JAVA 培训大纲

JavaTM 2 Platform Standard Edition

志诚（北京）科技
www.zc13.com

前言

目 录

第一章 JAVA 语言简介 8
JAVA 简介 8
JAVA 语言的特性 8
第二章 JDK 的下载-安装-配置 11
JDK 简介 11
JDK 下载 11
JDK 的安装 11
JDK 的配置 12
第三章 JAVA 语言的基础语法及面向对象 15
标识符 15
关键字 15
Java 常量 15
Java 变量 16
Java 变量分类 16
Java 基本数据类型 17
Java 运算符 21
break & continue 语句 31
this 关键字 32
static 关键字 33
package 和 import 关键字 33
第四章 JAVA 的面向对象 34
Java 的方法 34
Java 的成员变量 34
Java 的类 35
Java 的引用和引用类型 35
重载 36
继承 36
访问控制 39

重写.....39
super 关键字.....43
多态.....44
抽象类.....49
final 关键字.....50
interface.....54
构造器（构造函数、构造方法）.....58
内部类.....60
匿名内部类.....61
第五章　容器.....64
数组.....64
集合框架.....67
Collection 接口.....69
AbstractCollection 类.....70
Iterator 接口.....71
List 接口.....71
AbstractList 和 AbstractSequentialList 抽象类.....73
LinkedList 类.....73
ArrayList 类.....74
RandomAccess 接口.....74
Set 接口.....74
Hash 表.....75
Comparable 接口.....75
Comparator 接口.....75
SortedSet 接口.....76
AbstractSet 抽象类.....76
HashSet 类.....77
TreeSet 类.....77
LinkedHashSet 类.....77
Map 接口.....78

Map.Entry 接口 79
SortedMap 接口 79
AbstractMap 抽象类 80
HashMap 类 80
TreeMap 类 81
LinkedHashMap 类 81
WeakHashMap 类 82
IdentityHashMap 类 82
泛型 83
第六章 异常 85
什么是错误 85
抛出异常 85
捕获异常 85
异常的分类 85
异常的程序结构 86
异常的声明 87
自定义异常 87
第七章 I/O 89
输入/输出流的分类 89
InputStream 89
OutputStream 90
Reader 90
Writer 91
节点流 91
处理流 96
转换流 98
数据流 101
Print 流 102
Object 流 104
第八章 线程 106

什么是线程 106
线程和进程的区别 106
线程的状态转换 107
线程中的主要方法 109
线程同步 117
第九章　网络通信 122
计算机网络 122
网络通信协议 122
网络通信接口 123
通信协议的分层 123
IP 协议 123
TCP 协议 125
Socket 126
第十章　GUI 130
AWT 130
Component & Container 130
Frame 131
Panel 132
布局管理器 133
事件 139
附录 I 141
程序应该有良好的格式 141
eclipse 的使用 141
Eclipse 快捷键 143

Part I

Java 2 Standard Edition

第一章　JAVA 语言简介

JAVA 简介

Java 是由 Sun Microsystems 公司于 1995 年 5 月推出的 Java 程序设计语言（以下简称 Java 语言）和 Java 平台的总称。用 Java 实现的 HotJava 浏览器（支持 Java applet）显示了 Java 的魅力：跨平台、动态的 Web、Internet 计算。从此，Java 被广泛接受并推动了 Web 的迅速发展，常用的浏览器现在均支持 Java applet。另一方面，Java 技术也不断更新。

Java 平台由 Java 虚拟机（Java Virtual Machine）和 Java 应用编程接口（Application Programming Interface、简称 API）构成。Java 应用编程接口为 Java 应用提供了一个独立于操作系统的标准接口，可分为基本部分和扩展部分。在硬件或操作系统平台上安装一个 Java 平台之后，Java 应用程序就可运行。现在 Java 平台已经嵌入了几乎所有的操作系统。这样 Java 程序可以只编译一次，就可以在各种系统中运行。Java 应用编程接口已经从 1.1x 版发展到 1.2 版。目前常用的 Java 平台基于 Java1.4，最近版本为 Java1.7。

Java 分为三个体系 JavaSE(Java2 Platform Standard Edition)，JavaEE(Java 2 Platform,Enterprise Edition)，JavaME(Java 2 Platform Micro Edition)。

JAVA 语言的特性

Java 语言是简单的

Java 语言的语法与 C 语言和 C++语言很接近，使得大多数程序员很容易学习和使用 Java。另一方面，Java 丢弃了 C++ 中很少使用的、很难理解的、令人迷惑的那些特性，如操作符重载、多继承、自动的强制类型转换。特别地，Java 语言不使用指针，并提供了自动的废料收集，使得程序员不必为内存管理而担忧。

Java 语言是一个面向对象的

Java 语言提供类、接口和继承等原语，为了简单起见，只支持类之间的单继承，但支持接口之间的多继承，并支持类与接口之间的实现机制（关键字为 implements）。Java 语言全面支持动态绑定，而 C++ 语言只对虚函数使用动态绑定。总之，Java 语言是一个纯的面向对象程序设计语言。

Java 语言是分布式的

Java 语言支持 Internet 应用的开发，在基本的 Java 应用编程接口中有一个网络应用编程接口（java.net），它提供了用于网络应用编程的类库，包括 URL、URLConnection、Socket、ServerSocket 等。Java 的 RMI(远程方法激活)机制也是开发分布式应用的重要手段。

Java 语言是健壮的

Java 的强类型机制、异常处理、废料的自动收集等是 Java 程序健壮性的重要保证。对指针的丢弃是 Java 的明智选择。Java 的安全检查机制使得 Java 更具健壮性。

Java 语言是安全的

Java 通常被用在网络环境中，为此，Java 提供了一个安全机制以防恶意代码的攻击。除了 Java 语言具有的许多安全特性以外，Java 对通过网络下载的类具有一个安全防范机制（类 ClassLoader），如分配不同的名字空间以防替代本地的同名类、字节代码检查，并提供安全管理机制（类 SecurityManager）让 Java 应用设置安全哨兵。

Java 语言是体系结构中立的

Java 程序（后缀为 java 的文件）在 Java 平台上被编译为体系结构中立的字节码格式（后缀为 class 的文件），然后可以在实现这个 Java 平台的任何系统中运行。这种途径适合于异构的网络环境和软件的分发。

Java 语言是可移植的

这种可移植性来源于体系结构中立性，另外，Java 还严格规定了各个基本数据类型的长度。Java 系统本身也具有很强的可移植性，Java 编译器是用 Java 实现的，Java 的运行环境是用 ANSI C 实现的。

Java 语言是解释型的

如前所述，Java 程序在 Java 平台上被编译为字节码格式， 然后可以在实现这个 Java 平台的任何系统中运行。在运行时，Java 平台中的 Java 解释器对这些字节码进行解释执行，执行过程中需要的类在联接阶段被载入到运行环境中。

Java 是高性能的

与那些解释型的高级脚本语言相比，Java 的确是高性能的。事实上，Java 的运行速度随着 JIT(Just-In-Time)编译器技术的发展越来越接近于 C++。

Java 语言是多线程的

在 Java 语言中，线程是一种特殊的对象，它必须由 Thread 类或其子（孙）类来创建。通常有两种方法来创建线程：其一，使用型构为 Thread(Runnable) 的构造子将一个实现了 Runnable 接口的对象包装成一个线程，其二，从 Thread 类派生出子类并重写 run 方法，使用该子类创建的对象即为线程。值得注意的是 Thread 类已经实现了 Runnable 接口，因此，任何一个线程均有它的 run 方法，而 run 方法中包含了线程所要运行的代码。线程的活动由一组方法来控制。Java 语言支持多个线程的同时执行，并提供多线程之间的同步机制（关键字为 synchronized）。

Java 语言是动态的

Java 语言的设计目标之一是适应于动态变化的环境。Java 程序需要的类能够动态地被载入到运行环境，也可以通过网络来载入所需要的类。这也有利于软件的升级。另外，Java 中的类有一个运行时刻的表示，能进行运行时刻的类型检查。

第二章　JDK 的下载-安装-配置

JDK 简介

JDK（Java Development Kit）也叫 Java SDK，是 SUN 公司针对 Java 开发人员发布的。

JDK 包含的基本组件包括：

javac – 编译器，将源程序转成字节码

jar – 打包工具，将相关的类文件打包成一个文件

javadoc – 文档生成器，从源码注释中提取文档

jdb – debugger，查错工具

JDK 下载

JDK 在 SUN 的网站上提供下载。登陆 http://java.sun.com/javase/downloads/index.jsp，在 Downloads 里选择需要的版本，点击 download，选择运行平台和语言，即可下载。目前最新版本是 Java SE Development Kit 6update13，如果想下载之前的版本，可以在 http://java.sun.com/javase/downloads/index.jsp 点击 Previous Releases，来选择以前的版本。

JDK 的安装

JDK 在 windows 下的安装很简单，运行已下载的 JDK 安装文件，默认安装即可完成。JDK 安装完成后会继续安装 JRE（Java Runtime Environment），JRE 是为其他一些 Java 程序提供运行环境。JDK 中包含了 JRE，所以实际上安装 2 个 JRE。

JDK 的配置

JDK 安装完成后，在 cmd 输入命令 javac，会提示说不是内部或外部命令，也不是可运行的程序和批处理文件。Javac 是 java 的编译命令，输入 javac 实际上是运行安装目录的 bin 目录下的 javac.exe。在 windows 下运行命令需要指定命令的路径，进入 JDK 的安装目录的 bin 目录下运行 javac，出现如下信息说明 JDK 已经安装成功了。

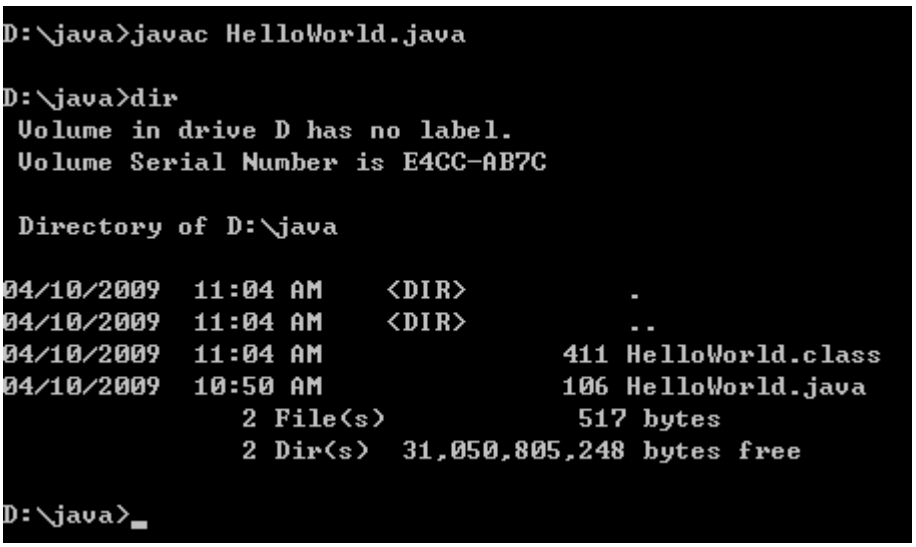
```
D:\>cd D:\Program Files\Java\jdk1.5.0_16\bin

D:\Program Files\Java\jdk1.5.0_16\bin>javac
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
  -nowarn                    Generate no warnings
  -verbose                   Output messages about what the compiler is doing
  -deprecation               Output source locations where deprecated APIs are u
sed
  -classpath <path>          Specify where to find user class files
  -cp <path>                 Specify where to find user class files
  -sourcepath <path>         Specify where to find input source files
  -bootclasspath <path>      Override location of bootstrap class files
  -extdirs <dirs>            Override location of installed extensions
  -endorseddirs <dirs>       Override location of endorsed standards path
  -d <directory>             Specify where to place generated class files
  -encoding <encoding>       Specify character encoding used by source files
  -source <release>          Provide source compatibility with specified release

  -target <release>          Generate class files for specific VM version
  -version                   Version information
```

创建一个文件夹用于存放 java 文件（以 D:\JAVA 为例），新建一个 HelloWorld.java 文件，内容如下：

```
public class HelloWorld{

    public static void main(String[] args){

        System.out.println("HelloWorld");

    }
```

}

JAVA 文件要运行首先要编译成.class 文件，在由解释器解释运行。HelloWorld.java 文件可以用 javac 命令编译。需要完整的命令路径才能够正确执行。为了不用每次都写完整的路径，可以在环境变量中配置。打开我的电脑》属性》高级》环境变量，在系统变量 path 中加入 JDK 的 bin 目录，和之前的变量用分号隔开，然后确定。重新打开 cmd，直接使用 javac 命令便可成功编译。



编译完成后在 java 目录下多了一个 HelloWorld.class 文件。

然后用 java 命令运行。

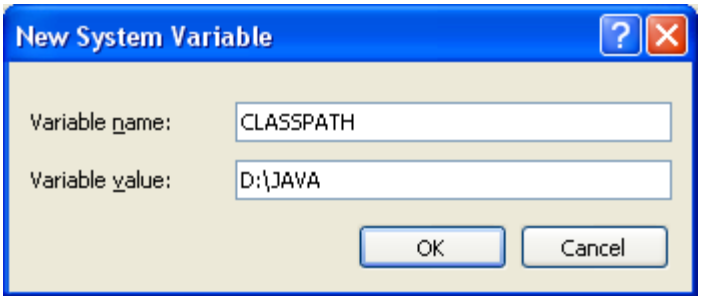
运行成功。

如果没有在 java 目录下执行 java 命令，将出现如下信息：

需要加入文件路径

```
D:\>java -classpath d:\java HelloWorld
HelloWorld
```

也可以在环境变量 classpath 中加入 D:\java



点击确定。重新打开 cmd，直接输入 java 命令便可执行。

```
C:\Documents and Settings\Administrator>java HelloWorld
HelloWorld
```

第三章　JAVA 语言的基础语法及面向对象

标识符

Java 语言对各种变量、方法和类等要素命名时使用的字符序列成为标识符。

标识符的命名规则：

（1）标识符有字符、下划线、美元符号或数字组成

（2）标识符应以字符、下划线或美元符号开头

（3）Java 语言的标识符大小写敏感，长度无限制

（4）不能与 Java 关键字重复

最好能做到见名知意。

关键字

Java 语言中一些具有特定含义，用作专门用途的字符串称为关键字。Java 语言所有的关键字都是小写。goto 和 const 虽然未被使用，但是仍被 Java 关键字保留。

Java 常量

整型常量：123

实型常量：3.14

字符常量：’a’

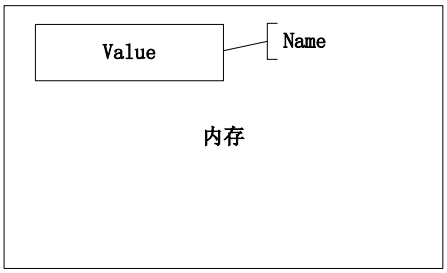
字符串常量：”abc”

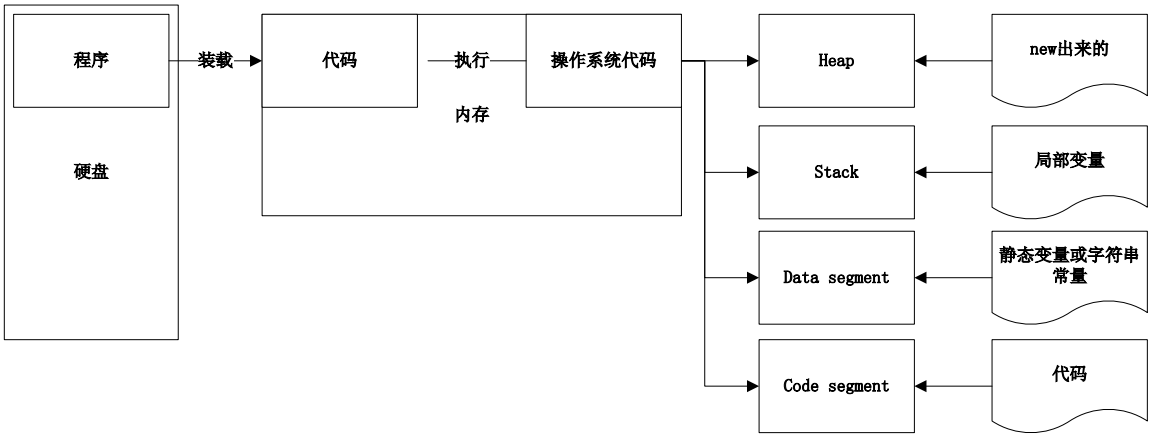
逻辑常量：true、false

值不可以再改变的变量：final 关键字

Java 变量

Java 变量是程序中最基本的存储单元，包括变量名、变量类型和作用域。Java 程序中的每一个变量都属于特定的数据类型，使用前必须先声明，赋值后才能使用。从本质上讲，变量其实是内存中的一小块区域，使用变量名字来访问这块区域。





Java 变量分类

按被声明的位置划分：

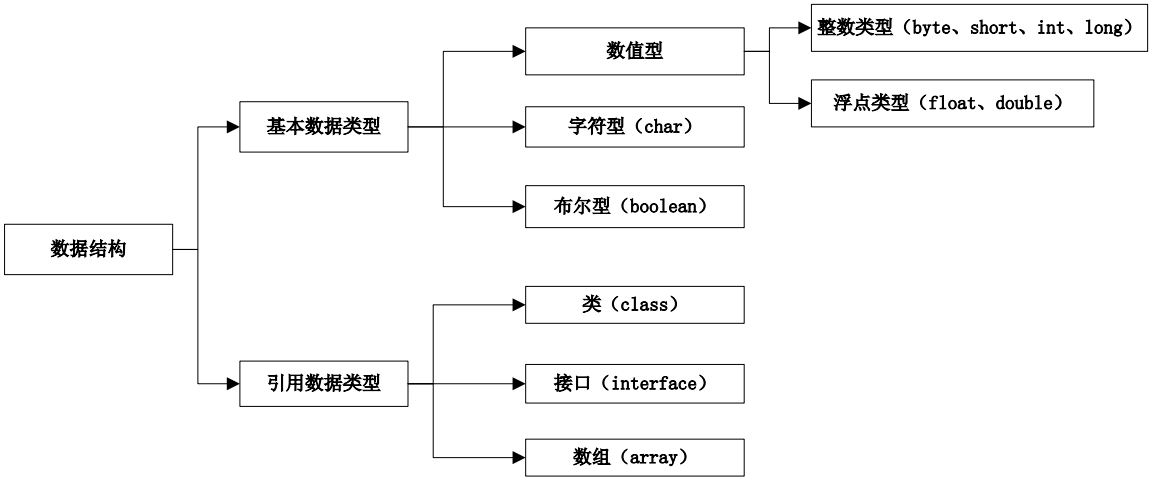
（1）局部变量：方法或语句块内部定义的变量

（2）成员变量：方法外部，类的内部定义的变量

按所属数据类型划分：

（1）基本数据类型变量

（2）引用数据类型变量



Java 基本数据类型

Java 中定义了 4 类 8 种基本数据类型：

（1）整数类型：byte、short、int、long

（2）浮点类型：float、double

（3）字符类型：char

（4）布尔类型：boolean

Java 语言中各整数类型有固定的范围和长度，以此来区分 4 种类型，并且不受操作系统的影响，以保证移植性。

Java 语言常用的 3 中表示：

十进制：如 10

八进制：如 010

十六进制，如：0x10

Java 语言整型常量默认是 int 型，后加’l’或’L’表示 long 型。

类型	长度	范围
byte	1 字节	-128~127
short	2 字节	-2^{15}~2^{15}-1
int	4 字节	-2^{31}~2^{31}-1
long	8 字节	-2^{63}~2^{63}-1

浮点类型也有自身固定的长度和范围，并且不受操作系统的影响。

浮点类型表示方法有两种：

（1）十进制表示：3.14

（2）科学计数法：3.14E2

Java 语言中浮点类型默认是 double 类型，后加’F’表示 float 类型。

类型	长度	范围
float	4	-3.403E38~3.403E38
double	8	-1.798E308~1.798E308

boolean 类型用于逻辑运算、流程控制。boolean 只能取 true 和 false。

char 类型表示字符，常用单引号括起来的单个字符表示。如：char c = ‘a’;Java 中采用 Unicode 编码，Unicode 有两种编码方式：UTF-8 和 UTF-16。Java 语言采用的是 UTF-16，每个字符占两个字节。Java 语言还可以使用转义字符’\’来将后边的字符变成其他含义，如 char c = ‘\n’;。

基本数据类型之间的转换

Java 语言中基本数据类型之间可以相互转换，需要遵守以下规则：

（1）除 boolean 以外，其他类型之间可以自由转换

（2）容量小的类型自动转换成容量大的数据类型

（3）容量由小到大依次为：byte、short、char<int<long<float<double

（4）byte、short、char 之间不会转换，运算时都转换成 int

（5）容量大的数据类型转换成容量小的数据类型需要加强制转换符，但会损失精度或溢出

（6）多种数据类型运算的时候，系统先将所有数据转换成容量最大的数据类型再进行运算

练习

```
public class Test {
    public static void main(String arg[]) {
        int i1 = 123;
        int i2 = 456;
        double d1 = (i1+i2)*1.2;//系统将转换为 double 型运算
        float f1 = (float)((i1+i2)*1.2);//需要加强制转换符
        byte b1 = 67;
```


```
        byte b2 = 89;
        byte b3 = (byte)(b1+b2);//系统将转换为 int 型运算，需
                                //要强制转换符
        System.out.println(b3);
        double d2 = 1e200;
        float f2 = (float)d2;//会产生溢出
        System.out.println(f2);

        float f3 = 1.23f;//必须加 f
        long l1 = 123;
        long l2 = 30000000000L;//必须加 l
        float f = l1+l2+f3;//系统将转换为 float 型计算
        long l = (long)f;//强制转换会舍去小数部分（不是四舍五入）
    }
}

public class Test{
    public static void main(String[] args){
        int i = 1,j;
```

```
        float f1 = 0.1;

        float f2 = 123;

        long l1 = 12345678,l2 = 8888888888;

        double d1 = 2e20,d2 = 123;

        byte b1 = 1,b2 = 2,b3 = 129;

        j = j+10;

        i = i/10;

        i = i*0.1;

        char c1 = 'a',c2 = 125;

        byte b = b1-b2;

        char c = c1+c2-1;

        float f3 = f1+f2;

        float f4 = f1+f2*0.1;

        double d = d1*i+j;

        float f = (float)(d1*5+d2);

    }

}
```

Java 运算符

Java 语言支持如下运算符：

（1）算数运算符：+、-、*、/、%、++、--

++、--在前表示先运算在取值，++、--在后表示先取值在运算。

（2）关系运算符：>、<、>=、<=、==、!=

（3）逻辑运算符：!、&、|、^、&&、||

a	b	!a	a&b	a|b	a^b	a&&b	a||b
true	true	false	true	true	false	true	true
true	false	false	false	true	true	false	true
false	true	true	false	true	true	false	true
false	false	true	false	false	false	false	false

（4）位运算符：&、|、^、~、>>、<<、>>>

（5）赋值运算符：=

当运算符两侧数据类型不一致时，可以用默认类型转换或使用强制类型转换原则

（6）扩展的赋值运算符：+=、-=、*=、/=

运算符	表达式	等效表达式
+=	a+=b	a=a+b
-=	a-=b	a=a-b
*=	a*=b	a=a*b
/=	a/=b	a=a/b
%=	a%=b	a=a%b

（7）字符串连接符：+

字符串运算符两侧的操作数中只要有一个是字符串类型，系统会自动将另一个操作数转换为字符串再进行运算。

```
public class Operator {

    public static void main(String[] args) {

        //算数运算符

        int i1 = 10, i2 = 20;

        int i = (i2++);

        System.out.print("i= " + i);

        System.out.println(" i2 = " + i2);

        i = (++i2);

        System.out.print("i = " + i);

        System.out.println(" i2 = " + i2);

        i = (--i1);

        System.out.print("i = " + i1);

        System.out.println(" i1 = " + i1);

        i = (i1--);

        System.out.print("i = " + i);

        System.out.println(" i1 = " + i1);

        //逻辑运算符

        boolean a, b, c;

        a = true; b = false;
```


```
        c = a & b;
        System.out.println(c);
        c = a | b;
        System.out.println(c);
        c = a ^ b;
        System.out.println(c);
        c = !a;
        System.out.println(c);
        c = a && b;
        System.out.println(c);
        c = a || b;
        System.out.println(c);
        int x = 1, y = 2;
        boolean flag1 = (x > 3) && ((x + y) > 5);//第二个操作数不再计算
        boolean flag2 = (x < 2) || ((x + y) < 6);//第二个操作数不再计算
    }
}
```

表达式

表达式是符合特定语法规则的运算符和操作数的序列，如 a、5.0 + a、（a-b）* c – 4、i < 30 && i % 10 != 0 。

表达式中操作数运算得到的结果称为表达式的值。

表达式的值的数据类型即为表达式的数据类型。

表达式的运算顺序应按照运算符的优先级的顺序进行，优先级相同的运算符按照约定的结合方向进行。

结合方向	运算符（从上到下优先级降低）
	、()、{}、;、,、
R to L	. ++、--、~、！
L to R	*、/、%
L to R	+、-
L to R	<<、>>、>>>
L to R	<、>、<=、>=、instanceof
L to R	==、!=
L to R	&
L to R	^
L to R	|
L to R	&&
L to R	||
R to L	?:
R to L	=、*=、/=、%=、+=、-=、<<=、>>=、>>>=、&=、^=、|=

条件语句

条件语句分为两种：if 语句和 switch 语句。

if 语句有一下几种形式：

（1）if

（2）if…else

（3）if…else if..

（4）if…else if…else if…

（5）if…else if…else if…else…

```
public class TestIF {
    public static void main(String[] args) {
        int i = 20;
        if(i < 20) {
            System.out.println("<20");
            System.out.println("<20");
        } else if (i < 40) {
            System.out.println("<40");
        } else if (i < 60) {
            System.out.println("<60");
        } else
```

```
                System.out.println(">=60");
            System.out.println(">=60");
        }
}
```

switch 语句的格式如下：

```
    switch（）{
            case xxx :  语句;
            case xxx :  语句;
            ……
            default :  语句;
    }
```

switch 语句一般会使用 break，防止 case 连续执行

多个 case 可以合并

defualt 可以省略，带不推荐

Java 语言的 switch 中只能接受 int 类型

```
public class TestSwitch {
    public static void main(String[] args) {
            int i = 8;
            switch(i) {
```

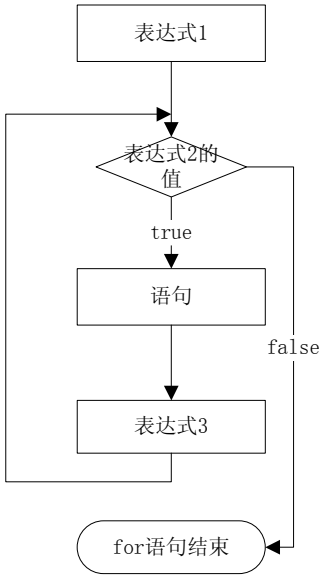

```
                case 8 :
                    System.out.println("=8");
                    break;
                case 3 :
                    System.out.println("=3");
                    break;
                case 2 :
                    System.out.println("=2");
                    break;
                case 9 :
                    System.out.println("=9");
                    break;
                default:
                    System.out.println("error");
            }
        }
    }
```

循环语句

循环语句分为 for 循环、while 循环和 do 循环。

for 循环格式如下：

```
for（表达式 1; 表达式 2; 表达式 3）{
    语句;
}
```




```
public class Sum {
    public static void main(String[] args) {
        int result = 0;
        for(int i=1; i<=99; i++) {
            result += i;
        }
        System.out.println("result=" + result);
    }
```

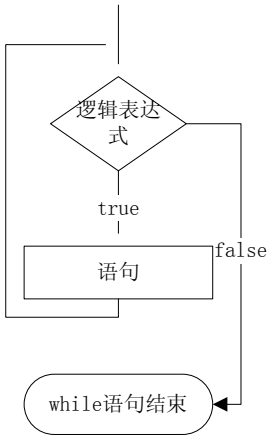

}

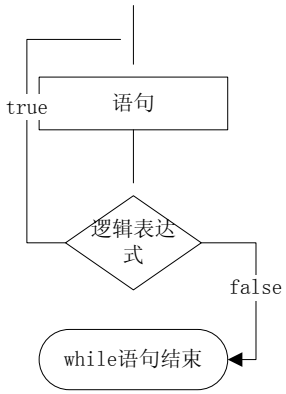
while 语句格式化如下：

```
while（逻辑表达式）{
    语句;
}
```

do-while 语句格式如下：

```
do{
    语句;
}while(逻辑表达式);
```






```
public class TestWhile {
    public static void main(String[] args) {
        int i = 0;
        while(i < 10) {
            System.out.println(i);
            i++;
        }
```

```
        i = 0;
        do {
            i++;
            System.out.println(i);
        } while(i < 10);
    }
}
```

break & continue 语句

break 语句用于终止某个程序块的执行。用在循环体中可以强制跳出循环。

Continue 语句用于终止某次循环，开始下一次循环。

```
public class TestBreak {
    public static void main(String args[]) {
        int num = 4;
        for (int i = 1; i <= 10; i++) {
            //当 i 等于 num 时，退出循环
            if (i == num) break;
            System.out.println(" i= " + i);
        }
```


```
        }
    }
    public class TestContinue {
        public static void main(String[] args) {
            int num = 4;
            for(int i=1;i<5;i++){
                //当 i 等于 4 跳出档次循环
                if (i == num) continue;
                System.out.println("i = " + i);
            }
        }
    }
```

this 关键字

在类的方法中使用 this 关键字代表使用该方法的对象的引用。

this 可以处理方法中成员变量和参数重名的问题。

this 可以看作是一个变量，值为当前对象的引用。

static 关键字

在类中使用 static 声明的变量为静态的成员变量，是该类的公共变量，在第一次使用是初始化，只有一个。

在类中使用 static 声明的方法为静态方法，在调用时，不会传递对象的引用，所以在 static 中不能访问非 static 的成员变量。

声明为 static 的成员变量可以直接访问，不需要实例化。

package 和 import 关键字

Java 引入 package 机制，提供类的多重命名空间，以解决类名的冲突问题。

package 语句必须写在 Java 文件的第一行。格式为：package cn.zc.java;

Java 文件存在于对应的目录下：cn\zc\java

当一个类中用到其他类的时候需要 import 被使用的类。例如：import java.util.*;

处于同一目录下的类不需要 import。

为保证程序能够正确执行，必须保证包的最上层目录位于 classpath 中。

第四章　JAVA 的面向对象

Java 的方法

Java 语言的方法和其他语言一样，是用来完成某种特定功能的代码片断。例如：

```
public String run(String s) {
    s = “abc”;
    return s;
}
```

其中 s 为形式参数用于接受外界的输入，传给 s 的值为实参，实参的个数、类型等需要和形参匹配。传递参数的时候，基本类型传递的是数值本身，引用类型传递的是对象的引用。return 后为执行完后需要返回给调用程序的值，类型为 String，如果没有返回值，其类型为 void。

Java 的成员变量

成员变量可以使用 Java 中任意数据类型（包括基本类型和引用类型）。

成员变量的作用域为整个类体。

在定义成员变量时可以对其进行初始化，如果没有初始化，系统会默认对其进行初始化。

成员变量类型	默认值
byte	0
short	0
int	0
long	0L
char	‘\u0000’
float	0.0F

double	0.0D
boolean	false
引用类型	null

Java 的类

Java 的类由方法和成员变量构成。例如：

```
class Person {
    //成员变量
    private String name;
    //方法
    public void setName(String name){
        name = name;
    }
}
```

Java 的引用和引用类型

Java 中除基本类型之外，其他都是引用类型。Java 中的对象就是通过引用操作的。在内存中，基本类型只占用一块内存区域，引用类型占用两块内存区域。例如：String s = new String(“abc”);

重载

重载是指一个类中可以定义有相同名字，但参数不同的多个方法。调用时，会根据不同的参数表选择对应的方法。例如：

```
class Person {
    public void sayHello(){
        System.out.println(“hello”);
    }
    Public void sayHello(String name) {
        System.out.println(“hello ” + name);
    }
}
```

继承

Java 中使用 extends 关键字实现继承机制，例如：

```
public class Student extends Person {}
```

Java 只支持单继承，不允许多继承，一个子类只能有一个基类，一个基类可以派生出多个子类。

通过继承，子类拥有基类的所有成员变量和方法，包括 peivate 的。被父类声明为 private 的变量和方法可以被子类通过继承持有，但不能使用。

```
public class TestExtends {

    public static void main(String[] args){

        Student s = new Student();
        s.getAge();

        Teacher t = new Teacher();
        t.getName();
    }
}

class Person {

    public String name;
    public int age;
    private String sid;

    public int getAge() {
```


```
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getSid() {
        return sid;
    }
    public void setSid(String sid) {
        this.sid = sid;
    }
}
```

```
class Student extends Person {

}


class Teacher extends Person{

}
```

访问控制

Java 的访问修饰符又叫权限控制符，用来限制其他对象对该类对象的访问权限。

修饰符	类内部	同一包下	子类	任何地方
private	Y			
default	Y	Y		
protected	Y	Y	Y	
public	Y	Y	Y	Y

重写

重写是指在同一范围内可使用同一个标识符来表示多个项。Java 中，可对方法进行重写，但不能对变量或运算符进行重写。方法重载可以让类以统一的方式处理不同类型的数据。Java 的方法重载，就是在类中可以创建多个方法，它们具有相同的名字，但具有不同的参数和不同的定义。调用方法时通过传递给它们的不同个数和类型的参数来决定具体使用哪个方法，是多态性的一种体现。

Java 方法重写的规则：

（1）在子类中可以重写基类中的方法。

（2）重写的方法必须和被重写的方法具有相同的名称，参数列表和返回类型。

（3）重写方法不能使用比被重写方法更严格的访问权限。

```
class Person {
    private String name;

    private int age;

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

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
```

```
        return age;
    }

    public String getInfo() {
        return "Name: " + name + "\n" + "age: " + age;
    }
}

class Student extends Person {
    private String school;

    public String getSchool() {
        return school;
    }

    public void setSchool(String school) {
        this.school = school;
    }
```


```
    public String getInfo() {
        return "Name: " + getName() + "\nage: " + getAge() + "\nschool: "
                + school;
    }
}

public class TestOverWrite {
    public static void main(String arg[]) {
        Student student = new Student();
        Person person = new Person();
        person.setName("zhangsan");
        person.setAge(25);
        student.setName("lisi");
        student.setAge(18);
        student.setSchool("new school");
        System.out.println(person.getInfo());
        System.out.println(student.getInfo());
    }
}
```

super 关键字

Java 语言中可以使用 super 来引用基类中的内容，即调用父类的构造函数。使用 super 调用有一些限制条件,不能在 super 调用中使用实例变量作为参数，而且在构造函数中,调用必须写在继承类的构造函数定义的第一行。

```
class SuperClass {


    SuperClass() {
        System.out.println("SuperClass()");
    }


    SuperClass(int n) {
        System.out.println("SuperClass(" + n + ")");
    }
}


class SubClass extends SuperClass {


    SubClass(int n) {
        super();
```


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

    SubClass() {
        super(300);
        System.out.println("SubClass()");
    }
}

public class TestSuper {
    public static void main(String arg[]) {
        //SubClass sc1 = new SubClass();
        SubClass sc2 = new SubClass(400);
    }
}
```

多态

多态是指通过发送消息给某个对象，让该对象自行决定响应何种行为，即通过将子类对象引用赋值给超类对象引用变量来实现动态方法调用。多态也叫动态绑定，是指在执行期间判断所引用的对象的实际类型，根据实际的类型调用相应的方法。当超类对象引用变量引用子类对

象时，被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但是这个被调用的方法必须是在超类中定义过的，也就是说被子类覆盖的方法。

实现多态的条件：

（1）要有继承

（2）要有重写

（3）要有基类的引用指向子类对象

```
class Book {
    private String name;

    Book(String name) {
        this.name = name;
    }

    public void getPages() {
        System.out.println("the book has 100 pages.");
    }
}

class MathBook extends Book {
```


```
    private int page;

    MathBook(String n, int c) {
        super(n);
        this.page = c;
    }

    public void getPages() {
        System.out.println("math book has 500 pages.");
    }
}

class ChineseBook extends Book {
    private int page;

    ChineseBook(String n, int c) {
        super(n);
        this.page = c;
    }
```

```
    public void getPages() {
        System.out.println("chinese book has 1000 pages.");
    }
}

class EnglishBook extends Book {
    EnglishBook() {
        super("EnglishBook");
    }

    public void getPages() {
        System.out.println("english book has 200 pages.");
    }
}

class Student {
    private String name;
```


```
    private Book book;

    Student(String name, Book book) {
        this.name = name;
        this.book = book;
    }

    public void getMyBookPages() {
        book.getPages();
    }
}

public class TestPolymoph {
    public static void main(String args[]) {
        MathBook m = new MathBook("MathBook", 500);
        ChineseBook c = new ChineseBook("ChineseBook", 1000);
        EnglishBook e = new EnglishBook();
        Student s1 = new Student("s1",m);
        Student s2 = new Student("s2", c);
```

```
        Student s3 = new Student("s3", e);
        s1.getMyBookPages();
        s2.getMyBookPages();
        s3.getMyBookPages();
    }
}
```

抽象类

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来却不是这样。并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。抽象类往往用来表征我们在对问题领域进行分析、设计中得出的抽象概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象。比如：如果我们进行一个图形编辑软件的开发，就会发现问题领域存在着圆、三角形这样一些具体概念，它们是不同的，但是它们又都属于形状这样一个概念，形状这个概念在问题领域是不存在的，它就是一个抽象概念。正是因为抽象的概念在问题领域没有对应的具体概念，所以用以表征抽象概念的抽象类是不能够实例化的。Java 中使用 abstract 关键字来修饰抽象类。使用 abstract 关键字修饰的方法称为抽象方法。抽象类具有以下特征：

（1）含有抽象方法的类必须被声明为抽象类。

（2）抽象类必须被继承。抽象方法必须被重写。

（3）抽象类不能被实例化。

（4）抽象方法只需要声明，不需要实现。

在上一节程序中，Book 类被其他类继承，getPages()方法被重写。Book 类是多个具体概念的抽象，getPages()在子类中全都被重写，父类中的 getPages()方法的实现没有被用到，那么父类中的 getPages()方法的实现是没有意义的，可将其改为抽象方法。如下：

```
abstract class Book {

    private String name;



    Book(String name) {

        this.name = name;

    }



    abstract void getPages();

}
```

Book 的子类也可以使抽象的，同样 getPages()方法不需要实现，而交由其子类来实现。

final 关键字

final 根据程序上下文环境，Java 关键字 final 有“这是无法改变的”或者“终态的”含义，它可以修饰非抽象类、非抽象类成员方法和变量。final 类不能被继承，因此 final 类的成员方法没有机会被覆盖，默认都是 final 的。final 方法不能被子类的方法覆盖。编译器在遇到调用 final 方法时候会转入内嵌机制，大大提高执行效率。final 成员变量表示常量，只能被赋值一次，赋值后值不再改变。final 不能用于修饰构造方法。父类的 private 成员方法是不能被子类方法覆盖的，因此 private 类型的方法默认是 final 类型的。

```
public class TestFinal {
```

```
    public static void main(String[] args) {
        T t = new T();
        // t.i = 8;
    }
}

final class T {
    final int i = 8;

    public final void m() {
        // j = 9;
    }
}

class TT extends T { //编译不能通过
}
public class TestFinal1 {
    private final String S = "final 实例变量 S";
```


```
private final int A = 100;

public final int B = 90;

public static final int C = 80;

private static final int D = 70;

public final int E; // final 空白,必须在初始化对象的时候赋初值

public Test3(int x) {
    E = x;
}

public static void main(String[] args) {
    Test3 t = new Test3(2);
    // t.A=101; //出错,final 变量的值一旦给定就无法改变
    // t.B=91; //出错,final 变量的值一旦给定就无法改变
    // t.C=81; //出错,final 变量的值一旦给定就无法改变
```

```
        // t.D=71; //出错,final 变量的值一旦给定就无法改变


        System.out.println(t.A);

        System.out.println(t.B);

        System.out.println(t.C); //  不推荐用对象方式访问静态字段

        System.out.println(t.D); //  不推荐用对象方式访问静态字段

        System.out.println(Test3.C);

        System.out.println(Test3.D);

        // System.out.println(Test3.E); //出错,因为 E 为 final 空白,依据不同对象值有所不同.

        System.out.println(t.E);


        Test3 t1 = new Test3(3);

        System.out.println(t1.E); // final 空白变量 E 依据对象的不同而不同

}


private void test() {

        System.out.println(new Test3(1).A);

        System.out.println(Test3.C);

        System.out.println(Test3.D);
```


```
    }

    public void test2() {
        final int a; // final 空白,在需要的时候才赋值
        final int b = 4; //  局部常量--final 用于局部变量的情形
        final int c; // final 空白,一直没有给赋值.
        a = 3;
        // a=4;  出错,已经给赋过值了.
        // b=2;  出错,已经给赋过值了.
    }
}
```

interface

接口是抽象方法和常量值的集合。接口是一种特殊的抽象类，值包含方法的声明没有具体实现。例如：

```
public interface Person {
    public static final int i = 0;
    public void speak();
}
```

多个类可以实现同一个接口。

一个类可以实现多个接口。

接口与实现类之间存在多态性。

接口可以多重实现。

接口中属性的声明默认为 public static final 的。

接口中只能定义 public 的抽象方法。

接口可以继承接口，并添加新的属性和方法。

对象的创建和使用的规则如下：

（1）必须使用 new 关键字创建对象。

（2）使用.来引用成员变量或调用对象的方法。

（3）同一个类的每个对象有不同的成员变量存储空间。

（4）同一个类的每个对象共享该类的方法。

同样可以将之前多态中的例子改写：

```
interface Book {

    public static final String name = "book";



    public void getPages();

}
```


```
class MathBook implements Book {
        private int page;

        public void getPages() {
                System.out.println("math book has 500 pages.");
        }
}

class ChineseBook implements Book {
        private int page;

        public void getPages() {
                System.out.println("chinese book has 1000 pages.");
        }
}

class EnglishBook implements Book {

        public void getPages() {
```

```
            System.out.println("english book has 200 pages.");
        }
}

class Student {
        private String name;

        private Book book;

        Student(String name, Book book) {
            this.name = name;
            this.book = book;
        }

        public void getMyBookPages() {
            book.getPages();
        }
}
```


```
public class TestPolymoph {
    public static void main(String args[]) {
        MathBook m = new MathBook();
        ChineseBook c = new ChineseBook();
        EnglishBook e = new EnglishBook();
        Student s1 = new Student("s1",m);
        Student s2 = new Student("s2", c);
        Student s3 = new Student("s3", e);
        s1.getMyBookPages();
        s2.getMyBookPages();
        s3.getMyBookPages();
    }
}
```

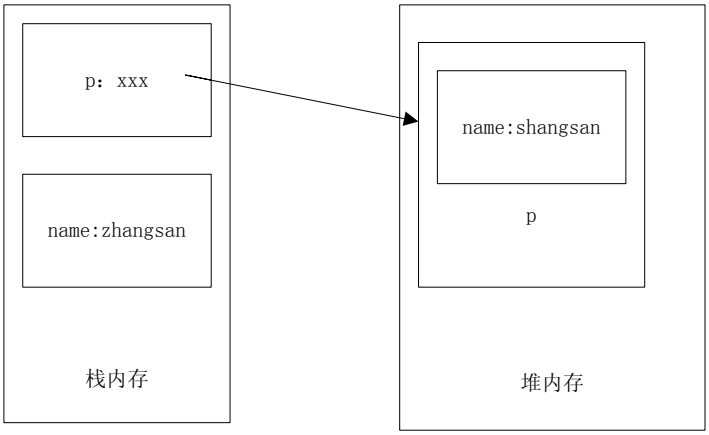
构造器（构造函数、构造方法）

构造器是定义在 Java 类中用来初始化对象的一个方法。构造器与类重名且没有返回值。例如：

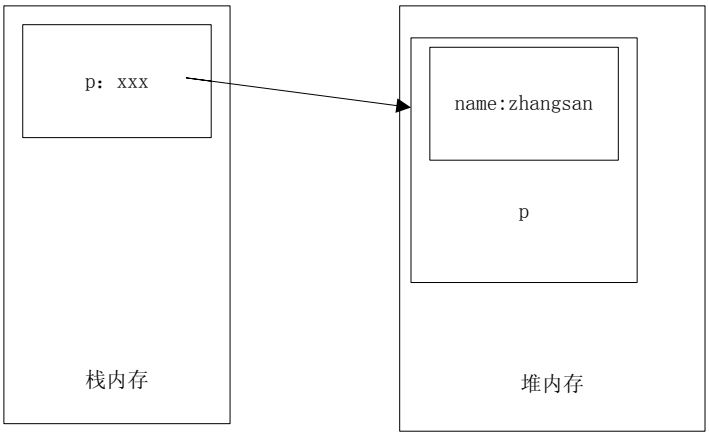
```
public class Person {
    String name;
    public Person(){}//无参数的构造器
```

```
    public Person(String name){//带参数的构造器
        name = name;
    }
}
```

创建对象时，是使用构造器来初始化对象。例如：Person p = new Person(“zhangsan”);



执行时会在栈中为形参分配一个内存。



执行完后会将战中的内存释放掉。

内部类

在一个类中定义另外一个类，这个类就叫做内部类(inner class) 。内部类的定义和普通类的定义没什么区别，它可以直接访问和引用它的外部类的所有变量和方法（包括 private），就像外部类中的其他非 static 成员的功能一样。区别是，外部类只能声明为 public 和 default，而内部类可以声明为 private 和 protected。

当我们建立一个 inner class 时，其对象就拥有了与外部类对象之间的一种关系，这是通过一个特殊的 this reference 形成的，当内部类的成员方法中访问某个变量/方法时，如果在该方法和内部类中都没有定义过这个变量，调用就会被传递给内部类中保存的那个外部类对象的引用（OuterClass.this），通过那个外部类对象的引用去调用这个变量。

举例简单的例子，如果 Test2 需要访问 Test1 中的成员变量，必须在 Test2 中实例化 Test1，才能够调用 Test1 中的 a 和 b。

```
public class Test1 {

    int a = 1;

    int b = 2;

}

class Test2 {

    void t() {

        Test t = new Test();

        System.out.println(t.a + t.b);

    }

}
```

对于这样的情况，可以将 Test2 声明为内部类。例如：

```
public class Test1 {
    int a = 1;
    int b = 2;
    class Test2 {
        public void t() {
            System.out.println(a + b);
        }
    }
}
```

这样 Test2 就自动持有 Test1 的引用了。由此举例可以看到内部类的优点：

（1）能够持有其他对象的引用，访问其他对象的属性

（2）可以程序结构变得更优雅

（3）可以防止不必要的调用

内部类很好的体现了封装的思想。

匿名内部类

```
public class Test {
    int a = 1;
```


```
    int b = 2;
    void test(){}
}

class Test1 {
    void fun1(){
        Test t = new Test();
        fun2(t);
    }

    void fun2(Test t){
        System.out.println(t.a+ t.b);
    }
}
```

用匿名内部类可以将上边的例子改写：

```
public class Test {
    int a = 1;
    int b = 2;
    void test() {
```

```
    }
    class Test1 {
        void fun1() {
            fun2(new Test(){});
        }
        void fun2(Test t) {
            System.out.println(a + b);
        }
    }
}
```


第五章　容器

数组

数组是一组具有相同类型和名称的变量的集合，并用一个数组下标来区分或指定每一个数。数组的变量是引用类型，数组的元素可以是任何数据类型，包括基本类型和引用类型。

一维数组的声明方式举例：

int[] i;或 int i[];

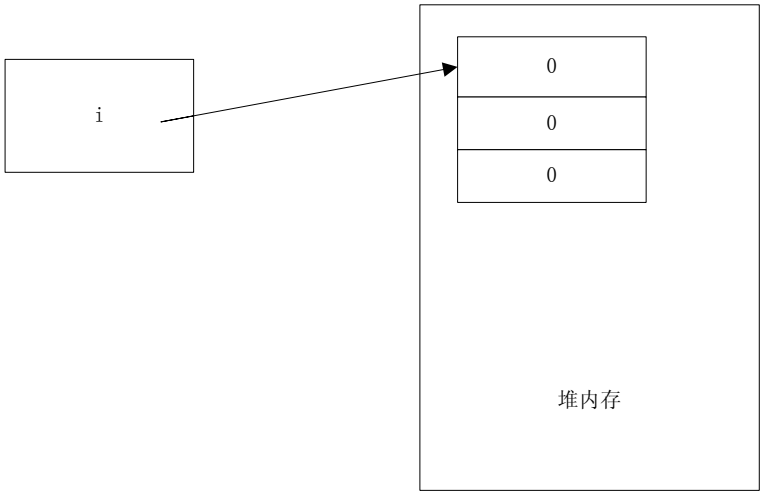
Java 语言中不能在声明数组的时候指定数据的长度，数组的长度由系统指定。Java 的数组是引用类型，不能在栈中声明，而只能在堆中声明。

不能做如下声明：

int[10] i;

Java 中使用 new 关键字来创建数组对象，例如：

int i[] = new int[3];



如果数组的元素为引用类型，那么每个元素都需要实例化。例如：

```
public class Test {
    public static void main(String[] args) {
        Person[] p = new Person[3];
        for(int i=0;i<3;i++) {
            p[i] = new Person(i);
        }
    }
}

class Person {
    int id;
    Person(int id){
        this.id = id;
    }
}
```

数组的初始化分为动态初始化和静态初始化。动态初始化是数组的定义与为数组元素分配空间和赋值的操作分开进行，例如：

```
String[] s = new String[2];
```


s[0] = “a”;

s[1] = “b”;

在数组定义的同时就为数组元素分配空间并赋值叫做静态初始化，例如：

String s [] = {“a”, “b”};

数组是引用类型，其元素相当于类的成员变量，所以，在数组被创建后。每个元素按照成员变量的默认值被隐式的初始化，例如：

```
public class TestInit {
    public static void main(String[] args) {
        int i[] = new int[3];
        boolean b[] = new boolean[2];
        Person[] p = new Person[4];
        System.out.println(i[0]);
        System.out.println(b[1]);
        System.out.println(p[3]);
    }
}

class Person {}
```

输出的结果为：0、false、null，和成员变量的默认初始化规则一致。

只有数组被定义，并且为其分配空间之后，才能引用数组中的每一个元素。格式为 array[数组下标]。数组下标取值从 0 开始，到下标-1 结束。每个数组通过 length 属性指出数组的长度。

二维数组可以看作是以数组为元素的数组，例如：

int[][] i = {{1,2},{3,4,5},{6,7,8,9}};

二维以上的数组的声明和初始化应从高维开始，例如：

String[][] s = new String[2][];

s[0] = new String[2];

s[1] = {“b”};

不能如下进行声明：

String[][] s = new String[][3];

二维数组的静态初始化：String[][] s = {{“a”},{“b”},{“c”}};

二维数组的动态初始化：String[][] s = new String[3][];String[][] s = new String[4][5];

集合框架

Java 平台提供了一个全新的集合框架。“集合框架”主要由一组用来操作对象的接口组成。不同接口描述一组不同数据类型。

集合接口：6 个接口（短虚线表示），表示不同集合类型，是集合框架的基础。

抽象类：5 个抽象类（长虚线表示），对集合接口的部分实现。可扩展为自定义集合类。

实现类：8 个实现类（实线表示），对接口的具体实现。

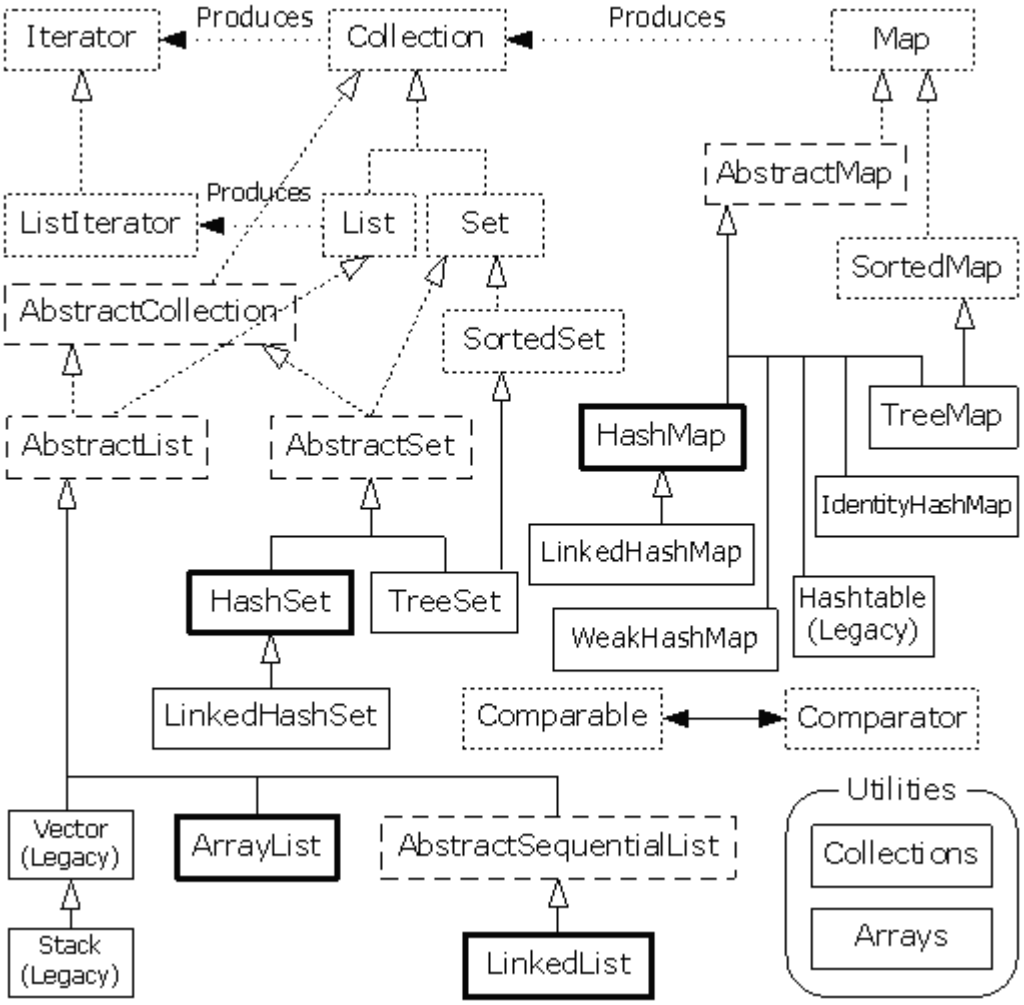
Collection 接口是一组允许重复的对象。

Set 接口继承 Collection，但不允许重复，使用自己内部的一个排列机制。

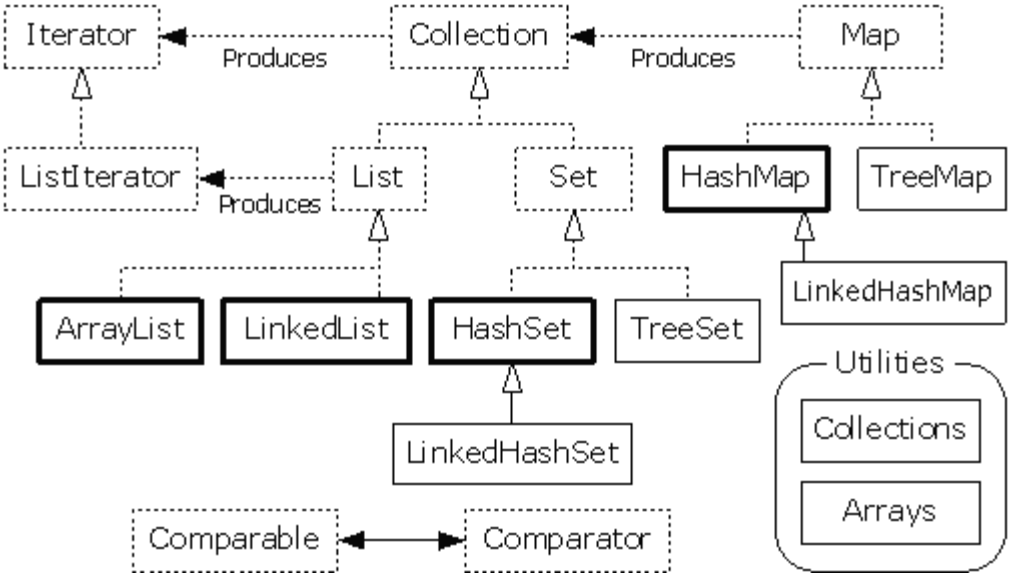
List 接口继承 Collection，允许重复，以元素安插的次序来放置元素，不会重新排列。

Map 接口是一组成对的键－值对象，即所持有的是 key-value pairs。Map 中不能有重复的 key。拥有自己的内部排列机制。

容器中的元素类型都为 Object。从容器取得元素时，必须把它转换成原来的类型。



将上图简化如下：



Collection 接口

Collection 接口用于表示任何对象或元素组。想要尽可能以常规方式处理一组元素时，就使用这一接口。

（1）单元素添加、删除操作：

boolean add(Object o):将对象添加给集合

boolean remove(Object o): 如果集合中有与 o 相匹配的对象，则删除对象 o

（2）查询操作：

int size() ： 返回当前集合中元素的数量

boolean isEmpty() ： 判断集合中是否有任何元素

boolean contains(Object o) ： 查找集合中是否含有对象 o

Iterator iterator() ： 返回一个迭代器，用来访问集合中的各个元素

（3）组操作 ： 作用于元素组或整个集合

boolean containsAll(Collection c): 查找集合中是否含有集合 c 中所有元素

boolean addAll(Collection c) : 将集合 c 中所有元素添加给该集合

void clear(): 删除集合中所有元素

void removeAll(Collection c) : 从集合中删除集合 c 中的所有元素

void retainAll(Collection c) : 从集合中删除集合 c 中不包含的元素

（4）Collection 转换为 Object 数组 ：

Object[] toArray() ： 返回一个内含集合所有元素的 array

Object[] toArray(Object[] a) ： 返回一个内含集合所有元素的 array。运行期返回的 array 和参数 a 的型别相同，需要转换为正确型别。

此外，您还可以把集合转换成其它任何其它的对象数组。但是，您不能直接把集合转换成基本数据类型的数组，因为集合必须持有对象。

Collection 不提供 get()方法。如果要遍历 Collectin 中的元素，就必须用 Iterator。

AbstractCollection 类

AbstractCollection 类提供具体“集合框架”类的基本功能。虽然您可以自行实现 Collection 接口的所有方法，但是，除了 iterator()和 size()方法在恰当的子类中实现以外，其它所有方法都由 AbstractCollection 类来提供实现。如果子类不覆盖某些方法，可选的如 add()之类的方法将抛出异常。

Iterator 接口

Collection 接口的 iterator()方法返回一个 Iterator。Iterator 接口方法能以迭代方式逐个访问集合中各个元素，并安全的从 Collection 中除去适当的元素。

boolean hasNext(): 判断是否存在另一个可访问的元素

Object next(): 返回要访问的下一个元素。如果到达集合结尾，则抛出 NoSuchElementException 异常。

void remove(): 删除上次访问返回的对象。本方法必须紧跟在一个元素的访问后执行。如果上次访问后集合已被修改，方法将抛出 IllegalStateException。

List 接口

List 接口继承了 Collection 接口以定义一个允许重复项的有序集合。该接口不但能够对列表的一部分进行处理，还添加了面向位置的操作。

（1）面向位置的操作包括插入某个元素或 Collection 的功能，还包括获取、除去或更改元素的功能。在 List 中搜索元素可以从列表的头部或尾部开始，如果找到元素，还将报告元素所在的位置 ：

void add(int index, Object element): 在指定位置 index 上添加元素 element

boolean addAll(int index, Collection c): 将集合 c 的所有元素添加到指定位置 index

Object get(int index): 返回 List 中指定位置的元素

int indexOf(Object o): 返回第一个出现元素 o 的位置，否则返回-1

int lastIndexOf(Object o) ： 返回最后一个出现元素 o 的位置，否则返回-1

Object remove(int index) ：删除指定位置上的元素

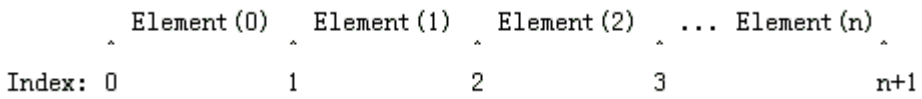
Object set(int index, Object element) ：用元素 element 取代位置 index 上的元素，并且返回旧的元素

(2) List 接口不但以位置序列迭代的遍历整个列表，还能处理集合的子集：

ListIterator listIterator() : 返回一个列表迭代器，用来访问列表中的元素

ListIterator listIterator(int index) : 返回一个列表迭代器，用来从指定位置 index 开始访问列表中的元素 List subList(int fromIndex, int toIndex) ：返回从指定位置 fromIndex（包含）到 toIndex（不包含）范围中各个元素的列表视图

ListIterator 接口继承 Iterator 接口以支持添加或更改底层集合中的元素，还支持双向访问。ListIterator 没有当前位置，光标位于调用 previous 和 next 方法返回的值之间。一个长度为 n 的列表，有 n+1 个有效索引值：



void add(Object o): 将对象 o 添加到当前位置的前面

void set(Object o): 用对象 o 替代 next 或 previous 方法访问的上一个元素。如果上次调用后列表结构被修改了，那么将抛出 IllegalStateException 异常。

boolean hasPrevious(): 判断向后迭代时是否有元素可访问

Object previous()：返回上一个对象

int nextIndex(): 返回下次调用 next 方法时将返回的元素的索引

int previousIndex(): 返回下次调用 previous 方法时将返回的元素的索引

AbstractList 和 AbstractSequentialList 抽象类

有两个抽象的 List 实现类：AbstractList 和 AbstractSequentialList。像 AbstractSet 类一样，它们覆盖了 equals() 和 hashCode() 方法以确保两个相等的集合返回相同的哈希码。若两个列表大小相等且包含顺序相同的相同元素，则这两个列表相等。这里的 hashCode() 实现在 List 接口定义中指定，而在这里实现。

除了 equals()和 hashCode()，AbstractList 和 AbstractSequentialList 实现了其余 List 方法的一部分。因为数据的随机访问和顺序访问是分别实现的，使得具体列表实现的创建更为容易。需要定义的一套方法取决于希望支持的行为。永远不必亲自提供的是 iterator 方法的实现。

LinkedList 类

在“集合框架”中有两种常规的 List 实现：ArrayList 和 LinkedList。使用两种 List 实现的哪一种取决于您特定的需要。如果要支持随机访问，而不必在除尾部的任何位置插入或除去元素，那么，ArrayList 提供了可选的集合。但如果，您要频繁的从列表的中间位置添加和除去元素，而只要顺序的访问列表元素，那么，LinkedList 实现更好。

void addFirst(Object o): 将对象 o 添加到列表的开头

void addLast(Object o)：将对象 o 添加到列表的结尾

Object getFirst(): 返回列表开头的元素

Object getLast(): 返回列表结尾的元素

Object removeFirst(): 删除并且返回列表开头的元素

Object removeLast():删除并且返回列表结尾的元素

LinkedList(): 构建一个空的链接列表

LinkedList(Collection c): 构建一个链接列表，并且添加集合 c 的所有元素

ArrayList 类

ArrayList 类封装了一个动态再分配的 Object[]数组。每个 ArrayList 对象有一个 capacity。这个 capacity 表示存储列表中元素的数组的容量。当元素添加到 ArrayList 时，它的 capacity 在常量时间内自动增加

在向一个 ArrayList 对象添加大量元素的程序中，可使用 ensureCapacity 方法增加 capacity。这可以减少增加重分配的数量。

void ensureCapacity(int minCapacity): 将 ArrayList 对象容量增加 minCapacity

void trimToSize(): 整理 ArrayList 对象容量为列表当前大小。程序可使用这个操作减少 ArrayList 对象存储空间。

RandomAccess 接口

这是一个特征接口。该接口没有任何方法，不过你可以使用该接口来测试某个集合是否支持有效的随机访问。ArrayList 和 Vector 类用于实现该接口。

Set 接口

Set 接口继承 Collection 接口，而且它不允许集合中存在重复项，每个具体的 Set 实现类依赖添加的对象的 equals()方法来检查独一性。Set 接口没有引入新方法，所以 Set 就是一个 Collection，只不过其行为不同。

Hash 表

Hash 表是一种数据结构，用来查找对象。Hash 表为每个对象计算出一个整数，称为 Hash Code(哈希码)。Hash 表是个链接式列表的阵列。

Comparable 接口

在“集合框架”中有两种比较接口：Comparable 接口和 Comparator 接口。像 String 和 Integer 等 Java 内建类实现 Comparable 接口以提供一定排序方式，但这样只能实现该接口一次。对于那些没有实现 Comparable 接口的类、或者自定义的类，您可以通过 Comparator 接口来定义您自己的比较方式。在 java.lang 包中，Comparable 接口适用于一个类有自然顺序的时候。假定对象集合是同一类型，该接口允许您把集合排序成自然顺序。

int compareTo(Object o): 比较当前实例对象与对象 o，如果位于对象 o 之前，返回负值，如果两个对象在排序中位置相同，则返回 0，如果位于对象 o 后面，则返回正值

利用 Comparable 接口创建您自己的类的排序顺序，只是实现 compareTo()方法的问题。通常就是依赖几个数据成员的自然排序。同时类也应该覆盖 equals()和 hashCode()以确保两个相等的对象返回同一个哈希码。

Comparator 接口

若一个类不能用于实现 java.lang.Comparable，或者您不喜欢缺省的 Comparable 行为并想提供自己的排序顺序(可能多种排序方式)，你可以实现 Comparator 接口，从而定义一个比较器。

int compare(Object o1, Object o2): 对两个对象 o1 和 o2 进行比较，如果 o1 位于 o2 的前面，则返回负值，如果在排序顺序中认为 o1 和 o2 是相同的，返回 0，如果 o1 位于 o2 的后面，则返回正值

boolean equals(Object obj): 指示对象 obj 是否和比较器相等。

SortedSet 接口

“集合框架”提供了个特殊的 Set 接口：SortedSet，它保持元素的有序顺序。SortedSet 接口为集的视图(子集)和它的两端（即头和尾）提供了访问方法。添加到 SortedSet 实现类的元素必须实现 Comparable 接口，否则您必须给它的构造函数提供一个 Comparator 接口的实现。TreeSet 类是它的唯一一份实现。

Comparator comparator(): 返回对元素进行排序时使用的比较器，如果使用 Comparable 接口的 compareTo()方法对元素进行比较，则返回 null

Object first(): 返回有序集合中第一个(最低)元素

Object last(): 返回有序集合中最后一个(最高)元素

SortedSet subSet(Object fromElement, Object toElement): 返回从 fromElement(包括)至 toElement(不包括)范围内元素的 SortedSet 视图(子集)

SortedSet headSet(Object toElement): 返回 SortedSet 的一个视图，其内各元素皆小于 toElement

SortedSet tailSet(Object fromElement): 返回 SortedSet 的一个视图，其内各元素皆大于或等于 fromElement

AbstractSet 抽象类

AbstractSet 类覆盖了 Object 类的 equals()和 hashCode()方法，以确保两个相等的集返回相同的哈希码。若两个集大小相等且包含相同元素，则这两个集相等。按定义，集的哈希码是集中元素哈希码的总和。因此，不论集的内部顺序如何，两个相等的集会有相同的哈希码。

HashSet 类

HashSet(): 构建一个空的哈希集

HashSet(Collection c): 构建一个哈希集，并且添加集合 c 中所有元素

HashSet(int initialCapacity): 构建一个拥有特定容量的空哈希集

HashSet(int initialCapacity, float loadFactor): 构建一个拥有特定容量和加载因子的空哈希集。LoadFactor 是 0.0 至 1.0 之间的一个数。

TreeSet 类

TreeSet():构建一个空的树集

TreeSet(Collection c): 构建一个树集，并且添加集合 c 中所有元素

TreeSet(Comparator c): 构建一个树集，并且使用特定的比较器对其元素进行排序

TreeSet(SortedSet s): 构建一个树集，添加有序集合 s 中所有元素，并且使用与有序集合 s 相同的比较器排序

LinkedHashSet 类

LinkedHashSet 扩展 HashSet。如果想跟踪添加给 HashSet 的元素的顺序，LinkedHashSet 实现会有帮助。LinkedHashSet 的迭代器按照元素的插入顺序来访问各个元素。它提供了一个可以快速访问各个元素的有序集合。同时，它也增加了实现的代价，因为哈希表元中的各个元素是通过双重链接式列表链接在一起的。

LinkedHashSet(): 构建一个空的链接式哈希集

LinkedHashSet(Collection c): 构建一个链接式哈希集，并且添加集合 c 中所有元素

LinkedHashSet(int initialCapacity): 构建一个拥有特定容量的空链接式哈希集 LinkedHashSet(int initialCapacity, float loadFactor): 构建一个拥有特定容量和加载因子的空链接式哈希集。LoadFactor 是 0.0 至 1.0 之间的一个数

Map 接口

（1）添加、删除操作：

Object put(Object key, Object value): 将互相关联的一个关键字与一个值放入该映像。如果该关键字已经存在，那么与此关键字相关的新值将取代旧值。方法返回关键字的旧值，如果关键字原先并不存在，则返回 null

Object remove(Object key): 从映像中删除与 key 相关的映射

void putAll(Map t): 将来自特定映像的所有元素添加给该映像

void clear(): 从映像中删除所有映射

键和值都可以为 null。但是，不能把 Map 作为一个键或值添加给自身。

(2) 查询操作：

Object get(Object key): 获得与关键字 key 相关的值，并且返回与关键字 key 相关的对象，如果没有在该映像中找到该关键字，则返回 null

boolean containsKey(Object key): 判断映像中是否存在关键字 key

boolean containsValue(Object value): 判断映像中是否存在值 value

int size(): 返回当前映像中映射的数量

boolean isEmpty() ：判断映像中是否有任何映射

(3) 视图操作 ：处理映像中键/值对组

Set keySet(): 返回映像中所有关键字的视图集

Collection values():返回映像中所有值的视图集

Set entrySet(): 返回 Map.Entry 对象的视图集，即映像中的关键字/值对

Map.Entry 接口

Map 的 entrySet()方法返回一个实现 Map.Entry 接口的对象集合。集合中每个对象都是底层 Map 中一个特定的键/值对。

Object getKey(): 返回条目的关键字

Object getValue(): 返回条目的值

Object setValue(Object value): 将相关映像中的值改为 value，并且返回旧值

SortedMap 接口

“集合框架”提供了个特殊的 Map 接口：SortedMap，它用来保持键的有序顺序。SortedMap 接口为映像的视图(子集)，包括两个端点提供了访问方法。除了排序是作用于映射的键以外，处理 SortedMap 和处理 SortedSet 一样。

添加到 SortedMap 实现类的元素必须实现 Comparable 接口，否则您必须给它的构造函数提供一个 Comparator 接口的实现。TreeMap 类是它的唯一一份实现。

Comparator comparator(): 返回对关键字进行排序时使用的比较器，如果使用 Comparable 接口的 compareTo()方法对关键字进行比较，则返回 null

Object firstKey(): 返回映像中第一个(最低)关键字

Object lastKey(): 返回映像中最后一个(最高)关键字

SortedMap subMap(Object fromKey, Object toKey): 返回从 fromKey(包括)至 toKey(不包括)范围内元素的 SortedMap 视图(子集)

SortedMap headMap(Object toKey): 返回 SortedMap 的一个视图，其内各元素的 key 皆小于 toKey

SortedSet tailMap(Object fromKey): 返回 SortedMap 的一个视图，其内各元素的 key 皆大于或等于 fromKey

AbstractMap 抽象类

和其它抽象集合实现相似，AbstractMap 类覆盖了 equals()和 hashCode()方法以确保两个相等映射返回相同的哈希码。如果两个映射大小相等、包含同样的键且每个键在这两个映射中对应的值都相同，则这两个映射相等。映射的哈希码是映射元素哈希码的总和，其中每个元素是 Map.Entry 接口的一个实现。因此，不论映射内部顺序如何，两个相等映射会报告相同的哈希码。

HashMap 类

集合框架”提供两种常规的 Map 实现：HashMap 和 TreeMap (TreeMap 实现 SortedMap 接口)。在 Map 中插入、删除和定位元素，HashMap 是最好的选择。但如果您要按自然顺序或自定义顺序遍历键，那么 TreeMap 会更好。使用 HashMap 要求添加的键类明确定义了 hashCode()和 equals()的实现。

这个 TreeMap 没有调优选项，因为该树总处于平衡状态。

HashMap(): 构建一个空的哈希映像

HashMap(Map m): 构建一个哈希映像，并且添加映像 m 的所有映射

HashMap(int initialCapacity): 构建一个拥有特定容量的空的哈希映像

HashMap(int initialCapacity, float loadFactor): 构建一个拥有特定容量和加载因子的空的哈希映像

TreeMap 类

TreeMap():构建一个空的映像树

TreeMap(Map m): 构建一个映像树，并且添加映像 m 中所有元素

TreeMap(Comparator c): 构建一个映像树，并且使用特定的比较器对关键字进行排序

TreeMap(SortedMap s): 构建一个映像树，添加映像树 s 中所有映射，并且使用与有序映像 s 相同的比较器排序

LinkedHashMap 类

LinkedHashMap 扩展 HashMap，以插入顺序将关键字/值对添加进链接哈希映像中。象 LinkedHashSet 一样，LinkedHashMap 内部也采用双重链接式列表。

LinkedHashMap(): 构建一个空链接哈希映像

LinkedHashMap(Map m): 构建一个链接哈希映像,并且添加映像 m 中所有映射

LinkedHashMap(int initialCapacity): 构建一个拥有特定容量的空的链接哈希映像

LinkedHashMap(int initialCapacity, float loadFactor): 构建一个拥有特定容量和加载因子的空的链接哈希映像

LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder): 构建一个拥有特定容量、加载因子和访问顺序排序的空的链接哈希映像

protected boolean removeEldestEntry(Map.Entry eldest): 如果你想删除最老的映射，则覆盖该方法，以便返回 true。当某个映射已经添加给映像之后，便调用该方法。它的默认实现方法返回 false，表示默认条件下老的映射没有被删除。但是你可以重新定义本方法，以便有选择地在最老的映射符合某个条件，或者映像超过了某个大小时，返回 true。

WeakHashMap 类

WeakHashMap 是 Map 的一个特殊实现，它使用 WeakReference(弱引用)来存放哈希表关键字。使用这种方式时，当映射的键在 WeakHashMap 的外部不再被引用时，垃圾收集器会将它回收，但它将把到达该对象的弱引用纳入一个队列。WeakHashMap 的运行将定期检查该队列，以便找出新到达的弱应用。当一个弱引用到达该队列时，就表示关键字不再被任何人使用，并且它已经被收集起来。然后 WeakHashMap 便删除相关的映射。

WeakHashMap(): 构建一个空弱哈希映像

WeakHashMap(Map t): 构建一个弱哈希映像,并且添加映像 t 中所有映射

WeakHashMap(int initialCapacity): 构建一个拥有特定容量的空的弱哈希映像

WeakHashMap(int initialCapacity, float loadFactor): 构建一个拥有特定容量和加载因子的空的弱哈希映像

IdentityHashMap 类

IdentityHashMap 也是 Map 的一个特殊实现。在这个类中，关键字的哈希码不应该由 hashCode()方法来计算，而应该由 System.identityHashCode 方法进行计算(即使已经重新定义了 hashCode 方法)。这是 Object.hashCode 根据对象的内存地址来计算哈希码时使用的方法。另外，为了对各个对象进行比较，IdentityHashMap 将使用==，而不使用 equals 方法。

换句话说，不同的关键字对象，即使它们的内容相同，也被视为不同的对象。IdentityHashMap 类可以用于实现对象拓扑结构转换(topology-preserving object graph transformations)(比如实现对

象的串行化或深度拷贝)，在进行转换时，需要一个“节点表”跟踪那些已经处理过的对象的引用。即使碰巧有对象相等，“节点表”也不应视其相等。另一个应用是维护代理对象。比如，调试工具希望在程序调试期间维护每个对象的一个代理对象。

IdentityHashMap (): 构建一个空的全同哈希映像，默认预期最大尺寸为 21

IdentityHashMap (Map m): 构建一个全同哈希映像,并且添加映像 m 中所有映射

IdentityHashMap (int expectedMaxSize): 构建一个拥有预期最大尺寸的空的全同哈希映像。放置超过预期最大尺寸的键/值映射时，将引起内部数据结构的增长，有时可能很费时。

泛型

在 JDK1.4 之前，在使用集合的时候，装入集合的类型都被当做 Object，失去的自身的数据类型，当从集合中取出来的时候，才需要转型确定类型，效率低，容易产生错误。在 JDK1.5 引入了泛型的范念，在定义集合的同时也定义集合中对象的数据类型，这样能够保证装入集合的对象的类型，在从集合中取出来的时候不用再做类型转换，增强了程序的可读性和稳定性。泛型的实现很复杂，但是使用起来很简单，例如：ArrayList<Person> 在 arrayList 中只能放 Person 类型的对象，如果放入别的类型就会报错。

```
import java.util.*;

public class TestHashMap {

    public static void main(String[] args) {

        Map<String,Cat> map = new HashMap<String,Cat>();

        Cat cat = new Cat();

        Dog dog = new Dog();

        map.put("one", cat);
```


```
        //map.put("two", dog);


        //Cat c = (Cat)map.get("one");
        //c.say();
        (map.get("one")).say();
    }
}

class Dog {
    public void say() {
        System.out.println("I 'm a dog!");
    }
}

class Cat {
    public void say() {
        System.out.println("I 'm a cat!");
    }
}
```

第六章　异常

Java 异常是 Java 语言提供的一种用于处理程序错误的机制。Java 的异常处理非常类似于 C++。Java 语言的异常处理继承里 C++异常处理中的优点，并且比 C++异常处理更安全、强大和丰富。

什么是错误

错误是指在程序运行的过程中发生的一些例外事件。

抛出异常

Java 程序的执行过程中如果出现例外事件，会生成一个异常类对象，该对象封装了例外时间的信息并会提供给 Java 运行时系统，这个过程叫作抛出异常。

捕获异常

Java 运行时系统接收到异常对象时，会寻找一段能够来处理这一异常的代码，并且将当前的对象交给其处理，这个过程叫作捕获异常。

异常的分类

Java 语言中有很多异常类，这些类都继承自 Throwable 类，每一个异常对象都是 Throwable 的一个实例，如果这些类不能满足需求，我们还可以自己创造自定义的实例。

Throwable 下边有 2 个分支：error 和 exception。Error 类层次结构描述了 Java 运行时系统的内部错误和资源耗尽错误。应用程序不能抛出这样类型的对象，并对此种错误无能为力。

Exception 是进行程序设计时需要关注的，它又分为两类：other exception 和 runtimeException。runtimeException 是由程序引起的异常。

Java 语言规范将 runtimeException 和 error 类的异常称为未检查异常，其他的称为以检查异常。

异常的程序结构

```
public void test() {
    try{
        ...
    } catch(OmeException e1) {
        ...
    } catch(TwoException e2) {
        ...
    } finally {
        ...
    }
}
```

try 中包含可能产生例外的代码段，try 后可以跟一个或多个 catch，每个 catch 声明一种特定的 exception 并对其处理。出现异常时，程序会终止当前的流程而转到相应的 catch 中执行。finally 中的内容无论出现异常与否，都必须执行。

一段程序可能产生并抛出一种或多种类型的异常，catch 分别对这些异常做相应的处理。如果没有出现异常，所有的 catch 都会被忽略。

catch 用于处理可能产生的不同类型的异常对象。catch 中的声明封装了异常事件的信息，在 catch 中可以对这个异常对象进行处理，常用的方法如：getMessage（）和 printStackTrace（）。

finally 为程序提供了一个统一的出口，使程序在转到其他程序以前，都能对程序的状态进行统一的管理。finally 中的语句总会被执行，所以通常在 finally 中进行资源的清除操作，如关闭打开的文件，数据库连接等等。

异常的声明

Java 的异常处理机制使例外的程序沿着被调用的顺序向上寻找符合此中类型异常的处理程序。

如果程序出现异常，没有处理或无法处理，会自动将异常向上一级程序抛出，直到找到可以处理的程序。一般来说，如果程序出现异常可以使用 throws 关键字直接抛出。如果一个方法抛出异常，那么调用该方法的程序在不对异常进行处理的情况下也必须抛出异常。声明异常的方式如下：

```
Public void test throws OneException {
    …
}
```

catch 处理多个可能抛出的异常时，应当先写子类异常在写基类异常。

自定义异常

如果现有异常类型不能满足需要，可以使用下面步骤自定义异常：

（1）通过继承 java.lang.Exception 类声明自己的异常类

（2）在方法的适当位置生成自定义异常的实例，并用 throw 语句抛出

（3）在方法的声明部分用 throws 语句声明该方法肯能抛出的异常

```
class MyException extends Exception {



}



public class Test {
    public void test() throws MyException {
        ...
        throw new MyException();
    }
}
```

第七章　I/O

Java 语言中，数据的输入输出操作是以流的方式进行的，Java 中提供了多种流的类用以获取不同的数据类型，在程序中通过标准的方法输入或输出数据。

输入/输出流的分类

java.io.包中定义了多个流类型来实现输入输出功能，流有以下几种分类：

按数据流的方向分为：输入流和输出流

按处理数据的单位分为：字节流和字符流

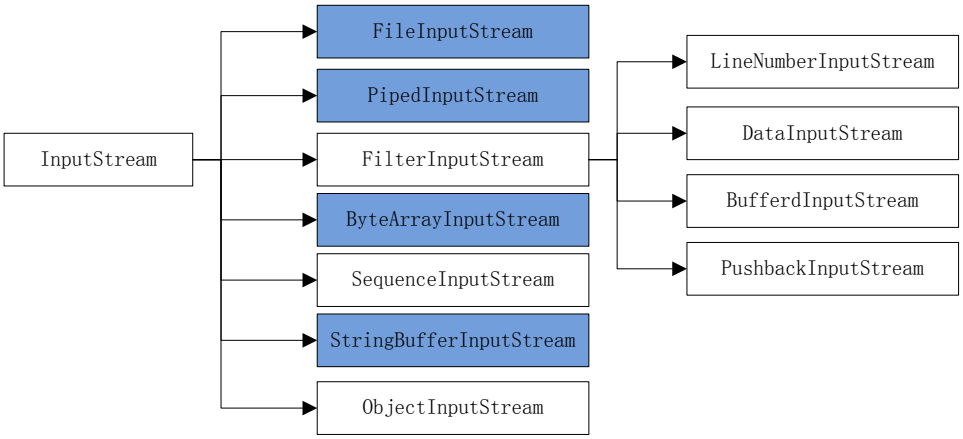
按功能分为：节点流和处理流

节点流可以从一个特定的数据源读写数据。处理流在已存在的流之上，通过对数据的处理为程序提供更强大的读写功能。

	字节流	字符流
输入流	InputStream	Reader
输出流	OutputStream	Writer

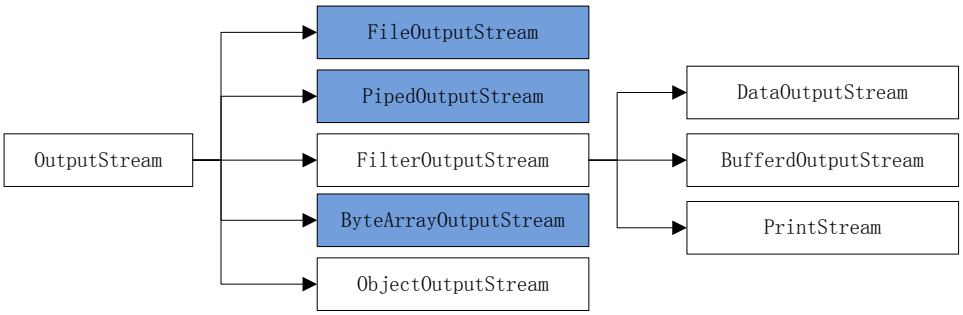
InputStream

继承自 InputStream 的流都是用于向程序中输入数据的，并且数据的单位为字节（8bit）。



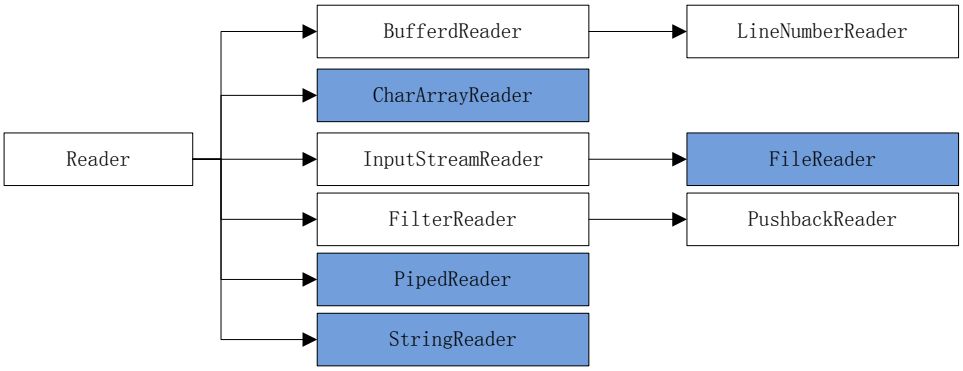
OutputStream

继承自 OutputStream 的流都是用于向程序中输出数据的，并且数据的单位为字节（8bit）。



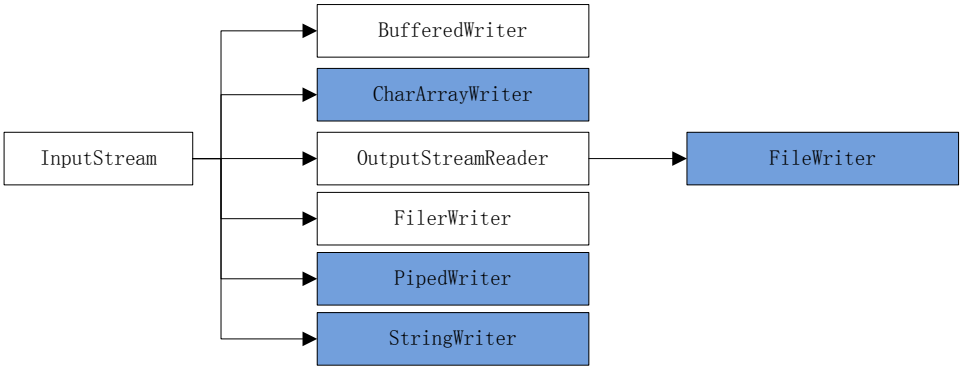
Reader

继承自 Reader 的流都是用于向程序中输入数据的，并且数据的单位为字节（16bit）。



Writer

继承自 Writer 的流都是用于向程序中输入数据的，并且数据的单位为字节（16bit）。



节点流

类型	字符流	字节流
File	FileReader	FileInputStream
	FileWriter	FileOutputStream
Memory Array	CharArrayReader	ByteArrayInputStream
	CharArrayWriter	ByteArrayOutputStream
Memory String	StringReader	
	StringWriter	
Pipe	PipedReader	PipedInputStream
	PipedWrite	PipedOutputStream

FileInputStream 和 FileOutputStream 分别继承了 InputStream 和 OutputStream，它们用于向文件中输入和输出字节。它们支持其父类中的所提供的所有数据读写的方法。在实例化 FileInputSteam 和 FileOutStream 是要用 try-catch 语句处理可能抛出的 FileNotFoundException，在读写数据时，也要用 try-catch 语句处理可能抛出的 IOException。

```
import java.io.*;
```


```
public class TestFileInputStream {

    public static void main(String[] args) {

        int b = 0;

        FileInputStream in = null;

        try {

            in = new FileInputStream("d:\\java\\io\\TestFileInputStream.java");

        } catch (FileNotFoundException e) {

            System.out.println("找不到指定文件");

            System.exit(-1);

        }



        try {

            long num = 0;

            while ((b = in.read()) != -1) {

                System.out.print((char) b);

                num++;

            }

            in.close();

            System.out.println();
```

```
            System.out.println("共读取了 " + num + " 个字节");
        } catch (IOException e1) {
            System.out.println("文件读取错误");
            System.exit(-1);
        }
    }
}
```

因为 FileInputStream 一次只读一个字节，所以中文不能够正确读取。

```
import java.io.*;
public class TestFileOutputStream {
    public static void main(String[] args) {
        int b = 0;
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("d:\\java\\io\\HelloWorld.java");
            out = new FileOutputStream("d:/java/io/test/HelloWorld.java");
            while ((b = in.read()) != -1) {
                out.write(b);
```


```
                }
                in.close();
                out.close();
            } catch (FileNotFoundException e2) {
                System.out.println("找不到指定文件");
                System.exit(-1);
            } catch (IOException e1) {
                System.out.println("文件复制错误");
                System.exit(-1);
            }
            System.out.println("文件已复制");
        }
}
import java.io.*;
public class TestFileReader {
        public static void main(String[] args) {
            FileReader fr = null;
            int c = 0;
            try {
```

```
                fr = new FileReader("d:\\java\\io\\HelloWorld.java");
                int ln = 0;
                while ((c = fr.read()) != -1) {
                    // char ch = (char) fr.read();
                    System.out.print((char) c);
                    // if (++ln >= 100) { System.out.println(); ln = 0;}
                }
                fr.close();
            } catch (FileNotFoundException e) {
                System.out.println("找不到指定文件");
            } catch (IOException e) {
                System.out.println("文件读取错误");
            }
        }
}
import java.io.*;
public class TestFileWriter {
    public static void main(String[] args) {
        FileWriter fw = null;
```


```
            try {
                fw = new FileWriter("d:\\java\\io\\test\\bak.txt");
                for (int c = 0; c <= 100; c++) {
                    fw.write(c);
                }
                fw.close();
            } catch (IOException e1) {
                e1.printStackTrace();
                System.out.println("文件写入错误");
                System.exit(-1);
            }
        }
    }
```

处理流

处理类型	字符流	字节流
Buffering	BufferedReader	BufferedInputStream
	BufferedWriter	BufferedOutputStream
Filtering	FilterReader	FilterInputStream
	FilterWriter	FilterOutputSteam
Converting between btyes and character	InputStreamReader	

	OutputStreamWriter	
Object Serialization		ObjectInputStream
		ObjectOutputStream
Data Convertion		DataInputSteam
		DataOutputStream
Counting	LineNumberReader	LineNumberInputStream
Printing	PrintWriter	PrinteStream

Buffered 缓冲流套接在相应的节点流之上，对读写数据提供了缓冲的功能，提高了读写的效率。缓冲流支持父类的 mark 和 reset 方法，并提供了 readLine 方法用于读取一行字符串，提供了 newLine 方法用于写入一个换行符。对于缓冲流，数据会存储在缓冲区中，可以使用 flush 方法将数据写出。

```
import java.io.*;

public class TestBufferStream2 {

    public static void main(String[] args) {

        try {

            BufferedWriter bw = new BufferedWriter(new
                                FileWriter("d:\\java\\io\\test\\bak.txt"));

            BufferedReader br = new BufferedReader(new
                                FileReader("d:\\java\\io\\HelloWorld.java"));

            String s = null;

            for (int i = 1; i <= 100; i++) {

                s = String.valueOf(Math.random());

                bw.write(s);
```


```
                bw.newLine();
            }
            bw.flush();
            while ((s = br.readLine()) != null) {
                System.out.println(s);
            }
            bw.close();
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

转换流

InputStreamReader 和 OutputStreamWriter 用于字节数据到字符数据之间的转换。InputStreamReader 需要和 InputStream 套接，OutputStreamWriter 需要和 OutputStream 套接。并且在构造转换流时可以指定字符集编码，例如：InputStream is = new InputStreamReader(System.in,”ISO-8859-1”）；

```
import java.io.*;
```

```
public class TestInputStreamReader {
    public static void main(String args[]) {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String s = null;
        try {
            s = br.readLine();
            while (s != null) {
                if (s.equalsIgnoreCase("exit"))
                    break;
                System.out.println(s.toUpperCase());
                s = br.readLine();
            }
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```


```
import java.io.*;
public class TestOutputStreamWriter {
	public static void main(String[] args) {
		try {
			OutputStreamWriter osw = new OutputStreamWriter(
					new FileOutputStream("d:\\java\\io\\test\\a.txt"));
			osw.write("qwertyuiopasdfghjkl");
			System.out.println(osw.getEncoding());
			osw.close();
			osw = new OutputStreamWriter(new FileOutputStream(
					"d:\\java\\io\\test\\b.txt", true), "ISO8859_1"); // latin-1
			osw.write("qwertyuiopasdfghjkl");
			System.out.println(osw.getEncoding());
			osw.close();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

数据流

DataInputStream 和 DateOutputStream 分别继承自 InputStream 和 OutputSteam，它们属于处理流，需要分别套接在 InputStream 和 OutputStream 类型的节点流之上。DataInputStream 和 DateOutputStream 提供了可以存取与机器无关的 Java 原始类型数据的方法。其构造方法为：DataInputStream(InputStream in)和 DateOutputStream(OutputStream out)

```
import java.io.*;


public class TestDataStream {
    public static void main(String[] args) {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        try {
            dos.writeDouble(Math.random());
            dos.writeBoolean(true);
            ByteArrayInputStream bais = new ByteArrayInputStream(baos
                    .toByteArray());
            System.out.println(bais.available());
            DataInputStream dis = new DataInputStream(bais);
            System.out.println(dis.readDouble());
            System.out.println(dis.readBoolean());
```


```
                dos.close();
                dis.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

Print 流

PrintWriter 和 PrintStream 都属于输出流，分别针对字节和字符。它们提供了重载的 print 方法。println 方法用于多种数据类型的输出。print 的操作不会抛出异常，所有异常在内部被处理，用户通过检验错误状态获取错误信息。PrintWriter 和 PrintStream 具有自动 flush 的功能。

```
import java.util.*;
import java.io.*;
public class TestPrintStream {
    public static void main(String[] args) {
        String s = null;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        try {
            FileWriter fw = new FileWriter("d:\\java\\io\\test\\log.log", true);
```

```
                PrintWriter log = new PrintWriter(fw);
                while ((s = br.readLine()) != null) {
                    if (s.equalsIgnoreCase("exit"))
                        break;
                    System.out.println(s.toUpperCase());
                    log.println("-----");
                    log.println(s.toUpperCase());
                    log.flush();
                }
                log.println("->" + new Date() + "<-");
                log.flush();
                log.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
}
```


Object 流

Object 流用于对象的读写操作。将对象用流的方式读写到硬盘或网络上的操作叫做序列化。对于对象中包含的属性可以不用使用具体的数据流，而是将所有信息封装在一个对象中，对对象进行读写操作。用 transient 关键字的属性在序列化的时候会被忽略。用于读写的对象需要实现 serializable 接口，seriazable 是一个空接口，其中没有定义方法，由 JDK 内部实现。一个类实现了 seriazable 接口，就为该类添加了一个标记，表示可序列化，JKD 会自动处理。Extenalizable 是 seriazable 的子接口，可以通过重写接口中的方法自定义序列化的方式，一般建议由 JDK 自己实现序列化。

```
import java.io.*;

public class TestObjectStream {

    public static void main(String args[]) throws Exception {

        T t = new T();

        t.k = 8;

        FileOutputStream fos = new FileOutputStream("d:\\java\\io\\test\\c.txt");

        ObjectOutputStream oos = new ObjectOutputStream(fos);

        oos.writeObject(t);

        oos.flush();

        oos.close();



        FileInputStream fis = new FileInputStream("d:\\java\\io\\test\\c.txt");

        ObjectInputStream ois = new ObjectInputStream(fis);
```

```
            T tr = (T) ois.readObject();
            System.out.println(tr.i + " " + tr.j + " " + tr.d + " " + tr.k);
        }
}

class T implements Serializable {
        int i = 10;
        int j = 9;
        double d = 2.3;
        transient int k = 15;
}
```


第八章　线程

什么是线程

在目前大多数操作系统都是支持多任务的，我们可以一边听歌一边写程序，再我们看来这些事情是同时进行的，但是，实际上 CPU 并是不同时来执行所有的程序的，而是 CPU 将时间分成小片，利用其高速的运行能力来轮流执行。一个执行中的程序就是一个进程，每一个进程都有自己独立的内存空间，每个内存的数据和状态都是独立的。产生一个进程的系统开销比较大，所以产生了线程的概念。线程简单的说就是顺序控制流，也就是程序的执行路径。一条程序的分支就是一个线程。多线程则指的是在单个程序中可以同时运行多个不同的线程，执行不同的任务。多线程意味着一个程序的多行语句可以看上去几乎在同一时间内同时运行。Java 的线程是通过 java.lang.Thread 类来实现的，可以通过创建 Thread 的实例来创建新的线程。每一个线程都是通过某个特定的 Thread 对象所对应的 run 方法来完成操作的，方法 run 称作线程体。JVM 启动时创建一个由主方法定义的主线程。通过调用 Tread 类的 start 方法来启动一个线程。

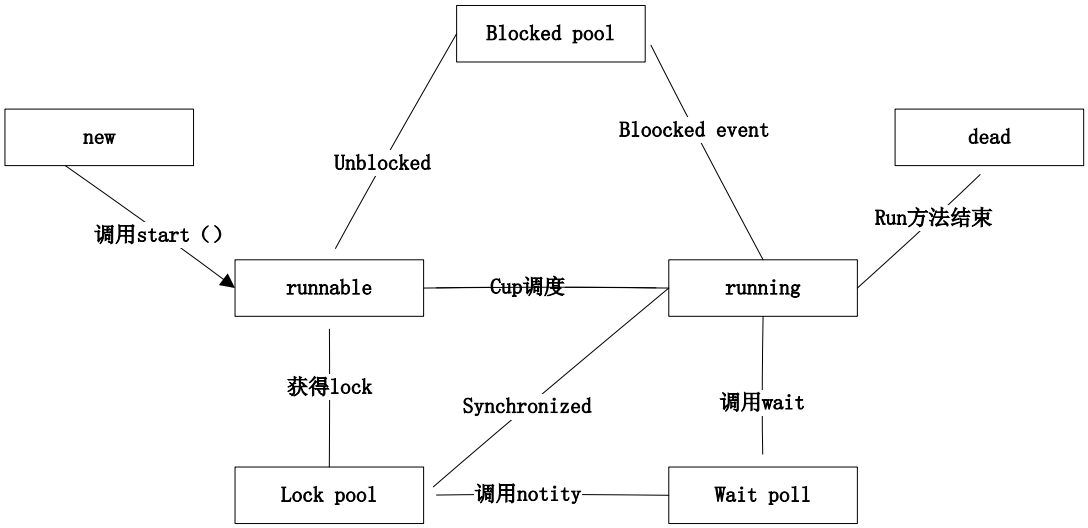
线程和进程的区别

每个进程都有独立的代码和数据空间，进程间切换会有很大的开销。

线程可以看作是轻量级的进程，同一类线程共享代码和数据空间，每一个线程有独立的运行栈和计数器，线程间切换开销较小。一个进程可以包含多个线程。

在操作系统中同时运行多个任务叫多进程，在同一应用程序中多个顺序流同时运行叫做多线程

线程的状态转换



线程的状态表示线程正在进行的活动以及在此时间段内所能完成的任务。线程有创建、可运行、运行中、阻塞、死亡五种状态。一个具有生命的线程总是处于这五种状态之一：

使用 new 运算符创建一个线程后，该线程仅仅是一个空对象，系统没有分配资源，称该线程处于创建状态(new thread)

使用 start()方法启动一个线程后，系统为该线程分配了除 CPU 外的所需资源，使该线程处于可运行状态(Runnable)

Java 运行系统通过调度选中一个 Runnable 的线程，使其占有 CPU 并转为运行中状态(Running)，此时系统真正执行线程的 run()方法

一个正在运行的线程因某种原因不能继续运行时进入阻塞状态(Blocked)

线程结束后处于死亡状态(Dead)

同一时刻如果有多个线程处于可运行状态，则他们需要排队等待 CPU 资源。此时每个线程会获得一个优先级(priority)。优先级的高低反映线程的重要或紧急程度。可运行状态的线程按优先级排队，线程调度依据优先级基础上的“先到先服务”原则。

线程调度管理器负责线程排队和 CPU 在线程间的分配，并由线程调度算法进行调度。当线程调度管理器选种某个线程时，该线程获得 CPU 资源而进入运行状态。

线程调度是先占式调度，即如果在当前线程执行过程中一个更高优先级的线程进入可运行状态，则这个线程立即被调度执行。先占式调度分为：独占式和分时方式

独占方式下，当前执行线程将一直执行下去，直到执行完毕或由于某种原因主动放弃 CPU 或 CPU 被一个更高优先级的线程抢占

分时方式下，当前运行线程获得一个时间片，时间到时，即使没有执行完也要让出 CPU，进入可运行状态，等待下一个时间片的调度。系统选中其他可运行状态的线程执行分时方式的系统使每个线程工作若干步，实现多线程同时运行。

线程调度规则：

如果两个或是两个以上的线程都修改一个对象，那么把执行修改的方法定义为被同步的（Synchronized）,如果对象更新影响到只读方法，那么只度方法也应该定义为同步的。

如果一个线程必须等待一个对象状态发生变化，那么它应该在对象内部等待，而不是在外部等待，它可以调用一个被同步的方法，并让这个方法调用 wait()。

每当一个方法改变某个对象的状态的时候，它应该调用 notifyAll()方法，这给等待队列的线程提供机会来看一看执行环境是否已发生改变

wait()、notify()、notifyAll()方法属于 Object 类，而不是 Thread 类，仔细检查看是否每次执行 wait()方法都有相应的 notify()或 notifyAll()方法，且它们作用与相同的对象在 java 中每个类都有一个主线程，要执行一个程序，那么这个类当中一定要有 main 方法，这个 man 方法也就是 java class 中的主线程。你可以自己创建线程，有两种方法，一是继承 Thread 类，或是实现 Runnable 接口。一般情况下，最好避免继承，因为 java 中是单继承，如果你选用继承，那么你的类就失去了弹性，当然也不能全然否定继承 Thread,该方法编写简单,可以直接操作线程,适用于单继承情况。至于选用那一种，具体情况具体分析。

线程中的主要方法

isAlive()方法用来判断线程是否终止，getPriority()方法用来得到线程的优先级，线程的优先级分为 10 个等级，默认是 5。setPriority()方法可以修改线程的优先级，例如：

```
public class TestIsAlive {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        System.out.println(t.isAlive());
        System.out.println(t.getPriority());
        t.setPriority(t.getPriority() + 3);
        System.out.println(t.getPriority());
        t.start();
    }
}

class MyThread extends Thread {
    public void run() {
    }
}
```


sleep()方法是 Thread 类的静态方法，可以直接调用。可以指定 sleep 的时间，单位是毫秒或纳秒，例如：

```
public class TestSleep {

    public static void main(String[] args) {

        MyThread t = new MyThread();

        t.start();

        for(int i=0;i<100;i++){

            if(i % 10 == 0){

                try {

                    Thread.sleep(1000);

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

            System.out.println("MainThread :" + i);

        }

    }

}
```

```
class MyThread extends Thread {
    public void run(){
        for(int i=0;i<100;i++){
            if(i % 10 == 0){
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("MyThread :" + i);
        }
    }
}
```

join()方法用于合并 2 个线程，在哪个线程中 join，则该线程最后执行，例如：

```
public class TestJoin {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
```


```
            try {
                t1.join();
            } catch (InterruptedException e) {
            }
            for (int i = 1; i <= 10; i++) {
                System.out.println("i am main thread");
            }
        }
}

class MyThread extends Thread {
        public void run() {
            for (int i = 1; i <= 10; i++) {
                System.out.println("i am thread " + i);
                try {
                    sleep(1000);
                } catch (InterruptedException e) {
                    return;
                }
```

```
            }
        }
    }
```

yield()方法用于让出 CPU，进入 runnable 状态，让出 CPU 只是暂时不执行，而不是永远失去执行机会，例如：

```
public class TestYield {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("t1");
        MyThread t2 = new MyThread("t2");
        t1.start();
        t2.start();
    }
}

class MyThread extends Thread {
        MyThread(String s){
            super(s);
        }
        public void run(){
```


```
            for(int i =1;i<=100;i++){
                System.out.println(getName()+": "+i);
                if(i%10==0){
                    yield();
                }
            }
        }
    }
```

wait()、notify()、notifyAll()不是 Thread 类中的方法，是 Object 类中的方法，所以每个对象都有这些方法来操作锁。

wait()允许我们将线程变成睡眠状态，即 blocked 状态，同时又积极地等待条件的改变，只有在 notify()或 notifyAll()发生变化的时候，线程才被唤醒。wait()类似 sleep()方法，存在区别。区别在于 wait()同时又“积极”地等待条件发生改变，sleep()无法做到。因为我们有时候需要通过同步 synchronized 来防止线程之间的冲突，而一旦使用同步，就要锁定对象，也就是获取对象锁，其它要使用该对象锁的线程都只能排队等着，等到同步方法或者同步块里的程序全部运行完才有机会。在同步方法和同步块中，sleep()不可能自己被调用的时候解除锁定，他们都霸占着正在使用的对象锁不放。而 wait()却可以，它可以让同步方法或者同步块暂时放弃对象锁，而将它暂时让给其它需要对象锁的人(这里应该是程序块或线程)，这意味着可在执行 wait()期间调用线程对象中的其他同步方法，在其它情况下(sleep)这是不可能的。

wait()方法放弃对象锁后可以通过两种方式拿回对象锁。第一种方法，限定借出去的时间。在 wait()中设置参数，比如 wait(1000)，以毫秒为单位，表明我只借出去 1 秒，一秒钟之后自动收回对象锁。第二种方法，让获得对象锁的人通知我，即使用 notify()方法。因此，我们可将一

个 wait()和 notify()置入任何同步方法或同步块内部，也只能在同步方法或者同步块里面调用 wait()和 notify()。

我们可以通过下边的例子来观察 wait()和 sleep()的区别：

```
public class TestWaitAndNotify {

    public static void main(String[] args) {

        MyThread b = new MyThread();

        b.start();

        System.out.println("b is start....");

        synchronized (b)

        {

            System.out.println("Waiting for b to complete...");

            b.notify();

            System.out.println("Completed.Now back to main thread");

        }

        System.out.println("end");

    }

}


class MyThread extends Thread {
```


```
    public void run() {
        synchronized (this) {
            System.out.println("MyThread is running..");
            for (int i = 0; i < 5; i++) {
                if(i == 2){
                    try {
                        wait();     //sleep(10000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("thread : " + i);
            }
        }
    }
}
```

线程同步

线程同步是一个非常复杂的问题，由于同一进程的多个线程共享同一片存储空间，在带来方便的同时，也带来了访问冲突这个严重的问题。Java 语言提供了专门机制以解决这种冲突，有效避免了同一个数据对象被多个线程同时访问。

首先我们来想象一个实际当中银行取钱存钱的例子。张三和李四共用一个账户，张三和李四在不同的营业厅办理存取，原来的账户上有余额 3000 元，张三和李四分别取出 2000 元，假设是同时进行操作。在实际当中，当一个人操作成功后，另一个人应为余额不足而不能取钱。这就是线程同步的结果，如果没有线程同步，则在他们同时操作的时候，有可能都能去出 2000 元，并且余额为 1000 元。显然这样的情况是必须要避免的，那么就必须考虑线程同步的问题了。下面的例子简单的模拟了这个场景，save（） 和 get（）方法加上 synchronized 和不加 synchronized 会出现不同的结果，请大家仔细体会。

```
/**
 *  模拟存钱取钱的线程同步问题
 */
public class User implements Runnable {
    Account account;// 账户
    String name;// 操作账户的人
    public void run() {
        account.set();    //account.get();
    }
    User(Account account, double num) {
```


```
            this.account = account;
            account.setNum(num);
        }
        public static void main(String[] args) throws Exception {
            Account a = new Account(3000);
            User u = new User(a, 2000);
            Thread t1 = new Thread(u, "shangsan");
            Thread t2 = new Thread(u, "lisi");
            t1.start();
            t2.start();
        }
}
class Account {
        double num;// 操作金额
        double money;// 账户余额
        Account(double money) {
            this.money = money;
        }
        /**
```

```
 * 获得账户余额
 */
public double getMoney() {
    return money;
}
/**
 * 获得操作金额
 */
public void setNum(double num) {
    this.num = num;
}
public double getNum() {
    return num;
}
/**
 * 存钱操作
 */
public synchronized void save() {
    double d1 = this.getMoney();// 这里将操作分开写为了模拟同时操作
```


```
        double d2 = this.getNum();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.money = d1 + d2;
        System.out.println(money);
    }
    /**
     * 取钱操作
     */
    public synchronized void get() {
        if (this.money >= num) {
            double d1 = this.getMoney();// 这里将操作分开写为了模拟同时操作
            double d2 = this.getNum();
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
            this.money = d1 - d2;
            System.out.println(money);
        } else {
            throw new ArithmeticException("账户余额不足账户余额:" + getMoney());
        }
    }
}
```


第九章　网络通信

计算机网络

计算机网络的最简单定义是：一些相互连接的、以共享资源为目的的、自治的计算机的集合。最简单的计算机网络就是只有两台计算机和连接它们的一条链路，即两个节点和一条链路。因为没有第三台计算机，因此不存在交换的问题。最庞大的计算机网络就是因特网。它由非常多的计算机网络通过许多路由器互联而成。因此因特网也称为“网络的网络”。

计算机网络的功能主要表现在硬件资源共享、软件资源共享和用户间信息交换三个方面。

（1）硬件资源共享。可以在全网范围内提供对处理资源、存储资源、输入输出资源等昂贵设备的共享，使用户节省投资，也便于集中管理和均衡分担负荷。

（2）软件资源共享。允许互联网上的用户远程访问各类大弄数据库，可以得到网络文件传送服务、远地进程管理服务和远程文件访问服务，从而避免软件研制上的重复劳动以及数据资源的重复存贮，也便于集中管理。

（3）用户间信息交换。计算机网络为分布在各地的用户提供了强有力的通信手段。用户可以通过计算机网络传送电子邮件、发布新闻消息和进行电子商务活动。

网络通信协议

在计算机网络中计算机与计算机之间的通信离不开通信协议，通信协议实际上是一组规定和约定的集合。两台计算机在通信时必须约定好本次通信做什么，怎样通信，什么时间通信等。因此，通信双方要遵从相互可以接受的协议（相同或兼容的协议）才能进行通信，如目前因特网上使用的 TCP/IP 协议等，任何计算机连入网络后只要运行 TCP/IP 协议，就可访问因特网。

通信协议三要素：

（1）语法：　确定通信双方通信时数据报文的格式

（2）语义：　确定通信双方的通信内容

（3）时序规则：指出通信双方信息交互的顺序，如：建链，数据传输，数据重传，拆链等。

网络通信接口

为了使两个节点之间能够通信，必须在他们之间建立通讯接口，使彼此之间能够进行信息交换。

通信协议的分层

由于通信的节点之间联系复杂，在制定协议时，将复杂的成份分成一些简单的成份，在将他们整合起来。例如常见的层式结构，上一层可以和下一层发生联系，而与下下一层没有直接的关系，各层之间互不影响。

通常将应用程序作为最高层，将物理通信线路作为最底层，之间又分为若干层，每一层都有自己的协议和接口标准，例如 ISOOSI 的 7 层模型和 TCP/IP 的 4 层模型。

IP 协议

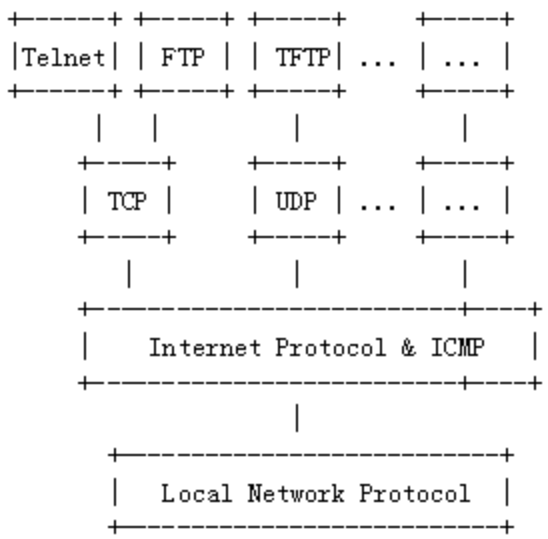
IP（Internet Protocol）协议是网络层次的主要协议之一，用于将多个包交换网络连接起来的，它在源地址和目的地址之前传送一种称之为数据报的东西，它还提供对数据大小的重新组装功能，以适应不同网络对包大小的要求。IP 的责任就是把数据从源传送到目的地。它不负责保证传送可靠性，流控制，包顺序和其它对于主机到主机协议来说很普通的服务。IP 协议由主机到主机协议调用，而此协议负责调用本地网络协议将数据报传送以下一个网关或目的主机。例如 TCP 可以调用 IP 协议，在调用时传送目的地址和源地址作为参数，IP 形成数据报并调用本地网络（协议）接口传送数据报。IP 实现两个基本功能：寻址和分段。IP 可以根据数据报报头中包括的目的地址将数据报传送到目的地址，在此过程中 IP 负责选择传送的道路，这种选择道路称

为路由功能。如果有些网络内只能传送小数据报，IP 可以将数据报重新组装并在报头域内注明。IP 模块中包括这些基本功能，这些模块存在于网络中的每台主机和网关上，而且这些模块（特别在网关上）有路由选择和其它服务功能。对 IP 来说，数据报之间没有什么联系，对 IP 不好说什么连接或逻辑链路。

IP 使用四个关键技术提供服务：服务类型，生存时间，选项和报头校验码。服务类型指希望得到的服务质量。服务类型是一个参数集，这此参数是 Internet 能 够提供服务的代表。这种服务类型由网关使用，用于在特定的网络，或是用于下下一个要经过的网络，或是下一个要对这个数据报进行路由的网关上选择实际的传送 参数。生存时间是数据报可以生存的时间上限。它由发送者设置，由经过路由的地方处理。如果未到达时生存时间为零，抛弃此数据报。对于控制函数来说选项是重 要的，但对于通常的通信来说它没有存在的必要。选项包括时间戳，安全和特殊路由。报头校验码保证数据的正确传输。如果校验出错，抛弃整个数据报。

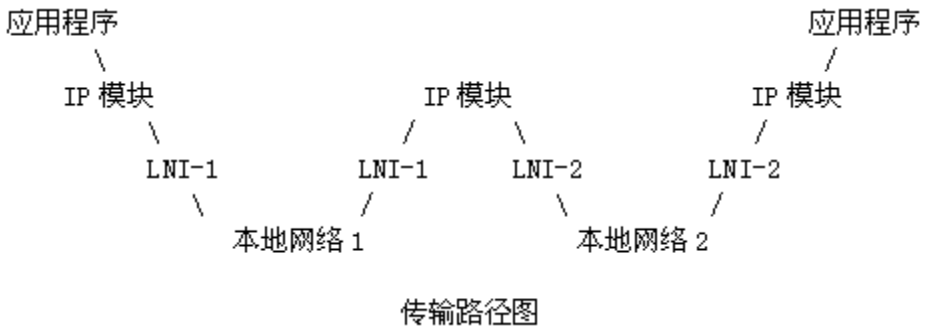
IP 不提供可靠的传输服务，它不提供端到端的或（路由）结点到（路由）结点的确认，对数据没有差错控制，它只使用报头的校验码，它不提供重发和流量控制。如果出错可以通过 ICMP 报告，ICMP 在 IP 模块中实现。

IP 协议在协议体系中的位置：



协议关系图

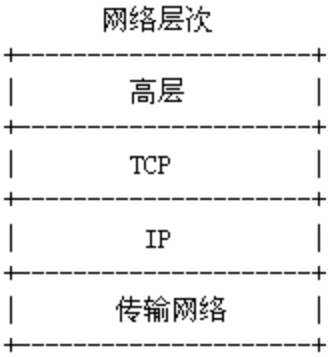
IP 协议的传输路径：

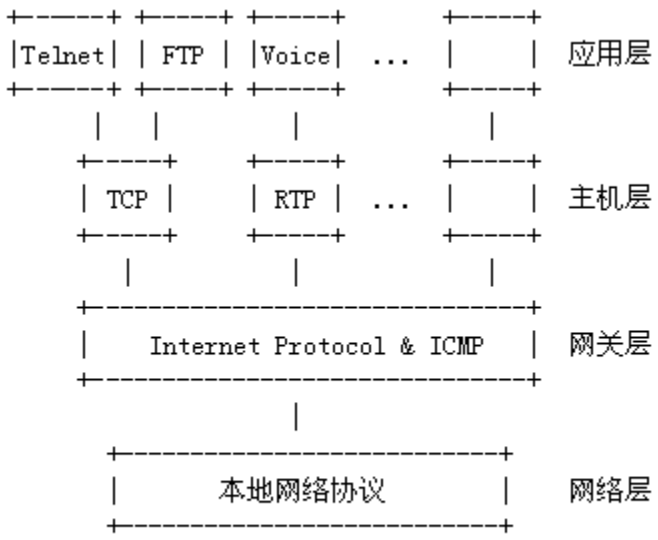


传输路径图

TCP 协议

TCP（transmission control protocol）TCP 协议是为了在主机间实现高可靠性的包交换传输。TCP 协议主要在网络不可靠的时候完成通信，TCP 是面向连接的端到端的可靠协议。它支持多种网络应用程序。TCP 对下层服务没有多少要求，它假定下层只能提供不可靠的数据报服务，它可以在多种硬件构成的网络上运行。TCP 的下层是 IP 协议，TCP 可以根据 IP 协议提供的服务传送大小不定的数据，IP 协议负责对数据进行分段，重组，在多种网络中传送。



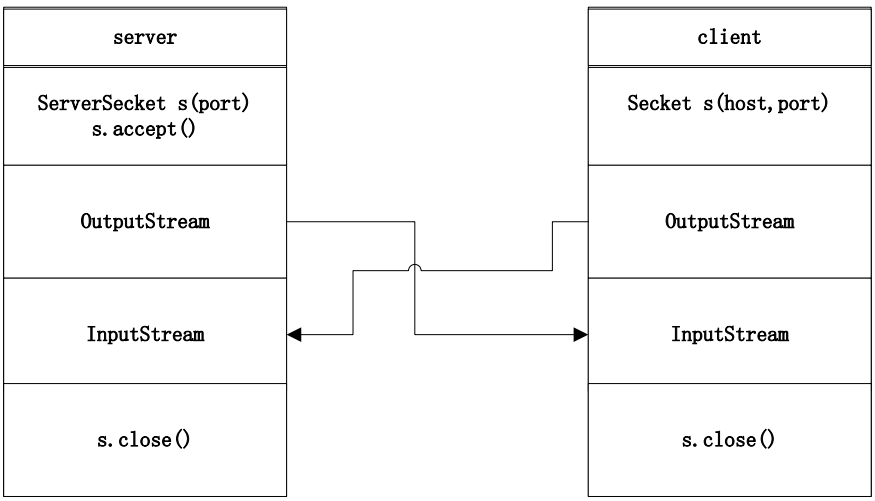


网络层次联系图

关于TCP更详细的说明可以参考 http://www.longen.org/s-z/details-z/TCP.htm。

Socket

两个 Java 应用程序可以通过一个双向的通信网络连接实现数据交换，双向网络的一段就是一个 Socket。Socket 通常用来实现 client-server 的连接。在 java.net 包中定义了两个类 Secket 和 ServerSecket，分别用来实现 client 和 server 端。在用 Secket 建立连接时需要远程计算机的 IP 地址和端口号。



```
import java.net.*;
import java.io.*;
public class TestServer {
    public static void main(String args[]) {
        try {
            ServerSocket s = new ServerSocket(8888);
            while (true) {
                Socket s1 = s.accept();
                OutputStream out = s1.getOutputStream();
                DataOutputStream dos = new DataOutputStream(out);
                dos.writeUTF("Hello," + s1.getInetAddress() + "port : "
                        + s1.getPort() + "    bye-bye!");
                dos.close();
                s1.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
            System.out.println("程序运行出错:" + e);
        }
```


```
        }
}
import java.net.*;
import java.io.*;
public class TestClient {
        public static void main(String args[]) {
                try {
                        Socket s1 = new Socket("127.0.0.1", 8888);
                        InputStream in = s1.getInputStream();
                        DataInputStream dis = new DataInputStream(in);
                        System.out.println(dis.readUTF());
                        dis.close();
                        s1.close();
                } catch (ConnectException connExc) {
                        connExc.printStackTrace();
                        System.err.println("服务器连接失败！ ");
                } catch (IOException e) {
                        e.printStackTrace();
                }
```

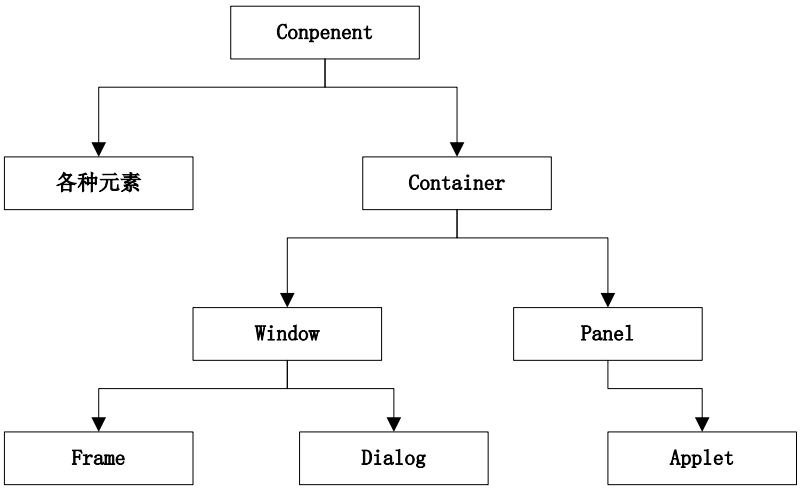
```
    }
}
```

注意运行时要首先运行 TestServer。

第十章　GUI

AWT

AWT（Abstract Window Toolkit）用于 Java Application 的 GUI（Graphic User Interface）编程，包括很多类和接口。GUI 的各种元素有 JAVA 类来实现，使用 AWT 所用到的类存在于 java.awt 包和其子包中，Container 和 Component 是 AWT 中的两个核心类。



Component & Container

Java 的图形用户界面的最基本组成部分是 Component，Component 类及其子类的对象用来描述以图形化的方式显示在屏幕上并能于用户进行交互的 GUI 元素。一般的 Component 对象不能独立的显示出来，必须将其放在一个 Container 对象中才可以显示。Container 是 Compoenet 的子类，Container 子类对象可以包含别的 Component 对象。Container 对象也可以被当做 Component 对象添加到其他 Container 对象中。常见的 Containter 有 Window 和 Panel。Window 对象表示自由停泊的顶级窗口，Panel 对象可以容纳其他 Component 对象，但不能独立存在，必须被添加到其他 Container 中。

Frame

Frame 是 Window 的子类，Frame 及其子类创建的对象表示一个窗体。常用的构造方法：Frame()和 Frame(String s)创建一个标题为 s 的窗体。

方法	说明
setBounds(int x,int y,int width,int height)	设置窗体的位置和大小，x、y 是左上角坐标，width 和 height 是宽度和高度
setSize(int width,int height)	调整组件的大小
setLocation(int x, int y)	将组件一道新的位置
setBackGround(Color c)	设置背景颜色
setVisible(Boolean b)	设置是否可见
setTitle(String s)	设置标题
getTitle(String s)	得到标题
setResizable(Boolean b)	设置是否可以调整大小

```
import java.awt.*;

public class TestFrame {

    public static void main(String args[]) {

        Frame f = new Frame("Frame Test");

        f.setLocation(300, 300);

        f.setSize(170, 100);

        f.setBackground(Color.blue);

        f.setResizable(false);

        f.setVisible(true);
```


```
    }
}
```

Panel

Panel 是最简单的容器类，应用程序可以将其他组件放在面板提供的空间内，Panel 对象可以容纳 Component，可以有自己的布局管理器，默认为 Flowlayout。

```
import java.awt.*;
public class TestPanel {
    public static void main(String args[]) {
        Frame f = new Frame("Panel Test");
        Panel p = new Panel(null);
        f.setLayout(null);
        f.setBounds(300, 300, 500, 500);
        f.setBackground(new Color(0, 0, 102));
        p.setBounds(50, 50, 400, 400);
        p.setBackground(new Color(204, 204, 255));
        f.add(p);
        f.setVisible(true);
    }
}
```

布局管理器

在 Java 语言中，提供了布局管理器类的对象可以管理 Component 在 Container 中的布局，不必直接设置 Component 的位置和大小。每一个 Container 都有一个布局管理器对象，当容器需要对某个组件进行定位或判断其大小时，就会调用对象的布局管理器，调用 Container 的 setLayout 方法改变布局管理器对象。AWT 提供了 5 中布局管理器：FlowLayout、BorderLayout、GridLayout、CardLayout、GridBayLayout。Panel 类默认的布局管理器是 FlowLayout。Frame 的默认布局管理器是 BorderLayout。当把 Panel 作为一个组件添加到某个容器中后，该 Panel 仍然可以有自己的布局管理器。使用布局管理器时，布局管理器负责各个组件的大小和位置，用户无法设置组件的大小和位置，所有的设置属性的方法都会被布局管理器覆盖。用户可以通过 setLayout(null)来取消容器的布局管理器。

FlowLayout 用于安排有向流中的组件，这非常类似于段落中的文本行。流的方向取决于容器的 componentOrientation 属性。

```
import java.awt.*;

public class TestFlowLayout {

    public static void main(String args[]) {

        Frame f = new Frame("Flow Layout Test");

        Button button1 = new Button("Ok");

        Button button2 = new Button("Open");

        Button button3 = new Button("Close");

        f.setLayout(new FlowLayout(FlowLayout.LEFT));

        f.add(button1);

        f.add(button2);
```


```
            f.add(button3);
            f.setSize(100, 100);
            f.setVisible(true);
        }
    }
```

BorderLayout 可以对容器组件进行安排，并调整其大小，使其符合下列五个区域：北、南、东、西、中。每个区域最多只能包含一个组件，并通过相应的常量进行标识：NORTH、SOUTH、EAST、WEST、CENTER。当使用边框布局将一个组件添加到容器中时，要使用这五个常量之一，例如：

```
import java.awt.*;
public class TestBorderLayout {
    public static void main(String args[]) {
        Frame f;
        f = new Frame("Border Layout");
        Button bn = new Button("NORTH");
        Button bs = new Button("SOUTH");
        Button bw = new Button("WEST");
        Button be = new Button("EAST");
        Button bc = new Button("CENTER");
```

```
            f.add(bn, BorderLayout.NORTH);
            f.add(bs, BorderLayout.SOUTH);
            f.add(bw, BorderLayout.WEST);
            f.add(be, BorderLayout.EAST);
            f.add(bc, BorderLayout.CENTER);

            f.setSize(200, 200);
            f.setVisible(true);
        }
    }
```

GridLayout 类是一个布局处理器，它以矩形网格形式对容器的组件进行布置。容器被分成大小相等的矩形，一个矩形中放置一个组件。

```
    import java.awt.*;
    public class TestGridLayout {
        public static void main(String args[]) {
            Frame f = new Frame("GridLayout Test");
            Button b1 = new Button("b1");
            Button b2 = new Button("b2");
            Button b3 = new Button("b3");
```


```
        Button b4 = new Button("b4");
        Button b5 = new Button("b5");
        Button b6 = new Button("b6");
        f.setLayout (new GridLayout(3,2));
        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);
        f.add(b6);
        f.pack();
        f.setVisible(true);
    }
}
```

CardLayout 对象是容器的布局管理器。它将容器中的每个组件看作一张卡片。一次只能看到一张卡片，容器则充当卡片的堆栈。当容器第一次显示时，第一个添加到 CardLayout 对象的组件为可见组件。卡片的顺序由组件对象本身在容器内部的顺序决定，例如：

```
import java.awt.*;
import java.awt.event.*;
public class TestCardLayout extends Frame implements ActionListener {
```

```
CardLayout cardLayout;

Panel panelCard;

Panel panelButton = new Panel();

Panel panel1 = new Panel();

Panel panel2 = new Panel();

TextField textField1 = new TextField("text1");

TextField textField2 = new TextField("text2");

Button button = new Button("next");

public TestCardLayout(String str) {

    super(str);

    cardLayout = new CardLayout();

    panelCard = new Panel();

    panelCard.setLayout(cardLayout);

    panel1.add(textField1);

    panel2.add(textField2);

    panelCard.add(panel1, "1");

    panelCard.add(panel2, "2");

    panelButton.add(button);

    button.addActionListener(this);
```


```
        this.add(panelCard, BorderLayout.CENTER);
        this.add(panelButton, BorderLayout.SOUTH);
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getModifiers() == InputEvent.ALT_DOWN_MASK) {
            cardLayout.first(panelCard);
        } else {
            cardLayout.next(panelCard);
        }
        System.out.println(e.getModifiers());
        System.out.println(InputEvent.ALT_DOWN_MASK);
    }
    public static void main(String args[]) {
        TestCardLayout c = new TestCardLayout("CardLayout Test");
        c.setSize(300, 200);
        c.setVisible(true);
    }
}
```

GridBagLayout 类是一个灵活的布局管理器，它不要求组件的大小相同便可以将组件垂直、水平或沿它们的基线对齐。每个 GridBagLayout 对象维持一个动态的矩形单元网格，每个组件占用一个或多个这样的单元，该单元被称为*显示区域*。具体实现和属性设置请分别参考 http://java.sun.com/docs/books/tutorial/uiswing/layout/gridbag.html 和 JDK 的 API 文档。

事件

在 TestCardLayout 例子中我们已经用到了事件。Java java 中的事件机制有 3 种角色：event object、event source、event listener。

event object：就是事件产生时具体的“事件”，用于 listener 的相应的方法之中，作为参数，一般存在与 listerner 的方法之中。

event source：具体的接受事件的实体，比如说，你点击一个 button，那么 button 就是 event source，这样你必须使 button 对某些事件进行响应，你就需要注册特定的 listener，比如说 MouseEvent 之中的 MouseClicked 方法，这是他就必须有了 add 方法。

event listener：具体的对监听的事件类，当有其对应的 event object 产生的时候，它就调用相应的方法，进行处理。在 windows 程序设计里边这种相应使用 callback 机制来实现。

我们回过头来在看 TestCardLout 这个例子。首先，在 java 控件对象 button 上添加一个监控对象 button.addActionListener(this)，这就相当于你要对 button 进行监听，先要在他身上绑定一个窃听器一样，这里“button”就是你要监听的对象，this 就是你自己的一个窃听器。第二步就是要考虑怎样造这个窃听器了，我们首先要搞清楚它要实现的功能：它不仅要能够监听 button 的一举一动，还要能够把监听到的事件告诉系统，并让系统对这个事件做出相应的处理。Java 中是通过接口实现这样的功能的。这些接口请参见 jdk 中 java.awt.event 包，里面那几个 XXXListener 就是（不是很多，常用的更少）。在这些接口中定义了一些方法，这些方法就是相应的事件处理方式，它们只由系统调用并且都有一个事件类的对象作为参数，这个参数正是联系发生事件主体 button 和操作系统的纽带。当然接口是虚的，不能产生对象的，所以想必你也猜到，上面的“窃听器”this 的类型肯定是某个实现了某个 Listener 接口的类。好了，现在在回顾一下这个

过程：

（1）用户通过鼠标在 button 对象上做动作（点击鼠标）

（2）这个动作被 this 监听到并产生事件对象 e（即 ActionEvent 的对象）

（3）this 通过事件 e 对象向系统报告

（4）系统调用 this 中实现的 actionPerformed 方法对事件进行处理，并向方法传送事件 e

现在我们已经知道在事件处理过程中我们需要做的事情：

（1）实现某个某些监听接口，重写接口中定义的方法

（2）为需要监听的对象添加监听器

Java 中的事件处理的机制同其他语言类似，请读者认真理解。

附录 I

程序应该有良好的格式

程序的格式非常的重要，我们写出的程序不光是要给自己看，还会给别人看，所以需要我们写出的程序要让别人看着舒服，可读性强。程序的格式有很多，每个公司都会有自己的程序的格式，其中有一些共同点需要我们遵守。

（1）大括号对齐

（2）遇{缩进

（3）方法之间加空行

（4）并排语句之间加空格

（5）运算符两侧加空格

（6）{前加空格

（7）{}成对书写

eclipse 的使用

Eclipse 是一个流行的针对 Java 编程的集成开发环境(IDE)。它还可以用作编写其他语言(比如 C++和 Ruby)的环境，合并各种种类工具的框架，以及创建桌面或服务器应用程序的富客户端平台。如今，Eclipse 开源社区拥有数十个项目，其范围从商务智能到社会网络等各个方面。Eclipse 同时也是管理这些项目的非赢利性组织的名称。

使用 eclipse 能够使我们的开发更方便快捷，掌握 eclipse 的使用技巧也是我们进行复杂项目发开的第一步。

Eclipse是完全开源免费的，我们可以去 http://www.eclipse.org/上下载我们需要的版本。目前最新版本是 3.4.2，可以在 http://archive.eclipse.org/eclipse/downloads/index.php下载之前的版本。在 http://www.eclipseplugincentral.com/还可以下载各种插件来丰富eclipse。

在以后项目开发中使用比较多的是 MyEclipse，MyEclipse 是 Eclipse 的一个插件，6.0 版本以后形成了一个包含 eclipse 的独立的 IDE。在以后的讲解中也都是基于 MyEclipse 的。

MyEclipse可以去 www.myeclipseide.com下载，但是MyEclipse是收费的，可以试用 30 天。

下载安装 Myeclipse 之后，启动进入 MyEclipse。在开始使用 MyEclipse 之前，需要先对 MyEclipse 进行一些设置，能是我们的开发更富有效率。

选中 Wndow>Peferences>General>Workspace>Text file encoding，将字符编码修改成需要的编码。

选中 Wndow>Peferences>Java>Installed JREs，添加自定义版本的 JDK

选中 Wndow>Peferences>Java>Compiler>JDK Comliance，设置编译器的版本

如需进行 web 开发，在安装 web 服务器之后，选中 Wndow>Peferences>MyEclipse>Servers，选择合适的服务器，以 tomcat5 为例，选则 Tomcat>Tomcat 5.x，Tomcat server 设置 Enable，并设置 Tomcat 的安装路径，在 Tomcat>Tomcat 5.x>JDK 中设置 Tomcat 的 JDK 版本。

为方便以后的开发还需要设置，选中 Wndow>Peferences>General>Content Types>Text>JSP，修改 Default encoding 为 workspace 的编码方式。选中 Wndow>Peferences>MyEclipse>JSP>Creating files，修改编码同 worksspace 编码。

由于 MyEclipse 继承了很多插件，速度上比 eclipse 慢，可以关掉 validateion、Startup and Shutdown 等来提高启动速度。当然不是必须的。

如果需要安装其他插件，方法同 eclipse，选择 Help>Software Updates>Find and Install>Search for new features to install 进行安装。

Eclipse 快捷键

Ctrl+1 快速修复(最经典的快捷键,就不用多说了)

Ctrl+D: 删除当前行

Ctrl+Alt+↓ 复制当前行到下一行(复制增加)

Ctrl+Alt+↑ 复制当前行到上一行(复制增加)

Alt+↓ 当前行和下面一行交互位置(特别实用,可以省去先剪切,再粘贴了)

Alt+↑ 当前行和上面一行交互位置(同上)

Alt+← 前一个编辑的页面

Alt+→ 下一个编辑的页面(当然是针对上面那条来说了)

Alt+Enter 显示当前选择资源(工程,or 文件 or 文件)的属性

Shift+Enter 在当前行的下一行插入空行(这时鼠标可以在当前行的任一位置,不一定是最后)

Shift+Ctrl+Enter 在当前行插入空行(原理同上条)

Ctrl+Q 定位到最后编辑的地方

Ctrl+L 定位在某行 (对于程序超过 100 的人就有福音了)

Ctrl+M 最大化当前的 Edit 或 View (再按则反之)

Ctrl+/ 注释当前行,再按则取消注释

Ctrl+O 快速显示 OutLine

Ctrl+T 快速显示当前类的继承结构

Ctrl+W 关闭当前 Editer

Ctrl+K 参照选中的 Word 快速定位到下一个

Ctrl+E 快速显示当前 Editer 的下拉列表(如果当前页面没有显示的用黑体表示)

Ctrl+/(小键盘) 折叠当前类中的所有代码

Ctrl+×(小键盘) 展开当前类中的所有代码

Ctrl+Space 代码助手完成一些代码的插入(但一般和输入法有冲突,可以修改输入法的热键,也可以暂用 Alt+/来代替)

Ctrl+Shift+E 显示管理当前打开的所有的 View 的管理器(可以选择关闭,激活等操作)

Ctrl+J 正向增量查找(按下 Ctrl+J 后,你所输入的每个字母编辑器都提供快速匹配定位到某个单词,如果没有,则在 stutes line 中显示没有找到了,查一个单词时,特别实用,这个功能 Idea 两年前就有了)

Ctrl+Shift+J 反向增量查找(和上条相同,只不过是从后往前查)

Ctrl+Shift+F4 关闭所有打开的 Editer

Ctrl+Shift+X 把当前选中的文本全部变成大写

Ctrl+Shift+Y 把当前选中的文本全部变为小写

Ctrl+Shift+F 格式化当前代码

Ctrl+Shift+P 定位到对于的匹配符(譬如{}) (从前面定位后面时,光标要在匹配符里面,后面到前面,则反之)

Alt+Shift+R 重命名 (是我自己最爱用的一个了,尤其是变量和类的 Rename,比手工方法能节省很多劳动力)

Alt+Shift+M 抽取方法 (这是重构里面最常用的方法之一了,尤其是对一大堆泥团代码有用)
Alt+Shift+C 修改函数结构(比较实用,有 N 个函数调用了这个方法,修改一次搞定)

Alt+Shift+L 抽取本地变量(可以直接把一些魔法数字和字符串抽取成一个变量,尤其是多处调用的时候)

Alt+Shift+F 把 Class 中的 local 变量变为 field 变量 (比较实用的功能)

Alt+Shift+I 合并变量(可能这样说有点不妥 Inline)

Alt+Shift+V 移动函数和变量(不怎么常用)

Alt+Shift+Z 重构的后悔药(Undo)

全局 查找并替换 Ctrl+F

文本编辑器 查找上一个 Ctrl+Shift+K

文本编辑器 查找下一个 Ctrl+K

全局 撤销 Ctrl+Z

全局 复制 Ctrl+C

全局 恢复上一个选择 Alt+Shift+↓

全局 剪切 Ctrl+X

全局 快速修正 Ctrl1+1

全局 内容辅助 Alt+/

全局 全部选中 Ctrl+A

全局 删除 Delete

全局 上下文信息 Alt+?

Alt+Shift+?

Ctrl+Shift+Space

Java 编辑器 显示工具提示描述 F2

Java 编辑器 选择封装元素 Alt+Shift+↑

Java 编辑器 选择上一个元素 Alt+Shift+←

Java 编辑器 选择下一个元素 Alt+Shift+→

文本编辑器 增量查找 Ctrl+J

文本编辑器 增量逆向查找 Ctrl+Shift+J

全局 粘贴 Ctrl+V

全局 重做 Ctrl+Y

全局 放大 Ctrl+=

全局 缩小 Ctrl+-

全局 激活编辑器 F12

全局 切换编辑器 Ctrl+Shift+W

全局 上一个编辑器 Ctrl+Shift+F6

全局 上一个视图 Ctrl+Shift+F7

全局 上一个透视图 Ctrl+Shift+F8

全局 下一个编辑器 Ctrl+F6

全局 下一个视图 Ctrl+F7

全局 下一个透视图 Ctrl+F8

文本编辑器 显示标尺上下文菜单 Ctrl+W

全局 显示视图菜单 Ctrl+F10

全局 显示系统菜单 Alt+-

Java 编辑器 打开结构 Ctrl+F3

全局 打开类型 Ctrl+Shift+T

全局 打开类型层次结构 F4

全局 打开声明 F3

全局 打开外部 javadoc Shift+F2

全局 打开资源 Ctrl+Shift+R

全局 后退历史记录 Alt+←

全局 前进历史记录 Alt+→

全局 上一个 Ctrl+,

全局 下一个 Ctrl+.

Java 编辑器 显示大纲 Ctrl+O

全局 在层次结构中打开类型 Ctrl+Shift+H

全局 转至匹配的括号 Ctrl+Shift+P

全局 转至上一个编辑位置 Ctrl+Q

Java 编辑器 转至上一个成员 Ctrl+Shift+↑

Java 编辑器 转至下一个成员 Ctrl+Shift+↓

文本编辑器 转至行 Ctrl+L

全局 出现在文件中 Ctrl+Shift+U

全局 打开搜索对话框 Ctrl+H

全局 工作区中的声明 Ctrl+G

全局 工作区中的引用 Ctrl+Shift+G

文本编辑器 改写切换 Insert

文本编辑器 上滚行 Ctrl+↑

文本编辑器 下滚行 Ctrl+↓

全局 保存 Ctrl+X

Ctrl+S

全局 打印 Ctrl+P

全局 关闭 Ctrl+F4

全局 全部保存 Ctrl+Shift+S

全局 全部关闭 Ctrl+Shift+F4

全局 属性 Alt+Enter

全局 新建 Ctrl+N

全局 全部构建 Ctrl+B

Java 编辑器 格式化 Ctrl+Shift+F

Java 编辑器 取消注释 Ctrl+\

Java 编辑器 注释 Ctrl+/

Java 编辑器 添加导入 Ctrl+Shift+M

Java 编辑器 组织导入 Ctrl+Shift+O

Java 编辑器 使用 try/catch 块来包围 未设置，太常用了，所以在这里列出,建议自己设置。也可以使用 Ctrl+1 自动修正。

全局 单步返回 F7

全局 单步跳过 F6

全局 单步跳入 F5

全局 单步跳入选择 Ctrl+F5

全局 调试上次启动 F11

全局 继续 F8

全局 使用过滤器单步执行 Shift+F5

全局 添加/去除断点 Ctrl+Shift+B

全局 显示 Ctrl+D

全局 运行上次启动 Ctrl+F11

全局 运行至行 Ctrl+R

全局 执行 Ctrl+U

全局 撤销重构 Alt+Shift+Z

全局 抽取方法 Alt+Shift+M

全局 抽取局部变量 Alt+Shift+L

全局 内联 Alt+Shift+I

全局 移动 Alt+Shift+V

全局 重命名 Alt+Shift+R

全局 重做 Alt+Shift+Y