义的类；

比如将上面 Hello 改为 HelloWorld，编译会报错：

```
$ javac Hello.java
Hello.java:1: 错误: 类 HelloWorld 是公共的, 应在名为 HelloWorld.java 的文件中声明
```

2) **class** 定义的类可以与文件名不一致，一个.java 文件可以有多个。

编译后生成对应多个.class 字节码文件。

**注意：**实际开发很少在一个.java 文件里定义多个 class；而是只定义一个 public class 就可以。但学习的时候定义很多.java 文件会很混乱；为了方便学习，还是会一个.java 文件定义多个 class。

② 主方法

主方法是所有程序执行的起点，必须定义在类中。

### ✧ 2.2 JShell

很多编程语言为了方便使用者代码开发，都会有 Shell 交互式编程环境。

有时只是为了进行简短的程序验证，但在 Java 里必须编写很多结构代码才行。JDK 9 开始提供 JShell，可以只编写代码就能快速验证。类似于 Python Shell，但貌似用处不大。

```
$ jshell
| 欢迎使用 JShell -- 版本 10.0.1
| 要大致了解该版本，请键入: /help intro

jshell> 23 + 45
$1 ==> 68

jshell> "hello" + " " + "world"
$2 ==> "hello world"
```

也可以将语句写入文件保存，再运行：

```
jshell> /open hello.txt
hello, hikari!

jshell> /exit
| 再见
```

### ✧ 2.3 CLASSPATH 环境属性

如在 D:/learn\_java/目录有 Hello.java 文件，javac 编译在该目录生成 Hello.class 字节码文件，可以 java Hello 解释运行；但是切换到其他没有 Hello.class 的目录如 C:/运行会报错：

```
C:\>java Hello
错误: 找不到或无法加载主类 Hello
原因: java.lang.ClassNotFoundException: Hello
```

修改 CLASSPATH 环境属性：

```
C:\>SET CLASSPATH=d:/learn_java
C:\>java Hello
hello world!
```

Java 程序解释的时候 JVM 通过 CLASSPATH 设置的路径进行类的加载。CLASSPATH 默认为当前目录。

修改 CLASSPATH 的坏处是：

```
C:\>javac A.java
C:\>java A
错误: 找不到或无法加载主类 A
原因: java.lang.ClassNotFoundException: A
C:\>SET CLASSPATH=.
C:\>java A
hello world!
```

当想运行 C 盘的字节码文件时，无法运行，因为设置的 CLASSPATH 目录没有 A.class；设为 . 表示当前目录，就可以运行了。  
所以 CLASSPATH 一般采用默认设置。

**注意：**CLASSPATH 是在一个命令行的配置，如果关闭命令行该配置也消失。  
最好做法是定义为全局属性：环境变量→新建→变量名为 CLASSPATH，变量值为 .

PATH 和 CLASSPATH 的区别：

- ① PATH 是操作系统提供的路径配置，定义所有可执行程序的路径；
- ② CLASSPATH 是 JRE 提供，用于设置 Java 程序解释时类加载路径，默认为 .  
可以通过 SET CLASSPATH=<路径>设置。

## 20180527

### 3 Java 基本概念

#### ✧ 3.1 注释

好的注释使得项目维护更加方便，但很多公司由于管理不善，不写注释。如果作为菜鸟进入这样的公司，维护时面对密密麻麻几万行没有注释的代码，心中定会有几万头草泥马飞奔而过。

编译器在编译时不会对注释内容进行编译处理，Java 有 3 种注释：

- ① 单行注释：//...
- ② 多行注释：/\* ... \*/
- ③ 文档注释：/\*\* ... \*/

文档注释里需要有很多选项，建议通过开发工具控制。

开发时，单行注释比较方便；对于一些重要的类和方法建议使用文档注释。

#### ✧ 3.2 标识符和关键字

- ① 标识符由字母、数字、\_、\$组成，不能以数字开头，不能是 Java 的关键字。用以标记变量名、方法名、类名等。

JDK1.7 增加了一个神奇的特性：标识符可以用中文！

但基本没人会这么用，对于 JDK 的新特性要保守使用。

- ② 关键字是系统对于一些结构的描述处理，有特殊的含义。一般 IDE 都会显示特殊的颜色，不需要死记硬背啦。

**注意：**

- 1) JDK 1.4 添加 **assert** 关键字，断言，用于程序调试；
- 2) JDK 1.5 添加 **enum** 关键字，用于枚举定义；
- 3) 未使用到的关键字(保留关键字)：**goto**、**const**；
- 4) 特殊含义的标记：**true**、**false**、**null**，严格来说不算关键字。

## 4 Java 数据类型

### ✧ 4.1 Java 数据类型

① 基本数据类型：描述一些具体的数字单元，不牵扯内存分配问题。

- 1) 数值型：
  - a) 整型：byte、short、int、long，默认值 0
  - b) 浮点型：float、double，默认值 0.0
- 2) 布尔型：boolean，默认值 false
- 3) 字符型：char，默认值 '\u0000'

不同类型所占大小不一样，保存数据范围也不同。

使用参考：

- 1) 通常，整数使用 int，小数使用 double；
- 2) 数据传输或文字编码转换使用 byte 类型(二进制操作)；
- 3) 处理中文使用 char 较方便；
- 4) 描述时间、内存或文件大小、描述表的主键列(自动增长)可以使用 long。

② 引用数据类型：牵扯到内存关系的使用。

数组、类、接口，默认值 null

### ✧ 4.2 整型

Java 任何一个整型常量都是默认 int 类型。

如果变量在处理过程中超过了最大的保存范围，就会数据溢出。

```
int MAX = Integer.MAX_VALUE; // int 最大值:  $2^{31}-1 = 2147483647$ 
int MIN = Integer.MIN_VALUE; // int 最小值:  $-2^{31} = -2147483648$ 
System.out.println("MAX + 1 = " + (MAX + 1)); // MAX + 1 = -2147483648
System.out.println("MIN - 1 = " + (MIN - 1)); // MIN - 1 = 2147483647
```

$2^{31}-1$  二进制为 0111,1111,1111,1111,1111,1111,1111,1111，加 1 后变为 1000,0000,0000,0000,0000,0000,0000,0000，因为最高位是符号位，所以对应十进制是其取反后的十进制+1 再加上负号，即  $-2^{31}$ 。

解决数据溢出就要使用范围更大的数据类型，比如 long：

```
// 可以(long)1、或 1L/1l 转换成 long 类型
System.out.println("MAX + 1 = " + ((long) MAX + 1)); // MAX + 1 = 2147483648
System.out.println("MIN - 1 = " + (MIN - 1L)); // MIN - 1 = -2147483649
```

运算符两边其中一个是 long，另一个是 int，int 会自动转换成 long，因为 long 的范围比 int 大，也就是数据范围小的类型会自动转换为范围大的类型。

long 类型赋值给 int 需要强制类型转换：

```
long a = -2147483649L;
// 范围大的转为范围小的类型需要强制类型转换
int b = (int) a;
System.out.println(b); // 2147483647
```

但会出现数据溢出。不是必须的话，不建议使用强制类型转换。

Java 对 byte 做了特殊处理，如果没超过范围，自动将 int 常量转为 byte；超过范围必须强制类型转换；对于变量也需要强制类型转换。

```
byte n = 20; // 20 在 byte 范围,自动转为 byte
System.out.println(n); // 20
byte m = (byte) 200; // 超过 byte 范围,强制类型转换
System.out.println(m); // -56
int a = 12;
byte s = (byte) a; // 变量,强制类型转换
System.out.println(s); // 12
```

200 转为 byte 为 1100,1000，最高位是符号位，剩余位取反为 110111 对应 55，加 1 为 56，符号位是 1 表示负数，结果是-56。

### ✧ 4.3 浮点型

#### ① double

```
double a = 12; // 自动类型转换
System.out.println(a); // 12.0
```

自动类型转换都是由范围小的类型向范围大的类型转换。

#### ② float

```
// 默认小数类型为 double,需要强制类型转换为 float
float a = 10.1F;
float b = (float) 10.2;
// 浮点数计算存在精度损失的问题,目前仍未解决
System.out.println(a * b); // 103.020004
```

注意：两个 int 相除得到 int，想要得到 double 需要将其中一个数转为 double：

```
int a = 18;
int b = 4;
// 两个 int 相除,地板除得到 int
System.out.println(a / b); // 4
// 想得到真实结果需要其中一个转换为 double
System.out.println((double) a / b); // 4.5
```

### ✧ 4.4 字符型

Java 的 char 使用 Unicode 编码，占 2 个字节，包括世界上常用的文字。

```
public class Hello {
    public static void main(String[] args) {
        char c='星';
        int n =c;
        System.out.println(n); // 26143
    }
}
```

由于 Win7 默认 GBK 编码，javac 使用系统默认编码，而此处 java 使用 UTF-8 编码，编译就会报错：编码 GBK 的不可映射字符

在编译需要指定编码为 UTF-8:

```
$ javac -encoding utf-8 Hello.java
$ java Hello
26143
```

#### ✧ 4.4 字符串

任何项目都会用到 **String**，是引用数据类型，一个特殊的类，可以像普通变量采用直接赋值。

可以使用+进行字符串拼接:

```
String s = "result:";
int a = 10;
int b = 24;
// 有 String, 所有类型无条件先变为 String
System.out.println(s + a + b); // result:1024
System.out.println(s + (a + b)); // result:34
```

### 5 运算符

运算符有优先级，但不用记，需要时优先计算直接使用()。

不要编写很复杂的计算，如:  $a-- + b++ * --b / a / b * ++a - --b + b++$

一般大学老师或奇葩的面试官才会出这种题。

#### ✧ 5.1 数学运算符

+、-、\*、/、%

简化运算符: +=、-=、\*=、/=、%=

自增自减: ++、--

```
int a = 23;
int b = 10;
// a++ 先使用变量后自增; --b 先自减后使用变量
System.out.println(a++ - --b); // 14
System.out.println("a=" + a + ", b=" + b); // a=24, b=9
```

这些代码是以前内存不大时提供的处理方式，但在硬件成本降低的现在，这种计算很繁琐。一般自增自减独立一行写较为直观。

#### ✧ 5.2 关系运算符

>、<、>=、<=、==，关系运算符返回布尔类型

#### ✧ 5.3 三目运算符

简化 if-else 语句，避免无谓的赋值运算处理。

```
int a = 10;
int b = 20;
int max = a > b ? a : b;
System.out.println(max); // 20
int c = 15;
int min = a < b ? (a < c ? a : c) : (b < c ? b : c);
```

```
System.out.println(min); // 10
```

三目运算可以嵌套，但可读性变差，根据实际情况使用。

#### ✧ 5.4 逻辑运算符

&、&&、|、||、!

&& / || 只要前面有一个为 false / true，直接返回 false / true，后面表达式不会判断。应该一直使用短路与和短路或。

#### ✧ 5.5 位运算符

&、|、^、~、>>、<<

```
System.out.println(12 & 7); // 4
System.out.println(12 | 7); // 15
System.out.println(3 << 4); // 48
```

12 对应二进制 1100，7 对应二进制 0111。(前面 28 个 0 省略)

按位与得到 0100 即 4；按位或得 1111 即 15。

$a \ll n$ : a 左移 n 位，相当于  $a \times 2^n$ ，比如  $3 \ll 4$  就是  $3 \times 2^4 = 48$

面试题：&和&&的区别

- ① &和&&都可以作为逻辑运算符：&是普通与，所有条件都要判断；&&是短路与，只要前面判断为 false，后面不再进行判断，直接返回 false。
- ② &可以作为位运算符，&&不可以。

## 6 程序逻辑结构

三种程序逻辑结构：顺序结构、分支结构、循环结构。

### ✧ 6.1 分支结构

① if-else

② switch-case

switch 最早只能 int 或 char 判断，JDK 1.5 支持枚举判断，JDK1.7 支持 String 判断；不支持布尔类型。

```
String s = "sun";
switch (s) {
    case "mon":
    case "tue":
    case "wed":
    case "thu":
    case "fri": {
        System.out.println("工作!");
        break;
    }
    case "sat":
    case "sun": {
        System.out.println("休息!");
        break;
    }
}
```



```

    }
    default: {
        System.out.println("输入错误!");
    }
}

```

## ✧ 6.2 循环结构

### ① while / do while 循环

do while 不管条件满不满足，循环都要至少执行一次；几乎不用 do while。

### ② for 循环

知道循环次数优先 for 循环；

不知道循环次数但知道循环结束条件时，使用 while 循环。

### ③ 循环控制

break 和 continue

### ④ 循环嵌套

练习：打印三角形

```

int line = 5; // 行数
for (int x = 0; x < line; x++) { // 打印几行
    for (int i = 0; i < line - x; i++) { // 打印空格
        System.out.print(" ");
    }
    for (int i = 0; i <= x; i++) { // 打印*
        System.out.print("* ");
    }
    System.out.println();
}

```

结果：

```

    *
   * *
  * * *
 * * * *
* * * * *

```

这只是程序逻辑的训练，与程序开发关系不大，主要面向应届生。

## 7 方法

程序中可能需要重复执行某段代码，可以将其封装成方法(method)，有的编程语言也叫函数(function)。

定义方法有利于重复调用，所有程序都是通过主方法开始执行的。

### ✧ 7.1 方法的重载

方法名称相同，参数的个数或类型不同时称为方法的重载。

```

public static void main(String[] args) {
    int a = sum(12, 34);
    int b = sum(12, 34, 56);
    double c = sum(1.2, 5.6);
    System.out.println("a=" + a + ", b=" + b + ", c=" + c); // a=46, b=102, c=6.8
}

public static int sum(int a, int b) {
    return a + b;
}

public static int sum(int a, int b, int c) {
    return a + b + c;
}

public static double sum(double a, double b) {
    return a + b;
}

```

方法重载与返回值类型没有关系，只和参数有关系。  
 实际开发建议方法重载返回值类型相同。  
 可以发现 `System.out.println()` 是系统提供的方法重载。

## ✧ 7.2 递归 (recursion)

需要设置递归结束的条件；每次调用时要修改传递的参数条件。

```

public static void main(String[] args) {
    System.out.println(fib(20)); // 6765
}

public static int fib(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    return fib(n - 1) + fib(n - 2);
}

```

实际开发编写的代码很少出现递归，常用的递归是系统内部提供的。  
 递归处理不当，容易造成栈内存溢出。  
 一般面试题常用递归，如汉诺塔、八皇后、二叉树遍历等问题。

# 8 类与对象

## ✧ 8.1 面向对象

Java 最大特点就是面向对象编程。也有开发者认为面向过程或函数式编程好。  
 C 语言是面向过程开发的代表，面向过程是面向一个问题的解决方案，更多情况不会考虑重复利用。

面向对象主要是设计形式的模块化设计，可以重用配置，面向对象更多考虑的是标准，使用时根据标准拼装。

面向对象的 3 个主要特征：

- ① 封装：内部的操作对外部不可见，安全；
- ② 继承：在已有结构的基础上进行功能扩充；
- ③ 多态：在继承的基础上扩充的概念，类型的转换处理

面向对象开发 3 步骤：

- ① OOA：面向对象分析；
- ② OOD：面向对象设计；
- ③ OOP：面向对象编程。

## ✧ 8.2 类与对象

类是某一类事物共性的抽象概念；对象是一个具体的产物。  
类是一个模板，对象是类创建的实例，先有类后有对象。

类一般有两个组成：

- ① 属性 (Field)：定义对象的属性
- ② 方法 (Method)：定义对象的行为

示例：

```
class Person {
    String name;
    int age;
    public void show() {
        System.out.println("我的名字是" + name + ", 今年" + age + "岁了!");
    }
}

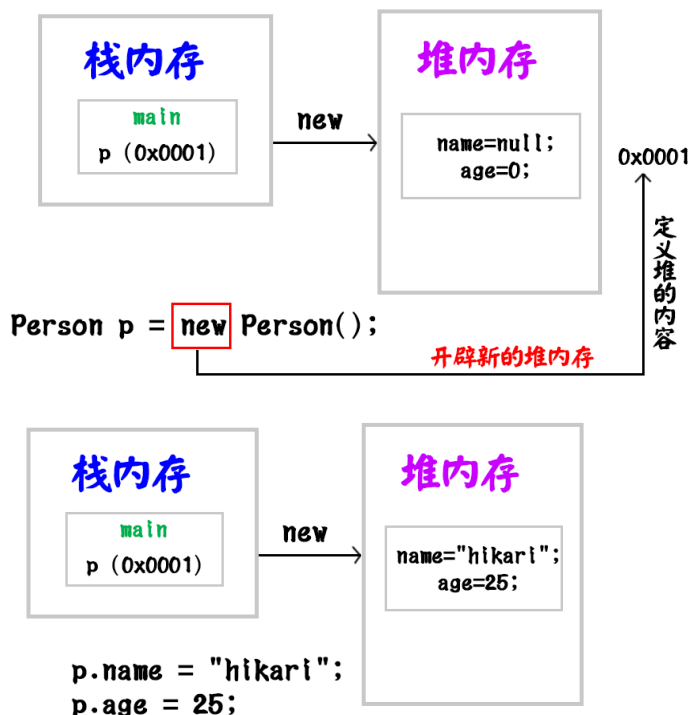
public class Hello {
    public static void main(String[] args) {
        Person p = new Person();
        // 未设置属性,使用默认值
        p.show(); // 我的名字是 null, 今年 0 岁了!
        p.name = "hikari";
        p.age = 25;
        p.show(); // 我的名字是 hikari, 今年 25 岁了!
    }
}
```

类是引用数据类型，其最大困难之处在于内存的管理，在操作时也会发生内存关系的变化。

最常用的内存空间：

- ① **堆内存**：保存对象的具体信息；堆内存空间的开辟通过 **new** 完成；
- ② **栈内存**：保存一块堆内存的地址，通过地址找到堆内存，找到对象内容

简单内存分析：



对象实例化语句可以是：

- ① 声明并实例化对象： `Person p = new Person();`
- ② 先声明对象，再实例化对象：

```
Person p = null;  
p = new Person();
```

对象调用类的属性或方法必须实例化完成后才能执行。

如果只声明对象，而没有实例化，调用会出错：

```
Exception in thread "main" java.lang.NullPointerException
```

抛出**空指针异常**，就是堆内存没有开辟时产生的问题，只有引用数据类型存在空指针异常。

### ✧ 8.3 引用传递

内存的**引用传递**：同一块堆内存空间可以被不同的栈内存所指向，也可以更换指向。一个栈内存只能保存一个堆内存地址数据。

```
Person p = new Person();  
p.name = "hikari";  
p.age = 25;  
p.show(); // 我的名字是 hikari, 今年 25 岁了!  
Person p1 = p; // 引用传递
```

```
p1.age = 20;
p.show(); // 我的名字是 hikari, 今年 20 岁了!
```

p 和 p1 指向同一个堆内存空间，通过 p1 修改对象属性，p 也会改变。

也可以通过方法实现引用传递：

```
public static void main(String[] args) {
    Person p = new Person();
    p.name = "hikari";
    p.age = 25;
    p.show(); // 我的名字是 hikari, 今年 25 岁了!
    change(p); // 通过方法实现引用传递
    p.show(); // 我的名字是 hikari, 今年 10 岁了!
}

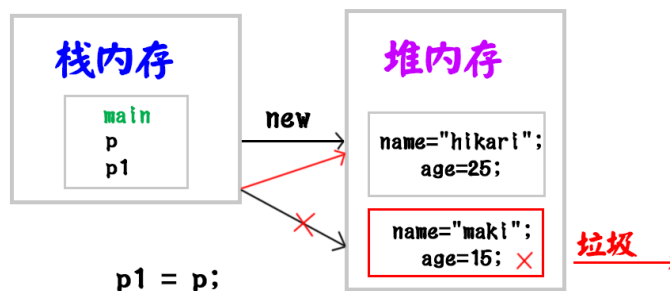
public static void change(Person a) {
    a.age = 10;
}
```

将 Person 的实例化对象 p (本质存的是内存地址)传递到 change()方法，a 也指向与 p 同一块堆内存空间。方法执行完毕，a 断开连接。

没有任何栈内存指向的堆内存空间就是垃圾空间，所有垃圾将被 GC (Garbage Collection)不定期进行回收，释放无用内存空间；如果垃圾过多，将影响 GC 的性能，从而降低整体程序性能。

```
Person p = new Person();
Person p1 = new Person();
p.name = "hikari";
p.age = 25;
p1.name = "maki";
p1.age = 15;
p1 = p; // 引用传递
p1.name = "maki";
p.show(); // 我的名字是 maki, 今年 25 岁了!
```

一开始 p 和 p1 指向各自的堆内存，后来 p1 指向 p 指向的堆内存；原来 p1 指向的堆内存没有任何栈内存指向，因而成为垃圾被回收。



## 20180528

### ✧ 8.4 封装性

以上代码对象可以在类的外部直接访问并修改属性 `name` 和 `age`。

但类似 `p.age = -10;` 的赋值虽然没有语法错误，但是存在业务逻辑错误，因为年龄不可能是负数。

一般而言，方法是对外提供服务，不会封装处理；而属性需要较高的安全性，此时需要封装性对属性进行保护。

将类的属性设置为对外不可见，可以使用 `private` 关键字定义属性：

```
private String name;  
private int age;
```

编译报错：`name / age 在 Person 中是 private 访问控制`

现在外部的对象无法直接调用类的属性了，也就是属性对外部不可见。

但是为了程序能正常使用，外部程序应该能间接操作类的属性，所以开发中对于属性一般要求：

- 1) 所有类中定义的属性都用 `private` 声明；
- 2) 如果属性要被外部使用，按要求定义相应的 `setter` 和 `getter` 方法

```
class Person {  
    private String name;  
    private int age;  
  
    public void show() {  
        System.out.println("我的名字是" + name + ", 今年" + age + "岁了!");  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
  
    public void setAge(int age) {  
        // 增加验证年龄，小于 0 使用默认值 0  
        if (age > 0) {  
            this.age = age;  
        }  
    }  
    public int getAge() {  
        return age;  
    }  
}
```

```

}

public class Hello {
    public static void main(String[] args) {
        Person p = new Person();
        p.setName("hikari");
        p.setAge(-10);
        p.show(); // 我的名字是 hikari, 今年 0 岁了!
    }
}

```

在开发中数据验证一般是其他辅助代码完成，而 `setter` 往往是简单的设置数据，`getter` 是简单的获取数据。

封装就是保证类内部定义在外部不可见，属性封装只是面向对象中封装最小的概念，还跟访问权限有关。

## ✧ 8.5 构造方法

如果类的属性有  $n$  个，按照上面方法需要调用  $n$  次 `setter` 方法设置属性，十分麻烦。

Java 提供[构造方法](#)实现实例化对象的[属性初始化](#)。构造方法定义要求：

- 1) 构造方法名称必须与类名称一致；
- 2) 构造方法不允许有返回值定义；
- 3) 构造方法在 `new` 实例化对象时自动调用。

之前代码可以改为：

```

class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public void show() {
        System.out.println("我的名字是" + name + ", 今年" + age + "岁了!");
    }
}

public class Hello {
    public static void main(String[] args) {
        Person p = new Person("hikari", 25);
        p.show(); // 我的名字是 hikari, 今年 25 岁了!
    }
}

```

分析 `Person p = new Person("hikari", 25);`

- ① `Person`: 定义对象的类型, 决定可以调用的方法;
- ② `p`: 实例化对象的名称, 所有操作通过对象来访问;
- ③ `new`: 开辟一块新的堆内存空间;
- ④ `Person("hikari", 25)`: 调用有参数的构造方法;

所有类都有构造方法。如果没有定义构造方法, 默认提供一个无参数、什么都不做的构造方法, 这个构造方法在编译时自动创建; 反之则不会自动创建。所以一个类至少存在一个构造方法。

既然构造方法不返回数据, 为什么不能使用 `void` 呢?

因为编译器根据代码结构进行编译, 执行时也根据代码结构处理。如果构造方法使用 `void`, 结构就和普通方法相同, 编译器就会认为是普通方法。

构造方法也具有重载的特点。多个构造方法定义时建议按一定顺序排列, 如按照参数个数升序/降序排列。

#### ✧ 8.6 匿名对象

定义对象时如 `Person p = new Person("hikari", 25);`

等号是右边实例化对象, 其也可以直接使用实例方法:

```
new Person("hikari", 25).show();
```

这种形式调用的对象由于没有名字, 称为匿名对象。

由于匿名对象没有任何引用, 使用一次后就变为垃圾, 被 GC 回收释放。

#### ✧ 8.7 this 关键字

**this**: 表示当前实例化对象

程序开发中, 只要访问本类属性时, 建议一定加上 `this`。

类似于 Python 的 `self`, 而且 Python 强制写 `self`。

除了属性, `this` 还可以实现方法的调用:

- 1) 构造方法: `this()`
- 2) 普通方法: `this.func()`

对于不同的构造方法, 需要执行相同的一段代码:

```
public Person() {
    System.out.println("***代表很长一段代码***");
    this.name = "匿名";
}

public Person(String name) {
    System.out.println("***代表很长一段代码***");
    this.name = name;
}
```



```
public Person(String name, int age) {
    System.out.println("***代表很长一段代码***");
    this.name = name;
    this.age = age;
}
```

在每个构造函数复制相同代码是不好的习惯。

评价代码的好坏：

- 1) 代码结构可以重复利用，提供一个中间独立的支持；
- 2) 尽量少的重复代码。

此时可以使用 this()调用构造方法简化代码：

```
public Person() {
    this("匿名", 0); // 调用本类双参构造方法
}

public Person(String name) {
    this(name, 0); // 调用本类参双参构造方法
}

public Person(String name, int age) {
    System.out.println("***代表很长一段代码***");
    this.name = name;
    this.age = age;
}
```

注意：

- 1) 构造方法必须在实例化对象时调用，this()语句必须放在构造方法的首行；
- 2) 构造方法可以调用普通方法，普通方法不能调用构造方法；
- 3) 构造方法互相调用需要保留程序出口，防止无限递归调用。

## 20180529

### ✧ 8.8 static 关键字

主要定义属性和方法。

#### ① static 定义属性

一般一个对象保存各自的属性，比如 Person 类新添加一个 country 属性，但大多数人都是 "China"，每个对象都保存一份有点浪费。而且如果此时想修改为 "中国"，对象已经实例化几千万个，那修改起来将是一场噩梦。

```
class Person {
    private String name;
    private int age;
    String country = "China";
    // ...
}
```

```

}

public class Hello {
    public static void main(String[] args) {
        Person p1 = new Person("张三", 20);
        Person p2 = new Person("李四", 23);
        Person p3 = new Person("王五", 25);
        p1.country = "中国";
        p1.show(); // name: 张三, age: 20, country: 中国
        p2.show(); // name: 李四, age: 23, country: China
        p3.show(); // name: 王五, age: 25, country: China
    }
}

```

因为每个对象各保存一份，修改 p1 的 country，p2 和 p3 没有受到影响。

static 修饰的属性是公共属性：

```
static String country = "China";
```

此时 p1、p2、p3 的 country 属性都变为“中国”。

static 属性存储在[全局数据区](#)，不是每个对象各自拥有，通过 p1 修改 static 属性，p2、p3 相应 static 属性都改变。

static 属性可以通过对象访问，但因为是公共属性，最好直接使用类名调用。

```
Person.country = "中国";
```

static 属性虽然定义在类中，但不受实例化对象控制，也就是 static 属性可以在没有实例化对象的时候使用。

类设计时，首选非 static 属性，如果要存储公共信息才会使用 static 属性。

## ② static 定义方法

static 定义的方法可以直接由类名在没有实例化对象时调用。

```

class Person {
    // ...
    private static String country = "China";
    public static String getCountry() {
        return country;
    }
    public static void setCountry(String country) {
        Person.country = country;
    }
    // ...
}

public class Hello {

```

```

public static void main(String[] args) {
    // ...
    Person.setCountry("中国");
    p1.show(); // name: 张三, age: 20, country: 中国
    p2.show(); // name: 李四, age: 23, country: 中国
    p3.show(); // name: 王五, age: 25, country: 中国
}
}

```

static 和非 static 方法:

- 1) static 方法: 只能调用 static 属性和 static 方法;
- 2) 非 static 方法: 无限制;
- 3) static 定义的属性和方法可以在没有实例化对象时使用; 非 static 则不行。

static 属性和方法在编写代码之初并不是需要考虑的, 只有在回避实例化对象调用并且描述公共属性时才考虑 static。

## ✧ 8.9 代码块

使用{}定义的结构就是代码块。根据代码块位置和定义关键字不同分为:

- 1) 普通代码块: {}内部定义的变量, 生命周期只在{}内部, 出了{}就消失; 可以在方法之中进行结构拆分, 防止相同变量名带来的影响。
- 2) 构造代码块: 构造块优先于构造方法执行, 每次序列化对象都会执行;
- 3) 静态代码块: 类初始化时对类的静态属性初始化, 只执行一次;
- 4) 同步代码块: 用于多线程

```

class Message {
    public static String getMsg() {
        // 此处数据可能从网络或数据库获取
        return "人生苦短, 我用Python!";
    }
}

class Person {
    private String name;
    private int age;
    private static String country = "China";
    private static String message;
    static { // 静态代码块可以多行
        System.out.println("静态代码块");
        message = Message.getMsg();
    }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

```

```

        System.out.println("构造方法");
    }
    {
        System.out.println("构造代码块");
    }
    // ...
}

public class Hello {
    public static void main(String[] args) {
        Person p1 = new Person("张三", 20);
        Person p2 = new Person("李四", 23);
        Person p3 = new Person("王五", 25);
    }
}

```

打印结果:

```

静态代码块
构造代码块
构造方法
构造代码块
构造方法
构造代码块
构造方法

```

所有执行顺序是：静态代码块-->构造代码块-->构造方法  
而且静态代码块只执行一次；而构造代码块和构造方法每次实例化都执行。

练习：设计一个用户类 User，属性有用户名、密码和记录用户的数量。定义三个构造方法(无参、用户名为参数、用户名和密码为参数)。

```

class User {
    private String username;
    private String password;
    private static int cnt = 0;
    public User() {
        this("匿名", "abc");
    }
    public User(String username) {
        this(username, "abc");
    }
    public User(String username, String password) {
        this.username = username;
        this.password = password;
        cnt++;
    }
    public static int getCnt() {

```

```

        return cnt;
    }
    public String getInfo() {
        return "用户名: " + this.username + ", 密码: " + this.password;
    }
    public void show() {
        System.out.println(this.getInfo());
    }
}

User a = new User();
User b = new User("maki");
User c = new User("rin", "kayochin");
a.show(); // 用户名: 匿名, 密码: abc
b.show(); // 用户名: maki, 密码: abc
c.show(); // 用户名: rin, 密码: kayochin
System.out.println("用户个数: " + User.getCnt()); // 用户个数: 3

```

## 9 数组

### ✧ 9.1 数组初始化

#### ① 数组动态初始化

Java 数组是引用数据类型，牵扯到内存分配，可以通过 new 创建。

```

int arr[] = new int[3];
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]); // int默认0
}

```

动态初始化每个元素的值都是对应数据类型的默认值。

#### ② 数组静态初始化

```

int arr[] = new int[] { 12, 34, 56, 78 }; // 完整格式
int arr[] = { 12, 34, 56, 78 }; // 省略格式, 不建议使用

```

int arr[]和 int[] arr 都可以。

### ✧ 9.2 for-each 循环遍历数组

JDK 1.5 为了减轻下标对程序的影响(数组下标越界)，参考了.NET 的设计引入了增强型的 for 循环——for-each 循环。

```

int arr[] = new int[] { 12, 34, 56, 78 }; // 完整格式
for (int i : arr) {
    System.out.println(i);
}

```

练习：求 int 数组总和、平均值、最大值、最小值

主方法所在类为主类，不希望涉及过于复杂代码。开发过程中，主方法相当于客户端，代码应该尽量简洁，所以可以定义一个工具类封装具体的操作过程；

而主类只关心如何操作。

```
class ArrayUtil {
    private int sum;
    private double average;
    private int max;
    private int min;
    public ArrayUtil(int[] arr) {
        if (arr == null || arr.length == 0) {
            return;
        }
        this.max = arr[0];
        this.min = arr[0];
        for (int i : arr) {
            this.sum += i;
            if (i > this.max) {
                this.max = i;
            }
            if (i < this.min) {
                this.min = i;
            }
        }
        this.average = this.sum / (double) arr.length;
    }

    public int getSum() {
        return this.sum;
    }
    public double getAverage() {
        return this.average;
    }
    public int getMax() {
        return this.max;
    }
    public int getMin() {
        return this.min;
    }
}

int arr[] = new int[] { 14, 22, 46, 75, 56, 63 }; // 完整格式
ArrayUtil util = new ArrayUtil(arr);
System.out.println("sum: " + util.getSum()); // sum: 276
System.out.println("average: " + util.getAverage()); // average: 46.0
System.out.println("max: " + util.getMax()); // max: 75
System.out.println("min: " + util.getMin()); // min: 14
```

其他练习如数组排序、反转等自己练练就就行了。实际开发可以使用内置方法就行，然而面试的时候却还是要会。

### ✧ 9.3 可变参数

JDK 1.5 方法支持可变参数，本质还是数组。

```
public static void main(String[] args) {
    System.out.println(sum(3, 4, 5)); // 12
    System.out.println(sum(1, 2)); // 3
    System.out.println(sum(new int[] { 1, 2, 3, 4, 5 })); // 15
}

private static int sum(int... args) { // 可变参数
    int s = 0;
    for (int i : args) {
        s += i;
    }
    return s;
}
```

### ✧ 9.4 对象数组

数组元素的类型还可以是自定义类的实例化对象：

```
Person[] persons = new Person[] { new Person(), new Person("hikari", 25), new
Person("maki", 15) };
for (Person p : persons) {
    p.show();
}
```

对象数组就是将对象放到数组统一管理。开发离不开对象数组，但数组最大缺点是长度固定，优点是数据线性保存，索引访问，速度快。实际开发不会直接开辟数组，内容通常是通过传递的数据动态生成，但会使用数组的概念。

## 10 数据表和简单 Java 类映射

### ✧ 10.1 引用传递的应用

引用传递是整个 Java 开发与设计过程中最重要的技术之一。

#### ① 类关联结构

例：使用面向对象设计 Person 类和 Car 类的关系。

假如 Person 只有 name 和 age 属性；Car 只有 name 和 price 属性，这两个是相互独立的类。所以 Person 类需要有 car 属性表明某个人拥有的某辆车，Car 类需要一个 person 属性，表明这辆车属于谁的。

```
class Car {
    private String name;
    private int price;
    private Person person;
```

```

    public Car(String name, int price) {
        this.name = name;
        this.price = price;
    }
    public Person getPerson() {
        return person;
    }
    public void setPerson(Person person) {
        this.person = person;
    }
    public String getInfo() {
        return "汽车品牌: " + this.name + ", 汽车价格: " + this.price;
    }
    public void show() {
        System.out.println(this.getInfo());
    }
}

class Person {
    private String name;
    private int age;
    private Car car;
    public Person() {
        this("匿名", 0);
    }
    public Person(String name) {
        this(name, 0);
    }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getInfo() {
        return "我的名字是" + this.name + ", 今年" + this.age + "岁了!";
    }
    public void show() {
        System.out.println(this.getInfo());
    }
    public Car getCar() {
        return car;
    }
    public void setCar(Car car) {
        this.car = car;
    }
}

```



```

}

Person p = new Person("hikari", 25);
Car c = new Car("法拉利", 3000000);
// 设置关系
p.setCar(c); // 一个人有一辆车
c.setPerson(p); // 一辆车属于一个人
p.getCar().show(); // 汽车品牌: 法拉利, 汽车价格: 3000000
c.getPerson().show(); // 我的名字是hikari, 今年25岁了!

```

## ② 自关联

一个人可以有孩子, 孩子也可以有车, 设计此时的关系。

此时 Car 类不变, 只需给 Person 类添加一个 child 的属性, 类似于链表。

如果有多个孩子, 需要使用对象数组。

```

class Person {
    private String name;
    private int age;
    private Car car;
    private Person[] children; // 对象数组
    // ...
    public void setChildren(Person[] children) {
        this.children = children;
    }
    public Person[] getChildren() {
        return this.children;
    }
}

// ...
Person p1 = new Person("maki", 15);
Person p2 = new Person("rin", 15);
p1.setCar(new Car("BMW X1", 300000));
p2.setCar(new Car("TOYOTA", 200000));
p.setChildren(new Person[] { p1, p2 });
// 根据人找到所有的孩子和孩子对应的汽车
for (Person i : p.getChildren()) {
    System.out.println("\t|- " + i.getInfo());
    System.out.println("\t\t|- " + i.getCar().getInfo());
}

```

结果:

```

汽车品牌: 法拉利, 汽车价格: 3000000
我的名字是hikari, 今年25岁了!
|- 我的名字是maki, 今年15岁了!

```

```
| - 汽车品牌: BMW X1, 汽车价格: 300000  
| - 我的名字是rin, 今年15岁了!  
| - 汽车品牌: TOYOTA, 汽车价格: 200000
```

这些关系的匹配都是通过引用数据类型的关联完成。

## ✧ 10.2 数据表和简单 Java 类映射

简单 Java 类是往往根据数据表的结果来实现。

数据表和简单 Java 类之间基本映射关系:

- 1) 数据表设计——类的定义
- 2) 表中字段——类的成员属性
- 3) 表的外键——引用关联
- 4) 表的一行记录——类的一个实例化对象
- 5) 表的多行记录——对象数组

先抛开关联字段写出类的基本组成, 再通过引用配置关联字段的关系。

## 20180530

上面人和车的关系就是一对一; 人和孩子的关系就是一对多, 而且是自关联。

示例:

用户 User 类、角色 Role 类、权限 Permission 类: 一个用户对应一个角色, 一个角色对应多个用户; 一个角色对应多个权限, 一个权限对应多个角色。

前者是一对多关系, 后者是多对多关系。

```
class User {  
    private String id;  
    private String username;  
    private Role role; // 一个用户对应一个角色  
    public User(String id, String username) {  
        this.id = id;  
        this.username = username;  
    }  
    public Role getRole() {  
        return role;  
    }  
    public void setRole(Role role) {  
        this.role = role;  
    }  
  
    public String toString() {  
        return "用户编号: " + this.id + ", 用户名: " + this.username;  
    }  
}
```

```

class Role {
    private static int cnt = 0;
    private int id;
    private String name;
    private User[] users; // 一个角色对应多个用户
    private Permission[] permissions; // 一个角色有多个权限
    public Role(String name) {
        this.id = ++cnt; // 编号自动累加
        this.name = name;
    }
    public User[] getUsers() {
        return users;
    }
    public void setUsers(User[] users) {
        this.users = users;
    }
    public Permission[] getPermissions() {
        return permissions;
    }
    public void setPermissions(Permission[] permissions) {
        this.permissions = permissions;
    }
    public String toString() {
        return "角色编号: " + this.id + ", 角色名: " + this.name;
    }
}

class Permission {
    private static int cnt = 0;
    private int id;
    private String name;
    private Role[] roles; // 一个权限对应多个角色
    public Permission(String name) {
        this.id = ++cnt; // 编号自动累加
        this.name = name;
    }
    public Role[] getRoles() {
        return roles;
    }
    public void setRoles(Role[] roles) {
        this.roles = roles;
    }
    public String toString() {
        return "权限编号: " + this.id + ", 权限名: " + this.name;
    }
}

```

```

    }
}

```

setter 和 getter 方法自己写太麻烦了，直接 eclipse 自动生成...

设置关系：

```

User hikari = new User("01", "hikari");
User maki = new User("muse-06", "西木野真姬");
User rin = new User("muse-05", "星空凛");
Role adminRole = new Role("管理员");
Role userRole = new Role("普通用户");
Permission p1 = new Permission("修改用户");
Permission p2 = new Permission("撰写文章");
Permission p3 = new Permission("浏览文章");
// 用户和角色关系设置
hikari.setRole(adminRole);
maki.setRole(userRole);
rin.setRole(userRole);
adminRole.setUsers(new User[] { hikari, });
userRole.setUsers(new User[] { maki, rin });
// 角色和权限关系设置
adminRole.setPermissions(new Permission[] { p1, p2, p3 });
userRole.setPermissions(new Permission[] { p3 });
p1.setRoles(new Role[] { adminRole, });
p2.setRoles(new Role[] { adminRole, });
p3.setRoles(new Role[] { adminRole, userRole });

```

测试：

```

System.out.println("*****根据用户查找角色和权限*****");
System.out.println(hikari.toString());
System.out.println("\t|- " + hikari.getRole().toString());
for (Permission p : hikari.getRole().getPermissions()) {
    System.out.println("\t\t|- " + p.toString());
}
System.out.println("*****根据角色查询用户*****");
System.out.println(userRole.toString());
for (User u : userRole.getUsers()) {
    System.out.println("\t|- " + u.toString());
}
System.out.println("*****根据权限查询用户*****");
System.out.println(p3.toString());
for (Role r : p3.getRoles()) {
    for (User u : r.getUsers()) {
        System.out.println("\t|- " + u.toString());
    }
}

```

结果:

```
*****根据用户查找角色和权限*****
用户编号: 01, 用户名: hikari
|- 角色编号: 1, 角色名: 管理员
   |- 权限编号: 1, 权限名: 修改用户
   |- 权限编号: 2, 权限名: 撰写文章
   |- 权限编号: 3, 权限名: 浏览文章
*****根据角色查询用户*****
角色编号: 2, 角色名: 普通用户
|- 用户编号: muse-06, 用户名: 西木野真姬
|- 用户编号: muse-05, 用户名: 星空凛
*****根据权限查询用户*****
权限编号: 3, 权限名: 浏览文章
|- 用户编号: 01, 用户名: hikari
|- 用户编号: muse-06, 用户名: 西木野真姬
|- 用户编号: muse-05, 用户名: 星空凛
```

## 11 字符串

字符串不是基本数据类型, Java 为了方便开发者编写, 利用 JVM 制造了一种可以简单使用的 `String` 类, 可以像基本数据类型直接赋值处理。

`String` 类内部定义了一个数组, 所有字符数据保存在数组里。

JDK 1.8 以前 `String` 保存的是 `char` 数组; JDK 1.9 之后是 `byte` 数组。

字符串是对数组的一种特殊包装应用, 字符串的内容是无法改变的。

### ✧ 11.1 字符串比较

`String` 比较 `==` 和 `equals()` 区别?

- 1) `==`: 对象比较的是两个内存地址数值;
- 2) `equals()`: `String` 类提供的比较方法, 可以直接进行字符串内容的比较。

如:

```
String a = "hikari"; // 直接赋值
String b = new String("hikari"); // 构造方法实例化
System.out.println(a == b); // false
System.out.println(a.equals(b)); // true
```

因为 `new` 会开辟内存空间, 而 `==` 比较的是对象的地址, 所以不同对象的 `==` 比较肯定是 `false`; 而 `String` 类的 `equals()` 覆写了 `Object` 类的 `equals()` 比较的是字符串内容。所以, 字符串比较通常使用 `equals()`。

### ✧ 11.2 字符串常量

字符串常量是 `String` 类的匿名对象。`String` 类对象的直接赋值是将一个 `String` 类匿名对象设置一个具体的引用名字。

比较字符串变量和字符串常量内容是否相等, 建议将常量写前面:

```
String a = new String("hikari");
String b = null;
System.out.println("hikari".equals(a)); // true
System.out.println("hikari".equals(b)); // false
```

因为如果调用变量的 `equals()`，可能变量为 `null`，此时会抛出空指针异常。而字符串常量是匿名对象，已经开辟了内存空间，不可能为 `null`。而且比较对象如果是 `null`，直接返回 `false`。

直接赋值和构造方法实例化比较：

① 直接赋值：只会产生一个对象，可以自动保存到字符串池，如果有相同的数据定义时，可以减少对象的产生，实现重用，提高性能。

```
String a = "你好 再见";
String b = "你好 再见";
System.out.println(a == b); // true
```

说明 `a` 和 `b` 指向同一块堆内存空间。

② 构造方法实例化：产生两个对象，不会自动入池，无法实现重用

```
String a = "你好 再见";
String c = new String("你好 再见");
System.out.println(a == c); // false
```

构造方法实例化，字符串常量(匿名对象)先开辟一块空间，`new` 再开辟一块空间，变量指向 `new` 创建的对象；如果匿名对象没有被任何栈变量指向，将作为垃圾被回收。

使用 `intern()` 方法可以实现手动入池，但是太麻烦了。

```
String a = "你好 再见";
String c = new String("你好 再见").intern();
System.out.println(a == c); // true
```

通常使用字符串常量。

### ✧ 11.3 String 对象(常量)池

对象池主要目的是实现数据的共享处理。Java 对象池分为：

- 1) 静态常量池：程序(.class)在加载时自动将程序中保存的字符串、普通常量、类、方法等全部进行分配；
- 2) 运行时常量池：程序加载后，里面的变量。

```
String a = "你好 再见";
String b = "你好" + " " + "再见"; // 字符串常量拼接
String p = "你好";
String c = p + " " + "再见"; // 变量拼接
System.out.println(a == b); // true
System.out.println(a == c); // false
```

前者字符串拼接时都是常量数据，得到的字符串存在于静态常量池，也就是 `a` 指向的字符串，所以 `b` 也指向此字符串；后者程序在加载时不确定 `p` 的内容，因为字符串拼接时 `p` 是变量，变量可以修改，得到的 `c` 存储在运行时常量池。

### ✧ 11.4 字符串内容不可修改

`String` 类内部包含一个数组，数组最大缺点是长度不可变，当设置字符串后，

自动数组空间开辟，开辟内容长度是固定的。

这是个不好的例子：

```
String s = "haha";
for (int i = 0; i < 100; i++) {
    s += i;
}
System.out.println(s);
```

字符串常量内容并没有改变，改变的是对象的引用，上面这个不好的例子引用改变了 100 次，会带来大量垃圾空间。

String 类在开发时不要进行内容的频繁修改。

### ✧ 11.5 主方法组成分析

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

- 1) **public**: 访问权限，主方法是程序的开始，开始点一定是公共的；
- 2) **static**: 程序执行通过类名完成，由类直接调用；
- 3) **void**: 起点一旦开始没有返回的可能；
- 4) **main**: 系统定义好的方法名；
- 5) **String[] args**: 字符串数组，可以接收程序的启动参数。

启动参数可以模拟数据输入：

```
for (String s : args) {
    System.out.println(s);
}
```

结果：

```
$ java StringDemo hello hikari "hello world"
hello
hikari
hello world
```

### ✧ 11.6 JavaDoc

开发时肯定需要大量查询 Java 的 API 文档([JavaDoc](#))。

JDK 1.9 之前，所有 Java 常用类库会 JVM 启动时全部加载，性能会下降；JDK 1.9 之后提供模块化设计，将一些程序类放在不同的模块。

在模块中包含大量程序开发包。String 类的相关定义在 java.base 模块的 java.lang 包。

### ✧ 11.7 String 类常用方法

#### ① 字符串与字符数组

1) 将字符数组转为字符串：

```
public String(char[] value)
public String(char[] value, int offset, int count)
```

offset 表示偏移，即字符数组的起点，count 表示字符个数

2) 获取字符串指定索引的字符

```
public char charAt(int index)
```

### 3) 字符串转为字符数组

```
public char[] toCharArray()
```

练习：判断字符串是不是全是由数字组成

```
public static boolean isNum(String s) {  
    // 判断字符串是不是全是数字组成  
    char[] tmp = s.toCharArray(); // 字符串变为字符数组操作  
    for (char c : tmp) {  
        if (c < '0' || c > '9') {  
            return false;  
        }  
    }  
    return true;  
}  
  
String s1 = "123";  
String s2 = "00hikari";  
System.out.println(isNum(s1)); // true  
System.out.println(isNum(s2)); // false
```

### ② 字符串与字节数组转换

```
public String(byte[] bytes)  
public String(byte[] bytes, int offset, int length, String charsetName)  
public byte[] getBytes()  
public byte[] getBytes(String charsetName)
```

charsetName 为编码名字，如果不支持该编码，抛出异常。

### ③ 字符串比较

#### 1) 比较是否相等，前者区分大小写

```
public boolean equals(Object anObject)  
public boolean equalsIgnoreCase(String anotherString)
```

#### 2) 比较大小，前者区分大小写

```
public int compareTo(String anotherString)  
public int compareToIgnoreCase(String str)
```

### ④ 字符串查找

#### 1) 子字符串是否存在(since JDK 1.5):

```
public boolean contains(CharSequence s)
```

#### 2) 查找位置，不存在返回-1:

```
public int indexOf(String str)  
public int indexOf(String str, int fromIndex)
```



```
public int lastIndexOf(String str)
public int lastIndexOf(String str, int fromIndex)
```

```
String s = "hikari";
String p = "ka";
// indexOf()也可以判断子字符串是否存在
System.out.println(s.indexOf(p) >= 0); // true
```

3) 是否以某个子字符串开始/结尾:

```
public boolean startsWith(String prefix)
public boolean startsWith(String prefix, int toffset)
public boolean endsWith(String suffix)
```

⑤ 字符串替换

```
public String replaceAll(String regex, String replacement)
public String replaceFirst(String regex, String replacement)
```

```
String s = "hikari";
System.out.println(s.replaceAll("i", "ou")); // houkarou
System.out.println(s.replaceFirst("i", "1")); // h1kari
```

⑥ 字符串拆分

```
public String[] split(String regex)
public String[] split(String regex, int limit)
```

```
String s = "192.168.1.0";
// .是正则表达式通配符,需要转义成\.; 然而\也需要转义,成了\\.
String[] ret = s.split("\\.");
for (String str : ret) {
    System.out.println(str);
}
```

⑦ 字符串截取

```
public String substring(int beginIndex)
public String substring(int beginIndex, int endIndex)
```

```
String s = "<p><a href=\"url长度不固定\">hikari</a>的博客</p>";
int start = s.indexOf(">", s.indexOf("href")) + 1;
int end = s.indexOf("</a>");
String name = s.substring(start, end);
System.out.println(name); // hikari
```

⑧ 字符串格式化

JDK 1.5 之后, 为了吸引更多传统开发人员, Java 提供了格式化数据操作,

类似于 C 语言格式化输出语句。

```
public static String format(String format, Object... args)
```

```
String name = "hikari";  
int age = 25;  
double salary = 1234.56789;  
String info = String.format("name:%s, age:%d, salary:%.2f", name, age, salary);  
System.out.println(info); // name:hikari, age:25, salary:1234.57
```

## ⑨ 其他方法

### 1) 判断是否为空字符串

```
public boolean isEmpty()
```

### 2) 获取字符串长度

```
public int length()
```

### 3) 去除字符串两边空格, 类似 Python 的 strip()

```
public String trim()
```

### 4) 字符串全部大写/小写

```
public String toUpperCase()  
public String toLowerCase()
```

练习: Java 没有提供字符串首字母大写的方法, 自己实现一下吧。

```
class StringUtil {  
    public static String capitalize(String s) {  
        // null或空字符串不处理,直接返回  
        if (s == null || "".equals(s)) {  
            return s;  
        }  
        // 利用字符串切片,首字母大写,其他小写  
        return s.substring(0, 1).toUpperCase() + s.substring(1).toLowerCase();  
    }  
}  
  
System.out.println(StringUtil.capitalize("")); // ""  
System.out.println(StringUtil.capitalize("a")); // "A"  
System.out.println(StringUtil.capitalize("hikari")); // "Hikari"
```

## 20180531

## 12 面向对象 (续)

### ✧ 12.1 继承性

良好的代码是结构性合理、易于维护、重用性高。

继承是在已有类的基础上进行功能扩充, 使用 **extends** 关键字。

子类实例化, 系统会自动调用父类的构造方法(实例化父类对象), 子类构造方法隐藏了 **super()**; 表示子类构造方法调用父类构造方法, 只能放在首行。如果父

类没有无参构造，子类必须指明调用哪个父类构造方法。  
构造方法也可以用 `this()` 调用本类其他构造方法，也必须放在首行。所以 `this()` 和 `super()` 不能同时出现。

```
class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        System.out.println("Person类实例化...");
        this.name = name;
        this.age = age;
    }
    public Person() {
        this("匿名", 0);
    }
    public String toString() {
        return "name: " + this.name + ", age: " + this.age;
    }
    public void show() {
        System.out.println(this.toString());
    }
}

class Student extends Person {
    private String school;
    public Student(String name, int age, String school) {
        super(name, age); // 调用父类双参构造方法
        System.out.println("Student类实例化...");
        this.school = school;
    }
    public String toString() {
        return super.toString() + ", school: " + this.school;
    }
}

Student s = new Student("hikari", 25, "皇家幼稚园");
s.show();
```

结果：

```
Person 类实例化...
Student 类实例化...
name: hikari, age: 25, school: 皇家幼稚园
```

注意：

① Java 只能单继承，不能多重继承(A 继承 B 和 C)，可以多层继承(A 继承 B，B 继承 C)。但多层继承也要有限度，不要搞个祖宗 20 代出来。

实际开发，继承关系不要超过 3 层。

② 子类可以继承父类所有操作结构；但对于私有属性方法属于隐式继承，所有非私有属于显式继承。

③ 子类没有现实中败家子的概念，子类至少会维持父类现有功能。

## ✧ 12.2 覆写(override)

子类能获取父类全部定义，但如果子类觉得父类的设计不足，而且需要保留父类方法或属性名字时可以使用覆写。

子类没有某个方法或属性，就会从父类寻找；但子类有和父类一模一样的方法或属性时，使用子类自己的，而不用父类的。

上面 Student 类的 toString()方法就是覆写了父类 People 的 toString()方法。People 类的 toString()覆盖了上帝 Object 类的 toString()方法。

**注意：**

被覆写的方法不能拥有比父类方法更严格的访问控制权限。

public > default (不写) > private

如果父类方法是 public，子类覆写的方法访问权限不能变严格，只能是 public。

overloading 和 override 的区别：

区别	overloading	override
含义	重载	覆写
概念	方法名相同，参数类型或个数不同	方法名、参数类型、个数相同
权限	没有权限限制	被覆写方法不能拥有更严格的控制权限
范围	发生在一个类中	发生在继承关系类中

属性的覆写和方法的覆写类似。

```
class A {
    private String s = "父类私有属性";
    private String get() {
        return "父类私有方法";
    }
    public void show() {
        System.out.println(this.get());
        System.out.println(this.s);
    }
}

class B extends A {
    private String s = "子类私有属性";
    public String get() { // 父类的get()方法私有，子类的get()不是覆写
        return "子类同名方法";
    }
}
```

```
}
```

```
B b = new B();
```

```
b.show(); // 调用父类的show()方法,打印的都是父类的私有属性或私有方法返回值
```

因为 `private` 与覆写无关，如果属性或方法进行了封装，子类和父类的私有属性或方法就没关系了，就算名字一样，也是子类自己定义的。

所以对于封装的属性或方法的覆写是没有意义的。

面试题： `super` 和 `this`

1) `this` 表示从本类查找需要的属性或方法，如果本类不存在，就查找父类；`super` 表示不查找子类，直接查找父类；

2) `this` 和 `super` 都可以进行构造方法的调用，`this()` 调用本类的构造方法，`super()` 为子类调用父类的构造方法；`this` 和 `super` 必须放在构造方法首行，所以不能同时使用；

3) `this` 可以表示当前对象，`super` 没有这种概念。

### ✧ 12.3 final 关键字

**final** 关键字描述的是终结器的概念，主要用于定义不能被继承的类、不能被覆写的方法、常量。

对于常量，不能修改，而且类每个对象都一样，应该设为全局常量，如：

```
public static final int ON = 1;
```

```
public static final int OFF = 0;
```

之前的字符串拼接问题，因为 `p` 是变量，在运行时常量池，拼接得到 `c` 会开辟内存空间，`a` 和 `c` 指向不是同一个对象；如果使用 `final` 定义 `p`，`p` 就是常量，在静态常量池，拼接后 `c` 指向 `a` 指向的字符串常量。

```
String a = "你好 再见";
```

```
final String p = "你好";
```

```
String c = p + " " + "再见"; // 使用final定义的常量拼接
```

```
System.out.println(a == c); // true
```

## 20180601

### ✧ 12.4 Annotation 注解

`Annotation` 从 `JDK 1.5` 提出的一个新的开发技术结构，利用 `Annotation` 可以有效减少程序配置的代码和进行一些结构化定义。`Annotation` 是以注解的形式实现的程序开发。

程序开发结构的历史：

1) 程序定义时将所有可能用到的资源全部定义在代码中。如果服务器相关地址发生了改变，程序需要修改源码，需要开发人员维护，很麻烦；

2) 引入配置文件，其中定义全部要使用的服务器资源。在配置项不多时，这种配置非常好用并简单。但复杂项目有可能会特别多的配置文件，所有操作都要通过配置文件完成，开发难度提升。

3) 将配置信息重新写回程序，利用特殊的标记与程序代码分离，这就是注解的作用，也是 Annotation 提出的基本依据，但全部都使用注解开发，难度颇高。

现在很多围绕配置文件+注解的形式开发。

### ① @Override: 准确覆写

子类继承父类，发现父类某些方法不足时可以覆写父类方法。

但实际开发可能出现问题：

- 1) 子类忘记写 extends，不是覆写；
- 2) 覆写的方法名写错，将会被当成一个新的方法，也不是覆写。

为了避免此类问题，可以在覆写的方法上追加一个注解：

```
@Override // 明确指定是覆写的方法，如果父类没有此方法编译会提示
public String toString() {
    return "name: " + this.name + ", age: " + this.age;
}
```

其主要帮助开发者在编译时检查出程序的错误。

### ② @Deprecated: 过期操作

过期操作是一个软件在开发过程中，某个方法或类在最初设计时考虑不周，导致新版本应用会有问题。但不能直接删除，采用过期声明，可以给老人一个过渡时间，而提示新人不要用了。

```
@Deprecated // 声明该方法已过时
public void show() {
    System.out.println(this.toString());
}
```

eclipse 在方法名上添加删除线，提示方法已经过期，但不会报错。

命令行编译提示：

```
$ javac Main.java
注: Main.java 使用或覆盖了已过时的 API。
注: 有关详细信息, 请使用 -Xlint:deprecation 重新编译。
```

### ③ @SuppressWarnings: 压制警告

不愿意看见提示信息，可以使用压制警告，让警告信息不出现：

```
@SuppressWarnings({ "deprecation" }) // 不显示方法过时的警告
public static void main(String[] args) {
    Student s = new Student("hikari", 25, "皇家幼稚园");
    s.show();
}
```

## ✧ 12.5 多态性

- 1) 方法的多态性：方法的重载和方法的覆写；
- 2) 对象的多态性：父子实例对象之间的转换处理：
  - a) 对象向上转型；b) 对象向下转型。

大多数情况考虑向上转型。

① 向上转型可以对参数进行统一设计：

```
class Worker extends Person {
    private String job;
    public Worker(String name, int age, String job) {
        super(name, age);
        this.job = job;
    }
    @Override
    public String toString() {
        return super.toString() + ", job: " + this.job;
    }
}

public static void main(String[] args) {
    Person p1 = new Student("hikari", 25, "皇家幼稚园"); // 向上转型
    Person p2 = new Worker("张三", 38, "码农");
    func(p1); // name: hikari, age: 25, school: 皇家幼稚园
    func(p2); // name: 张三, age: 38, job: 码农
}

public static void func(Person p) { // 可以接收Person类实例和其所有子类的实例
    p.show();
}
```

p1 是 Student 的实例，p2 是 worker 的实例，两个类都是 Person 的子类，p1、p2 都是 Person 的实例。

使用方法重载也可以实现类似效果，但是如果子类有 n 万个，方法重载的话，每次新增一个子类就需要添加一个重载的方法，不利于维护。

② 向下转型

向下转型主要需要使用子类自己特殊的属性或方法。

```
public class Superman extends Person {
    public Superman(String name, int age) {
        super(name, age);
    }
    public void fly() {
        System.out.println("I believe i can fly...");
    }
    public void attack() {
        System.out.println("代表月亮消灭你!");
    }
}

System.out.println("*****正常情况超人应该是一个普通人*****");
Person p = new Superman("hikari", 25); // 向上转型
p.show();
```

```

System.out.println("*****外星人来袭*****");
Superman sm = (Superman) p; // 向下转型
sm.fly();
sm.attack();

```

向上描述的是一些公共特征，向下描述的是子类特殊的定义。  
 向下转型不是安全的操作。在向下转型之前首先要发生向上转型。  
 如果上面 p 是 Person 类的实例，会抛出 java.lang.ClassCastException

所以向下转型时需要判断 p 是不是 Superman 的实例。  
 可以使用 instanceof 关键字：

```

public static void main(String[] args) {
    System.out.println("*****正常情况超人应该是一个普通人*****");
    Person p1 = new Superman("hikari", 25); // 向上转型
    Person p2 = new Worker("张三", 38, "码农");
    System.out.println("*****外星人来袭*****");
    henshin(p1); // true
    henshin(p2); // false
}

public static void henshin(Person p) {
    System.out.println(p instanceof Superman);
    if (p instanceof Superman) {
        Superman sm = (Superman) p;
        sm.fly();
        sm.attack();
    }
}

```

## ✧ 12.6 Object 类

方法参数设为 Object 类型，可以接收所有数据类型，解决参数统一问题。  
 Object 类是万能数据类型，适合程序的标准设计。

```

public static void main(String[] args) {
    print(123); // 123, 自动装箱?
    print("haha"); // haha
    print(new Person("hikari", 25)); // name: hikari, age: 25
    print(false); // false
}

public static void print(Object obj) {
    System.out.println(obj);
}

```

### ① 获取对象信息

```
public String toString()
```



返回对象的字符串表示信息，为了易于阅读，建议重写此方法。类似于 Python 类的 `__str__()` 方法

直接打印 `p` 就是打印 `toString()` 方法返回的字符串：

```
Person p = new Person("hikari", 25);
System.out.println(p.toString());
System.out.println(p); // 两者打印结果相同
```

## ② 对象比较

```
public boolean equals(Object obj)
```

因为两个对象内存地址不同，`equals()` 默认比较两个对象的地址，不同对象结果肯定 `false`；如果需要比较两个对象内容是否相同，需要重写此方法。

```
public class Person {
    // ...
    @Override
    public boolean equals(Object obj) {
        if (obj == null) { // 过滤null
            return false;
        }
        if (!(obj instanceof Person)) { // 类型不同
            return false;
        }
        if (obj == this) { // 同一地址
            return true;
        }
        Person p = (Person) obj;
        return this.name.equals(p.getName()) && this.age == p.getAge();
    }
}

Person p = new Person("hikari", 25);
Person p2 = new Person("hikari", 25);
System.out.println(p.equals(p2)); // true
System.out.println(p.equals(null)); // false
System.out.println(p.equals("hikari")); // false
```

## ✧ 12.7 抽象类

在实际开发中很少继承一个已经完善的类，因为父类无法对子类做强制性要求（强制必须覆写某些方法），此时需要继承抽象类。抽象类用于父类设计。

有抽象方法的类就是抽象类，抽象类和抽象方法定义使用关键字 `abstract`，抽象方法没有方法体。

抽象类不能直接实例化，使用抽象类原则：

1) 抽象类必须被子类继承；

- 2) 抽象类的子类需要覆写抽象类全部抽象方法，否则子类还是抽象类；
- 3) 抽象类对象实例化可以利用多态性通过向上转型完成。

```
abstract class Action { // 抽象类
    public static final int EAT = 1; // 0000,0001
    public static final int SLEEP = 2; // 0000,0010
    public static final int WORK = 4; // 0000,0100
    public void command(int code) {
        switch (code) {
            case EAT: {
                this.eat();
                break;
            }
            case SLEEP: {
                this.sleep();
                break;
            }
            case WORK: {
                this.work();
                break;
            }
            case EAT + SLEEP + WORK: {
                this.eat();
                this.sleep();
                this.work();
                break;
            }
        }
    }
    public abstract void eat(); // 3个抽象方法,抽象方法没有方法体
    public abstract void sleep();
    public abstract void work();
}
// 三个子类继承抽象类,覆写所有抽象方法
class Robot extends Action {
    @Override
    public void eat() {
        System.out.println("我要充电!");
    }
    @Override
    public void sleep() {}
    @Override
    public void work() {
        System.out.println("机器人按固定套路工作...");
    }
}
```

```

    }
}

class Worker extends Action {
    @Override
    public void eat() {
        System.out.println("我要安静地吃饭!");
    }
    @Override
    public void sleep() {
        System.out.println("我要睡觉!");
    }
    @Override
    public void work() {
        System.out.println("搬砖...");
    }
}

class Pig extends Action {
    @Override
    public void eat() {
        System.out.println("吃剩饭");
    }
    @Override
    public void sleep() {
        System.out.println("倒地就睡");
    }
    @Override
    public void work() {}
}

    Action r = new Robot();
    Action w = new Worker();
    Action pig = new Pig();
    System.out.println("*****机器人行为*****");
    r.command(Action.SLEEP);
    r.command(Action.WORK);
    System.out.println("*****工人行为*****");
    w.command(Action.SLEEP + Action.EAT + Action.WORK);
    System.out.println("*****猪的行为*****");
    pig.command(Action.SLEEP);
    pig.command(Action.WORK);

```

结果:

```

*****机器人行为*****
机器人按固定套路工作...

```

```
*****工人为*****
我要安静地吃饭!
我要睡觉!
搬砖...
*****猪的行为*****
倒地就睡
```

#### 注意:

- 1) 抽象类不能用 `final` 关键字修饰, 因为抽象类必须有子类继承, 而 `final` 定义的类不能有子类。
- 2) 抽象类允许没有抽象方法, 但也无法直接实例化。
- 3) 抽象类可以有 `static` 方法, 静态方法不受实例化对象限制, 可以直接通过类名调用。
- 4) 抽象方法不能使用 `private` 修饰, 因为私有的方法子类无法覆写;
- 5) 抽象方法不能使用 `static` 修饰, 因为静态方法可以直接通过类名调用, 而抽象方法没有方法体, 调用没有意义。

综上, `abstract` 不能和 `final`、`private`、`static` 组合使用。

抽象类的好处是对子类方法统一管理; 子类如果想调用抽象类提供的普通方法, 必须覆写父类抽象方法, 而不是自己去实现其他方法。

#### ✧ 12.8 包装类

Java 设计之初认为一切皆对象, 而基本数据类型不是一个类, 最早可以人为定义一个类解决此问题。Java 提供包装类, 将基本数据类型进行包装后, 可以像对象一样进行引用传递, 也可以用 `Object` 类进行接收。

##### ① 八种包装类分为:

对象型包装类(`Object` 直接子类): `Boolean`、`Character`

数值型包装类(`Number` 直接子类): `Byte`、`Short`、`Integer`、`Long`、`Float`、`Double`

`Number` 类是抽象类, 其中 6 个方法的功能是从包装类获取基本数据:

```
public byte byteValue()
public short shortValue()
public abstract int intValue()
public abstract long longValue()
public abstract float floatValue()
public abstract double doubleValue()
```

##### ② 装箱和拆箱

- 1) 数据装箱: 将基本数据类型保存到包装类中, 一般可以利用构造方法完成;
- 2) 数据拆箱: 从包装类获取基本数据类型。

JDK 1.9 开始所有包装类的构造方法都变为已过时, 不建议继续使用, 因为 JDK 1.5 提供了自动装箱和拆箱操作。

```
Integer obj = 23; // 自动装箱
int num = obj; // 自动拆箱
```

```
obj++; // 包装类对象可以直接参与数学运算
System.out.println("obj=" + obj + ", num=" + num); // obj=24, num=23
```

自动装箱的最大好处是可以实现 Object 接收基本数据类型。

注意：

```
Integer x = 127;
Integer y = 127;
System.out.println(x == y); // true
x = 128;
y = 128;
System.out.println(x == y); // false
```

在-128~127 范围内自动装箱的对象==比较返回 true；超过返回 false。也就是包装类占位长度如果超过了一位就需要 equals()比较，不超过可以使用==判断。当然，new 创建的两个包装类对象==比较肯定为 false。所以包装类的相等判断一定要使用 equals()！

### ③ 基本数据类型和字符串的转换

包装类可以实现基本数据类型和字符串之间的转换。

#### 1) 字符串转为基本数据类型

```
public static int parseInt(String s)
public static double parseDouble(String s)
public static boolean parseBoolean(String s)
```

```
// 字符串由数字构成，格式错误抛出NumberFormatException
int a = Integer.parseInt("123");
System.out.println(a * a); // 15129
double b = Double.parseDouble("1.23");
System.out.println(b * 100); // 123.0
boolean flag = Boolean.parseBoolean("true");
System.out.println(flag); // true
// 只要不是"true"返回都是false
flag = Boolean.parseBoolean("hikari");
System.out.println(flag); // false
```

#### 2) 基本数据类型转为字符串

```
int num = 123;
// 方法1，与空字符串拼接，但是产生无用的垃圾
String s = num + "";
System.out.println(s.length()); // 3
// 方法2，String类提供一系列valueOf()方法，避免一些垃圾的产生
String s1 = String.valueOf(num); // 调用包装类的toString()方法
System.out.println(s1.length()); // 3
```

20180602

## 13 接口

### ✧ 13.1 接口定义

抽象类可以实现子类覆写方法的控制，但抽象类还有保留一些普通方法，而普通方法可以涉及一些安全或隐私的操作问题。如果要对外部隐藏全部实现细节，可以通过**接口**(interface)描述。

接口的最原始定义是只包含抽象方法和全局常量的抽象类。JDK 1.8 引入了 Lambda 表达式后，接口还可以定义普通方法或静态方法。但是从设计角度来说，接口还是应该以抽象方法和全局常量为主。

接口的使用原则：

- 1) 接口需要被子类实现(implements)，一个子类可以实现多个父接口；
- 2) 子类必须覆写接口全部抽象方法，否则子类是抽象类；
- 3) 接口对象可以利用子类对象的向上转型进行实例化。

接口不允许继承父类，所以接口不是 Object 的子类，但是接口可以通过 Object 接收，因为 Object 可以接收所有数据类型。

```
interface IUSB { // 加I是为了区分类和接口，不强求
    public final static String TYPE = "USB 3.0"; // 全局常量
    public abstract void show(); // 抽象方法
}

interface IClickable {
    public abstract void click();
}

class Mouse implements IUSB, IClickable {
    @Override
    public void show() {
        System.out.println("我是鼠标");
    }
    @Override
    public void click() {
        System.out.println("鼠标点击...");
    }
}

// 此实例对象既是IUSB的实例,也是IClickable的实例,还是Object的实例
IUSB mouse = new Mouse();
mouse.show(); // 我是鼠标
IClickable click = (IClickable) mouse;
click.click(); // 鼠标点击...
```

接口描述的是一个公共的定义标准，所有抽象方法访问权限一定是 public，所以写与不写是一样的。所以可以简写为：

```
interface IUSB {
    String TYPE = "USB 3.0"; // 全局常量
    void show(); // 抽象方法
}
```

一个接口可以继承多个父接口，即接口多继承。

```
interface A {
    public void run();
}

interface B {
    public void eat();
}

interface C extends A, B { // 接口多继承
    public void say();
}

class P implements C { // 覆写所有抽象方法
    @Override
    public void run() {
        System.out.println("running...");
    }
    @Override
    public void eat() {
        System.out.println("eating food...");
    }
    @Override
    public void say() {
        System.out.println("hello...");
    }
}
```

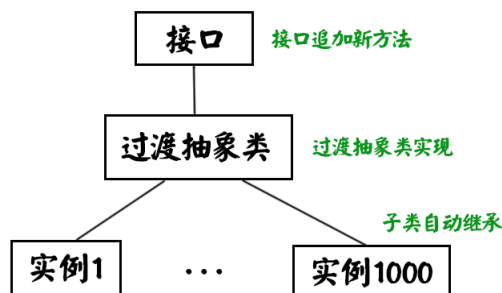
接口使用三种形式：

- 1) 进行标准设置；
- 2) 表示一种操作能力；
- 3) 暴露远程方法视图，一般在 RPC 分布式开发使用。

### ✧ 13.2 接口定义加强

如果接口有 1000 个子类，要为接口添加新方法，而且所有子类方法实现相同，需要复制粘贴 1000 次。

可以考虑先让一个抽象类(作为过渡)实现接口，再让 1000 个子类继承抽象类。如果要为接口添加新方法，只需要在抽象类中实现，子类自动继承。



JDK 1.8 后，为了解决接口设计缺陷，允许在接口中定义普通方法，但必须追加 default 声明。

```

interface IUSB {
    // ...
    public default boolean connect() { // 接口中default声明的普通方法
        System.out.println("连接成功...");
        return true;
    }
}

class Mouse implements IUSB, IClickable {
    // ...
    @Override
    public void click() {
        if (this.connect()) {
            System.out.println("鼠标点击...");
        }
    }
}
  
```

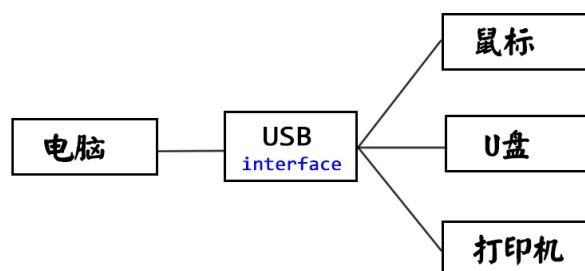
注意：此操作属于挽救功能，不是必须情况，不应作为设计首选。

接口还可以定义 static 方法。如果接口里定义了普通方法和 static 方法，就可以取代抽象类的功能，但是接口设计最好还是以抽象方法为主。

### ✧ 13.3 使用接口定义标准

接口在开发中最重要的应用就是进行标准的制定。

例如，电脑可以插入各种 USB 设备，电脑只认 USB 标准，不关心标准的具体实现。





```

interface IUSB { // 定义USB标准
    public boolean connect();
    public void work();
}

class Computer {
    public void plugin(IUSB usb) { // 符合USB标准就能在电脑类中使用
        if (usb.connect()) {
            usb.work();
        }
    }
}

class Mouse implements IUSB { // 定义符合USB标准的类
    @Override
    public boolean connect() {
        System.out.println("鼠标已经连接...");
        return true;
    }
    @Override
    public void work() {
        System.out.println("鼠标点击...");
    }
}

class Printer implements IUSB {
    @Override
    public boolean connect() {
        System.out.println("没有发现打印机...");
        return false;
    }
    @Override
    public void work() {
        System.out.println("打印文章...");
    }
}

Computer computer = new Computer();
computer.plugin(new Mouse());
computer.plugin(new Printer());

```

结果:

```

鼠标已经连接...
鼠标点击...
没有发现打印机...

```

### ✧ 13.4 工厂设计模式 (Factory)

主类相当于客户端，客户端一般不需要知道具体的子类，只需要知道接口的标准，主类代码需要以不变应万变。如同现实中一个人去吃饭，只要食物能吃就行，不管食物从哪里来，不管食物是怎么做的。

```
interface IFood { // 定义食物标准
    public void eat();
}

class Bread implements IFood {
    @Override
    public void eat() {
        System.out.println("吃面包...");
    }
}

class Milk implements IFood {
    @Override
    public void eat() {
        System.out.println("喝牛奶...");
    }
}

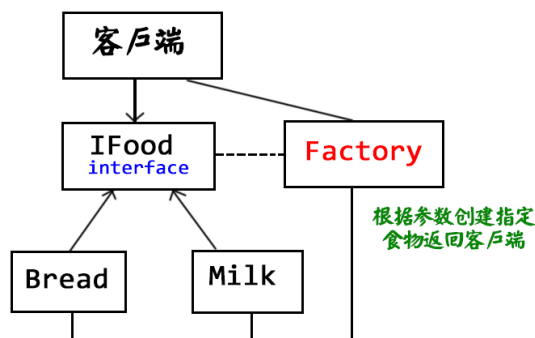
IFood food = new Bread();
food.eat();
```

上面代码中，客户端必须知道具体的子类如 Bread，然后去 new 一个 Bread；如果不想吃面包想吃牛奶，客户端代码需要修改。

也就是代码的耦合性太强了。造成耦合的原因是关键字 new。

Java 实现可移植性的关键是 JVM，JVM 的核心是利用一个虚拟机运行 Java 程序，与具体的操作系统无关。良好的设计应避免耦合。

工厂设计模式：



```
class Factory {
    public static IFood getInstance(String className) {
```

```

        if ("bread".equalsIgnoreCase(className)) {
            return new Bread();
        }
        if ("milk".equalsIgnoreCase(className)) {
            return new Milk();
        }
        return null;
    }
}

for (String s : args) {
    IFood food = Factory.getInstance(s);
    food.eat();
}

```

结果:

```

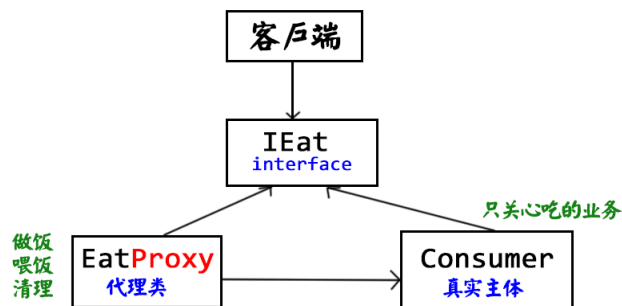
$ javac FactoryDemo.java
$ java FactoryDemo milk bread
喝牛奶...
吃面包...

```

这样，客户端与 IFood 接口的子类没有任何关联，所有关联通过 Factory 类完成，程序初始化通过初始化参数进行要使用的子类定义。如果日后要子类扩充，只需要修改 Factory 类。

### ✧ 13.5 代理设计模式(Proxy)

代理设计模式主要功能是帮助开发中集中精力与核心业务。



代理设计模式主要特点是一个接口提供两个子类，一个是真实业务操作类，另一个是代理业务操作类。没有代理业务，真实业务无法进行。

```

interface IEat {
    public void eat();
}

class Consumer implements IEat { // 真实业务，顾客吃
    @Override
    public void eat() {
        System.out.println("[consumer] 得到一份食物,开始吃...");
    }
}

```

```

class EatProxy implements IEat { // 服务代理
    private IEat con; // 为吃服务
    public EatProxy(IEat con) { // 代理项
        this.con = con;
    }

    public void prepare() { // 准备过程
        System.out.println("[proxy] 1. 准备食材");
        System.out.println("[proxy] 2. 处理食材");
    }
    public void clear() {
        System.out.println("[proxy] 3.收拾碗筷");
    }
    @Override
    public void eat() { // 准备,调用顾客吃,清理
        this.prepare();
        this.con.eat();
        this.clear();
    }
}

// 此处应该使用工厂设计模式
IEat eat = new EatProxy(new Consumer());
eat.eat();

```

结果:

```

[proxy] 1. 准备食材
[proxy] 2. 处理食材
[consumer] 得到一份食物,开始吃...
[proxy] 3.收拾碗筷

```

### ✧ 13.6 抽象类和接口的区别

抽象类和接口定义形式很像，而 JDK 1.8 接口能定义 static 和 default 方法，这一点就更明显了。

区别	抽象类	接口
定义关键字	<b>abstract class</b>	<b>interface</b>
权限	各种权限	只能 <b>public</b>
子类使用	<b>extends</b> 继承一个抽象类	<b>implements</b> 实现多个接口
两者关系	抽象类可以实现多个接口	接口不能继承抽象类，但可以实现多个父接口
使用	1) 抽象类或接口必须定义子类； 2) 子类必须覆写抽象类或接口的所有抽象方法； 3) 通过子类的向上转型实现抽象类或接口的实例化。	

都可以使用时优先考虑接口，可以避免子类单继承的局限性。

20180603

练习：定义 Shape 类表示一般二维图形，具有抽象方法 area()和 perimeter()表示计算面积和周长。定义一些 Shape 的子类，如圆形、矩形等。

```
abstract class Shape { // 定义抽象类
    public abstract double area();
    public abstract double perimeter();
}

class Circle extends Shape {
    private static final double PI = 3.14159;
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public double area() {
        return PI * this.radius * this.radius;
    }
    @Override
    public double perimeter() {
        return 2 * PI * this.radius;
    }
}

class Rectangle extends Shape {
    private double width;
    private double height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    @Override
    public double area() {
        return this.width * this.height;
    }
    @Override
    public double perimeter() {
        return 2 * (this.width + this.height);
    }
}

class Factory { // 工厂类
    public static Shape getInstance(String className, double... args) {
        // 暂时忽略健壮性判断
        if ("circle".equalsIgnoreCase(className)) {
```

```

        return new Circle(args[0]);
    }
    if ("rectangle".equalsIgnoreCase(className)) {
        return new Rectangle(args[0], args[1]);
    }
    return null;
}
}

Shape c = Factory.getInstance("circle", 5);
Shape r = Factory.getInstance("rectangle", 3, 5);
System.out.println("圆的面积: " + c.area() + ", 圆的周长: " +
c.perimeter());
System.out.println("矩形的面积: " + r.area() + ", 矩形的周长" +
r.perimeter());

```

使用工程设计模式主要目的是向客户端隐藏实现的子类。

### ✧ 13.7 单例设计模型

有时候类不需要重复产生对象，如一个程序启动，需要有一个类负责保存程序加载的数据信息，程序运行中保证一直存在一个实例化对象。

比如 Windows 只有一个回收站，每个磁盘共享，就是单例模式的实际应用。

特点：构造方法私有化，类内部提供 static 方法获取实例化对象，保证外部只有一个实例化对象。

单例设计模式分为[恶汉式](#)和[懒汉式](#)。

① 恶汉式：系统加载就自动提供实例化对象：

```

class Singleton {
    // final保证类内部只有一个实例化对象
    private static final Singleton INSTANCE = new Singleton();
    private Singleton() {} // 构造方法私有化，外部无法创建实例化对象
    public static Singleton getInstance() {
        return INSTANCE;
    }
    public static void print() {
        System.out.println("hello, hikari!");
    }
}

Singleton s1 = Singleton.getInstance();
Singleton s2 = Singleton.getInstance();
System.out.println(s1 == s2); // true

```

② 懒汉式：第一次使用时实例化对象：

```

class Singleton {
    private static Singleton instance;
    private Singleton() {} // 构造方法私有化，外部无法创建实例化对象
    public static Singleton getInstance() {
        if (instance == null) { // 第一次使用,实例化对象
            instance = new Singleton();
        }
        return instance;
    }
}

```

## ✧ 13.8 多例模式

如性别类只有两个对象，颜色基色类只有三个对象，可以使用多例模式。本质与单例模式相同。

```

class Color {
    private static final Color RED = new Color("红色");
    private static final Color GREEN = new Color("绿色");
    private static final Color BLUE = new Color("蓝色");
    private String name;
    private Color(String name) {
        this.name = name;
    }
    public static Color getInstance(String color) {
        switch (color) {
            case "red": return RED;
            case "green": return GREEN;
            case "blue": return BLUE;
            default: return null;
        }
    }
    public String toString() {
        return this.name;
    }
}

Color c = Color.getInstance("red");
System.out.println(c); // 红色

```

## 14 泛型

泛型从 JDK 1.5 引入，主要目的是解决向下转型时的 ClassCastException 问题。

### ✧ 14.1 泛型的引入

现设计一个坐标类 Point 有属性 x 和 y，x 和 y 类型相同，可以是 int、double 或 String。这时最直接的想法是 x 和 y 用 Object 定义。

```

class Point {
    private Object x;
    private Object y;
    public Object getX() {
        return x;
    }
    public void setX(Object x) {
        this.x = x;
    }
    public Object getY() {
        return y;
    }
    public void setY(Object y) {
        this.y = y;
    }
}

Point p = new Point();
p.setX(10); // 自动装箱,再向上转型
p.setY(20);
int x = (Integer) p.getX(); // 向下转型,再自动拆箱
int y = (Integer) p.getY();
System.out.println("(" + x + ", " + y + ")"); // (10, 20)

```

但是如果此时改为:

```

p.setX(10);
p.setY("北纬 30°");

```

程序编译不会有任何问题,但是运行就会报错: `java.lang.ClassCastException`  
 所以本程序存在安全隐患,因为使用了 `Object` 类型,范围太大了。如果这种错误能在编译时及时暴露,就能避免运行时的尴尬。

## ✧ 14.2 泛型的基本定义

避免 `ClassCastException` 最好方法是不进行向下转型。泛型的本质是类的属性或方法参数与返回值的类型可以在对象实例化时动态决定。

改为使用泛型定义 `Point` 类:

```

class Point<T> { // T是Type简写,可以定义多个泛型
    private T x;
    private T y;
    public Point(T x, T y) {
        this.x = x;
        this.y = y;
    }
    public T getX() {

```



```

        return x;
    }
    public void setX(T x) {
        this.x = x;
    }
    public T getY() {
        return y;
    }
    public void setY(T y) {
        this.y = y;
    }
}

// 不设置泛型默认Object类, 会有警告
Point<Integer> p = new Point<Integer>(10, "北纬20°"); // 编译就会报错

```

泛型设置仅针对该对象，如果泛型设置了 `Integer`，传入 `String` 编译就会报错。因为使用了泛型，获取数据时不需要向下转型：

```

Point<Integer> p = new Point<Integer>(10, 20);
int x = p.getX();
int y = p.getY();
System.out.println("(" + x + ", " + y + ")"); // (10, 20)

```

泛型的注意点：

- 1) 泛型只能使用引用数据类型，操作基本数据类型要使用包装类；
- 2) JDK 1.7 后泛型实例化=后面的<>里面可以省略：

```

Point<Integer> p = new Point<>();

```

使用泛型可以解决大部分对象类型强制转换问题，也就是说有了泛型以后不建议使用 `Object` 类接收参数。

### ✧ 14.3 泛型通配符

泛型的引入给对象的引用传递带来的一些麻烦：

如定义方法：

```

private static void print(Point<Integer> p) {
    System.out.println("(" + p.getX() + ", " + p.getY() + ")");
}

Point<Integer> p = new Point<Integer>(10, 20);
print(p); // (10, 20)
Point<String> p2 = new Point<>("东经120°", "北纬30°");
print(p2); // 报错, 类型不匹配

```

此时 `print()` 方法只能接收 `Point<Integer>` 对象，不能接收 `Point<String>` 对象。而且泛型与对象类型无关，`Point<Integer>` 和 `Point<String>` 都是 `Point` 对象，类型相同，无法利用方法的重载。

如果方法参数不设置泛型呢？可以正常输出，但是也可以随意修改其中数据：

```
private static void print(Point p) {  
    p.setX(22); // 不设置泛型,可以对数据随意修改,不安全  
    System.out.println("(" + p.getX() + ", " + p.getY() + ")");  
}
```

结果：

```
(22, 20)  
(22, 北纬 30°)
```

这显然更不好...

使用泛型通配符？可以接收任意泛型类型，而且不能修改其中数据(可以获取)

```
private static void print(Point<?> p) {  
    System.out.println("(" + p.getX() + ", " + p.getY() + ")");  
}
```

- 1) 设置泛型上限：如<? extends Number>，只能使用 Number 及其子类
  - 2) 设置泛型下限：如<? super String>，只能使用 String 及其父类
- 此类代码开发不常见，但是系统类库中有大量通配符使用，所有需要能看懂。

#### ✧ 14.4 泛型接口

```
interface IMessage<T> { // 泛型接口  
    public String echo(T t);  
}  
  
// 1. 子类继续使用泛型  
class Message1<T> implements IMessage<T> {  
    @Override  
    public String echo(T t) {  
        return "echo: " + t;  
    }  
}  
  
// 2. 直接定义具体泛型类型  
class Message2 implements IMessage<String> {  
    @Override  
    public String echo(String t) {  
        return "echo: " + t;  
    }  
}
```

#### ✧ 14.5 泛型方法

泛型方法不一定就要定义在泛型类里。

```
public static void main(String[] args) {  
    ArrayList<Integer> arr = new ArrayList<>();  
    getArr(arr, 1, 2, 3, 4, 5);  
}
```

```

        System.out.println(arr); // [1, 2, 3, 4, 5]
    }

    private static <T> void getArr(ArrayList<T> arr, T... args) {
        for (T t : args) {
            arr.add(t);
        }
    }

```

会警告 Potential **heap pollution** via varargs parameter args  
提示使用泛型可变参数会导致堆污染?

## 15 包

### ✧ 15.1 包的定义

操作系统要求同一个目录不能有同名的.class 文件，但是实际开发，特别是多人开发，很难避免同名问题。将.class 文件放到不同目录，可以很大程度避免重名问题。**包**就是存放一个或多个.class 文件的目录。

```

package hikari.hello; // 首行定义包名,hikari包的hello子包
public class Hello {
    public static void main(String[] args) {
        System.out.println("hello world");
    }
}

```

命令行操作:

```

$ javac -d . Hello.java
$ java hikari.hello.Hello
hello world

```

javac -d 表示打包编译，-d . 表示在当前目录生成 package 目录  
java 运行时要带包名运行程序类。

### ✧ 15.2 包的导入

使用 **import** 语句。

hikari/Muse.java:

```

package hikari;
public class Muse {
    public static final String[] MUSE = new String[] { "高坂穗乃果", "绚濑绘里",
        "南ことり", "国田海未", "星空凛", "西木野真姬", "东条希", "小泉花阳", "矢泽にこ" };
    public static final String GROUPNAME = "μ's";
}

```

java0603/JavaDemo.java:

```

package java0603;
import hikari.Muse; // 导入自定义的包
public class JavaDemo {
    public static void main(String[] args) {

```

```

        System.out.println(Muse.MUSE.length); // 9
        System.out.println(Muse.GROUPNAME); // μ's
    }
}

```

不同的包只能导入 public 的类，没有 public 定义的类只能在同一个包中使用。

如果一个包要导入的类太多，可以使用\*全部导入，但并不是全部都加载，而是按需加载。唯一区别是，\*如果导入的不同包有相同的类名，如 hikari.Muse 和 hello.Muse，直接使用 Muse 会出现不明确的引用，简单解决方法是在类名前加上包名，以避免冲突。

通过具体类名导入，如果类名相同编译时直接报错...

### ✧ 15.3 包的静态导入

JDK 1.5 后，对于全部是静态方法组成的特殊类可以采用静态导入。

就像方法定义在主类中一样，可以由主方法直接调用。

```

package hikari;

public class MyMath { // 全部由静态方法组成的类
    public static int sum(int x, int y, int... args) {
        int s = x + y;
        for (int i : args) {
            s += i;
        }
        return s;
    }

    public static int max(int x, int y, int... args) {
        int tmp = x > y ? x : y;
        for (int i : args) {
            if (i > tmp) {
                tmp = i;
            }
        }
        return tmp;
    }
}

package java0603;

import hikari.MyMath;

public class JavaDemo {
    public static void main(String[] args) {
        System.out.println(MyMath.sum(87, 33, 12, 35, 56)); // 223
        System.out.println(MyMath.max(34, 12, 56, 8)); // 56
    }
}

```

普通导入需要类名.方法名()调用，如果引用方法多，每个方法都要添加类名，很麻烦。而使用静态导入可以避免写大量的类名：

```
import static hikari.MyMath.*; // 静态导入,属于功能弥补,很少用
public class JavaDemo {
    public static void main(String[] args) {
        System.out.println(sum(87, 33, 12, 35, 56)); // 223
        System.out.println(max(34, 12, 56, 8)); // 56
    }
}
```

#### ✧ 15.4 Jar 命令

一个项目开发完成肯定会有大量.class 文件，可以使用压缩结构的形式来维护，Java 之中称为 jar 文件。可以使用 JDK 提供的 jar 命令完成。

```
$ jar
用法: jar [OPTION...] [ [--release VERSION] [-C dir] files] ...
尝试使用 `jar --help' 获取详细信息。
```

##### ① 生成 jar 文件

比如 Muse.java 文件，进行编译：

```
$ javac -d . Muse.java
```

生成 hikari 包，里面有 Muse.class 文件。

打包为 hikari.jar：

```
$ jar -cvf hikari.jar hikari
已添加清单
正在添加: hikari/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: hikari/Muse.class(输入 = 601) (输出 = 483)(压缩了 19%)
```

-c: 创建新的 jar 文件；

-v: 得到详细的输出；

-f: 设置生成 jar 文件的名称，如 hikari.jar

##### ② 使用 jar 文件

每个.jar 文件都是一个独立的程序路径，需要通过 CLASSPATH 配置。

```
$ SET CLASSPATH=.;D:\hikari.jar
$ javac JavaDemo.java
$ java JavaDemo
9
μ's
```

CLASSPATH 设置为当前目录+hikari.jar 的目录，JavaDemo.java 可以在别处编译运行。如果程序编译后，CLASSPATH 发生变化，类无法加载，报错：

```
Exception in thread "main" java.lang.NoClassDefFoundError: hikari/Muse
```

出现这种错误的唯一原因是.jar 包没有配置正确。

#### ✧ 15.5 模块

JDK 1.9 之前所有历史版本提供的是一个所有类的.jar 文件(rt.jar、tools.jar)

JDK 1.9 后提供了模块化设计，将原本很大的要加载的.jar 文件变为若干个模块文件，启动是时候根据程序加载指定的模块，实现启动速度变快的效果。

模块在 JDK 安装目录的 jmods 目录里。

## ✧ 15.6 系统常见包

- 1) java.lang: 如 String、Number、Object 等类都在此包，默认导入；
- 2) java.lang.reflect: 反射机制处理包，Java 精髓，所有设计从此开始；
- 3) java.util: 工具类的定义，包括数据结构的定义；
- 4) java.io: 输入、输出流操作的包；
- 5) java.net: 网络程序开发包；
- 6) java.sql: 数据库编程开发；
- 7) java.applet: Java 最原始的使用形式，直接嵌套在网页上执行的程序类。现在的程序都以 Application 为主(有主方法的程序)；
- 8) java.awt、javax.swing: Java 的 GUI 包，awt 为重量级组件，swing 为轻量级；

少年，看官方文档吧！

## ✧ 15.7 访问控制权限

封装性主要依靠访问控制权限，共有四种访问权限。

- 1) private: 同一个类内部能访问；
- 2) default: 同一个包所有类能访问；
- 3) protected: 同一个包所有类、不同包子类可以访问；
- 4) public: 不同包的所有类都可以访问。

	类内部	本包	子类	外部包
private	√	×	×	×
default	√	√	×	×
protected	√	√	√	×
public	√	√	√	√

20180604

## 16 枚举

JDK 1.5 后才提供枚举，其主要作用是定义有限个数对象的一种结构。枚举属于多例设计(P55~13.8)，而且结构更加简单。

### ✧ 16.1 枚举的定义

使用关键字 enum 定义枚举类：

```
enum Color { // 枚举类
    RED, GREEN, BLUE; // 实例化对象
}

Color c = Color.RED; // 获取实例化对象
System.out.println(c); // RED
```

枚举和多例模式可以实现相同功能，但是枚举可以在编译时就判断实例化对象是否存在。比如获取 Color.PINK，编译时就会报错。

values()方法获取所有枚举对象组成的数组：

```
for (Color c : Color.values()) {
    System.out.println(c);
}
```

而且 switch 语句也支持枚举的判断；多例则需要编写大量 if 判断语句。

## ✧ 16.2 Enum 类

枚举本质是一个类，使用 enum 关键字定义的类默认继承于抽象类 Enum。

Enum 类主要方法：

```
protected Enum(String name, int ordinal)
public final String name()
public final int ordinal()
```

```
for (Color c : Color.values()) {
    System.out.println(c.ordinal() + " - " + c.name());
}
```

结果：0 - RED; 1 - GREEN; 2 - BLUE

枚举常量的序号是它在枚举声明的位置，初始为 0。

## ✧ 16.3 定义枚举结构

① 定义属性、构造方法(必须 private)、普通方法

```
enum Color { // 枚举类
    RED("红色"), GREEN("绿色"), BLUE("蓝色"); // 实例化对象写在首行
    private String name;
    private Color(String name) {
        this.name = name;
    }
    public String toString() {
        return this.name;
    }
}

for (Color c : Color.values()) {
    System.out.println(c.ordinal() + "-" + c.name() + "-" + c);
}
```

② 枚举类可以实现接口，与普通类一样。

③ 枚举类可以直接定义抽象方法，每个枚举对象都要独立覆写抽象方法。

```
enum Color { // 枚举类
    RED("红色") { // 每个对象都要覆写抽象方法
        @Override
        public String show() {
            return this.toString();
        }
    }
}
```

```

    },
    GREEN("绿色") {
        @Override
        public String show() {
            return this.toString();
        }
    },
    BLUE("蓝色") {
        @Override
        public String show() {
            return this.toString();
        }
    }; // 实例化对象写在首行
    private String name;
    private Color(String name) {
        this.name = name;
    }
    public String toString() {
        return this.name;
    }
    public abstract String show(); // 枚举类中定义抽象方法
}

```

```
System.out.println(Color.RED.show()); // 红色
```

上面代码结构很有意思，然后根本没人会这么用...

枚举定义十分灵活，然而实际中主要还是用于定义几个实例化对象。但是这样的话不用枚举也可以实现，只不过稍微麻烦点。

一句话，枚举爱用不用，能看懂就行。

## 17 异常

异常是导致程序执行中断的指令流。

下面代码 1/0 会抛出异常：

```

System.out.println("1. start~");
System.out.println("2. 计算1/0: " + 1 / 0);
System.out.println("3. over~");

```

程序只会输出 1. start~，在计算 1/0 时抛出异常，程序中止，后面不会执行。

实际中遇到非致命错误，需要处理异常保证流程顺利完成。如果一个机长被蚊子咬了(异常)，就要把飞机紧急降落，这是十分不合理的，真实情况是机长会把蚊子拍死或默默忍受，也不会中止。

所以为了保证出现了非致命错误，程序依然可以执行完成，需要有一个完善的异常处理机制。



## ✧ 17.1 异常处理结构

```
try { //... } catch (Exception e) { //... } finally { //... }
System.out.println("1. start~");
try {
    System.out.println("2. 计算1/0: " + 1 / 0);
} catch (ArithmeticException e) { // 可以多个catch捕获多个异常
    e.printStackTrace(); // 打印完整异常信息
} finally { // 无论有没有异常都执行
    System.out.println("3. over~");
}
```

结果:

```
1. start~
java.lang.ArithmeticException: / by zero
    at java0604.JavaDemo.main(JavaDemo.java:69)
3. over~
```

即使有了异常处理语句，如果没有正确捕获异常，程序也会中断，后面语句不会执行，但是 `finally` 语句一定执行。

多个 `catch` 语句可以捕获多个异常，然而都已经知道有哪些异常，为什么还需要异常处理？写几个 `if` 判断不就可以了？

所以 `try-catch` 简单处理不是 Java 异常处理的精髓。而且捕获所有异常虽好，但是代码编写复杂。

## ✧ 17.2 异常处理流程

异常处理流程实际操作的是一个异常类实例化对象。

- 1) 程序运行过程中产生异常，自动实例化相关异常类的对象；
- 2) 如果没有提供异常处理的支持，采用 JVM 默认异常处理方式(打印异常信息，程序中断)；反之产生的异常对象将被 `try` 语句捕获；
- 3) `try` 捕获的异常对象和 `catch` 的异常类型依次匹配：如果匹配成功，则使用此 `catch` 的异常处理；如果没有任何匹配，表示该异常无法处理；
- 4) 不管是否有异常都要执行 `finally` 语句，之后判断当前异常是否被处理：如果处理了，继续执行后面的代码；反之交给 JVM 执行默认处理。

异常最大的类是 `Throwable`，其有两个子类：

- 1) **Error**：程序未执行时发生的错误，开发者无法处理；
- 2) **Exception**：程序运行时发生的异常，开发者可以处理。

所有异常都可以使用 `Exception` 接收，捕获多个异常时范围大的异常放后面。不知道异常类型时，直接使用 `Exception` 捕获虽然方便，但是其屏蔽了异常的细节，代码多了，很难发现异常在哪。

## ✧ 17.3 throws 关键字

方法声明上使用 `throws` 关键字标注该方法可能会产生的异常类型。

```

public static void main(String[] args) {
    try {
        // 调用处需要处理异常,或继续向上抛出
        System.out.println(divide(1, 0));
    } catch (Exception e) {
        e.printStackTrace(); //Infinity
    }
}

private static double divide(int x, int y) throws Exception {
    // throws告知调用者此方法可能产生异常类型,异常抛给调用者
    return (double) x / y;
}

```

主方法也可以继续把异常向上抛出，抛给 JVM，但没意义。

#### ✧ 17.4 throw 关键字

**throw** 关键字用于手动抛出异常。手动创建一个异常实例化对象，向上抛出。

```

private static double divide(int x, int y) throws Exception {
    // throws告知调用者此方法可能产生异常类型,异常抛给调用者
    if (y == 0) { // throw手动抛出异常
        throw new Exception("除数不能为0!");
    }
    return (double) x / y;
}

```

throw 和 throws 的区别：

- 1) throw 在代码块中使用，用于手动抛出异常对象；
- 2) throws 在方法定义上使用，告诉调用者该方法可能产生的异常，由调用者处理异常。

#### ✧ 17.5 RuntimeException 类

- 1) RuntimeException 是 Exception 的子类；
- 2) RuntimeException 标注的异常可以不需要进行强制性处理；
- 3) 常见 RuntimeException：NumberFormatException、ClassCastException、NullPointerException 等。

```

public static void main(String[] args) {
    // RuntimeException异常不需要强制处理
    System.out.println(divide(1, 4));
}

private static double divide(int x, int y) throws RuntimeException {
    if (y == 0) { // 抛出RuntimeException
        throw new RuntimeException("除数不能为0!");
    }
}

```

```

        return (double) x / y;
    }

```

## ✧ 17.6 自定义异常类

JDK 提供特别多的异常类，但未必是想要的，可以自定义异常类。需要继承 `Exception` 或 `RuntimeException`。

```

// 自定义异常
class BombException extends RuntimeException {
    public BombException(String message) {
        super(message);
    }
}

class Consumer {
    public static void eat(int num) throws BombException {
        if (num > 10) {
            throw new BombException("吃撑了,肚子要炸了!");
        }
        System.out.println("不怕胖,继续吃!");
    }
}

Consumer.eat(11);

```

## ✧ 17.7 assert 关键字

JDK 1.4 新增断言功能，确认代码执行到某处时是期待的结果。

```

double x = divide(100, 50);
assert x == 3; // 断言 x 为 3, 如果 false 抛出 AssertionError

```

Java 没有将断言设置为必须执行的步骤，需要特定环境才能开启。

命令行：

```

$ javac JavaDemo.java
$ java -ea JavaDemo
Exception in thread "main" java.lang.AssertionError

```

# 18 内部类

## ✧ 18.1 内部类的定义

在类、方法、代码块等内部定义的类是内部类。

```

class Outer { // 外部类
    private String name = "hikari";
    public void show() {
        // 实例化内部类对象
        Inner in = new Inner();
    }
}

```

```

        in.print();
    }

    class Inner { // 内部类
        private String code = "01";
        public void print() {
            // 使用外部类的私有属性
            System.out.println(this.code + "-" + Outer.this.name);
        }
    }
}

Outer out = new Outer();
out.show();

```

虽然内部类破坏了程序的结构，不过内部类的优势是可以轻易地访问外部类的私有属性。如果不使用内部类，将 Inner 类移到外部需要：

- 1) Outer 类需要提供 getName()方法；
- 2) Inner 类需要 Outer 类的实例化对象，调用 getName()方法；
- 3) Inner 类私有属性 out，通过构造方法获取；
- 4) Outer 类的 show()方法实例化 Inner 对象时，需要传入 this。

```

class Outer {
    private String name = "hikari";
    public void show() {
        Inner in = new Inner(this);
        in.print();
    }
    public String getName() {
        return this.name;
    }
}

class Inner {
    private String code = "01";
    private Outer out;
    public Inner(Outer out) {
        this.out = out;
    }
    public void print() {
        System.out.println(this.code + "-" + out.getName());
    }
}

```

## ✧ 18.2 内部类相关

外部类也可以访问内部类的私有属性或方法。

使用内部类后，内部类和外部类的私有访问不需要 setter 和 getter 间接访问。

内部类编译后生成 Outer\$Inner.class 类文件，\$就是程序中的.，内部类全称就是 Outer.Inner。

在外部实例化 Inner 类对象：

```
Outer.Inner in = new Outer().new Inner();  
in.print();
```

内部类使用 private 修饰：private class Inner，则无法在外部实例化。

抽象类和接口都可以定义内部结构。

内部抽象类可以定义在普通类、抽象类、接口内部。

示例：内部接口：

```
interface IChannel {  
    public void send(IMessage msg);  
    interface IMessage { // 内部接口  
        public String getContent();  
    }  
}  
  
class Channel implements IChannel {  
    @Override  
    public void send(IMessage msg) {  
        System.out.println("发送消息: " + msg.getContent());  
    }  
  
    class Message implements IMessage {  
        @Override  
        public String getContent() {  
            return "hello world";  
        }  
    }  
}  
  
IChannel ch = new Channel();  
ch.send(((Channel) ch).new Message()); // 发送消息: hello world
```

接口的内部类可以实现该接口，接口的静态方法返回内部类的实例化对象。

示例：

```
interface IUSB {  
    public void connect();  
    class Mouse implements IUSB { // 内部类实现外部接口
```

```

@Override
public void connect() {
    System.out.println("鼠标已经连接...");
}
}

public static IUSB getInstance() {
    return new Mouse();
}
}

IUSB usb = IUSB.getInstance();
usb.connect();

```

可见，内部类是非常灵活的结构。

### ✧ 18.3 static 定义内部类

内部类使用 `static` 定义，就变为“外部类”，独立于类的结构，相当于独立的类。`static` 定义的内部类只能访问外部类 `static` 的属性或方法。

```

class Outer { // 外部类
    private static final String NAME = "hikari";
    static class Inner { // 内部静态类
        private String code = "01";
        public void print() {
            // 使用外部类的私有属性
            System.out.println(this.code + "-" + Outer.NAME);
        }
    }
}

// 实例化内部静态类对象
Outer.Inner in = new Outer.Inner();
in.print();

```

如果开发中发现类名有.首先应该想到是内部类。如果可以直接实例化极有可能是内部静态类。

`static` 定义的内部类不常用，`static` 定义的内部接口较为常用。

```

interface IMessageWrap {
    // 一组static内部接口,统一管理
    static interface IMessage {
        public String getContent();
    }
    static interface IChannel {
        public boolean connect();
    }
    public static void send(IMessage msg, IChannel ch) {

```

```

        if (ch.connect()) {
            System.out.println(msg.getContent());
        } else {
            System.out.println("消息通道建立失败,无法发送消息!");
        }
    }
}

class Message implements IMessageWrap.IMessage {
    @Override
    public String getContent() {
        return "hello world";
    }
}

class Channel implements IMessageWrap.IChannel {
    @Override
    public boolean connect() {
        return false;
    }
}

IMessageWrap.send(new Message(), new Channel());

```

#### ✧ 18.4 方法中定义内部类

内部类可以定义在任意结构，如类中、方法中、代码块中。实际开发方法中定义内部类的形式较多。

```

class Outer { // 外部类
    private static final String NAME = "hikari";
    public void show() {
        Date day = new Date(); // Date对象格式化转为字符串
        SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd");
        class Inner { // 内部类
            public void print() {
                // 使用外部类的私有属性和外部方法的局部变量
                System.out.println(Outer.NAME + ": " + df.format(day));
            }
        }
        // 方法中实例化内部类对象
        new Inner().print();
    }
}

new Outer().show(); // hikari: 2018-06-04

```

show()方法内部定义 Inner 类, JDK 1.8 开始, 内部类可以直接访问方法的局部变量。JDK 1.8 之前需要在方法局部变量前追加 final, 否则不能访问。

## ✧ 18.5 匿名内部类

匿名内部类是一个没有名字, 只能使用一次的内部类。

```
interface IMessage {
    public void send(String s);
    public static IMessage getInstance() { // 返回匿名内部类的实例,也是接口的实例
        return new IMessage() { // 继承外部接口的匿名内部类
            @Override
            public void send(String s) {
                System.out.println("hello, " + s + "!");
            }
        };
    }
}

IMessage.getInstance().send("hikari"); // hello, hikari!
```

# 19 Lambda 表达式

## ✧ 19.1 Lambda 表达式

JDK 1.8 开始为了简化代码的开发, 提供 Lambda 表达式, 实现函数式编程。函数式编程比较著名的语言有: haskell、Scala, 利用函数式编程可以避免面向对象中一些繁琐的处理问题。

面向对象的反对者认为面向对象设计过于复杂, 比如上面匿名内部类使用 Lambda 表达式改写为:

```
interface IMessage {
    public void send(String s);
    public static IMessage getInstance() { // 返回匿名内部类的实例,也是接口的实例
        return s -> { // Lambda表达式
            System.out.println("hello, " + s + "!");
        };
    }
}
```

要使用 Lambda 表达式, 需要满足 SAM(Single Abstract Method), 即只有一个抽象方法。

如果一个接口只有一个抽象方法(可以有非抽象方法)称为函数式接口(Functional Interface), 其可以转换为 Lambda 表达式。

为了区分, 可以给函数式接口添加注解:

```
@FunctionalInterface
interface IMessage { // ... }
```

添加了注解的函数式接口如果定义多个抽象方法, 编译会报错。



```
@FunctionalInterface
interface MyMath { // 函数式接口
    public int add(int x, int y);
}

MyMath math = (x, y) -> x + y;
System.out.println(math.add(23, 45));
```

```
@FunctionalInterface
interface Filter<T> { // 函数式接口
    public boolean test(T t);
}

public class JavaDemo {
    public static void main(String[] args) {
        ArrayList<Integer> arr = new ArrayList<>();
        for (int i = 1; i < 20; i++) {
            arr.add(i);
        }
        ArrayList<Integer> lst = filter(arr, x -> x % 3 == 0);
        System.out.println(lst); // [3, 6, 9, 12, 15, 18]
    }
    private static ArrayList<Integer> filter(ArrayList<Integer> arr,
        Filter<Integer> f) {
        ArrayList<Integer> lst = new ArrayList<>();
        for (Integer i : arr) {
            if (f.test(i)) {
                lst.add(i);
            }
        }
        return lst;
    }
}
```

## ✧ 19.2 方法引用

JDK 1.8 提供方法的引用，不同方法名可以引用同一个方法。

- 1) 引用静态方法：类名::静态方法名；
- 2) 引用某个实例的方法：实例对象::普通方法；
- 3) 引用特定类型的方法：特定类::普通方法；
- 4) 引用构造方法：类名::new。

### ① 引用静态方法，如 String 类的 valueOf()

```
@FunctionalInterface
interface IFunction<P, R> {
```

```

    public R change(P p);
}

    IFunction<Integer, String> f = String::valueOf;
    // change()方法引用String.valueOf()方法
    System.out.println(f.change(123).length()); // 3

```

② 引用实例对象的方法，如 String 对象的 toUpperCase()方法

```

@FunctionalInterface
interface IFunction<R> {
    public R upper();
}

    IFunction<String> f = "hikari"::toUpperCase;
    System.out.println(f.upper()); // HIKARI

```

③ 引用指定类中的方法，如 String 类的 compareTo()方法，这是一个普通方法，需要实例化对象，但只想引入方法，则可以使用特定类引入处理。

```

@FunctionalInterface
interface IFunction<P> {
    public int compare(P p1, P p2);
}

    IFunction<String> f = String::compareTo;
    System.out.println(f.compare("maki", "rin")); // -5

```

④ 引用构造方法

```

@FunctionalInterface
interface IFunction<R> {
    public R create(String s, int n);
}

    IFunction<Person> f = Person::new;
    System.out.println(f.create("hikari", 25)); // name: hikari, age: 25

```

方法引用更多情况也只是弥补对于引用的支持功能。

## ✧ 19.3 内建函数式接口

java.util.function 包提供了许多函数式接口，可以直接拿来使用。

① 功能型函数式接口：有接收参数有返回值

```

@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}

```

如字符串的 `startsWith()`方法:

```
Function<String, Boolean> f = "hikari"::startsWith;
System.out.println(f.apply("hi")); // true
```

② 消费型函数式接口: 有接收参数无返回

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

如 `System.out.println()`:

```
Consumer<String> c = System.out::println;
c.accept("hikari");
```

③ 供给型函数式接口: 无参数有返回

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

如字符串的 `toUpperCase()`方法:

```
Supplier<String> s = "hikari"::toUpperCase;
System.out.println(s.get()); // HIKARI
```

④ 断言型函数式接口: 进行判断处理

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

如字符串的 `equalsIgnoreCase()`方法:

```
Predicate<String> p1 = "hikari"::equalsIgnoreCase;
System.out.println(p1.test("hIkari")); // true
Predicate<Integer> p2 = x -> x % 3 == 0; // 还能这么用
System.out.println(p2.test(23)); // false
```

## 20180605

### 20 链表

链表本质是对象的动态数组。传统数组最大缺点的长度固定。实际开发传统数组应用有限, 依赖于索引的控制。对于存入后不需要修改的数据用数组存储没问题; 但很多情况要随时修改、增加、删除数据, 这就需要灵活保存数据, 比如自己实现可以动态扩容的数据结构。

#### ✧ 20.1 链表瞎扯

链表使用引用的逻辑关系实现类似数组的数据操作。每个结点存储数据和一个指向下一个结点的指针 `next`, 所有结点像火车一样串起来。一个结点可以定义

为 Node 类，还需要一个 LinkedList 类管理所有结点的引用关系。没必要将 Node 类暴露给客户端，因为客户端处理很多 Node 麻烦，而且用户关系的只是数据的存储和获取。

接口 LinkedList 定义链表的抽象方法，实现规则；SingleLinkedList 实现 LinkedList 接口；将 Node 类定义为内部类。

如果想定义单链表、循环链表、双向链表等，还是要把 Node 类拿出来...

## ✧ 20.2 链表实现

定义链表接口 LinkedList:

```
public interface LinkedList<E> {  
    public boolean isEmpty(); // 判断链表是否为空  
    public int length(); // 获取链表长度  
    public boolean contains(E val); // 判断是否含有元素  
    public ArrayList<E> toArrayList(); // 链表转为列表  
    public void traverse(); // 遍历  
    public void append(E e); // 尾插  
    public void prepend(E e); // 头插  
    public E pop(); // 头删  
    public E get(int index); // 获取指定索引的元素  
    public void set(int index, E val); // 修改指定索引的元素  
    public void remove(E val); // 删除指定元素  
    public void removeAll(E val); // 删除所有指定元素  
}
```

单链表类:

```
public class SingleLinkedList<E> implements LinkedList<E> {  
    private class Node { // 单个结点,内部类  
        private E val;  
        private Node next;  
        public Node(E val) {  
            this.val = val;  
        }  
        // 不写setter和getter是因为外部类也能访问内部类私有属性和方法  
    }  
    // *****分隔线*****  
    private Node head;  
    public SingleLinkedList() {}  
}
```

SingleLinkedList 类需要覆写的方法:

① isEmpty()和 contains()

```
@Override  
public boolean isEmpty() { // 判断是否为空  
    return this.head == null;  
}
```

```

    }

    @Override
    public boolean contains(E val) {
        for (Node p = this.head; p != null; p = p.next) {
            if (p.val.equals(val)) { // 使用equals()判断内容
                return true;
            }
        }
        return false;
    }
}

```

## ② 获取链表长度

```

    @Override
    public int length() {
        int cnt = 0; // 计数器
        for (Node p = this.head; p != null; p = p.next) {
            cnt++;
        }
        return cnt;
    }
}

```

## ③ 链表遍历

```

    @Override
    public ArrayList<E> toArrayList() { // 转为ArrayList
        ArrayList<E> lst = new ArrayList<>();
        for (Node p = this.head; p != null; p = p.next) {
            lst.add(p.val);
        }
        return lst;
    }

    @Override
    public void traverse() { // 遍历,打印ArrayList
        System.out.println(this.toArrayList());
    }

    @Override
    public String toString() { // 使用->拼接
        StringBuffer sb = new StringBuffer();
        for (Node p = this.head; p != null; p = p.next) {
            sb.append(p.val + "->");
        }
        sb.append("null");
    }
}

```

```

        return sb.toString();
    }

```

#### ④ 插入元素

```

@Override
public void append(E e) { // 尾插法
    if (e == null) { // 插入元素为null什么也不做
        return;
    }
    Node node = new Node(e);
    if (this.isEmpty()) { // 链表为空,直接设为头结点
        this.head = node;
        return;
    }
    Node p = this.head;
    while (p.next != null) {
        p = p.next;
    }
    p.next = node;
}

@Override
public void prepend(E e) { // 头插法
    if (e == null) {
        return;
    }
    Node node = new Node(e);
    node.next = this.head;
    this.head = node;
}

```

链表的头插法时间复杂度为  $O(1)$ ，尾插法为  $O(n)$ ，自然是头插法常用。

#### ⑤ 批量插入

```

public SingleLinkedList(E[] arr) { // 构造方法传入数组批量插入
    this.build(arr);
}

public void build(E[] arr) { // 重新构造链表, 原链表舍弃
    this.head = null;
    for (E e : arr) {
        this.append(e);
    }
}

```

## ⑥ 删除元素

```
@Override
public E pop() { // 头删法
    if (this.isEmpty()) {
        return null;
    }
    Node p = this.head;
    this.head = p.next;
    return p.val;
}

@Override
public void remove(E val) { // 删除第一个指定元素
    if (this.isEmpty()) {
        return;
    }
    if (this.head.val.equals(val)) {
        this.head = this.head.next;
        return;
    }
    // 快慢指针法遍历
    for (Node p = this.head.next, pre = this.head; p != null; pre = p, p =
p.next){
        if (p.val.equals(val)) {
            pre.next = p.next;
            return;
        }
    }
}

@Override
public void removeAll(E val) { // 删除所有指定元素
    while (this.contains(val)) {
        this.remove(val);
    }
}
```

## ⑦ 根据索引获取或修改元素

```
@Override
public E get(int index) { // 获取指定索引的数据
    if (index < 0 || index >= this.length()) {
        return null;
    }
    Node p = this.head;
```

```

        for (int cnt = 0; cnt < index; cnt++) {
            p = p.next;
        }
        return p.val;
    }

    @Override
    public void set(int index, E val) { // 修改指定索引的数据
        if (index < 0 || index >= this.length()) {
            return;
        }
        Node p = this.head;
        for (int cnt = 0; cnt < index; cnt++) {
            p = p.next;
        }
        p.val = val;
    }
}

```

测试:

```

LinkedList<Integer> ll = new SingleLinkedList<Integer>(new Integer[]
{ 23, 43, 56, 12, 23 });
System.out.println(ll); // 23->43->56->12->23->null
ll.append(88);
System.out.println(ll); // 23->43->56->12->23->88->null
ll.removeAll(23);
System.out.println(ll); // 43->56->12->88->null

```

链表主要内容就是链表的遍历，思想是工作指针后移，有时头结点需要单独处理。注意不要访问 null 的数据，防止空指针异常。

练习：超市购物车

买东西，东西放到购物车，最后到收银台结账。

1) 定义商品标准

```

public interface IGoods { // 商品标准
    public String getName();
    public int getPrice();
}

```

2) 购物车处理标准

```

public interface IShopCart { // 购物车标准
    public void add(IGoods g); // 添加商品
    public void delete(IGoods g); // 删除商品
    public ArrayList<IGoods> getAll(); // 获取全部商品
}

```



### 3) 定义购物车实现类

```
public class ShopCart implements IShopCart {  
    // 使用链表存储  
    private LinkedList<IGoods> ll = new SingleLinkedList<>();  
    @Override  
    public void add(IGoods g) {  
        ll.prepend(g);  
    }  
    @Override  
    public void delete(IGoods g) {  
        ll.remove(g);  
    }  
    @Override  
    public ArrayList<IGoods> getAll() {  
        return ll.toArray();  
    }  
}
```

### 4) 定义收银台

收银台也可以定义为接口，但就会变得麻烦，简化处理，只定义一个类。

```
public class Cashier { // 收银台  
    private IShopCart shopcart;  
    public Cashier(IShopCart shopcart) {  
        this.shopcart = shopcart;  
    }  
    public int totalPrice() { // 获取购物车商品总价  
        int s = 0;  
        for (IGoods g : this.shopcart.getAll()) {  
            s += g.getPrice();  
        }  
        return s;  
    }  
    public int totalCount() { // 获取购物车商品数量  
        return this.shopcart.getAll().size();  
    }  
}
```

### 5) 定义商品类

可以定义很多商品类，处理类似，所以可以先使用抽象类实现 IGoods 接口，再使用子类继承。此处简化处理，只定义一个商品 Book 类。

```
public class Book implements IGoods {  
    private String name;  
    private int price;  
    public Book(String name, int price) {
```

```

        this.name = name;
        this.price = price;
    }
    @Override
    public String getName() {
        return this.name;
    }
    @Override
    public int getPrice() {
        return this.price;
    }
    @Override
    public String toString() {
        return "Book [name=" + name + ", price=" + price + "]";
    }
    public boolean equals(Object obj) {
        // equals()需要覆写,否则商品无法删除
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof Book)) {
            return false;
        }
        if (obj == this) {
            return true;
        }
        Book b = (Book) obj;
        return this.name.equals(b.getName()) && this.price == b.getPrice();
    }
}

```

## 6) 测试

```

IShopCart cart = new ShopCart();
cart.add(new Book("Flask web开发", 50));
cart.add(new Book("MySQL 从删库到跑路", 99));
cart.add(new Book("Java 从入门到弃坑", 66));
cart.delete(new Book("MySQL 从删库到跑路", 99));
Cashier cas = new Cashier(cart);
System.out.println("总价: " + cas.totalPrice() + ", 数量: " +
cas.totalCount()); // 总价: 116, 数量: 2

```

## 21 Eclipse

Java 开发主流 IDE 是 eclipse, 一般下载免安装版 [package](#), Java 有 EE 版和普通版, 推荐使用 Java EE, 目前版本号为 oxygen 3a。除了 Java, eclipse 还可以用

来开发 C/C++、PHP、JavaScript 等。

建议只使用一个工作区，所有项目都扔在里面。

如果界面乱了，将工作区里面全部删了，重启后界面就可以重置。

创建的项目目录有两个子目录：

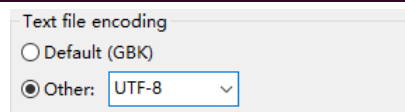
src/：保存所有.java 源文件；

bin/：保存所有编译后的.class 程序文件。

### ✧ 21.1 eclipse 编码问题

eclipse 默认使用系统的编码 GBK，需要修改为 **UTF-8**：

Window->Preferences->General->Workspace->Text file encoding->Other:UTF-8



### ✧ 21.2 eclipse 常用快捷键

① **Ctrl + 1**：代码纠正提示；

② **Alt + /**：代码提示；

③ **Ctrl + Shift + ↓**：复制当前一行代码

然而屏幕反转了，看来有冲突，修改成其他按键吧

④ **Ctrl + /**：单行注释；

⑤ **Ctrl + Shift + /**：多行注释；不建议使用

⑥ **Ctrl + Shift + F**：格式化代码；

⑦ **Ctrl + Shift + O**：自动导入需要的包；

⑧ **Ctrl + Shift + L**：查看快捷键；

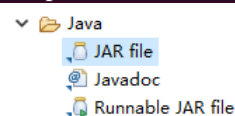
eclipse 代码生成操作：Source->GenerateXXX

生成构造方法、setter、getter 方法、toString()、equals()等

### ✧ 21.3 JAR 文件

① 导出为.jar 文件

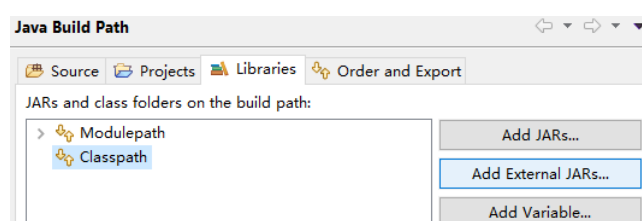
File->Export->Java->JAR file->选择要导出的文件->设置.jar 文件名称



② 导入.jar 文件

项目开发可能需要.jar 文件的导入，要针对某个项目上进行构建路径

项目右键->Properties->Java Build Path->Libraries->Classpath->Add External JARs...



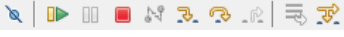
## ✧ 21.4 eclipse 配置初始化参数


类需要执行一次才能配置


右键->Run as->Run Configuration->Arguments


## ✧ 21.5 Debug


需要设置断点(Break Point), 点击蜘蛛🕸使用调试模式启动程序。

主要通过几个控制工具调试: 

F5 : 单步跳入(Step Into), 进入代码中执行程序观察;

F6 : 单步跳过(Step Over), 只关心结果, 不关心里面执行什么; 观察程序表面执行;

F7 : 单步返回(Step Return), 进入后如果不想看了, 点击直接返回;

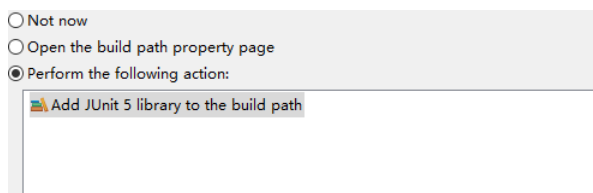
F8 : 恢复执行(Resume): 取消断点的影响, 程序正常执行完毕。

// 调试太麻烦了, print 大法好!

## ✧ 21.6 JUnit 测试工具

最流行的测试是用例测试, JUnit 就是用例测试的工具。

点击要测试的类, 右键->New->JUnit Test Case->修改包名和类名或使用默认->Next->选择要测试的方法->Finish



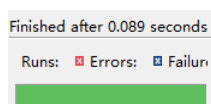
JUnit 是第三方的组件包, 需要在 Java Build Path 配置。

编写测试实例:

```
class JavaDemoTest {  
    @Test  
    void testAdd() {  
        TestCase.assertEquals(JavaDemo.add(10, 20), 30);  
        TestCase.assertEquals(JavaDemo.add(34, 43), 77);  
    }  
}
```

实际测试用例是由专门的测试人员编写。

测试结果:



绿条为测试成功, 红条为测试失败。