


# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)



# 第1章 Java语言概述

---

- 1.1 程序设计语言
- 1.2 面向对象的程序设计语言—Java
- 1.3 Java的开发和运行环境
- 1.4 第一个Java程序

# 1.1 程序设计语言

## ■ 程序设计语言的发展历史

### — 面向机器的编程语言

- 机器语言
- 汇编语言

### — 面向过程的编程语言

- 特点
- C、PASCAL

### — 面向对象的编程语言

- 特点
- VC、Delphi、JAVA



# 1.1 程序设计语言

---

- 面向对象编程语言的四个特点:
  - 对象
  - 继承
  - 封装
  - 消息

## 1.2 面向对象的程序设计语言—Java

---

### ■ 一、JAVA的发展历史

- SUN公司，GREEN小组，1991
- C++, OAK, JAVA
- JAVA名称来源

## 1.2 面向对象的程序设计语言—Java

### ■ 二、JAVA的特点：

- 1、简单性
- 2、面向对象
- 3、安全性
- 4、体系结构中立
- 5、多线程
- 6、分布性
- 7、执行效率

## 1.2 面向对象的程序设计语言—Java

### ■ 三、JAVA与C++的比较

比较内容	JAVA	C	C++
执行方式	解释式	编译式	编译式
是否产生机器码	否，产生字节码	是	是
是否跨平台	是	否	否
运行速度	较慢	快	快
是否有指针类型	否，对象引用	是	是
是否有继承功能	单重继承	否	多重继承



## 1.3 Java的开发和运行环境

- 一、JAVA开发环境
  - J2SDK及使用方法
- 二、JAVA编辑工具
  - JCreator及使用方法





## 1.4 第一个Java程序

---

- JAVA程序分类:
- 一、JAVA Application程序
- 二、JAVA Applet程序

## 1.4 第一个Java程序

---

### ■ JAVA Application程序

#### ■ 特点:

- 独立
- 解释运行
- 主类, `main()`方法

#### ■ 编写和执行

- 编写源代码
- 源代码的编译
- 运行字节码文件

#### ■ Hello world程序

## 1.4 第一个Java程序

---

- **JAVA Applet程序**

- **特点**

- 嵌入在HTML文件中
- 浏览器执行
- 无main()方法

- **编写和执行**

- 编写源代码
- 编写HTML文件调用applet
- 编译过程
- 解释执行

- **Hello world程序**




# 总结

---

- 理解JAVA的特点
- 课后习题练习

# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)



## 第2章 Java语言基础

---

- 2.1 Java的符号
- 2.2 Java的数据类型
- 2.3 基本数据类型
- 2.4 数据类型间的相互转换
- 2.5 运算符和表达式
- 2.6 简单的输出与输入

## 2.1 Java的符号

### ■ 简单程序举例

- `Import Java.io.*;` //引入类库, **Java.io**:包名, ' .':成员运算符,'\*':包中所有的类
- `Public class ex1_1` //定义类, **ex1\_1**:类名,';':一行语句结束标识符
- `{ // }`':作用域的界定符, 类体及方法体的起始标志
- `Public static void main(String [] args)` //**main()**方法, 程序入口
- `{//`公有, 静态, 无返回值, 方法名, 字符串数组
- `System.out.println("I like Java!");` // 标准输出语句
- `}` //带换行的打印输出, **print**为不带换行
- `} //` '//'':行注释符, 块注释符: `/*.....*/`

## 2.1 Java的符号

---

- JAVA符号集和标识符
- 字符集：采用16位编码的Unicode字符集；
- 标识符：命名记号，规则：
  1. 以字母、下划线(\_)、美元符号(\$)开始的一个字符序列，后面可以跟字母，数字，\_，\$。
  2. 严格区分大小写，没有长度限制。
  3. 可使用汉字做标识符。字母指英文字母和序号大于0xC0H的字符。



## 2.1 Java的符号

---

### ■ JAVA关键字

- 具有专门意义和用途的特殊字和保留字，都用小写字母表示。
  - 关键字主要用来：各种定义，程序流程控制，修饰说明及布尔常量，共49个；
  - 保留字：JAVA规范中未用到单被系统保留的，有8个。

## 2.1 Java的符号

### ■ JAVA注释和分隔符

#### ■ 注释符:

- 行注释: //
- 块注释: /\*.....\*/
- 文档注释: /\*\*.....\*/ 可被JavaDoc程序搜索并编译成程序开发文档。

#### ■ 分隔符

- ‘.’点号, ‘;’分号, ‘ ’空格, ‘{ }’花括号;
- ‘{ }’为语句块, 语句块可嵌套。

## 2.2 Java的数据类型

### ■ 一、数据类型的分类

- 基本（简单）数据类型：整数、浮点、字符，布尔类型；占用存储空间固定。
- 引用（复合）数据类型：类，接口，数组；数据存储取决于数据类型的定义。

数据类型	所占位数	所占字节	默认值	数的范围
char	16	2	\u0000	0 ~ 65535
byte	8	1	0	-2 <sup>7</sup> ~ 2 <sup>7</sup> -1
short	16	2	0	-2 <sup>15</sup> ~ 2 <sup>15</sup> -1
int	32	4	0	-2 <sup>31</sup> ~ 2 <sup>31</sup> -1
long	64	8	0L	-2 <sup>63</sup> ~ 2 <sup>63</sup> -1
float	32	4	0.0F	3.4e- <sup>38</sup> ~ 3.4e+ <sup>38</sup>
double	64	8	0.0D	1.7e- <sup>308</sup> ~ 1.7e+ <sup>308</sup>
boolean	8	1	false	true, false

## 2.2 Java的数据类型

### ■ 二、变量和常量

#### ■ 常量：

- 直接量；
- 定义的常量标识符，数据在执行过程中不可修改；  
`final` 常量类型 常量名=常量值；  
//例如： `final int NUM=100;`

#### ■ 变量：

- 先定义后使用；
- 变量名，类型，作用域；  
类型 变量名[=变量初值]；  
//例如： `int num=100;`

## 2.3 基本数据类型

- 一、整型数据：Long, int(默认), short, byte;
- 数值进制：
  - 十进制：非0开头,如：4, -15;
  - 八进制：以0开头，如：054;
  - 十六进制：以0x开头，如：0x12AF;
- 存储空间：
  - byte在机器中占8位，short占16位，int占32位，long占64位。
  - 64位长整数以l或L结尾
    - 没有以l或L结尾的数字，根据其实际值所属范围，可以被用作byte，short，或int型整数
    - 以l或L结尾的数字，无论其实际值所属范围怎样，都被用作long型整数

## 2.3 基本数据类型

### ■ 二、浮点型数据-实数

■ float以F/f结尾, double（默认）以D/d结尾;

■ 表示形式:

– 1. 十进制数形式

由数字和小数点组成，且必须有小数点，如  
**0.123, .123, 123., 123.0**

– 2. 科学计数法形式（底数\*10的幂）

如：**123e3**或**123E3**，其中**e**或**E**之前必须有数字，  
且**e**或**E**后面的指数必须为整数。

## 2.3 基本数据类型

### ■ 三、字符型数据

- 字符型数据代表16位的Unicode字符
- 字符常量是用单引号括起来的一个字符或以转义符(\)开头的字符。
  - ‘a’ ‘B’ ‘\n’ ‘\u0030’
- 字符型数据的取值范围为
  - 0~65535 或者说 \u0000~\uFFFF
  - \u0000为缺省值
- 示例
  - `char c1;`                    \\ 缺省值为0
  - `char c2 = '0';`            \\ 赋初值为字符 ‘0’
  - `char c3 = 32;`            \\ 用整数赋初值为空格

## 2.3 基本数据类型

### ■ 三、字符型数据

#### ■ 特殊字符的常量表示法:

- 反斜线 (Backslash) `'\\'`
- 退格 (Backspace) `'\b'`
- 回车 (Carriage return) `'\r'`
- 进纸符 (Form feed) `'\f'`
- 制表符 (Form feed) `'\t'`
- 换行 (New line) `'\n'`
- 单引号 (Single quote) `'\''`
- 双引号 (double quote) `'\"'`
- 八进制数 ( `'\0' ~ '\377'` ) `'\DDD'`
- Unicode字符 `'\uHHHH'`



## 2.3 基本数据类型

- 四、布尔型数据
- 布尔型数据只有两个值true和false，且它们不对应于任何整数值
- 布尔型变量的定义如：  
`boolean b = true;`
- 布尔型数据只能参与逻辑关系运算：  
– && || == != !

## 2.4 数据类型间的相互转换

- 自动转换（隐式）与强制转换
- JAVA规定：
  - 1、将占内存少的短数据类型转化为占内存多的长数据类型，系统将自动转换。
  - 2、将长数据类型转换为短数据类型，必须使用强制类型转换。

## 2.4 数据类型间的相互转换

- 自动类型转换
- 整型、实型、字符型数据可以混合运算。运算中，不同类型的数据先转化为同一类型，然后进行运算，转换从低级到高级：
  - 低----->高
  - `byte, short, char` → `int` → `long` → `float` → `double`

## 2.4 数据类型间的相互转换

### ■ 强制类型转换：

— 格式：变量 = （数据类型）表达式；

### ■ 数据类型转换必须满足如下规则：

1. 不能对**boolean**类型进行类型转换。
2. 不能把对象类型转换成不相关类的对象。
3. 在把容量大的类型转换为容量小的类型时必须使用强制类型转换。
  - 转换过程中可能导致溢出或损失精度
    - `int i = 8; byte b=(byte)i;`
    - `(byte)255 == -1 (byte)0x5634 == 0x34`
  - 浮点数到整数的转换是通过舍弃小数得到，而不是四舍五入
    - `(int)23.7 == 23 ;`                      `(int)-45.89f == -45`

## 2.5 运算符和表达式

### ■ 按操作数的个数分类:

- 一元运算符: `--`, `++`, `-`等;
  - 支持前后缀
- 二元运算符: 大多数运算符;
  - 使用中缀记号;
- 三元运算符: 仅一个 “`? :` ”;
  - 使用中缀记号
  - `op1?op2:op3` //如果`op1`成立执行`op2`,否则执行`op3`;

## 2.5 运算符和表达式

### 一、算术运算符和算术表达式

#### ■ (1) 运算符

– 算术运算符：  $+$ ， $-$ ， $*$ ， $/$ ， $\%$ ， $++$ ， $--$

#### ■ (2) 表达式

- 表达式是由操作数和运算符按一定的语法形式组成的符号序列。
- 一个常量或一个变量名字是最简单的表达式，其值即该常量或变量的值；
- 表达式的值还可以用作其他运算的操作数，形成更复杂的表达式。如：  $x$ ， $\text{num1}+\text{num2}$ ， $a*(b+c)+d$

## 2.5 运算符和表达式

### ■ 注:

- “/”对整数和浮点数不同，整数相除舍去小数部分，浮点数保留。（ $7.0/2=?$ ）
- “%”取余运算，多用于整数，结果余数的正负取决于被除数的正负。
- 自增减运算与++/--出现位置无关；
- 表达式的值与运算符位置有关；  
如： $x=2$ ,  $(++x)*3=9$ , 而  $(x++)*3=6$ ；//使用运算前的值还是运算后的值？

## 2.5 运算符和表达式

### ■ 二、关系运算符和关系表达式

- 关系运算符：  $>$ ， $<$ ， $>=$ ， $<=$ ， $==$ ， $!=$
- 结果为布尔值；

### ■ 三、逻辑运算符和逻辑表达式

- 布尔逻辑运算符： $!$  非， $\&\&$  与， $||$  或；
- 某些表达式不参与计算；



## 2.5 运算符和表达式

### ■ 四、位运算符和位表达式

- 移位运算符：>> 带符号右移（空位补最高符号位），<< 带符号左移（符号位不变，右边补0），>>> 不带符号右移（左边空位补0）。
- 按位逻辑运算：& 与，| 或，^ 异或，~ 取反。

### ■ 几种语言位运算符比较。

运算符	Java	C/C++	Delphi	VBasic
与	&	&	and	and
或	~	~	not	not
取反			or	or
异或	^	^	xor	xor
左移	<<	<<	Shl	无
右移	>>	>>	Shr	无

## 2.5 运算符和表达式

### ■ 五、赋值运算符和赋值表达式

- 赋值运算符：=，及其扩展赋值运算符如+=，-=，\*=，/=，%=，&=，>>=等。
- 赋值语句格式：变量名=表达式；
- 扩展赋值：op1+=op2 即 op1=op1+op2;

### ■ 六、条件运算符和条件表达式

- 条件运算符：?:
- 格式：条件? 表达式1: 表达式2;
- 条件成立，计算表达式1的值，否则表达式2。

## 2.5 运算符和表达式

### ■ 运算符的优先次序

优先级	运算符类型	运算符
优先级最高	一元运算符	[ ] . () (方法调用)
		! ~ ++ -- + - new () (强制类型转换)
优先级较高	算术运算符 位移运算符	* / % + -
		<< >>
优先级较低	关系运算符 位移运算符 逻辑运算符	< <= > > == !=
		& ^
		&&
		?: (三元判断运算符, 例如: A>B?X:Y)
优先级最低	赋值运算符	=
		+= -= *= /= %=

## 2.6 简单的输出与输入

---

- 一、简单的输出举例
- 两种常用标准输出方法：
- `System.out.print(输出内容);` //直接输出到屏幕上；
- `System.out.println(输出内容);` //换行输出

## 2.6 简单的输出与输入

### ■ 二、简单的输入举例

#### ■ 1、字符数据输入

- `System.in.read();`//从键盘读取单个字符;
- `Read()`方法得到的是字符的字节表示形式, 需要使用强制转换转化为字符形式, 如

`char c=(char)System.in.read();`

- 因为可能产生IO异常, 此方法要在`try.....catch`中使用。

#### ■ 例程:

```
try {  
    char c=(char)System.in.read();  
}  
catch (IOException ex) {System.out.println(ex); }
```

## 2.6 简单的输出与输入

- 2、字符串输入
- I、利用标准IO流方法

```
try{  
    BufferedReader in =  
        new BufferedReader (new InputStreamReader(System.in));  
    String s = in.readLine(); //从键盘读取一个字符串  
}  
catch (IOException ex) {System.out.println(ex);} //异常处理  
System.out.println(s); //打印输出前面输入的字符串
```

- II、利用命令行参数的方法
- 在main(String args[])方法的参数数组记录的是命令行的所有参数，每个参数为一个数组元素，数组大小取决于参数个数。可以用这种方法获得字符串输入，命令行参数间用空格隔开。
- System.out.println(args[0]+'&'+args[1]); //输出前面输入的命令行参数字符串

## 2.6 简单的输出与输入

### ■ 3、整数和双精度数的输入

- 其他类型的数据输入必须通过字符串输入方法得到数据，再通过基本数据类的方法将字符串转换为相应类型的数据。

- **Integer.parseInt(String);**//将数字字符串转换为整数
- **Double.parseDouble(String);**//将字符串形式的数字数据转化为双精度数。
- 例: **String x="123";**  
**int m= Integer.parseInt(x); //m=123;**


# 总结

---

- 1 Java的基本代码规则---符号，数据类型，运算符和表达式，变量与常量定义。
- 2 基本数据类型与类型间的相互转换。
- 3 简单的输出与输入方法。



# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)

# 第三章 JAVA流程控制

---

## ■ 3.1 程序的逻辑控制流程 P34

- 程序中的语句是按照编写时写入的顺序一条接一条地执行的，这一过程称为顺序执行。
- 任何程序流程均可以用顺序结构、选择结构、循环结构这三种基本控制结构实现。

## 3.2 选择结构

- Java提供了两种选择结构语句：if语句和switch语句。
- if语句：

```
if (条件表达式) {  
    //做某事的语句;  
    ...  
} //如果是单条语句，可不使用{}  
else {  
    //做另一件事的语句1;  
    ...  
} //如果是单条语句，可不使用{ }
```

## 3.2 选择结构

- switch语句:

```
switch (表达式) {  
    case 值1:  
        语句1;//满足表达式值1的条件执行语句  
        break;  
    case 值2:  
        语句2;//满足值2的条件执行语句  
        break;  
    ...  
    case 值n:  
        语句n;//满足值n的条件执行语句  
        break;  
    default:  
        缺省语句;  
}
```

## 3.3 循环结构

---

- 循环结构使用条件表达式来控制一个（一组）动作的重复执行。
- Java语言中支持的循环语句包括：  
`while`循环、`do-while`循环、`for`循环。

## 3.3 循环结构

---

**while**循环：当型循环，循环条件已知。

```
while (布尔表达式) {  
    语句;
```

```
    ...  
} //如果循环体是单条语句，可不使用{}
```

**do-while**循环：直到型循环，循环条件已知。

```
do {  
    语句;
```

```
    ...  
} while (布尔表达式);
```

## 3.3 循环结构

---

**for**循环：最常用。

```
for ( 初值表达式 ; 布尔表达式 ; 步进表达式 ) {  
    语句;
```

```
    ...  
}
```

//如果循环体是单条语句，可不使用{}

嵌套循环：多重循环，循环体中包括循环语句。

## 3.4 程序跳转语句

---

- **break** : 在while、do-while或for循环中，当某种条件满足时需要立即跳出循环时，使用**break**关键字，可以使程序的执行流程立即跳出循环。
- **continue** : 如果想跳过循环中的部分代码重新开始下一次循环，就使用**continue**关键字。



# 练习：

---

读过《射雕英雄传》的同学，一定还记得黄蓉遇上神算子瑛姑，瑛姑给她出的三道题目中有一题是这样的：今有物不知其数，三三数之剩二，五五数之剩三，七七数之剩二，问物几何？

也就是说，有一个未知数，这个数除以三余二，除以五余三，除以七余二，问这个数是多少？请用Java编程实现（此数在100以内）。

## 3.5 方法

### ■ 一、方法的定义

[修饰字] 返回值类型 方法名称(参数1,参数2,...)

```
{  
    ...语句;    //方法体：方法的内容  
}
```

1. 返回值类型可以是任意的Java数据类型，当一个方法不需要返回值时，返回类型为void。返回值类型和return语句后的表达式类型一致。
2. 参数表可由零个或多个参数组成，参数的类型可以是简单数据类型，也可以是引用数据类型（数组、类或接口），参数传递方式是值传递。
3. 方法体是对方法的实现。它包括局部变量的声明以及所有合法的Java指令。局部变量的作用域只在该方法内部。

## 二、方法的引用

- 方法的调用是实现方法的功能。调用方法时要用一些实际的参数替换方法定义中的参数表。实际参数的个数、顺序与定义中的参数表一致。
- 方法调用有两种形式：
  - 1、将方法调用作为一个表达式语句（适用于无返回值的方法）。`System.out.println("...");`
  - 2、将方法调用作为一个表达式。`S=area(2,3,4);`

# 三、方法调用时参数的传递和返回值

## ■ 1、方法调用时参数的传递

- 方法通过其参数将基本类型变量、数组和类对象，传递给方法体使用。
- 方法参数是数组时，调用方法向被调用方法传递数组的地址。实际参数的数组名称与方法定义中的形式参数的数组名虽然不同，但指向同一处连续存储区，所以被调用方法通过传递过来的数组地址，可以改变调用方法开辟的数组内的元素值。

## 2、方法的返回值

- 在方法体中，通过return语句把一个确定的数值返还给调用该方法的语句。
- Return语句的一般格式：
  - return 表达式;
  - 或
  - return;
- 1. 表达式的值就是方法的返回值，其类型应该与方法的返回值类型一致，不一致就要进行强制类型转换。
- 2. 如果方法没有返回值（void），则省略return后的表达式或不写return语句。

# 方法的例子：


```
public class app03{  
    public static void main(String args[])  
    {  
        int n1,n2,s;  
        n1=10,n2=3;  
        s=SofRectangle(n1,n2); //方法调用,n1,n2是实际参数  
        System.out.println("以n1,n2为边的矩形面积为： " +s);  
    }  
    static int SofRectangle(int side1, int side2) //方法定义  
    {int s; //局部变量， 以及： side1,side2是形式参数。  
      s=side1*side2;  
      return s; //方法的返回值  
    }  
}
```

# 作业1:

---

- P47, 实训2, 实训内容 (1)。
- 编写一个完整的java程序, 提示用户输入一个球的Int型半径, 然后调用方法sphereVolume(方法名) 来计算并显示球的体积。
- 球体积公式:  
$$\text{Volume} = (4/3)\text{Math.PI} * \text{Math.pow}(\text{radius}, 3).$$

# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)



# 第四章 数组和字符串处理

## ■ 4.1 一维数组

数组是一组相同数据类型的变量或者对象的集合，数组元素可以是基本数据类型，也可以是复合类型（如数组）。

在一个数组中：

- 每个元素的数据类型都是相同的。
- 数组中的各个元素是有顺序的。
- 所有元素共用一个数组名，利用数组名和下标来唯一的确定数组中每个元素的位置。  
如：**sum[2]**

# 一、一维数组的定义和创建

## ■ 定义:

■ 格式: 数据类型[] 数组名;

■ 或 数据类型 数组名[];

如: Student[] stu; //Student是自定义类

int sum[]; //定义一个int型数组sum

## ■ 说明:

■ 1、类型可以是任意数据类型, 包括类类型。

■ 2、数组名是合法标识符。

■ 3、[]指明定义的是一个数组类型的变量。

■ 4、数组声明不能在数组名后的扩号内指定数组元素个数。这是数组创建的工作。

如: **int sum[10];**是错误的。

# 创建： 给数组指定长度，分配内存，用new运算符

- 格式：一个数组是一个对象，需用new来创建。

数组名=new 数组类型[数组长度];

- 如：sum=new int[100];

- 也可定义和创建合写：

数据类型[] 数组名=new 数据类型[数组长度];

数据类型 数组名[]=new 数据类型[数组长度];

如：int sum[]=new int[100];

int[] sum=new int[100];

- 说明：

- 1、数组中的数据称为数组元素，数组长度即为数组元素个数。
- 2、数组长度必须为int值，或能自动转为int类型，如short型。
- 3、数组一旦被创建，长度确定，不能再改变大小，可使用“数组名.length”的方法得到数组长度。如：sum.length

## 二、一维数组的引用和初始化

- 一维数组的引用：利用数组下标方法引用。

格式：数组名[下标] 如： **sum[2]**

(1) 数组下标从0开始，最大下标是数组长度减1。

如：最大为： **sum[sum.length-1]**

(2) 下标可以是整型常数或表达式。如： **sum[n-1]**

- 一维数组的初始化(赋值)——四种方法：

1、定义时初始化，格式：

数据类型 数组名[]=**new** 数据类型[] {初值1, 初值2, .....}

如： **Int s[]=new int[]{1,2,3};**

则： **s[0]=1;s[1]=2;s[2]=3;**

## 2、格式(省略创建)：

数据类型 数组名[] = {初值1, 初值2, .....}

如：Int s[] = {1,2,3};     效果同：Int s[] = new int[]{1,2,3};

则：s[0]=1;s[1]=2;s[2]=3;

## 3、如下格式（第六章）：

类型 数组名[] = new 类型 {new 构造方法(参数列表), new 构造方法(参数列表), ..... }

或：类型 数组名[] = {new 构造方法(参数列表), new 构造方法(参数列表), ..... }

## 4、直接为数组的每个元素赋值。

如：s[0]=1; s[1]=2;.....

**注：**数组被定义创建之后，其内所有元素自动初始化，值为相应数据类型的默认值。如：int型数组的所有元素都被初始化为0；对象数组元素初始化为null。

用户不对数组的元素进行赋值，也能直接读取数组元素的值。

# 例子：

---

```
public class ArrayTest
{
    public static void main(String args[])
    {
        int i;
        int a[] = new int[5]; // 定义及创建
        for( i=0; i<5; i++ )
        {
            a[i]=i; // 直接赋值
        }
        for( i=a.length-1; i>=0; i-- ) // a.length 是数组 a 的长度
        {
            System.out.println("a["+i+"] = "+a[i]);
        }
    }
}
```

运行结果： a[4]=4;

.....  
a[0]=0;

## 4.2 二维数组

- Java中数组的元素也可以是数组，当一个数组中的每一个元素都是一个一维数组时就构成了一个二维数组。同理：当每一个元素都是 $n-1$ 维数组时就构成了 $n$ 维数组。
- 如： `int a[ ][ ]=new int[3][4];`  
二维数组，每个元素都是包含4个元素的一维数组。

## 4.2.1 二维数组的定义和创建

- 1、二维数组的定义，格式：

数据类型 数组名[ ][ ]; 或

数据类型[ ][ ] 数组名;

如：int sum2[ ][ ];或 int[ ][ ] sum2;

说明：

- 1、类型可以是任意数据类型，包括类类型。
- 2、数组名是合法标识符。
- 3、当定义多维数组时，用多对[ ]表示多位数组的维数，即n维数组用n对[ ]。



## 2、二维数组的创建

### ■ 两种方式：

#### ■ 1、直接为每一维分配内存

**数据类型 数组名[ ][ ]=new 数据类型[长度1][长度2]**

注：“长度1”和“长度2”都是大于0的整数，分别对应二维（高维）和一维（低维）的长度。

如：`int sum2[][]=new int[3][4];`

2、从高维开始，分别为每一维分配内存。**Java**语言中，由于把二维数组看作数组的数组，数组空间不是连续分配的，所以不要求二维数组每一维的大小相同，即多位数组的每一维的长度可以不同。

## 多位数组的每一维的长度可以不同

- 如: `int sum2[ ][ ];`
- `sum2=new int[3][ ];`//创建第二维长度
- `sum2[0]=new int[3];`//创建第一维长度
- `sum2[1]=new int[2];`
- `sum2[2]=new int[1];`
- 存储形式图:

sum2[0][0]	sum2[0][1]	sum2[0][2]
sum2[1][0]	sum2[1][1]	
sum2[2][0]		

## 4.2.2 二维数组的初始化

### ■ 1、在创建时初始化，格式：

数据类型 数组名=new 数据类型[长度1][ ]{{值00,值01,...},{值10,值11,...},...};

或

数据类型 数组名={{值00,值01,...},{值10,值11,...},...};

注：数组的维数由[ ]的个数来决定，数组的第二维的长度由{ }的对数来决定，第一维的长度由{ }内的值的个数来决定。

如：int a[ ][ ]=new int{{1,2,3},{5,6},{5}}; //3行3列的二维数组。

a[0][0]=1	a[0][1]=2	a[0][2]=3
a[1][0]=5	a[1][1]=6	
a[2][0]=5		

## 2、创建数组后，直接为数组的每个元素赋值

- 如：
- `int sum2[ ][ ]=new int[2][2];`
- `sum2[0][0]=1;`
- `sum2[0][1]=2;`
- `sum2[1][0]=3;`
- `sum2[1][1]=4;`

## 4.2.3 二维数组的引用

- 引用方式：
- 数组名[下标1][下标2]；
- 其中，下标1下标2分别代表二维数组的第二维和第一维（高维和低维）下标。同一维数组类似，数组下标从0开始，最大下标是数组长度减1。

## 4.3 数组的基本操作

### ■ 4.3.1 数组的复制

- **Java**类库**java.lang.System**类提供了一个静态方法用于数组的复制。从源数组复制到目标数组。

```
void arraycopy(Object src, int src_position,  
                Object dst, int dst_position, int length)
```

**src**是源数组，**src\_position**表示要复制“源数组”中此索引序号和以后（包括序号本身）的元素数据；**dst**是目标数组，**dst\_position**表示要将复制来的数据存放在目标数组中此索引序号（包括序号本身）的元素位置；**Object**是数组的类型，**length**表示要复制的数据的个数。（例子：P56，例4.3）

- 范围不能越界；
- 可对任何同类型的数组进行复制；
- 数组复制过程中做严格的类型检查；

## 4.3.2 数组排序举例

在**java.util**包中，专门有一个数组类**Arrays**，该类提供了一些方法用于排序、查找等操作。

其中，**sort()**方法用于对数组元素排序。

格式1: **public static void sort(数据类型[] 数组名)**

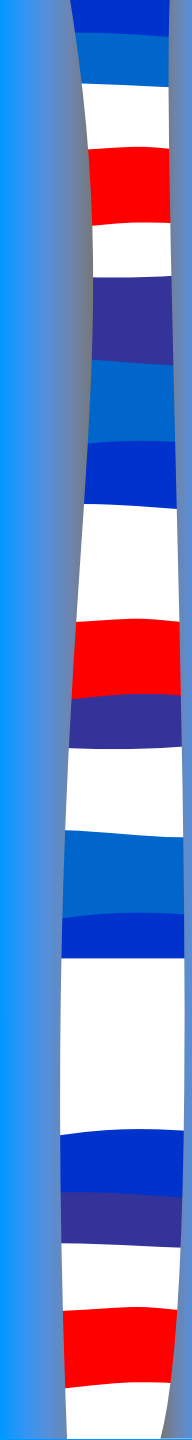
该方法用改进的快速排序方法对指定的数组进行排序，其中数组的类型为**char,byte,short,int,float,double**或**boolean**。

引用: **Arrays.sort(a);** //对数组**a**中元素进行排序。

格式2: **public static void sort(数据类型[] 数组名, int fromIndex, int toIndex)**

该形式只对指定的数组中指定范围内的元素（从数组名**[fromIndex]**到数组名**[int toIndex]**之间的元素，不包括数组名**[int toIndex]**）进行排序。

引用: **Arrays.sort(a, 4, 7);** //对数组**a**中从**a[4]**到**a[6]**元素进行排序。



```
import java.util.*;
public class ArrayTest
{
    public static void main( String args[ ] )
    {
        int a[ ]={9,1,3,4,2,5,7,6,8};
        System.out.println("数组a排序前为: ");
        for(int i=0;i<a.length;i++)
            System.out.print(a[i]+" ");
        System.out.print("\n数组a排序后为: ");

        Arrays.sort(a);//排序语句

        for(int i=0;i<a.length;i++)
            System.out.print(a[i]+" ");
        System.out.println();
    }
}
```



## 4.4 字符串

- 字符串是字符的序列，类似于字符的数组。
- 在**Java**中，字符串被当作对象来处理。
- 程序中需要用到的字符串可以分为两大类：
- 一：创建之后不会再做修改和变动的字符串常量；
- 二：创建之后允许再做更改和变化的字符串变量。

### ■ 4.4.1 字符串常量

- 对于字符串常量，由于程序中经常需要对它做比较，搜索之类的操作，所以通常把它放在一个具有一定名称的对象之中，由程序对该对象完成上述操作。
- 在**Java**中，存放字符串常量的对象用**String**类，有专门的属性来规定它的长度。对于所有用“”括起来的字符串常量，系统都会缺省的为它创建一个无名的**String**类型对象。

## 4.2.2 String类字符串

- 要创建String类的一个对象并进行初始化，需要调用String类的构造方法。String类的构造方法及其使用方法：

- 1、`public String();` //public 可省略不写

无参数的缺省构造方法，用来创建一个空字符串变量。如：

```
String s = new String( ); = String s="";
```

- 2、`public String( String value);`

利用已经存在的字符串常量创建一个新的String对象，该对象的内容与给出的字符串常量一致。此字符串常量可是另一个String对象，也可是一个双引号引起的字符串常量。如：`String s=new String("hello");`

### ■ 3、public String (StringBuffer buffer);

- ◆ Buffer为StringBuffer类对象，此构造函数利用一个已经存在的StringBuffer对象为新建的String初始化，StringBuffer对象代表内容、长度可变的字符串变量。

如： `String s = new String(Sb);`

- 分配一个新的字符串s，它包含字符串缓冲区参数中当前包含的字符序列（Sb的内容）。      参数： Sb --- 一个 StringBuffer对象。

### ■ 4、public String( char value[ ] );

通过给构造方法传递一个已存在的字符数组可以创建一个非空字符串。

如： `char chars[ ] = { 'a', 'b', 'c' };`

`String s = new String( chars );`

则： `s="abc";`

# String对象的声明和创建

- String的声明:

如: `String s;` //仅声明一个String对象, s值为null。

- String的创建:

如: `s=new String("ABC");`

- 合写: `String s=new String("ABC");`

- 省略方法: `String s="ABC";`

Java系统会自动为一个双括号引起的字符串常量创建一个String对象, 所以此语句的实际含义和效果与前一个句子完全相同。

## 4.2.2 String类字符串操作常用方法

1.求字符串长度。 `public int length( );`

返回字符个数，汉字也算一个Unicode字符。

2.求字符串某个位置的字符（ $0 \sim *.length-1$ ）.

`public char charAt(int i);`

如: `String s="abc"; s.charAt(1) = ? ;`

3.连接字符串（将指定字符串连接到此字符串的结尾）。

`public String concat(String str);` //也可用“+”运算符连接

4.查找某个字符（当前字符串中某特定字符出现的位置）

`public int indexOf(int ch) // s.index((int)'b')= ?`

返回指定字符在此字符串中第一次出现处的索引。

`public int indexOf(int ch, int fromIndex)`

返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。

# String类字符串操作常用方法

## 5. 查找字符串中的子串

`public int indexOf(String str) // s.indexOf("bc") = ?`

返回指定子字符串在此字符串中第一次出现处的索引。

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

返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始。

## 6. 比较两个字符串

`public int compareTo(String anotherString)`

按字典(a-b-...-z)从小到大顺序比较两个字符串,大为正,小为负,等为0。

`// s.compareTo("abd") = ?`

`public int compareToIgnoreCase(String str)`

按字典顺序比较两个字符串，不考虑大小写。

`public boolean equals(Object anObject)`

将此字符串与指定的对象比较。

`public boolean equalsIgnoreCase(String anotherString)`

将此 String 与另一个 String 比较，不考虑大小写。

注：在JDK API 1.6.0中有所有 `java.lang.String` 类的方法，遇到新问题可“查字典”解决。

## 4.5 命令行参数

---

### ■ 1. main方法。

java Application程序中，必须有且只有一类定义一个主方法`main( )`，程序名和包含主方法的类名相同。

格式：`public static void main(String args[ ])`

主方法必须是公共的`public`，静态存储`static`，无返回值`void`。  
方法参数是字符串类型的数组。

## 2. 命令行参数

---

- 主方法的参数是在输入**java**解释执行应用程序时输入的，又称命令行参数。
- 多个参数之间（包括参数和程序名之间）用空格隔开。
- 执行程序后，命令行参数由字符串数组 **args[]** 接收，并提供给主方法使用，是向 **java Application** 程序传入数据的有效手段。




# 命令行例子：

---

```
■ public class shiyan
■ {
  public static void main(String args[])
■ {
  int n=args.length;
  if(n==0)
    System.out.println("no");
  else
  {
■    System.out.println("number is:"+n);
■    for(int i=0;i<n;i++)
■      System.out.println("No."+i+": "+args[i]);
  }
  }
}
```

# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)



# 第五章 Java面向对象编程基础

- 面向对象的程序设计（OOP）已成为现代软件开发的必然选择。通过掌握面向对象的技术，能开发出复杂、高级的系统，这些系统是完整健全的，但又是可扩充的。OOP是建立在把对象作为基本实体看待的面向对象的模型上的，这种模型可以使对象之间能相互交互作用。
- 面向对象程序设计在一个好的面向对象程序设计语言（OOPL）的支持下能得到最好的实现。Java就是一种优秀的OOPL，它提供了用来支持面向对象程序设计模型所需的一切条件。Java有自己完善的对象模型，并提供了一个庞大的Java类库，并有一套完整的面向对象解决方案和体系结构。

# 5.1 面向对象的几个概念

## ■ 5.1.1 对象、类与实体

### ■ 一、对象：

#### ■ 对象有两个层次的概念：

- 现实生活中对象指的是客观世界的实体；
- 程序中对象就是一组变量和相关方法的集合，其中变量表明对象的状态，方法表明对象所具有的行为。

## 二、类

---

- 类是描述对象的“基本原型”，它定义一类对象所能拥有的数据和能完成的操作。在面向对象的程序设计中，类是程序的基本单元。
- 相似的对象可以归并到同一个类中去，就像传统语言中的变量与类型关系一样。
- 程序中的对象是类的一个实例，是一个软件单元，它由一组结构化的数据和在其上的一组操作构成。

## 5.1.2 对象的属性

---

- 1、状态属性

对象内部包含的各种信息，即变量。

- 2、行为属性

对象的操作，即方法。

- 3、标志

对象的名称

- 对象是具有唯一对象名和固定对外接口的一组属性和操作的集合。

## 5.2 类

---

### ■ 5.2.1 类的定义

对一个用户自定义的类，要为其取一个名字，并指明类中包含哪些变量和方法以及对应的类型、实现等，称为类的定义。

- 类是定义同一类所有对象变量和方法的蓝图原型，对象是一个具体的事物。
- 一个类定义了对对象的一个种类，一个对象则是一个类中的一个实例。
- **Java**中的类由类头和类体组成，其中类头定义类的性质，类体定义类的具体内容。

## 5.2.1 类的定义

### 1、类头定义的语法格式

[修饰符] **class** 类名 [**extends** 父类名] [**implements** 接口名列表]

■ 注释：

■ (1) **Java**的修饰符分为访问控制符和非访问控制符。**Java**的主类定义时必须被修饰为**public**，当缺省此选项时，则定义类为非公有的、非抽象、非最终的。

■ (2) **class**为**Java**关键字，定义一个类时，必须用此关键字。类名为新定义的类的名称。

■ (3) **extends**父类名，说明定义的类是一个已经存在的类的子类。通过此关键字实现类的继承。

■ (4) **implements**接口名列表，说明定义的类将要实现的接口，接口数目不固定。

■ (5) 类头定义的格式中，带方括号的选项为可选项。



## 5.2.1 类的定义

- 2.Java类体的定义
- 类体是由一对大括号括起来的成员方法和成员属性组成的。

[类的修饰字] **class** 类名称 [extends 父类名][implements 接口名列表]  
{

变量定义及初始化;

方法定义及方法体;

} 类体，其中定义了该类中所有的变量和该类所支持的方法，称为成员变量和成员方法。

}

修饰字: **[public] [abstract | final]**  
缺省方式为 **friendly**

## 5.2.2类的成员

- 1.类的成员属性（成员变量）
- 描述该类的内部信息，又称类的静态属性。可以是简单的变量，也可是对象、数组等复合的数据结构。
- （1）声明类属性为简单变量的格式：
- [修饰符] 变量类型 变量名 [=变量初值];
- 如： `int i=0;`
- （2）声明类属性为对象的格式：
- [修饰符] 类名 对象名 [=new 类名（实参表）];
- 如： `Label import = new Label(“请输入 “);`

## 5.2.2类的成员

- 在一个类中定义另外一个类的对象为本类属性时，则要求另一个类在此类中必须可见。
- 系统类可以被所有的Java程序使用，用户定义的类要想在其他类中可见，则只有两种方法：一、将类说明为public；二、与引用的类在同一个程序中。
- 在一个类中要访问其他类的属性，则需要有new运算符。

## 5.2.2类的成员

### ■ 2.类的成员方法

- 即成员函数，类的动态属性，标志了类所具有的功能和操作，用来把类和对象的数据封装在一起，是一段用来完成某种操作的程序片段。
- 成员变量的类型可以任意，变量名称在一个类中保证唯一即可。
- 在类的定义中，可加入对数据成员进行操作的方法。数据成员是面向对象的术语，用于表示类中的数据变量。
- 参数传递方式是值传递。

# 一个简单的类

```
■ class Car{
■     int car_number;
■     void set_number(int car_num)
■     {
■         car_number=car_num;
■     }
■     void show_number()
■     {
■         System.out.println ("My car No. is : "+car_number);
■     }
■ }
■ public class CarDemo{
■     public static void main(String args[])
■     {
■         Car DemoCar=new Car();
■         DemoCar.set_number(3838);
■         DemoCar.show_number();
■     }}
```

My car No. is : 3838

## 5.2.3 类的构造方法

- 创建类的对象时，利用new关键字和一个与类同名的方法来完成，这个方法就是构造方法。
- 构造方法:类中用来初始化新建的对象的方法。
- 构造方法特点：
  - (1) 构造方法名与类名相同；
  - (2) 没有返回值类型；
  - (3) 主要作用是完成对类对象的初始化操作。
  - (4) 不能由编程人员显式的直接调用，在创建一个类的新对象的同时，系统会自动调用该类的构造方法为新对象初始化。
  - (5) 每一个类可以有零个或多个构造方法。当一个类定义多个构造方法时，称构造方法的重载。
- 变量和方法称为类的成员（成员变量和成员方法），而构造方法不是类的成员。

## 5.2.3 类的构造方法

- 如果程序中没有定义构造方法，则创建实例时使用的是缺省构造方法，它是一个无内容的空方法(没有任何参数,没有**body**,不做任何事情)。
- 构造方法可以带参数，也可以重载。
- 系统在产生对象时会自动执行构造方法。
- 当一个类有自定义的构造方法，类的默认构造方法无效，但此时一定要用默认的构造方法时，则必须在程序中显式的定义一下默认的构造方法。
- 格式：  
[修饰符] 类名 ( ) { }    //无形参，无方法体。
- 构造方法应包含的内容：
  - 定义一些初值或内存配置工作。

## 5.2.3 类的构造方法

■ 例：

```
public class Thing
{
```

```
    private int x;
```

```
    public Thing()
```

```
        //构造方法
```

```
    {        x = 47;
```

```
    }
```

```
    public Thing( int new_ x)
```

```
        构造方法可被重载
```

```
    { x = new_ x;
```

```
    }
```

```
}
```

//构



## 5.3 对象

---

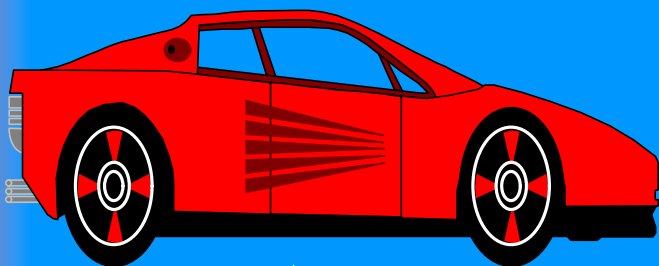
### ■ 5.3.1 对象与类的关系

- 定义类的最终目的是使用这些类，创建并操作某类的对象是使用该类的最主要手段。
- 对象是类的一个实例，类是同种对象的抽象集合，是创建对象的模板。
- 在程序中创建一个对象将在内存中开辟一块空间，其中包括该对象的属性和方法。
- 对象的生成过程(生命周期)：创建、使用、清除。

## 5.3.1 对象与类的关系

抽象数据类型

将对象抽象为类



现实生活中的对象

将类进行实例化

**Class Car**

```
{  
    int color_number;  
    int door_number;  
    int speed;  
  
    void brake() { ... }  
    void speedUp() {...}  
    void slowDown()  
    { ... }  
}
```

## 5.3.2 对象的创建

- 通过new运算符和类的构造方法实现。在定义类时如未定义构造函数系统，java会自动构造一个没有参数的构造函数。
- 格式有三：
  - 类名 对象名=new 类的构造方法名（参数表）；
- 1、将声明和分配内存空间分开定义。
  - 格式： `classname1 c1;`  
`c1= new classname1();`
- 注：如只说明，c1是空对象，系统不为其分配内存空间，无法使用。
- 2、声明同时分配内存。
  - 格式： `classname2 c2=new classname2();`

## 5.3.2 对象的创建

- 3、同时声明一个类的两个对象，对其中一个对象用new分配内存空间，而另一个则是第一个对象的备份。
- 格式：`classname3 c3;`
- `classname3 c4;`
- `c3=new classname3();`
- `c4=c3;`
- 注：C3，C4引用同一个实例对象，C4是C3的一个备份，它们共用相同的内存空间。因此改变一个对象的同时必将改变另一个。
- 例:P80，例5.5。

## 5.3.3 对象的引用

- 使用对象包括：
  1. 从对象中获得信息；
  2. 改变对象的状态；
  3. 使对象执行某些操作。
  
- 程序对对象的访问通过向对象发送消息实现，**Java**对象之间的通信也是通过消息机制。
  
- 实现途径：
  1. 引用对象的变量
  2. 调用对象的方法

## 5.3.3 对象的引用

### ■ 引用对象的变量

格式:

对象名.对象的变量;

```
rect.origin = new Point(15, 37);
```

```
area = rect.height * rect.width;
```

### ■ 说明:

- “.”称为成员运算符，在对象名和成员名之间起到连接作用，指明是哪个对象的成员。
- 可以像使用其它变量一样来使用对象的变量。 例如：  

```
area = rect1.height * rect1.width;
```

## 5.3.3 对象的引用

### ■ 调用对象的方法

格式: 对象名.对象方法名();  
或 对象名.对象方法名(参数表);

例

```
rect.move(15, 37);
```

说明:

- 对于返回值的方法,方法调用可以用于表达式中
- 调用一个对象的方法即是向该对象发送一个消息.

## 5.3.4 清除对象

- java运行使系统有一个垃圾回收进程负责清除不再使用的对象。
- 垃圾回收器
  1. 垃圾回收器定期扫描内存，对于被应用的对象加上标记，按可能的路径扫描结束后清除未加标记的对象。
  2. 被回收的对象是：
    1. 不再被任何引用变量引用的对象
    2. 引用变量自动放弃
    3. 人为地将引用变量置为null
  3. 当系统的内存用尽或程序中调用**System.gc()**要求进行垃圾收集时，垃圾收集线程与系统同步运行。否则垃圾收集器在系统空闲时异步地执行。



## 5.3.4 清除对象

### ■ **finalize ( )** 方法

在一个对象被垃圾回收器回收之前，java解释器会自动调用对象的**finalize ( )** 方法。通常在该方法内包括了释放系统资源的代码段。

### ■ **finalize( )**方法在类java.lang.Object中实现.

如：

```
protected void finalize ( ) throws throwable  
{
```

```
..... // clean up code for this class
```

```
super. finalize( ); //清除对象使用的所有资源， 包括由于继  
                    //承关系而获得的资源
```

```
}
```

# 作业II:

---

- 创建一个学生**Student**类，该类中有学生姓名，学号，生日，专业及年级。
- 其中有：
  - 构造方法初始化所有数据成员
  - 方法**Age()**:计算学生的当前年龄
  - 方法**Display()**: 在屏幕上输出所有学生成员的值
  - 方法**ChangeGrade(int)**:改变学生年级的方法
- 在**main()**中实现创建一个学生对象，计算学生的年龄，改变学生年级，并在屏幕上输出所有学生数据成员的值。

## 5.4 JAVA的修饰符

- 修饰符分类：
- 1、访问控制符：为一组起到限定类、属性或方法被程序里的其他部分访问和调用的修饰符。
- 如： `public,private,protected`,等等
- 2、非访问控制符：作用各不相同。
- 如：  
`static,final,abstract,native,volatile,synchronized`,等等。

## 5.4.1 Java的访问控制符

- 1、public修饰符
- 含义：公有的
- 使用范围：可修饰类、成员变量、成员方法。
- （1）public 修饰类  
例：5.7
- （2）public 修饰属性和方法

## 5.4.1 Java的访问控制符

- 2、`private`修饰符
- 含义：私有的；
- 范围：类的成员属性和成员方法。
- 注：私有成员变量和成员方法的访问权限：它只能由定义它的类本身来访问。
- 例：5.8

## 5.4.1 Java的访问控制符

- 3.protected修饰符
- 含义：保护型的；
- 范围：只能修饰成员属性和成员方法。
- 注：保护型成员变量的权限：能由定义它的类本身和定义它的类的子类（可在同一包中，也可不在），以及与它在同一个包中的其他的类访问。
- 例：5.9

## 5.4.1 Java的访问控制符

- 4.默认的修饰符friendly
- 特点：不是Java的关键字，不能直接使用friendly这个单词。
- 范围：变量，方法，类。
- 注：当一个变量、方法和类的定义没有具体的访问控制符修饰时，则成员变量、方法和类的权限默认为friendly.该成员变量、方法和类能够被与它在同一个包中的其他类访问，即具有包访问特性。
- 例：5.10

# Java的访问控制符的访问权限

限访问修饰符关键字	同一个类中	同一个包中	派生类中	其他包中
<b>public</b>	√	√	√	√
<b>protected</b>	√	√	√	
无访问修饰符关键字 (friendly)	√	√		
<b>private</b>	√			



## 5.4.2 Java的非访问控制符

- 1、Static修饰符
- 含义：静态的。
- 范围：变量，方法。
- 实例成员与类成员；
- static修饰的成员为类成员。
- （1）类变量：所有实例对象共享内存，可直接由类调用。
- （2）类方法：可直接用类来调用它，不用必须通过对象来调用。
- 注：main()的static: java程序均从main方法开始运行，调用main()的是类而不是某个对象，所以应该加static; 且main()均是在类之外被调用，所以还应该加public.
- 例：5.11

## 5.4.2 Java的非访问控制符

- 2. abstract修饰符
- 含义：抽象的。
- 范围：类和方法。
- （1）抽象类
- （2）抽象方法
- 例：5.12

## 5.4.2 Java的非访问控制符

- 3、final修饰符
- 含义：最终的。
- 范围：类、属性、方法。
  - （1）最终类：无子类。
  - （2）最终属性：常量。
  - （3）最终方法：不能覆盖。

## 5.5 其他修饰符和修饰符的混用

- 1.volatile修饰符
- 含义：可变的。
- 范围：属性。
- 注：如果一个类的属性被volatile修饰，则这个属性可能同时被几个线程控制和修改。即这个属性可被其他未知程序的操作影响和修改。

## 5.5 其他修饰符和修饰符的混用

- 2、native修饰符
- 含义：原生的。
- 范围：方法。
- 注：一般用来声明：用其他语言书写方法体并具体实现方法功能的特殊的方法。



## 5.5 其他修饰符和修饰符的混用

- 3、synchronized修饰符
- 含义：同步的；
- 范围：方法；
- 注：主要用于多线程共存的程序中的协调和同步。


## 5.5 其他修饰符和修饰符的混用

### ■ 4、修饰符的混合使用

#### ■ 注意：

- (1) **abstract**不能与**final**并用修饰一个类。
- (2) **abstract**不能与**private,static,final**或**native**并列修饰一个方法。
- (3) **abstract**类中不能有**private**的成员（包括属性和方法）。
- (4) **abstract**方法必须在**abstract**类中。
- (5) **static**方法中不能处理非**static**的属性。

# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)





# 第六章 Java面向对象高级编程

- 6.1 类的继承
- 6.2 类的多态
- 6.3 接口
- 6.4 包
- 6.5 Java的类库

## 6.1 类的继承

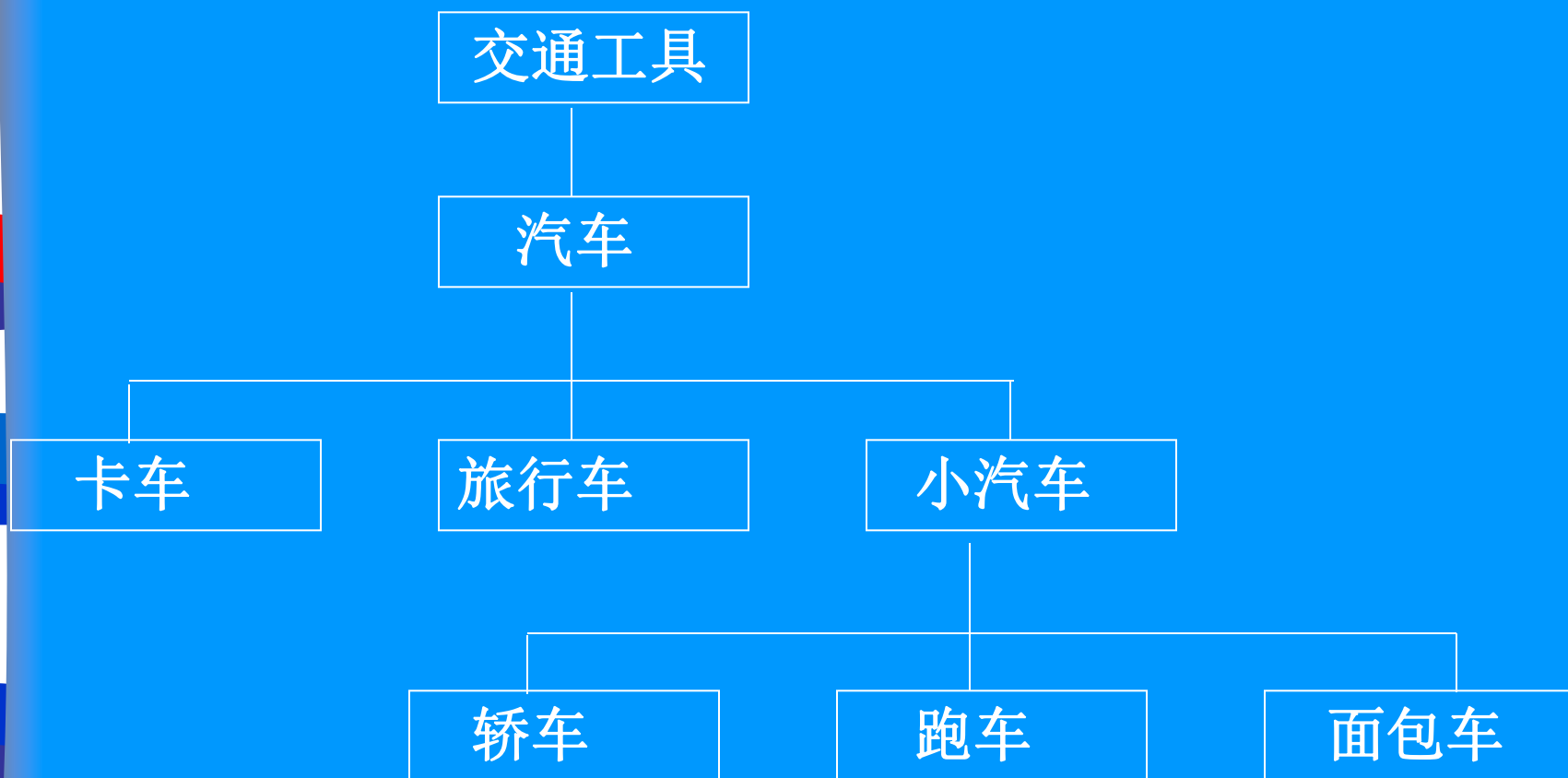
---

- 通过继承可以更有效的组织程序结构，明确类间关系，并充分利用已有的类来完成更复杂、深入的开发。
- **Java**语言以**Object**类作为所有类的最高父类，所有的类都是直接或间接的继承**Object**类得到的。
- **Java**提供不同层次的标准类，使用户可以按需要派生自己的类。

## 6.1.1 继承的概念

- 定义：继承是程序中两个类间的一种关系，在定义一个类时，如果指明了类A继承了类B（B已存在），则类A通常就拥有了类B的成员属性和成员方法，此时称类A和类B间具有继承的关系。
- 直接父类（直接超类）：被继承的类B称为类A的直接父类。父类包括所有直接或间接被继承的类。
- 子类、次类、衍生类：继承于父类的类。子类继承父类的状态和行为，同时也可以修改父类的状态或重写父类的行为，并添加新的状态和行为。

# 理解继承:



## 6.1.1 继承的概念

- 指明继承关系后，父类中的属性（成员）就无须在子类中重复描述，从而实现了一定程度的代码的复用。
- *Java*语言是单继承机制：在*Java*的继承层次中，一个父类可以有多个直接子类，但一个子类只能有一个直接父类，即*Java*不支持多重继承。
- *Java*的类继承不改变成员的访问权限，父类中的成员为公有的(**public**)或被保护的(**protected**)的，则其子类的成员访问权限也即成为公有的或被保护的，而父类中的私有的(**private**)成员变量是不能被继承来的。



## 6.1.2 继承的实现

---

- 1. 继承的实现
- 2. 子类继承父类的原则
- 3. 构造方法的继承

# 1. 继承的实现

---

- 在类头的定义处用**extends**关键字来实现类间的继承。
- 格式：
- [修饰符] **class** 新类名 [**extends** 父类名]
- **extends**为java的关键字，声明了新定义的类为一个已经存在类的子类。
- 如果缺省**extends**子句，则该类为**java.lang.Object**的子类。
- 例： 6.1

## 2. 子类继承父类的原则

- 直接子类继承直接父类应遵循的原则：
- (1)直接子类继承那些被声明为**public**或**protected**的直接父类成员。
- (2)直接子类继承同一包中父类里无访问控制修饰符的成员（包括变量和方法）。
- (3)子类不能继承父类中被**private**修饰的成员。
- (4)若子类中声明了一个与父类同名的成员，则子类不再继承该父类中的该同名的成员。



### 3.构造方法的继承

---

- 在创建子类的对象时，使用子类的构造方法对其初始化，不但要对自身的成员变量赋初值，还要对父类的成员变量赋初值。
- **Java**中允许子类继承父类的构造方法。继承原则有三条。

# 继承原则：

- (1)若父类是无参数的构造方法，则子类无条件的继承该构造方法。
- (2)若子类无自己的构造方法，则它将继承父类的无参数构造方法作为自己的构造方法；若子类有自己的构造方法，则在创建子类对象时，它将先执行继承自父类的无参数构造方法，然后执行自己的构造方法。
- (3)若父类是有参数的构造方法，子类可以通过在自己的构造方法中使用**super**关键字来调用它，但这个调用语句必须是子类构造方法的第一个可执行语句。
- 例：6.2

## 6.1.3 变量的隐藏

- 定义：子类对从父类那里继承来的变量重新加以定义称为变量的隐藏。
- 当在子类中定义了与父类中同名的成员变量时，若在该子类中访问此变量，则父类中的同名变量被隐藏，访问的是子类中定义的同名变量，而非父类中的。
- 如一定要引用父类中的被隐藏的变量，可使用 `super` 关键字。
- 例：6.3

## 6.1.4 方法的覆盖

---

- 定义：子类定义了与父类中同名的方法时，则父类中的方法被覆盖，即方法的结果是子类方法的执行结果。
- 例：6.4

## 6.1.5 this和super关键字

- 1、this的用法
- （1）**this**代表他所在的类本身的实例对象，相当于类对象的另一个名字，利用**this**可以调用当前对象的方法和变量。
- 特：当方法的参数与成员变量重名且该方法中同时使用参数和访问该方法所属类的成员变量时，无法区分哪一个是形参名，哪一个是变量名，这时要用到**this**。
- 例：6.5

## 6.1.5 this和super关键字

- (2)可用this来调用它所在的类的构造方法。
- 例： 6.6

```
class car{ int color; float speed;
```

```
car(int color)
```

```
{this.color=color;}
```

//构造方法I， this.color代表变量， color代表形参

```
car(float speed){
```

```
this(21);           //引用构造方法I， 必须是第一句。
```

```
this.speed=speed;}
```

## 6.1.5 this和super关键字

- 2、super的用法
- (1) 表示他所在的类的直接父类。通过super可以访问父类中被子类中隐藏的变量和覆盖的方法。
- 例：6.7
- (2)在子类中，用super调用父类的构造方法，初始化父类的成员变量。super语句要放在第一句。
- 例：6.8

## 6.1.4 this和super关键字

```
■ class father
■ {   int i,j,k;
■     String abc="I am father!";
■     father(int i,int j){this.i=i; this.j=j;}
■ }
■ class son extends father
■ {   int c;
■     String abc="I am son!";
■     son(int a,int b,int c)
■     {       super(a,b);//super调用父类的构造方法，必须在第一句
■             k=23;
■             this.c=c;
■             System.out.println("i="+i+"\nj="+j+"\nk="+k+"\nc="+c);
■     }
■     void output()
■     {   System.out.println(super.abc);    //super代表直接父类，访问被
隐藏的父亲类的变量。
■     }
■ }
```



## 6.2 类的多态

---

### ■ 6.2.1 方法的重载

- 含义：在同一个类中，以相同的名字定义多个方法，只是参数列表不同，即为方法的重载。
- 程序可以按参数决定调用对应的方法，此决定由编译器来做。即在调用时，**Java**将根据实参个数或参数类型选择匹配的方法。
- 例：6.10

## 6.2.1 构造方法的重载

---

- 即一个类中定义了多个构造方法，只是参数列表不同。实例化对象时，系统会根据传递参数的不同自动调用相应的构造方法。
- 详见5.2.3节。

## 6.3 接口

---

### ■ 6.3.1 接口的概念

- 接口（界面）：是用来组织应用中的各类并调节它们的相互关系的一种结构，是用来实现类间多重继承的结构。
- 接口是一种特殊的类，它是方法定义（没有方法的实现）和常量值的集合。
- 多重继承：一个子类可以有一个以上的直接父类，该子类可继承它所有直接父类的成员。

## 6.3.1 接口的声明

### ■ 接口声明格式:

[public] interface 接口名 [extends 父接口名列表]

{ //接口体

    [public] [static] [final] 域类型 常量名=常量值;

    //常量域声明, 可为多个

    [public] [abstract] [native] 返回值类型 方法名 ([参数列表]) [throws 异常列表];

    //抽象方法声明, 可为多个

}

## 6.3.1 接口的声明

- 由接口声明的语法格式可以看出，接口是一种由常量和抽象方法组成的特殊的类。**Java**中一个类只能有一个父类，但是它可以同时实现若干个接口，这种情况如果把接口理解为特殊类，则这个类利用接口实现了多重继承。
- 注：
  - （1）**interface**是定义接口的关键字。
  - （2）接口名只要符合**Java**对标识符的规定即可。
  - （3）接口的访问控制符只有一个**public**,用此修饰的接口为公共接口，可以被所有类和接口使用，否则只能被同一个包中的其他类和接口使用。

## 6.3.1 接口的声明

- (4) 接口具有继承性。定义一个接口时可通过**extends**声明该新接口是某个已存在的父接口的派生接口（子接口），它将继承父接口的所有属性和方法。与类不同的是一个接口可以有一个以上的父接口，它们之间用“,”隔开，形成父接口列表。新接口将继承所有父接口中的属性和方法。如果在子接口中定义了和父接口同名的常量或方法，则父接口中的常量被隐藏，方法被重写。
- (5) 接口体的声明是定义接口的重要部分。接口体由两部分组成，一部分是对接口中属性的声明，另一部分是对接口中方法的声明。
- (6) 接口中的所有属性都必须是**public static final**修饰的，这是系统默认的规定。所以接口属性也可以没有任何修饰符，其效果完全相同。
- (7) 接口中的所有方法都必须是抽象方法。

## 6.3.3 接口的实现

- 含义：为具体实现接口所规定的功能，则需要某个类为接口的抽象方法书写语句并定义实在的方法体，称为接口的实现。
- 类要实现某个或某几个接口时的步骤和注意：
  - （1）在类的声明部分，用**implements**关键字声明该类将要实现哪些接口。
  - （2）如果实现某接口的类不是**abstract**类，则在类的定义部分必须要实现指定接口的所有抽象方法，即为所有抽象方法定义方法体，而且方法头部分应该与接口中的定义完全一致，即有完全相同的返回值和参数列表。

## 6.3.3 接口的实现

- (3) 如果实现某接口的类是抽象类，则它可以不实现该接口所有的方法，但是对于这个抽象类任何一个非抽象的子类而言，它们父类所实现的接口中所有抽象方法都必须有实在的方法体。这些方法体可以来自抽象的父类，也可以来自子类本身，但是不允许存在未被实现的接口方法。
- (4) 一个类在实现某接口的抽象方法时，必须使用完全相同的方法头。如果所实现的方法与抽象方法有相同的方法名和不同的参数列表，则只是一个新的方法，而不是实现已有的抽象方法。
- (5) 接口的抽象方法，其访问限制符都已指定是 **public**，所以类在实现方法时，必须显式地使用 **public** 修饰符，否则将被系统警告为缩小了接口定义的方法和访问控制范围。



## 6.3.3 接口的实现

在使用接口的类体中可以使用：

- 接口中定义的常量
- 必须实现接口中定义的所有方法。
- 在类中实现接口所定义的方法时，方法的声明必须与接口中所定义的完全一致。
- 接口可以作为一种引用类型来使用，所以可以象其它类型的一样使用。

例：

假如 `Sleeper` 是接口

```
private Sleeper[] sleepers = new  
Sleeper[MAX_CAPACITY];
```

# 关于接口的两个问题讨论：

i: 接口是否能看成多重继承呢？

接口虽然能完成多重继承的相似功能，但是完全不同的：

- 一个类仅仅继承接口中的常量。
- 一个类不能继承接口中的方法。
- 接口继承独立于类的继承。执行相同接口的类，不一定在类继承上相关。

ii: 接口的用处主要体现在哪几个方面？

- 通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系。
- 通过接口可以声明多个类需要实现的方法。
- 通过接口可以了解对象的交互界面，而不需了解对象所对应的类。

## 6.4 包

---

### ■ 定义:

包是一组相关的类或接口的集合, 它提供了访问保护和名字空间管理。**Java**编译器使用文件系统目录来保存包。

### ■ 使用包的好处:

1. 程序员能很容易确定同一包中的类是相互关联的。
2. 程序员能很方便地了解到在哪里可以找到能完成特定功能的类。
3. 由于每个包都创建了一个名字空间, 个人创建的类名不会和其它包中的类名名发生冲突。
4. 可以使同一包中的类彼此不加限制地访问, 而同时对其它包中的类提供访问控制。

## 6.4.1 包的创建

### 1. 创建包

格式: `package 包名1[. 包名2[. 包名3]];`

`package`语句必须放在源文件的开始处, 第一条语句。

例如: `java.net java.io java.util java.lang`  
`java.applet javax.swing`

```
package graphics;  
public class Circle {  
    . . .  
}
```

## 6.4.1 包的创建

---

### 说明：

- package的作用域是整个源文件；
- 如果在同一个源文件中定义了多个类，最多仅可以有一个类用public修饰，且源文件名必须与该类名相同；
- 当未使用package语句时，类和接口放在无名缺省包里；
- 包的命名习惯：将Internet域名作为包名。  
例如： `com.company.region.package`.

## 6.4.2 包的引用

1.包的使用：使用长名字引用包中的公共成员，即在类名前注明包名。

- 要引用其他包中类的成员，需要在类名前再加上包名前缀。包名前缀.类名前缀.成员名；

- 例：

```
java.awt //包名.类名
```

```
java.awt.Label mc1= new java.awt.Label();
```

- 注：包中的所有类中只有public类能被包外部的类访问。

## 2.包的导入

- 用import关键字来导入一个包，使得该包的某些类或所有类能被直接使用。在Java源程序中，如果有package语句，则import语句紧接在其后，否则import语句应放在程序首行。
- import语句有三种形式：
  - 1) 格式： **import 包名;** //这种形式允许使用只用包名的最后一部分来指定包名。  
例： 如： `import java.awt;` //这条语句指定这个包中的文件存储在目录 `path/java/awt` 下,包层次的根目录 `path` 是由环境变量 `CLASSPATH` 来确定的。
- 则： `awt.Label mc1=new awt.Label();`

## 6.4.2 包的引用

2) 格式: **import** 包名.类名;

例: 如: `import java.awt.Label;`

则: `Label mc1=new Label();`

3) 使用import语句引进包中的所有类和接口

格式: `import 包名.*;`

■ 例: 如: `import java.awt.*;`

则: `Label mc1=new Label();`

■ 注: 这种形式使得该包中所有类都可以直接用类名来访问, 但同时会浪费内存的开销。





```
package com.bruceeckel.util;
```

```
/*文件都置于自己系统的一个子目录中:
```

```
C:\DOC\JavaT\com\bruceeckel\util */
```

```
public class Vector {
```

```
    public Vector()
```

```
    { System.out.println("com.bruceeckel.util.Vector"); }}
```

```
/*需预先设置环境变量classpath:
```

```
    CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT */
```

```
package c05;
```

```
import com.bruceeckel.util.*; //另外包中的类用Vector 类:
```

```
public class LibTest {
```

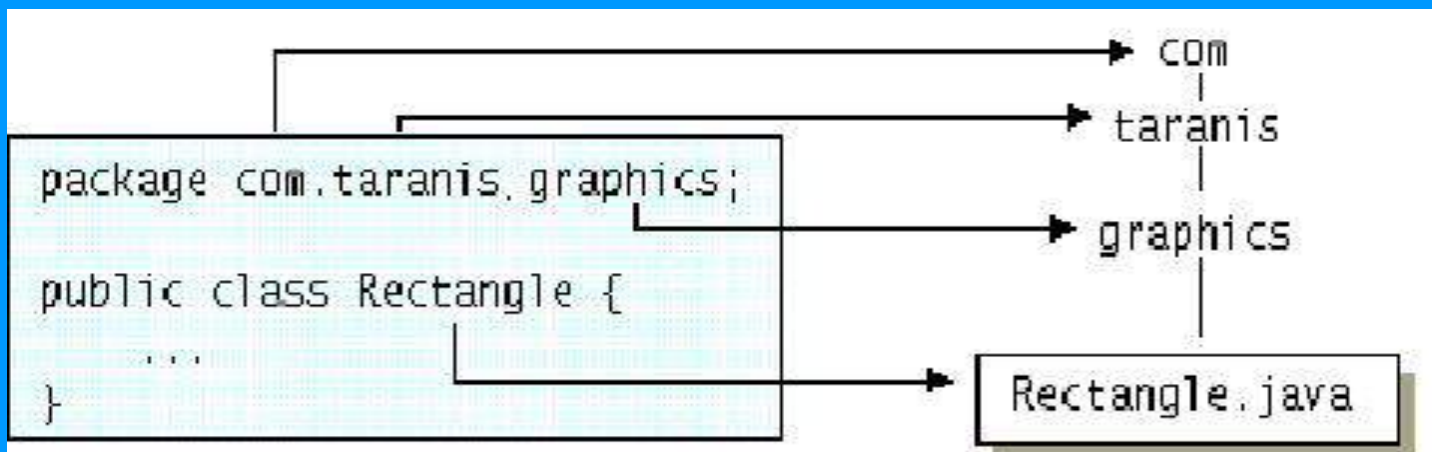
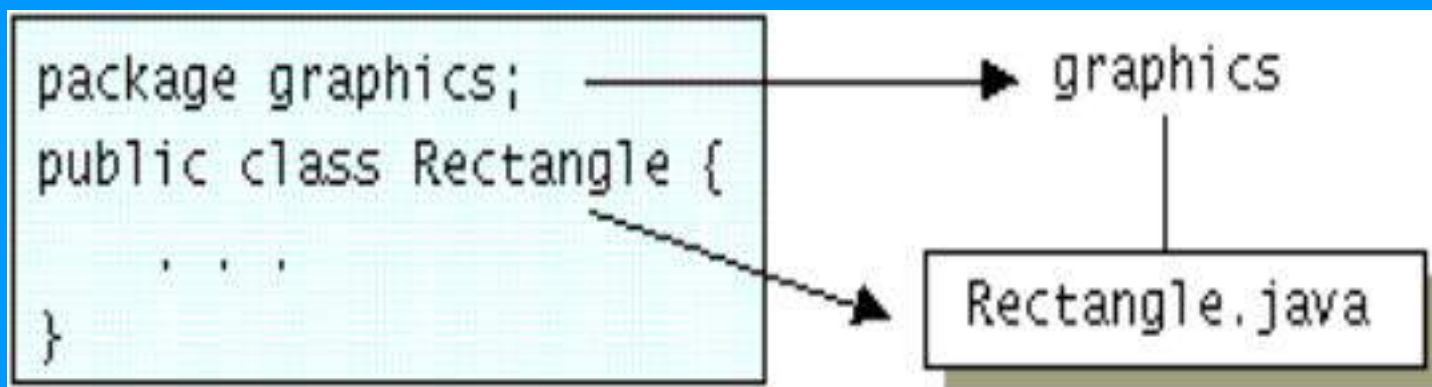
```
    public static void main(String[] args)
```

```
    { Vector v = new Vector(); }
```

```
}
```

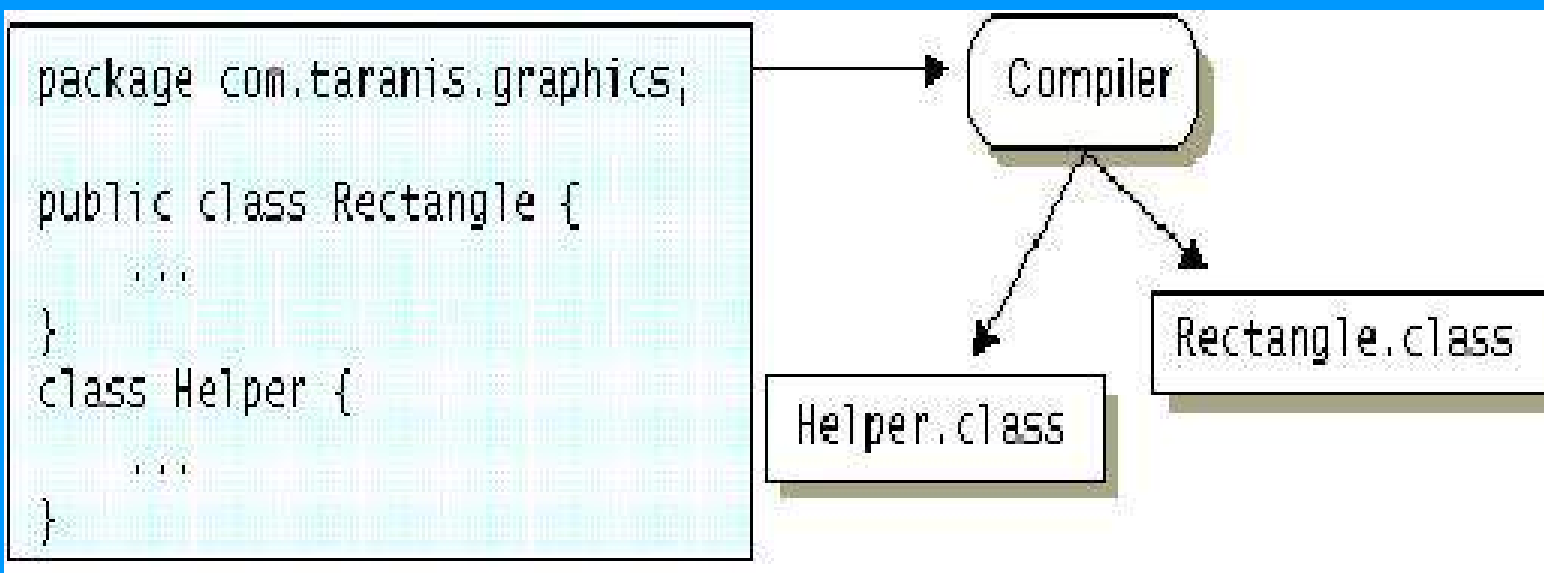
### 3. 源文件和类文件的管理

- JDK利用文件系统的层次结构来管理源文件和类文件。源文件和类文件所在的目录名应与其中的类所在的包名对应，编译器和解释器按此机制来查找类。如：



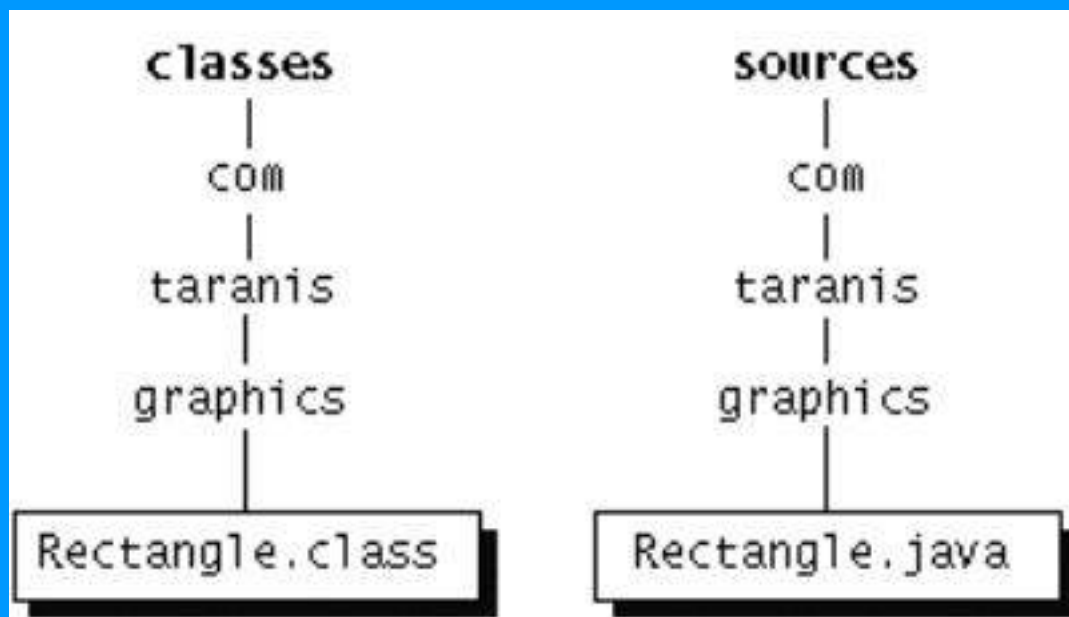
### 3. 源文件和类文件的管理

- 当源文件中有多个类定义时，编译后将产生多个类文件。



### 3. 源文件和类文件的管理

- 虽然类文件所在的目录序列必须与包名对应，但不一定与源文件在相同的目录中。



```
javac -d d:\classes Rectangle.java
```

# 3. 源文件和类文件的管理

说明:

- 包对应于文件系统的目录管理方式可以帮助编译器和解释器找到程序中的类和接口存放的地方。

- CLASSPATH指出用户的类库所在目录, 在类路径 (CLASSPATH) 中只需指出包路径之上的顶层路径

CLASSPATH= . ; d:\classes

- 设置类路径的方法

1. 设置环境变量CLASSPATH (不推荐)

set classpath=. ; d:\classes

2. 在调用编译器或解释器时使用-classpath选项  
例

```
java -classpath d:\classes com.taranis.grathpics.Test
```

# 类成员的访问控制

限访问修饰符关键字	同一个类中	同一个包中	不同包 子类中	其他包 其他类中
<b>public</b>	√	√	√	√
<b>protected</b>	√	√	√	
无访问修饰符关键字 (friendly-package)	√	√		
<b>private</b>	√			

## 6.5 Java的类库

- 6.5.1 Java类库简介--当前版本1.6。
- Java系统事先设计并实现了一些体现了常用功能的标准类。这些系统标准类根据实现的功能不同，可划分为不同的集合，每个集合是一个包，合称为类库。
- 一个用户程序中系统标准类使用得越多、越全面、越准确，这个程序的质量就越高。
- Java的标准类和类库类似于C语言的库函数，都是一种应用编程接口，简称为**API (Application Program Interface)**，是开发编程人员所必须了解和掌握的。
- 开发者的Java编程能力在相当程度上取决于他对java类库的熟悉程度。
- Java的标准类和类库的详细内容见JDK文档。  
[JDK API 1.6 zh\\_CN.CHM](#)。

## 6.5.2 语言基础类库


- java.lang包—语言基础类库简介
- 1.Object类—所有类的直接或间接父类，也是类库中所有类的父类。包含了所有Java类的公共属性。
- 2.数据类型类—基本数据类型的变换和操作。
- 3.Math类—数学函数类，提供了基本的数学运算。
- 例：6.13



## 6.5.3 Java常用工具类库

- `java.util`包一低级的实用工具类。如：日期类，堆栈类，随机数类，向量类，等等。
- 1.(Gregorian)Calendar:日历类。
- 2.Vector:动态分配对象列表。
- 3.Stack:先进先出的对象栈。
- 4.Dictionary:关键字和值的数据对存储的集合。

# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)



# 第七章 Java的Applet

---

- 7.1 Applet概述
- 7.2 Applet的编写和执行
- 7.3 Applet中图形用户界面GUI
- 7.4 Applet的多媒体支持
- 作业IV

# 7.1 Applet概述

- Java语言中程序分两类：
- 一、Java应用程序，标志是包含主方法main()，可以独立运行，应用范围较广。
- 二、Java小应用程序Applet，无main()方法，应用在Internet上。只能在支持Java的浏览器上运行，applet的运行必须依赖HTML文档。当然applet也可在集成开发环境（IDE）下，如小程序查看器（applet viewer）中直接运行。
- 注：Java小应用程序有Applet和JApplet之分，分别是基于AWT的Java Applet和基于Swing的Java JApplet。但目前IE不支持Swing，故除非在浏览器中安装了Sun发布的Java插件，否则JApplet不能在IE浏览器中运行。此书中，我们主要学习较为基础广泛的Applet。

# 7.1 Applet概述

## ■ 7.1.1 Applet举例

- 学习和使用applet，可以使我们轻松地将Java用于网络编程，增加网页设计的功能，体会到internet的乐趣。

### ■ 例7.1 关键句：

- `import java.awt.*;` //需要引用awt包
- `import java.applet.Applet;` //需要引用Applet包
- `public class AppletDemo extends Applet{ }`  
//Java Applet程序的正式入口。文件名一定要和Applet子类的类名（AppletDemo）一致。

## 7.1.2 Applet的运行原理

一个applet是一个嵌入到HTML页中的Java类，它可通过浏览器下载并在浏览器中运行。Java给静态Html页面带来了动态能力。

### ■ 运行Applet程序的步骤

1. 编写Applet源程序 : MyApplet.java
2. 编译源程序: MyApplet.java 生成 MyApplet.class
3. 编写包括Applet小程序的HTML文件 <HTML>  
<BODY>

...

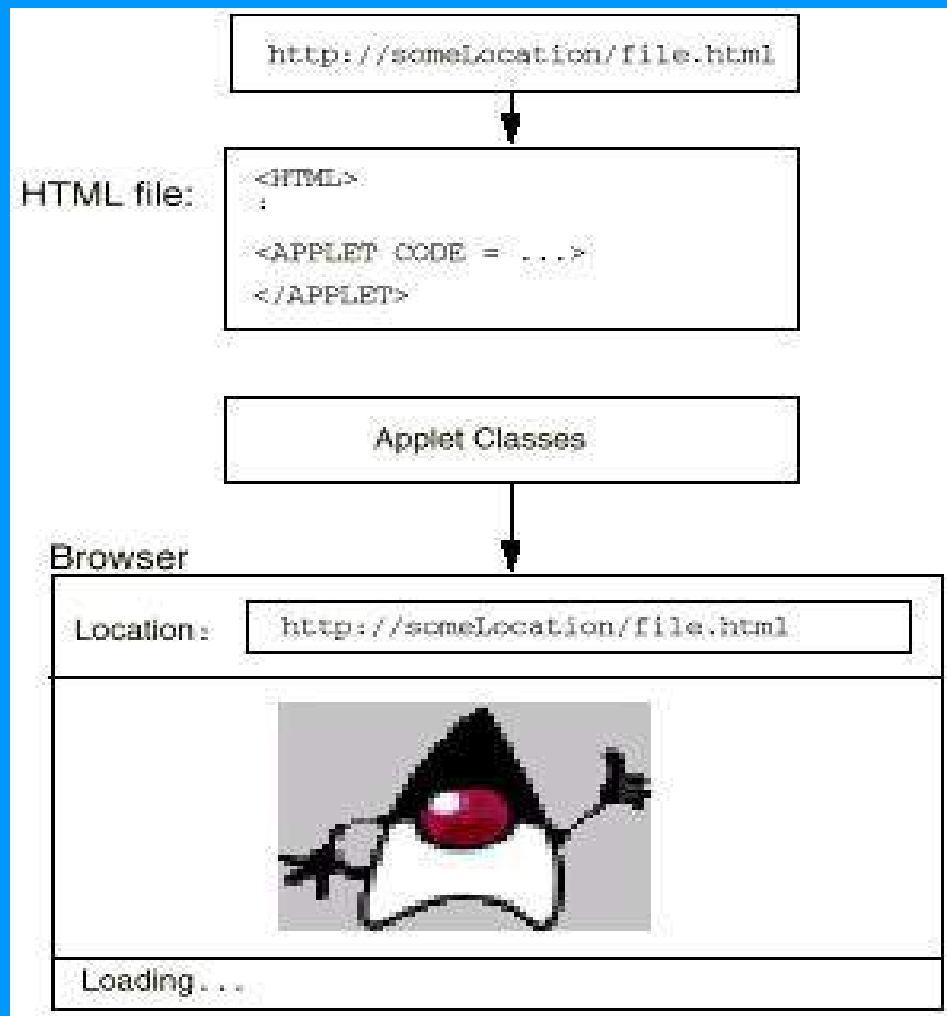
```
<APPLET CODE= "MyApplet.class" WIDTH=150  
        HEIGHT=25>
```

```
</APPLET>
```

```
</BODY>
```

4. 运行小程序 : 在浏览器地址栏中输入html文件地址运行

## 7.1.2 Applet的运行原理



1.浏览器装载URL地址。

2.浏览器装载HTML文件。

3.浏览器装载applet程序。

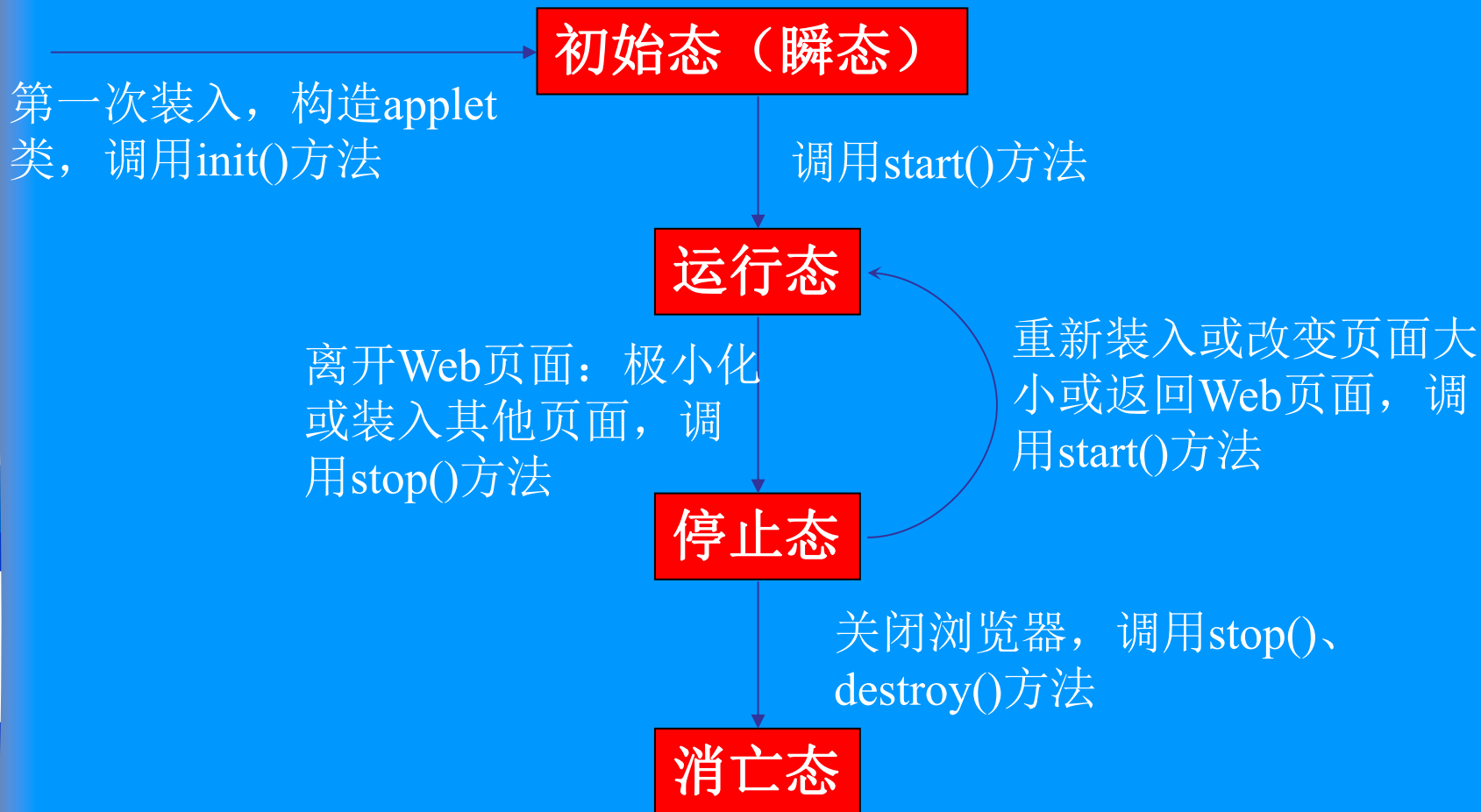
4.浏览器运行applet程序。

## 7.1.3 Applet的生命周期和主要方法

- Applet的生命周期是指Applet存在的时间，即某些方法还在被调用的时间。根据构造Applet框架的四个方法：  
`init()`, `start()`, `stop()`, `destroy()`，可得出Applet的生命周期如下图：



# Applet的生命周期



# Applet的6个主要方法

## Applet的方法

### ■ `public void init()`

- 当**Applet**创建时调用，第一次运行**Applet**时调用；
- 可以被用来初始化数据：对参数进行处理以及增加用户界面组件；

### ■ `public void start()`

- 当**applet** 变成可见时调用，在**init()**调用之后；
- 会被多次调用，实现需多次调用的代码，以及在用户离开当前页面时需要挂起的操作；

### ■ `public void stop()`

- 当**applet** 变成不可见时调用，在用户离开对应页面时调用，可多次调用；一般不直接调用该方法；
- 实现某些耗时操作（动画、音视频、计算等），在用户离开页面时，可停止那些耗时操作，节约系统资源。

# Applet的主要方法

## ■ public void **paint**(Graphics g)

- 在HTML页面中的Applet部分被显示、调整浏览器窗口、刷新页面时调用；
- 可多次调用，浏览器会自动创建一个Graphics对象并传给paint()方法，用户只需导入Graphics类所在的包即可；

## ■ public void *destroy*()

- 在卸载applet之前，进行最后的清除处理，回收资源；在浏览器正常退出时调用。



## 7.2 Applet的编写和执行

- 7.2.1 用户Applet的定义
- 7.2.2 HTML文件的编写
- 7.2.3 Applet的执行

## 7.2.1 用户Applet的定义

- 编写Applet步骤:
- 1、导入java.applet包中的类:
  - import java.applet.\*;
  - 或 import java.applet.Applet;
- 2、类头定义:
  - public class Myapplet extends Applet;
- 3、编写类体（编写几个常用方法，无main()方法）。
- 4、嵌入到html页面中，运行applet。
  - <applet .....> ..... </applet>标签。
- 5、浏览器中执行Applet的步骤（见7.1.2）。
  - 请求—发现—获取—验证—执行

## 7.2.2 HTML文件的编写

例：<html>

- <head>
- <title> Java 小程序试验平台 </title>
- </head>
- <body>
- <applet code="li71.class" height="200" width="500">
- </applet>
- </body>
- </html>

注：<applet></applet>标签一般加在<body></body>标签中。

## 7.2.2 HTML文件的编写

- **applet**的嵌入格式:
- **<APPLET**
- **CODE = *appletFile***           *//必需的*
- **WIDTH = *pixels* HEIGHT = *pixels***
- **[ARCHIVE = *archiveList* ]**       *//可选的*
- **[CODEBASE = *codebaseURL*]**
- **[ALT = *alternateText*]**
- **[NAME = *appletInstanceName*]**
- **[ALIGN = *alignment*]**
- **[object = *appletFile*]**
- **[VSPACE = *pixels*] [HSPACE = *pixels*]**
- **>**
- **[< PARAM NAME = *appletParameter1* VALUE = *value* >]**
- **[< PARAM NAME = *appletParameter2* VALUE = *value* >]**
- **...**
- **[*alternateHTML*]**
- **</APPLET>**

注: **<param>**标记在**<applet></applet>**标签之间设置, 然后由Applet自带的方法获取;

# 属性列表解释：

**CODE** = *appletFile*

指明需要运行的Applet类文件（.class文件），该文件是与 *codebaseURL* 相关的。

**WIDTH** = *pixels*   **HEIGHT** = *pixels*

Applet在浏览器中所显示的高度和宽度（以象素为单位）。

**ARCHIVE** = *archiveList [File1, File2, ...]*

给出了Applet类及其运行时所需的类所在的卷文件名。

**CODEBASE** = *codebaseURL*

指明Applet类文件所在URL根目录，在Applet类中，方法 *getCodeBase()* 可以获取该属性。如果这个属性没有指明，那么Applet类文件必须与包含它的HTML页面在同一个地方，方法 *getCodeBase()* 返回的值与 *getDocumentBase()* 相同。

**NAME** = *appletInstanceName*

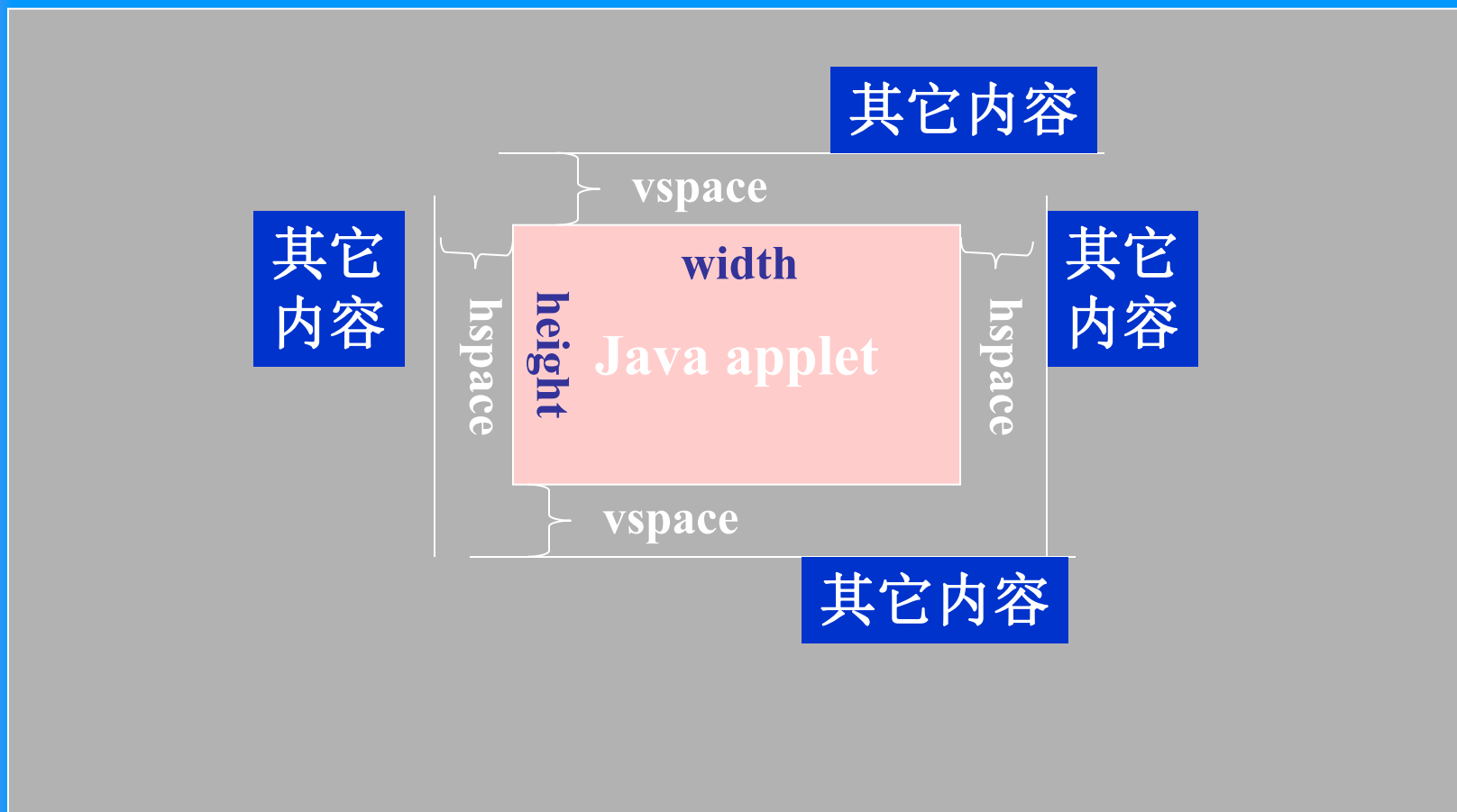
给出了Applet类运行时的实例名，这使得同处于一个页面的不同Applet之间能够相互通信。方法 ***getAppletContext()*** 可以得到同一个页面中其它Applet类。



# 属性列表解释：

- **ALT** = *alternateText*
  - 如果浏览器支持Applet但不能运行它，就显示 *alternateText* 中所给出的文字。否则就忽略它。
- **WIDTH** = *pixels*   **HEIGHT** = *pixels*
  - Applet在浏览器中所显示的高度和宽度（以像素为单位）。
- **ALIGN** = *alignment*
  - Applet在浏览器中显示时的对齐方式，其含义与效果与图片在HTML中的一样。其取值有： *left*, *right*, *top*, *texttop*, *middle*, *absmiddle*, *baseline*, *bottom*, *absbottom*。
- **VSPACE** = *pixels*   **HSPACE** = *pixels*
  - Applet在浏览器中显示时上下、左右要预留的高度和宽度，其含义与效果与图片在HTML中的一样。
- **object** = *objectname.obj*
  - 使用object属性指定applet类文件，需要浏览器支持，安装java插件。

# 属性列表解释：



# 属性列表解释：

- **< PARAM NAME = *appletParameter1* VALUE = *value* >**
  - <PARAM>属性可以使得Applet能够从页面中获取所需的参数。
  - Applet可以用其方法getParameter()获取<PARAM>属性指定的参数。
  - <PARAM>属性的作用与应用程序中main()方法里参数 *String args[]*的作用是一样的。
  - 一个好的小应用程序，应该提供用户能够设置外部参数的功能，以使得用户能够根据自己的需要来应用它。方法 **getParameterInfo()**可以得到有关<PARAM>属性的说明信息。
- ***alternateHTML***
  - 如果浏览器不支持Applet，就解释*alternateHTML*所给出的HTML代码。否则就忽略它。

## 7.2.3 Applet的执行

- 两种方法（都需要编辑HTML文件）；
- 一、appletviewer查看：借助Jcreator实现，先用Jcreator编辑好相应HTML文件，点击“运行”，即可调用appletviewer执行applet。
- 二、直接打开相应的包含applet的html文件。（xxx.htm，xxx.html）

## 7.3 Applet中图形用户界面GUI

### ■ 7.3.1 实例

- 本节提到的各种方法的详细内容可以查看JDK API 中关于java.awt.Graphics类的介绍。
- 一、输出文字
- 二、使用颜色
- 三、编制低级图形

# 一、输出文字

- 三种方法实现以指定字体和颜色在指定位置输出文字：
- 1、drawString(String str, int x, int y)  
使用此图形上下文的当前字体和颜色绘制由指定 **string** 给定的文本。
- 2、drawChars(char[] data, int offset, int length, int x, int y)  
使用此图形上下文的当前字体和颜色绘制由指定字符数组给定的文本。
- 3、drawBytes(byte[] data, int offset, int length, int x, int y)  
使用此图形上下文的当前字体和颜色绘制由指定 **byte** 数组给定的文本。

## 二、使用颜色和字体

### ■ **setColor(Color c)**

——将此图形上下文的当前颜色设置为指定颜色。  
(c是一个Color对象。)

#### – **Color(int r, int g, int b)**

——创建具有指定红色、绿色和蓝色值的不透明的  
sRGB 颜色，这些值都在 (0 - 255) 的范围内。

### ■ **setFont(Font f)**

——将此图形上下文的当前字体设置为指定字体。  
(f是一个Font对象。)

#### – **Font(String name, int style, int size)**

——根据指定字体名称、样式(0,1,2代表普通，加粗，  
斜体或PLAIN、BOLD、ITALIC ) 和磅值大小，创  
建一个新Font。

# 三、编制低级图形

---

- 方法很多，详见JDK API 文档。
- 如：
- `drawRect();`//画空心矩形框；
- `fillRect();`//画实心矩形
- `drawRoundRect()` ;//画空心圆角矩形框
- `fillRoundRect()` ;//画实心圆角矩形



## 7.3.2 基于Swing的Applet图形用户界面

- 能够实现GUI的类库有2个：`java.awt`和`java.swing`。`java.swing`是`java.awt`的扩充，它支持JFC/Swing组件体系结构，与后者有些不兼容的地方，在后面的章节将详细介绍，本节以掌握`java.awt`为主要任务。



## 7.4 Applet的多媒体支持

---

- 7.4.1 显示图像
- 7.4.2 动画制作
- 7.4.3 播放声音

## 7.4.1 显示图像

- java支持三种图像格式: .gif, .jpg, .png
- 主要使用AWT.Graphics类.(详见JAVA\_API 1.6).
- 需要导入: `java.awt.*`;
- 显示图像的步骤:
  - 建立Image对象: `Image img1`;
  - 例如 `Image img`;
  - 为Image对象赋值: `img1 = getImage( URL url, String Name)`;
  - 例如: `img1= getImage( getCodeBase(), "bull.gif")`;
  - 输出图像(在 `paint()`方法中):
  - `drawImage( Image img, int x, inty, ImageObsever obsever)`;  
obsever: 在构造Image时, 接受有关信息, 多用this.
  - 例如: `g.drawImage( img1, 10, 10, this)`;

## 例7.5:

---

```
■ import java.awt.*;
■ import java.applet.*;
■ public class myapplet extends Applet{
■     Image img1; //创建图片对象
■     public void init()
■     {
■         img1=getImage(getDocumentBase(),"flag1.gif");
■         //在init()中,给图片对象赋值
■     }
■     public void paint(Graphics g)
■     {
■         g.drawImage(img1,150,10,this); //输出图片
■     }
■ }
```

## 7.4.2 动画制作

- 原理：连续输出一系列帧（图片），利用人眼的视觉暂停来造成运动的感觉。（电影胶片是 24帧/秒）
- Java动画的两大要素：
  - 动画图片
  - 播放速度
- 动画图片：多用Image数组存放，图片名要事先准备好。
- 播放速度：利用多线程功能实现(java.lang.Thread包)
  - 在程序public class中实现Runnable接口。
  - 建立一个Thread对象。
  - 实现run()方法：图片的循环播放和暂停。
  - 实现paint()方法：输出一副帧（图形）。



## 7.4.2 动画制作

---

## 7.4.2 动画制作

- 建立Image数组的技巧

- 例：

```
Image pic[]=new Image[7]; //定义一个图片对象数组
for(i=0;i<7;i++) { //一般在init()方法中实现图片装载
pic[i]=getImage(getCodeBase(),"pics/pic"+i+".jpg");
//给图片数组赋值,图片在当前目录的pics子目录下, 图片
  名为: pic0.jpg~pic6.jpg}
```

## 7.4.2 动画制作，例7.6

### ■ 建立新线程的技巧

例：

- 1.) public class myapplet extends Applet implements Runnable{
- 2.) Thread animator; //定义一个线程对象
- 3.) 在start()方法中实现线程，启动它
- animator=new Thread(this); //创建线程  
animator.start(); //启动线程，开始运行run()方法
- 4.) 实现run()方法  
public void run(){  
    while (animator!=null) {  
        repaint(); //运行paint()方法  
        try{Thread.sleep(100); //线程暂停（睡眠100ms）}  
        catch( InterruptedException e){ break; // 有异常退出}  
    }  
}
- 5.) 在stop方法中停止，并清理线程
- animator.stop(); //线程停止stop(){  
animator=null; //停止run()方法运行
- 6.) 最后在paint()方法中输出图形。



## 7.4.3 播放声音

- Java支持的声音文件格式主要有：
  - .au; .aif; .mid; .midi; .rfm; .wav; 等。
- 播放声音主要使用AudioClip接口。
  - 使用getAudioClip(URL)方法获得AudioClip对象，其URL地址为由getCodeBase()方法得到的声音文件地址。( 加入：import java.applet.AudioClip;)
- AudioClip类有三种方法：
  - play():播放一次;
  - loop():循环播放;
  - stop():停止播放;

# 按钮的实现

- 在Applet中实现按钮，根据引用的包的不同，有基于awt的Button 和基于Swing的JButton两种，这次我们主要学习较为简单、基础的Button用法，关于JButton将在下一章中介绍。
- 书中例7.7是用JButton实现的。
- 按钮实现步骤：
  - 1、建立Button对象； `//Button bt1=new Button();`
  - 2、设置Button属性； `//`
    - `bt1.setFont(new Font("宋体",0,12));` //按钮上的字体
    - `bt1.setLabel("播放");` //按钮上的文字内容
    - `bt1.setActionCommand("play");` //按钮的动作名称

# 按钮的实现

3、注册Button动作；

```
//bt1.addActionListener(this);
```

4、在容器中加入按钮；

```
//this.add(bt1,null);
```

```
//把按钮加入到容器（applet面板）上
```

5、编写实现Button动作的代码。

```
public void actionPerformed(ActionEvent actionEvent){
```

```
//实现actionPerformed()方法，响应按钮动作；
```

```
■ String btn=actionEvent.getActionCommand();
```

```
■ //得到被按下按钮的动作名称
```

```
■ if(btn.equals("play"))
```

```
■ {audio.play();//播放音乐}
```


# 播放声音

- 播放声音的步骤:
- 1、引用相应包:
  - import java.awt.event.\*;
  - import java.applet.AudioClip;
- 2、实现ActionListener接口(如果不需控制播放, 则可以省略)
  - public class myapplet extends Applet implements ActionListener{ }
- 3、建立声音对象// AudioClip audio;
- 4、给声音对象赋值——装载声音文件
  - audio=this.getAudioClip( getCodeBase(), "guoge.wav");
- 5、按控制要求播放声音
  - audio.play();//播放音乐
  - audio.loop();//循环播放
  - audio.stop();//停止播放音乐
- 6、控制按钮的实现（见上节）

# 作业IV:

- 利用Java Applet实现这样一个HTML页面，
- 基本功能：
  - 1、可以用至少两个按钮控制（播放、停止）播放一段声音；
  - 2、显示一副图片（最好与声音能有关，协调起来）作为播放声音的背景。
  - 3、在图片上以文字形式输出声音的名字（或者再加上其他信息）。
- 发挥部分：
  - 声音信息用动画表示；连续输出多幅图片形成动画效果；欢迎其他更好的发挥。

# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)



# 第八章 Java的图形用户界面设计

- 8.1 图形用户界面设计概述
- 8.2 AWT图形用户界面
- 8.3 布局管理
- 8.4 事件处理
- 8.5 Swing图形用户界面
- 作业V:

# 8.1 图形用户界面设计概述

- 面向对象的图形用户界面:
- JAVA语言基于“**面向对象**”的思想互相交换信息，即尽可能在屏幕上用形象的图标和窗口等来代表有用的资源和可启用的对象。
- **图形界面编程的主要特征**:
  - 图形界面对象及其框架（图形界面对象之间的包含关系）
  - 图形界面对象的布局（图形界面对象之间的位置关系）
  - 图形界面对象上的事件响应（图形界面对象上的动作）



# 8.1 图形用户界面设计概述

- 通过图形用户界面（Graphical User Interface, GUI），用户和程序之间可以方便友好地进行交互。在Java语言中，Java的基础类（Java Foundation Classes, JFC）是开发GUI的API集，它包括以下几个部分：
  - 抽象窗口工具包（AWT）：Java开发用户界面最初的工具包，是建立JFC的主要基础；
  - 2D API：实现高质量的二维图形；
  - Swing组件：建立在AWT之上，新的、功能更强大的图形组件包；
  - etc.



## 8.1 图形用户界面设计概述

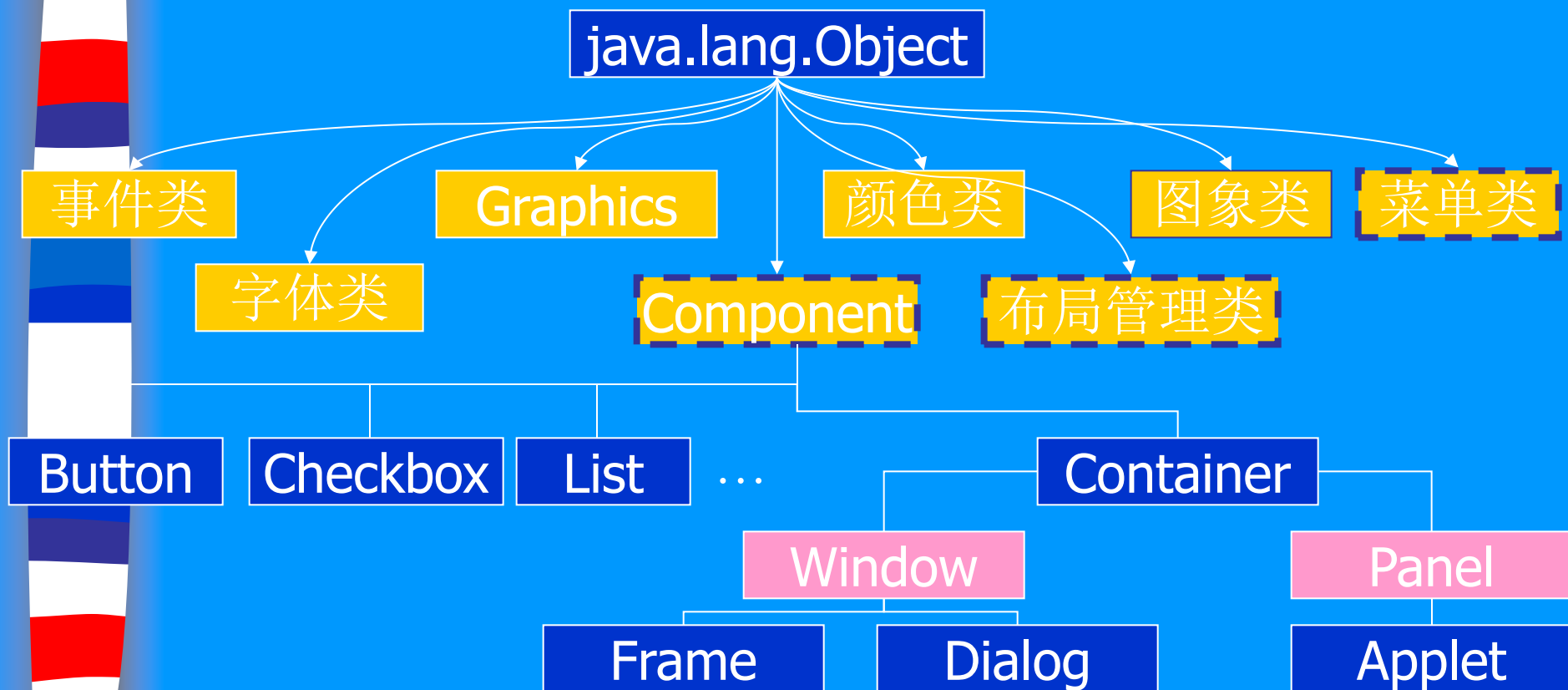
- 进行GUI设计的三点思路：
  - 1、界面中的构件如何放置；
  - 2、如何让构件响应用户的操作；
  - 3、掌握每种构件的显示效果和响应用户操作。

## 8.2 AWT图形用户界面

- 图形用户界面（**GUI**）使用户可以和程序进行可视化交互。
- 图形用户界面包含了许多独立的显示元素，供用户交互。它们由程序中的**GUI**组件组成。**GUI**组件是用户通过键盘或鼠标进行交互的对象，菜单、按钮、文本域、标签和下拉列表框等**GUI**组件是图形用户界面的常用部分。
- **AWT (Abstract Window Toolkit)**用来处理**GUI**的最基本方式，可用来创建**Applet**和窗口，支持**GUI**编程的主要功能。

## 8.2.1 java.awt包

java.awt及相关包中包含了一个完整的类集以支持GUI程序的设计，其中的类及相互关系可以用下图来描述（重量级构件）：



## 8.2.1 java.awt包

- java.awt包提供了基本的java程序的GUI设计工具。主要包括下面三个概念，每个概念对应一个类：
  - 1、构件：Component，awt的核心，是一个抽象类，其他构件都是从此类衍生而来的。
  - 2、容器：Container，从Component继承而来，用来管理构件。
  - 3、布局管理器：LayoutManager，确定容器内构件的布局。

## 8.2.2 构件和容器类

- 一、构件类（Component类）
- 最核心的类，它是构成Java图形用户界面的基础，大部分构件都是由该类派生出来的。
- Component类是一个抽象类，其中定义了构件所具有的一般功能（方法）：
  - 基本的绘画支持(paint, repaint, update等);
  - 字体和颜色等外形控制(setFont, setForeground等);
  - 大小和位置控制(setSize, setLocation等);
  - 图象处理(实现接口ImageObserver);
  - 组件状态控制(setEnabled, setVisible, isEnabled, isValid等)
  - 获得构件对象(getComponentAt, getName, getFont, getForeground, getSize等)
- 常见的构件包括：Button, Checkbox, CheckboxGroup, Choice, Label, List, Canvas, TextComponent, Scrollbar等。

## 8.2.2 构件和容器类

- 二、容器类(Container)
- Container类是由Component类派生出来的一种特殊子类，用来表示各种GUI构件的容器，可以容纳多个构件，其主要功能包括：
  - 构件的管理：方法add()可以向其中添加一个构件，remove()删除其中的一个构件，...
  - 布局管理：每个Container类都和一个布局管理器相联，以确定其中构件的布局。Container类可以通过setLayout()方法设置某种布局方式。
- 常见的Container类有：Frame, Panel, Applet等。

## 8.2.2 构件和容器类

### ■ 三、布局管理器类

- 用来管理构件放置在容器中的位置和大小。每个容器都有一个布局管理器，当容器需要对某个构件进行定位或判断其大小尺寸时，就会调用其对应的布局管理器。为保证GUI的平台无关性，不能使用直接设置构件位置和大小的方式。



# 常用容器类(Container)

- 1、窗口 (Frame)
- 在Java中，生成一个窗口通常是用Window的子类Frame来进行实例化，不直接用Window类。
- 生成的窗口，如果不添加响应代码，不能响应用户的操作，如“关闭窗口”等。
- 例：
  - public class li81 extends Frame{
  - public static void main(String args[]){
  - li81 fr=new li81(“This is a contianer”);//窗口名
  - }}

# 常用容器类(Container)

---

- 2、面板 (Panel)
- 透明容器，无标题，无边框。
- 既是容器，又是构件。必须先作为构件放到其他容器中，然后再当容器容纳其他构件。
- `Panel pan=new Panel();`

## 8.3 布局管理

- Java中提供了各种布局管理器类来管理各种组件在容器中的放置状态，这些类都是实现了LayoutManager接口的。标准的布局管理器类(在java.awt包中)有：
  - FlowLayout
  - BorderLayout
  - GridLayout
  - CardLayout
  - GridBagLayout
- 还可以通过实现LayoutManager接口来定义自己的布局管理器。



# FlowLayout布局管理器

- Panel和Applet的缺省布局管理器
- FlowLayout布局方式:
- 将组件一排一排地依次放置, 它自动调用组件的 `getPreferredSize()` 方法, 使用组件的最佳尺寸来显示组件。当容器被重新设置大小后, 布局也会随之发生改变: 各组件的大小不变, 但相对位置会发生变化。
- 组件配置过程:
  - (1) 由左到右配置在长方形格子里;
  - (2) 格子大小为组件大小;
  - (3) 配置完一行, 再配置下一行, 直到组件配置完成。

# FlowLayout布局管理器

- FlowLayout类有三种构造方法,即声明方式有三:
- **public FlowLayout()**
  - 使用缺省居中对齐方式, 组件间的水平和垂直间距为缺省值5个像素。
- **public FlowLayout(int alignment)**
  - 使用指定的对齐方式 (FlowLayout.LEFT, FlowLayout.RIGHT, FlowLayout.Center), 水平和垂直间距为缺省值5像素。
- **public FlowLayout(int alignment, int horizontalGap, int verticalGap)**
  - 使用指定的对齐方式, 水平和垂直间距也为指定值。

# BorderLayout布局管理器

- Window、Dialog和Frame的缺省布局管理器
- BorderLayout布局方式提供了更复杂的布局控制方法，它包括5个区域：North, South, East, West和Center，其方位依据上北下南左西右东。当容器的尺寸发生变化时，各组件的相对位置不变，但中间部分组件的尺寸会发生变化，南北组件的高度不变，东西组件的宽度不变。
- BorderLayout类有二种构造方法：
  - public BorderLayout()  
各组件间的水平和竖直间距为缺省值0个像素。
  - public BorderLayout(int horizontalGap, int verticalGap)  
各组件间的水平和竖直间距为指定值。

# BorderLayout布局管理器

- 如果容器使用了BorderLayout布局方式，则用add()方法往容器中添加组件时必须指明添加的位置，否则组件将无法正确显示（不同的布局管理器，向容器中添加组件的方法也不同）。

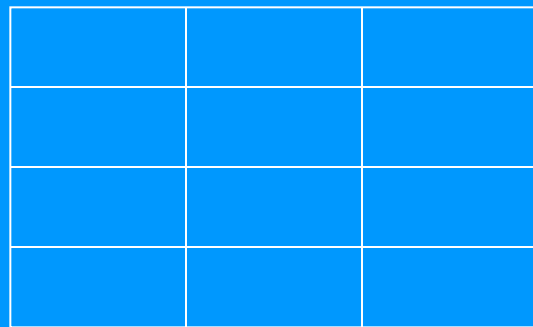
如：  
`add("West", new Button("West"));`  
`add("North", new Button("North"));`  
`add(new Button("West"), BorderLayout.SOUTH);`

若没有指明放置位置，则表明为默认的“Center”方位。

- 每个区域只能添加一个组件，若添加多个，则只能显示一个。如果想在在一个区域添加多个组件，则必须先在该区域放一个Panel容器，再将多个组件放在该Panel容器中。
- 若每个区域或若干个区域没有放置组件，东西南北区域将不会有预留，而中间区域将置空。

# GridLayout布局管理器

- GridLayout布局方式可以使容器中的各组件呈网格状分布。容器中各组件的高度和宽度相同，当容器的尺寸发生变化时，各组件的相对位置不变，但各自的尺寸会发生变化。
- 各组件的排列方式：从左到右，从上到下。
- 与BorderLayout类相类似，如果想在在一个网格单元中添加多个组件，则必须先在该网格单元放一个Panel容器，再将多个组件放在该Panel容器中。





# GridLayout布局管理器

- GridLayout类有三种构造方法：
- `public GridLayout()`
  - 在一行中放置所有的组件，各组件间的水平间距为0像素。
- `public GridLayout(int rows, int cols)`
  - 生成一个`rows`行，`cols`列的管理器，最多能放置`rows*cols`个组件。（课本有误）。
  - `rows`或`cols`可以有一个为0。若`rows`为0，这表示每行放置`cols`个组件，根据具体组件数，可以有任意多行；若`cols`为0，这表示共有`rows`行，根据具体组件数，每行可以放置任意多个组件。
  - 组件间的水平和竖直间距为0像素。
- `public GridLayout(int rows, int cols, int horizontalGap, int verticalGap)`
  - 各组件间的水平和竖直间距为指定值。

# CardLayout布局管理器

- CardLayout布局方式可以帮助用户处理两个或更多的组件共享同一显示空间。共享空间的组件之间的关系就像一摞牌一样，它们摞在一起，只有最上面的组件是可见的。CardLayout可以象换牌一样处理这些共享空间的组件：为每张牌定义一个名字，可按名字选牌；可按顺序向前或向后翻牌；也可以直接选第一张或最后一张牌。类似于VB中的选项卡。
- 常用方法：
  - `public void show(Container parent, String name)`  
//显示指定名称的卡片
  - `public void next(Container parent)`
  - `public void previous(Container parent)`
  - `public void first(Container parent)`
  - `public void last(Container parent)`
  - 其中，Container是拥有该CardLayout布局管理器的容器。

# CardLayout布局管理器

- CardLayout类有二种构造方法：
- `public CardLayout()`
  - 组件距容器左右边界和上下边界的距离为缺省值0个像素。
- `public CardLayout(int horizontalGap, int verticalGap)`
  - 组件距容器左右边界和上下边界的距离为指定值。
- 与BorderLayout类和GridLayout类相类似，每张牌中只能放置一个组件，如果想在一张牌放置多个组件，则必须先在该牌放一个容器，再将多个组件放在该容器中。
- 采用CardLayout布局方式时，向容器中添加组件时可以为组件取一个名字，以供更换显示组件时使用：
- `add(String, Component);`



# GridBagLayout布局管理器

- GridBagLayout布局方式是AWT中最灵活、同时也是最复杂的一种布局方式。与GridLayout相同，它也是将容器中的组件按照行、列的方式放置，但各组件所占的空间可以互不相同。
- GridBagLayout根据对每个组件所施加的空间限制、每个组件自身所设定的最小尺寸和最佳尺寸来为每个组件分配空间。对组件施加空间限制是通过类GridBagConstraints来实现的。

# GradBagLayout布局管理器

- 类GridBagConstraints中提供了一些相应的属性和常量来设置对组件的空间限制：
  - gridx, gridy
  - //指明组件显示区域左上角坐标（在容器中）
  - gridwidth, gridheight
  - //指明组件在一行中所占的网格单元数（宽度，高度）。
  - fill //指明当组件所在的网格单元的区域大于组件所请求的区域时，是否改变组件的尺寸
  - ipadx, ipady
  - //指明组件的宽/高度与指定的最小宽/高度之间的关系
  - insets //指明了组件与其显示区边缘之间的距离
  - anchor
  - //指明了当组件的尺寸小于其显示区时，在显示区中如何放置改组件的位置
  - weightx, weighty
  - //指明当容器扩大时，如何在列，行间为组件分配额外的空间

# GradBagLayout布局管理器

- 当一个容器的布局方式为GridBagLayout时，往其中添加一个组件，必须先用GridBagConstraints设置该组件的空间限制。
  - //在一个容器中
  - GridBagLayout g = new GridBagLayout();
  - GridBagConstraints c = new GridBagConstraints();
  - setLayout( g );
  - Button b = new Button(“Test”); //生成组件b
  - c.fill = ... //设置c的值
  - .....
  - g.setConstraints( b, c ); //根据c的值对组件b进行空间限制
  - add( b );



# Swing布局管理器

---

- Swing除了可用上述五种布局管理器外，还提供其他选择。其中：
- ScrollPaneLayout和ViewportLayout是内置的。
- BoxLayout和OverlayLayout的使用类似AWT的布局管理器。
- BoxLayout
- ScrollPaneLayout
- 详见javax.swing包中。

# ■ null布局管理器

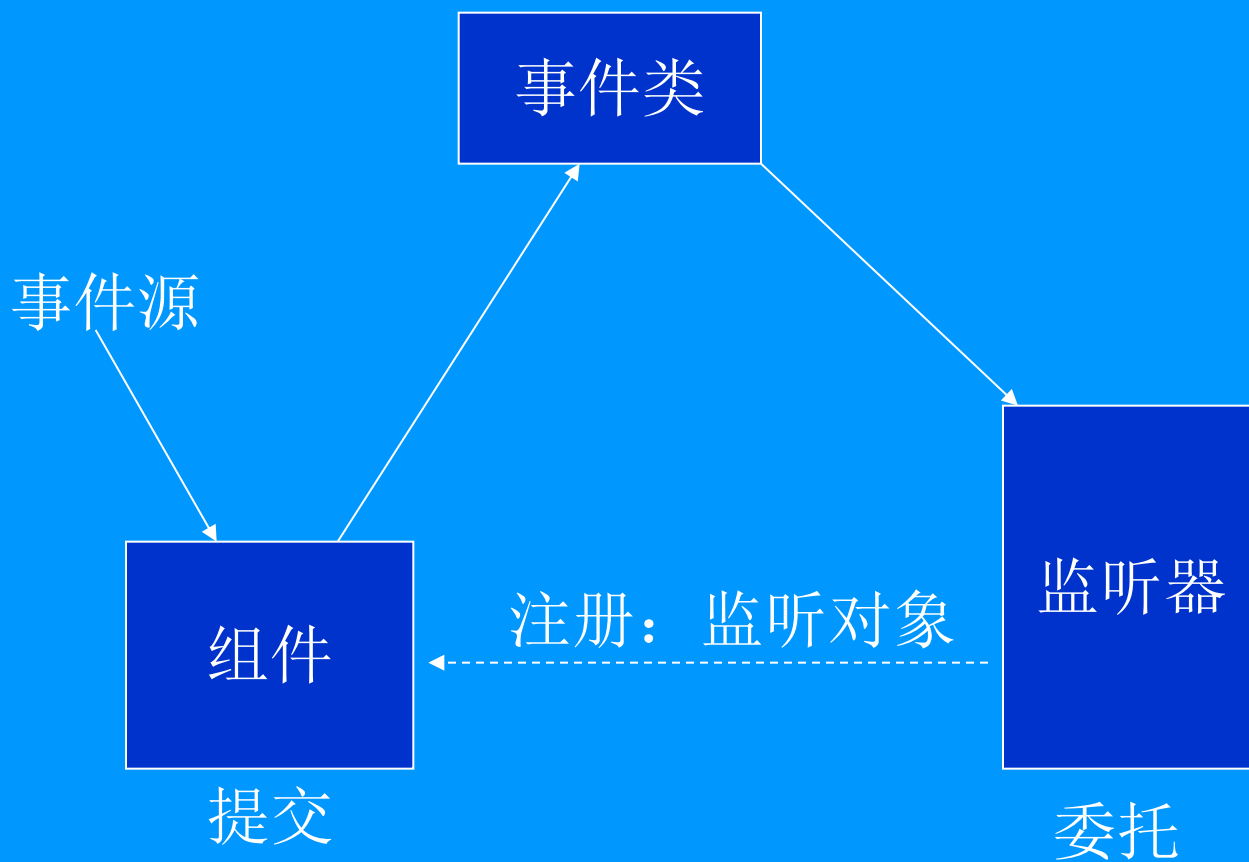
- 用户不想使用布局管理器，需要自己设置组件的位置和大小，需取消布局管理器，然后再进行设置，否则用户自定义的设置将会被布局管理器覆盖。
- 取消方法：
  - `setLayout(null)`
- 设置方法：
  - `setLocation()`
  - `setSize()`
  - `setBounds()`方法
- 注：这种方法会导致程序的运行效果与系统有关。
- 用户也可以通过实现LayoutManager接口来定义自己的布局管理器。



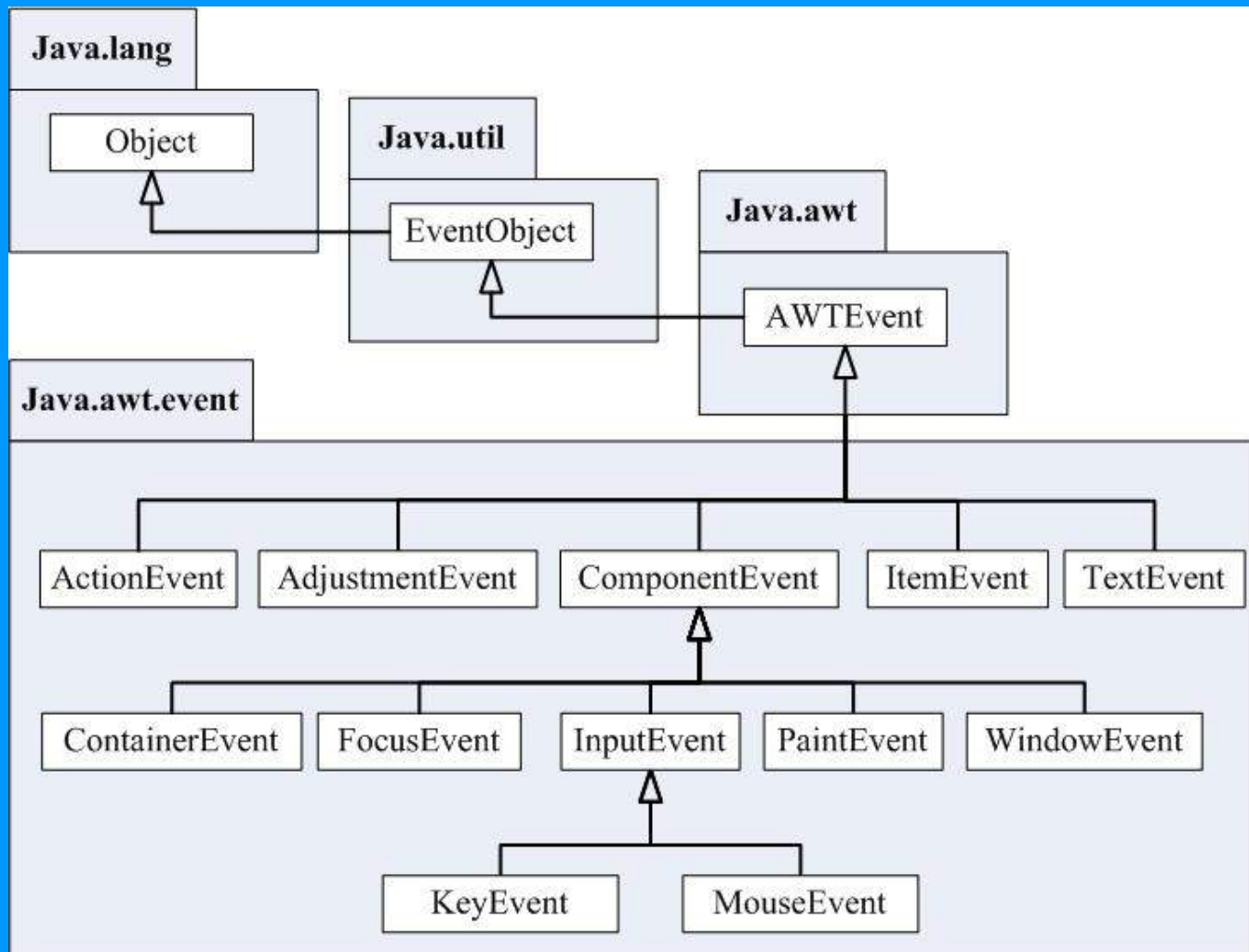
## 8.4 事件处理

- 事件是由程序检测到的行为和动作。
- 事件是发生在用户界面上的用户交互行为所产生的一种效果。
- 事件是系统对其可能处于的某种状态或者某种操作的预先定义。
- Java采用一种叫“事件授权模型”的事件处理机制，即委托型事件处理模式。当用户与GUI程序交互时，会触发相应的事件。
- 事件源：产生事件的组件称为事件源。
- 监听器：触发事件后系统会自动创建事件类的对象，组件本身不会处理事件，而是将事件对象提交给Java运行时系统，系统将事件对象委托给专门的实体——监听器。
- 事件源与监听器建立联系的方式是将监听器注册给事件源。

# 事件处理模型



## 8.4.1 java的事件类



## 8.4.1 java的事件类

- 事件都放在包java.awt.event中，这些事件都从java.util.Event对象派生而来，对于与AWT有关的事件，则由java.awt.AWTEvent类派生，可分为两大类：
- 低级事件：基于构件和容器的事件。
  - ComponentEvent, 构件尺寸的变化、移动。
  - ContainerEvent, 容器事件，构件增加、移动。
  - WindowEvent, 窗口事件，关闭，还原，最小化。
  - FocusEvent, 焦点事件，焦点的获得和丢失。
  - KeyEvent, 键盘事件，键按下、释放。
  - MouseEvent, 鼠标事件，鼠标单击、移动。
- 高级事件（语义事件，可以不和特定的动作相关联，而依赖于触发此事件的类。一般的GUI编程，只需对这类事件进行处理即可）
  - ActionEvent, 动作事件，按下按钮或按下文本框中的回车键产生。
  - AdjustmentEvent, 调节事件，用户在滚动条上移动滑块产生。
  - ItemEvent, 项目事件，选择项目等产生。
  - TextEvent, 文本事件，文本对象的改变。

## 8.4.2 事件处理模式

---

- 一个Java事件处理模型的具体步骤:

- 1.创建事件监听器
- 2.注册事件监听器
- 3.创建事件对象
- 4.通知事件发生
- 5.执行事件处理程序

## 8.4.2 事件处理模式

- 事件处理过程中主要涉及3类对象：
  - **Event**: 事件，用户对界面操作在Java语言上的描述，以类的形式出现。
  - **Event Source**: 事件源，事件发生的场所，通常就是各个构件。
  - **Event Handler**: 事件处理者（监听器），接收事件对象并对其进行处理的对象。

## 8.4.2 事件处理模式

---

- 授权处理机制
- 事件源与监听器分开的机制。
- 事件处理者（监听器）通常是一个类，该类如果要处理某种类型的事件，就必须实现与该事件类型相对应的接口。

## 8.4.3 AWT事件及其相应的监听接口

- 共10类事件11个接口。表8.1
- 每个事件类对应一个监听器接口，事件类中每个具体事件类型都有一个具体的抽象方法与之对应，当具体事件发生时，这个事件将被封装成一个事件类的对象作为实际参数传递给与之对应的具体方法，由这个具体方法负责响应并处理发生的事件。
- 注：InputEvent类
- MouseEvent类



## 8.4.4 事件适配器

- 由于通过实现接口XXXListener来完成事件处理时，要同时实现该接口中的所有方法。通常我们只是需要对其中的某些方法做处理，而不想实现所有的无关方法。
- 因此，为了方便起见，JDK1.1为某些监听器接口提供了适配器类（XXXAdapter），当需要对某种事件进行处理时，只需让事件处理类继承事件所对应的适配器类，只重写需要关注的方法即可，而无关的方法就不必实现了。

## 8.5 Swing图形用户界面

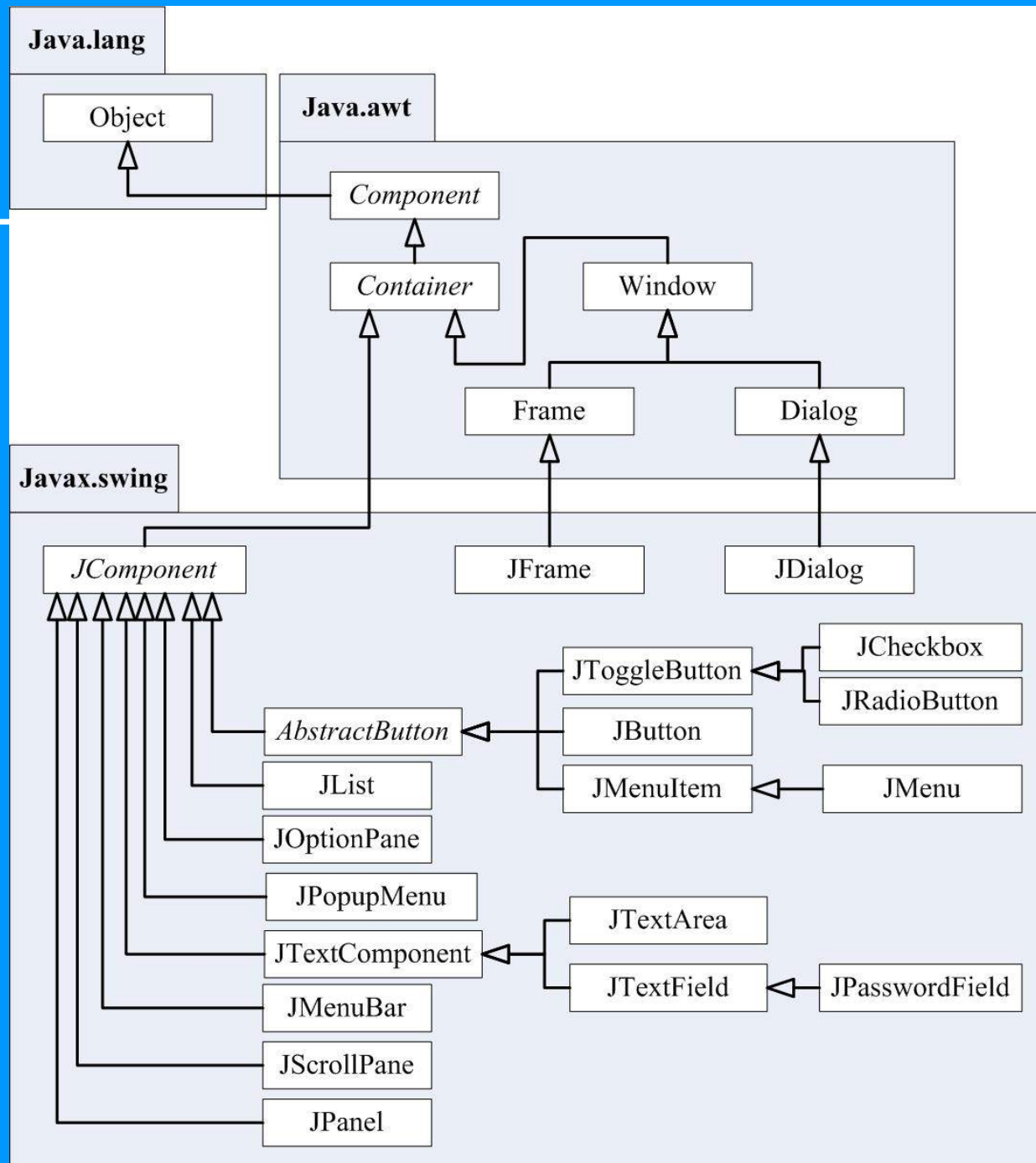
### 8.5.1. Swing概述

- 是第二代GUI开发工具
- 建立在AWT之上，但用新版本的组件替代了旧版本的组件。
- 由纯Java实现的轻量级(light-weight)构件AWT, AWT组件是建立在对等模型的基础上的重量级组件。
- 提供了许多新的组件和相关的API
- Swing与AWT的区别：P161
- Swing的常用组件：详见JDK\_API\_1.6

## 8.5.2 Swing 的层次结构

---

- AWT与Swing的结构关系以及Swing的组件关系
- 我们经常用的有两个包：
  - javax.swing
  - javax.swing.event



# Swing常用组件

**Swing**组件从功能上分可分为：

- 顶层容器 有JFrame、JApplet、JDialog、JWindow共4个。
- 中间容器 有JPanel、JScrollPane、JSplitPane、JToolBar、Box等。
- 特殊容器 在GUI上起特殊作用的中间层，例如：JInternalFrame、JLayeredPane、JRootPane。
- 基本组件 实现GUI交互的组件，例如：JButton、JComboBox、JList、JTextField等。
- 不可编辑信息的组件 向用户显示不可编辑信息的组件，例如：JLabel、JProgressBar。
- 可编辑信息的组件 向用户显示能被编辑的格式化信息的组件，例如：JColorChooser、JFileChooser、JTable、JTextArea。



# Swing常用组件

---

- 8.5.3 常用容器组件
  - 1.JFrame
  - 2.JPanel
  - 3.JApplet
- 8.5.4 标签 JLabel
- 8.5.5按钮 JButton
- 8.5.6文本组件
  - 1.JTextField
  - 2.JPasswordField
  - 3.JTextArea
- 8.5.7复选框 JCheckBox
- 8.5.8单选按钮 JRadioButton



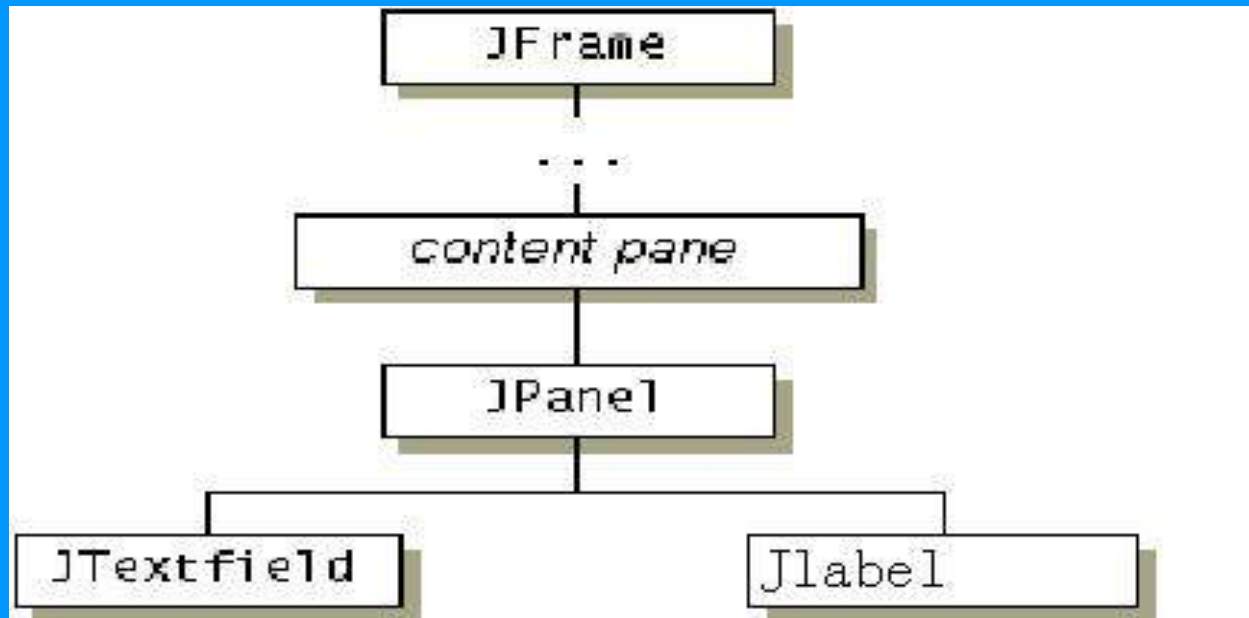
# Swing常用组件

---

- 8.5.9 列表框 JList
- 8.5.10 组合框 JComboBox
- 8.5.11 滚动条 JScrollBar
- 8.5.12 菜单
  - 1. 菜单条 JMenuBar
  - 2. 菜单 JMenu
  - 3. 菜单项 JMenuItem
  - 4. 弹出式菜单 JPopupMenu
- 8.5.13 对话框
  - 1. JOptionPane
  - 2. JFileChooser
- 8.5.14 表格

# Swing组件和容器

容器与组件包含继承关系图表：

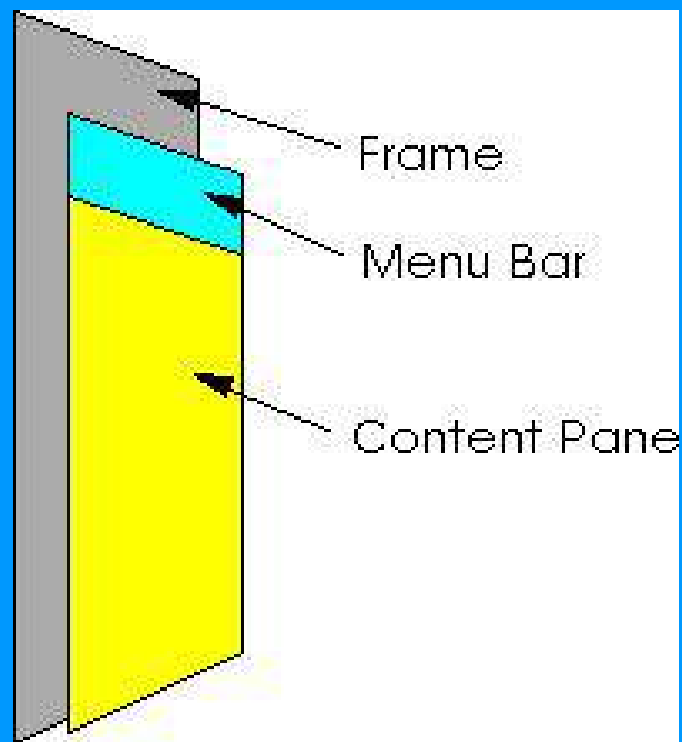
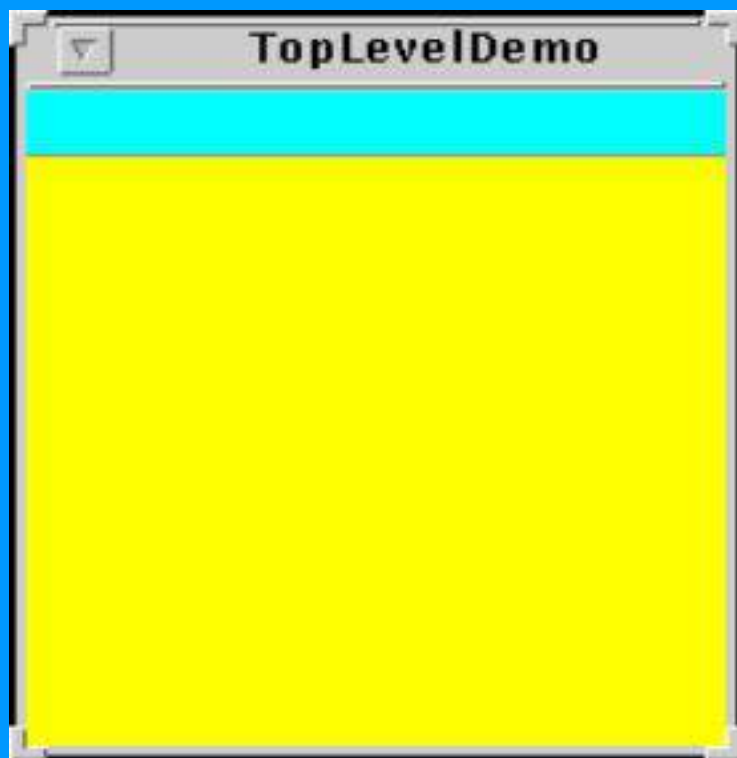




# 容器

## 顶层容器

java 提供了三个顶层容器：JFrame, JDialog, 和 JApplet。



# 容器

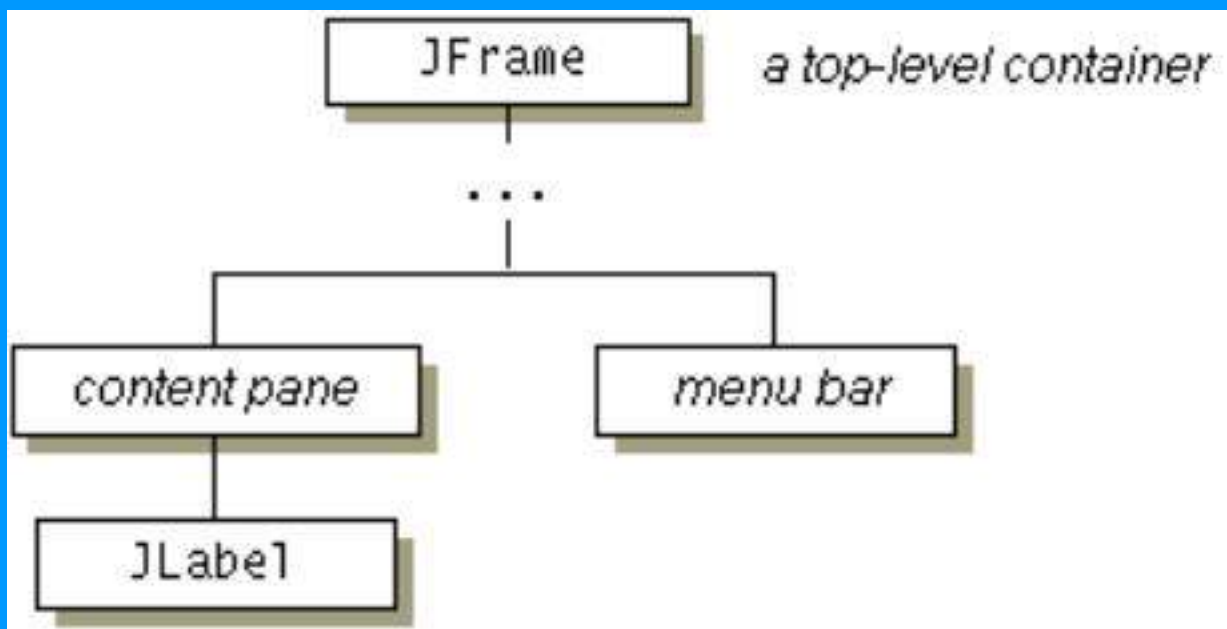
---

## 特点：

- 显示在屏幕上的每个组件都必须在一个包含继承中。每一个包含继承都有一个顶层容器作为它的根。
- 每一个一个顶层容器都有一个content pane，它包含了顶层容器中的所有组件
- 菜单在顶层容器中，但在content pane 之外。

# 容器

包含继承图如下：





# 在Content Pane 中添加组件

有两种方式:

```
topLevelContainer.getContentPane().add(  
yellowLabel, BorderLayout.CENTER);
```

或:

- `JPanel contentPane = new JPanel();`

... ..

```
topLevelContainer.setContentPane(contentPane);
```

## 2. 添加菜单条

---

- `JMenuBar cyanMenuBar = new JMenuBar();`
- `frame.setJMenuBar(cyanMenuBar);`

## 2. 添加菜单条

### 创建菜单:

```
1. JFrame f = new JFrame(" Menu");
2. JMenuBar mb = new JMenuBar();
3. JMenu m1 = new JMenu(" File");
4. JMenu m2 = new JMenu(" Edit");
5. JMenu m3 = new JMenu(" Help");
6. mb.add( m1);
7. mb.add( m2);
8. f.setMenuBar( mb);
9. JMenuItem mi1 = new JMenuItem("New");
10. JMenuItem mi2 = new JMenuItem("Save");
11. JMenuItem mi3 = new JMenuItem("Load");
12. JMenuItem mi4 = new JMenuItem("Quit");
13. m1.add(mi1);
14. m1.add(mi2);
15. m1.add(mi3);
16. m1.addSeparator();
17. m1.add(mi4);
```

## 2. 添加菜单条

---

创建菜单:

步骤:

1. 创建一个 `MenuBar` 对象, 将其加入一个菜单容器, 例如: `Frame` .
2. 创建 一个或多个 `Menu` 对象, 将其加入 *menu bar* 对象.
3. 创建 一个或多个 `MenuItem` *objects*, 对象, 将其加入相应的 *menu object*.


# 作业V： 简易绘图板

---

- 在Swing中实现一个绘图板程序，要求基于Jframe设计。
- 基本要求：显示若干按钮，点击一个按钮，就会在屏幕上输出相应图形，至少要有3种图形的绘制。
- 发挥扩展：用菜单表示选项，能输出图像，能手动绘制图形，能选择颜色，能选择字体，能选择线条格式，等等，欢迎更好的发挥。



# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)



# 第九章 Java的异常处理

---

- 9.1 异常概述
- 9.2 异常处理

# 9.1 异常概述

---

- 错误可分为编译错误和运行错误。
- 编译错误：语法错误，编译器能够检查出，程序无法运行，编译完成即已改正。
- 运行错误：程序运行过程中产生的错误。即使编译通过，也会在程序运行时因为主客观原因造成程序运行错误。在JAVA中这类错误称为异常（Exception）

# 9.1 异常概述

---

- 排除运行错误的常规手段：单步运行，设置断点，暂停程序，一步一步发现错误。
- 常规手段的不足：运行之后才发现错误，无法避免程序运行时造成的损失。
- 为了加强程序在运行期间的强壮性（鲁棒性，**robust**），程序设计时，必须考虑到可能发生的异常事件并做出相应的处理，这就是**异常机制**。

## 9.1.1 异常的基本概念

---

### 1. 什么是异常？

异常是在程序运行过程中所发生的破坏了正常的指令流程的事件。软件和硬件错误都可能导致异常的产生。

### 2. 常见异常：

1. 除0溢出： `ArithmeticException`
2. 数组越界： `ArrayIndexOutOfBoundsException`
3. 空指针异常： `NullPointerException`

## 9.1.1 异常的基本概念

---

### ■ 发生异常后的情况：

1. **命令行界面的应用程序：** 发生异常，且未被处理，程序终止运行，并在控制台输出异常信息。
2. **图形化界面的应用程序：** 发生异常，且未被处理，控制台输出异常信息，但程序不会中止运行。

## 9.1.1 异常的基本概念

- 简单的异常，可在编程中加代码预防。
- 复杂的异常，无法避免，需要使用JAVA的异常处理机制来处理。
- **JAVA的异常处理机制：**
- 用面向对象的方法处理异常，一个异常事件由一个异常类的对象来代表。所有的异常都是以类的类型存在，除了内置的异常类之外，也可自行定义异常类。允许抛出异常，保证程序允许。

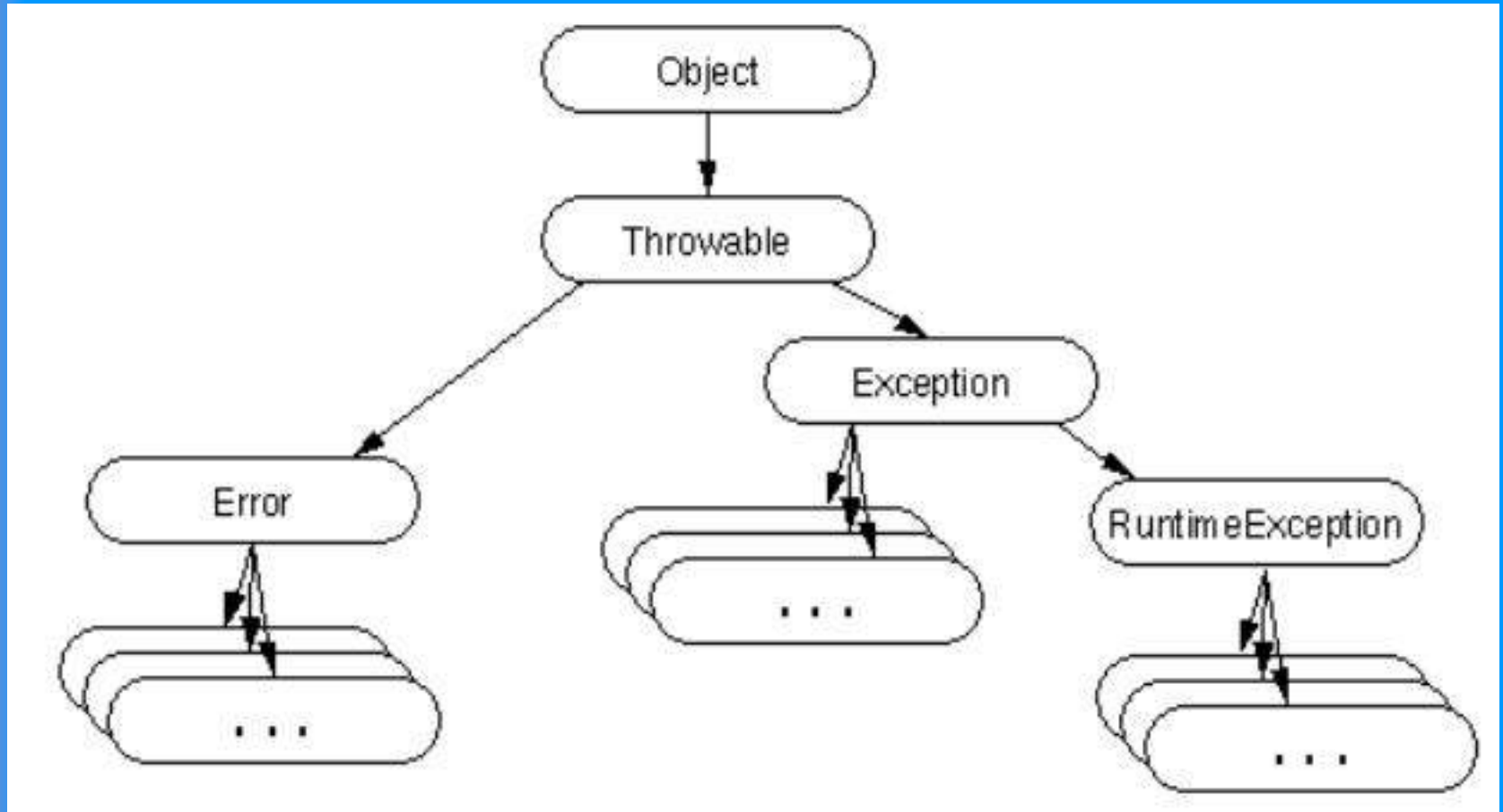
## 9.1.2 异常类的层次

---

- **异常类**：用来描述和处理异常的类。
  - 每个异常类反映一类运行时错误，类定义中包含了该类异常的信息和对异常进行处理的方法。
  - 当程序运行过程中发生了某个异常事件，系统将产生一个相应的异常类对象，并交由系统中的相应机制进行处理。



# 异常类的层次： `java.lang.Throwable`



# 异常类的层次

- **Throwable**有两个直接子类：Error和Exception

- **Error**类：用来处理运行环境方面的异常。

如：虚拟机错误、装载错误、动态连接错误。这类异常主要是和硬件有关系，而不是由程序本身抛出。通常，Java程序不对这类异常进行处理。

- **Exception**是Throwable的一个主要子类：

- 运行时异常（**RuntimeException**）：java程序运行时常常遇到的各种异常的处理，可以不做处理。
- 非运行时异常：如输入输出异常**IOException**等。Java编译器要求Java程序必须捕获或声明所有的非运行时异常。

## 9.1.3 异常处理机制

- 抛出和处理错误机制：
  1. 抛出（**Throw**）异常： **JAVA**程序运行过程中如果出现异常，会自动生成一个异常类对象，该对象将被提交给**JAVA**运行时环境。
  2. 捕获（**Catch**）异常： **JAVA**运行时环境接收到一个异常对象时，会寻找能处理这一异常的代码，并把当前异常对象交给它处理。如果找不到可以捕获异常的方法，则运行时环境将终止，相应的**JAVA**程序退出。
  3. 异常类的常用方法。 P205

## 9.2 异常处理

---

异常处理方法：

1. 使用try-catch-finally语句捕获处理异常
2. 通过throws和throw子句抛出(声明)异常

## 9.2.1. try-catch-finally语句

- 用try-catch-finally语句来捕获一个或多个异常

- 基本格式为：

**try** { 可能发生异常的程序代码 }

**catch** (异常类型1 异常对象名1)

{异常处理代码}

**catch** (ExceptionType2 ExceptionObject2)

{ Exception Handling }

.....

**finally** { 必然要执行的代码 }

# try-catch-finally语句说明：

- **try {...}**：定义可能产生异常的代码段。  
如果出现异常，程序中断运行，并抛出该异常产生的对象。
- **catch (Etype e) {...}**：用于捕获一个异常。  
当try抛出异常对象时，catch识别，如果与catch语句参数中声明的异常类相同或是它的子类对象时，捕获这个异常对象，显示简短信息，处理。注意捕获顺序--先子后父。
- **finally {...}**：用于做统一的事后处理，如释放资源，必然会被执行的代码。

## 9.2.2 throw语句和throws语句

- 抛出异常的两种方式：
- **throw**： 程序中通过此语句抛出异常，同时在该方法中解决抛出的异常。
- **throws**： 在声明方法时使用，抛出但不捕获异常，让调用该方法的其他方法捕获异常。

# 1. throw语句

---

- 格式:

`throw e;` //e为异常类对象

- 抛出异常，同时在该方法中解决抛出的异常：`try{...}`中抛出，`catch(){...}`中解决。
- 异常抛出点后的代码在抛出异常后不再执行，即异常的抛出终止了代码段的执行。



## 2. throws语句

---

- 格式:

方法名（形参列表） throws 异常类名1,  
异常类名2, .....

{...}

- 仅仅抛出，由调用该方法的其他方法使用try...catch...finally捕获并处理异常。

## 9.2.3 用户自定义的异常

- 当我们在设计自己的类包时，应尽最大的努力为用户提供最好的服务，并且希望用户不要滥用我们所提供的方法，当程序出现某些异常事件时，我们希望程序足够健壮以从程序中恢复，这时就需要用到异常。
- 在选择异常类型时，可以使用**Java**类库中已经定义好的类，也可以自己定义异常类。自定义异常类不是由**Java**系统监测到的异常（如数组下标越界，被0除等），而是由用户自己定义的异常。
- 自定义异常同样要用**try-catch-finally**捕获，但必须由用户自己抛出（**throw**）。

# 1.自定义的异常

## ■ 格式:

```
class 自定义异常类 extends 父异常类名  
{ ... //类体; }
```

## ■ 注:

- 抛出的异常必须是`Throwable`或其子类的实例。建议:
  - 异常一定是不经常发生的故障, 应避免把控制流程作为异常处理
  - 尽量使用JDK提供的异常类: 重用、便于理解
  - 用`Exception/ RuntimeException`类: 编译时异常、运行时异常。
  - 一般不把自定义异常作为`Error`的子类, 因为`Error`通常被用来表示系统内部的严重故障。
- 在类体中, 定义用户异常类的属性和方法或覆盖父类的属性和方法, 使得出现该类异常时输出相应的信息。

## 2. 抛出自定义异常

### ■ 格式:

[修饰符] 返回类型 方法名(参数列表) **throws** 自定义异常

```
{ ...;  
    if(条件)  
        throw new myException();  
    else  
        ...;  
}
```


注：自定义异常不可能依靠系统自动抛出，必须通过**throw**语句抛出，通常是通过条件判断确定是否抛出这个异常类的新对象。

## 9.2.4 多异常处理

---

- 一段程序中既有系统异常类，又有程序员自己定义的异常类。
- 分类处理。
  - 1.系统异常类：自动抛出，人工捕获；
  - 2.自定义类：人工抛出，人工捕获。

# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)



# 第十章 Java的输入输出

---

- 10.1 输入/输出流概述
- 10.2 字节输入/输出流
- 10.3 字符输入/输出流
- 10.4 文件输入/输出流
- 作业IV

# 10.1 输入/输出流概述

## ■ 10.1.1 输入/输出流的概念

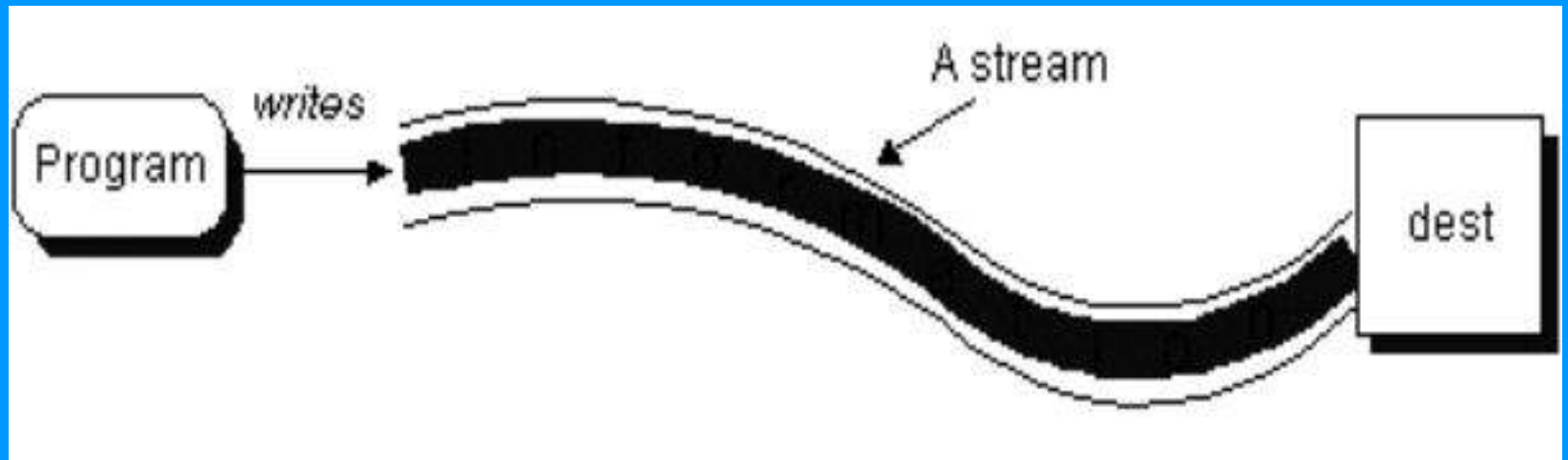
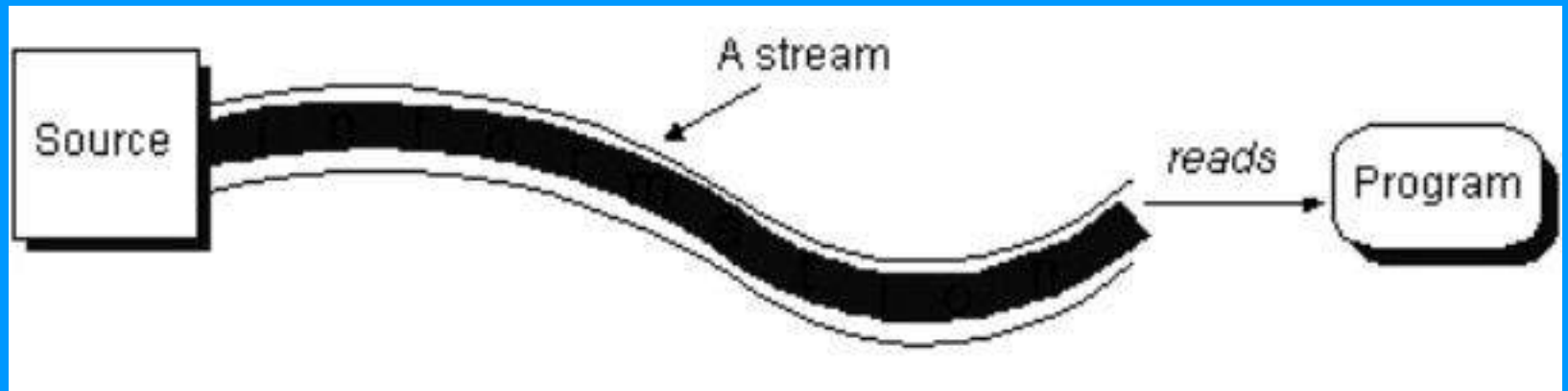
- 流：计算机中流动的数据缓冲区，分：
  - 输入流：从外设流向CPU，数据提供者，可从中读取数据出来，如键盘；
  - 输出流：从CPU流向外设，数据接收者，可往其中写数据，如显示器；
  - 特：文件，当向其中写数据时，它就是一个输出流；当从其中读取数据时，它就是一个输入流。
- Java中的所有输入输出，都通过流完成。
- 流就像一个管道，连通了信息的源及其目的地。



# 10.1 输入/输出流概述

## ■ 10.1.2 I/O流类

- 在Java开发环境中，主要是由包java.io中提供的一系列类和接口来实现输入/输出处理。
- I/O类层次结构，图10.1。
- 按所读写的数据类型分两类：
  - 字符流类（*Character Streams*）：用于向字符流读写16位二进制字符。多用于读写文本数据。
  - 字节流类（*Byte Streams*）：用于向字节流读写8位二进制的字节。一般地，字节流类主要用于读写诸如图象或声音等的二进制数据。
- 基本I/O流类：InputStream和OutputStream类，都是字节流类，都是抽象类，不能直接创建对象。



[back](#)

## 10.1.2 I/O流类

---

### ■ 1. InputStream类

- InputStream类层次关系图，图10.2。
- 缓冲区：数据传输时，用来暂存数据的存储区域
- 主要方法：
  - read(): 从流中读入数据
  - skip(): 跳过流中若干字节数
  - available(): 返回流中可用字节数
  - close(): 关闭流
- read()方法的三种形式：

# InputStream类

- 在InputStream类中，方法read()提供了三种从流中读数据的方法：
  - **int read():** 从输入流中读一个字节，形成一个0~255之间的整数返回（是一个抽象方法）。
  - **int read(byte b[]):** 读多个字节到数组b中，填满整个数组。
  - **int read(byte b[], int off, int len):** 从输入流中读取长度为len的数据，写入数组b中从索引off开始的位置，并返回读取得字节数。
- 对于这三个方法，若返回-1，表明流结束（当前位置没有数据），否则，返回实际读取的字节数。

## 10.1.2 I/O流类

---

### ■ 2. OutputStream类

– OutputStream类层次关系图，图10.3。

– 主要方法：

- `write(int b)`: 将一个整数输出到流中（只输出低位字节，为抽象方法）
- `write(byte b[])`: 将字节数组**b**中的数据输出到流中
- `write(byte b[], int off, int len)`: 将数组**b**中从**off**指定的位置开始，长度为**len**的数据输出到流中
- `flush()`: 刷空输出流，并将缓冲区中的数据强制送出
- `close()`: 关闭流，释放资源。

## 10.1.3 标准输入/输出流

- 语言包java.lang中的System类管理标准输入/输出流和错误流。
- **System.in**，从InputStream中继承而来，用于从标准输入设备中获取输入数据(通常是键盘)。
- **System.out**，从PrintStream中继承而来，把输出送到缺省的显示设备(通常是显示器)。
- 每当main方法被执行时，就自动生成上述二个对象。

# System.in

---

- 标准输入流，InputStream类的对象。
- `int read()`方法：从键盘缓冲区读取一个字节数据，得到的返回值为一个int数据。返回值中的高字节都是0,低字节是读出的输入字符的编码，可判断是哪个字符。
- 可能产生IOException异常，需要try...catch语句处理。



# System.out

---

- 标准输出流，是打印输出流PrintStream类(FilterOutputStream的子类)的对象.
- print()与println();



## 10.2 字节输入/输出流

- 字节流：以byte为基本处理单位的流。
- 由InputStream和OutputStream的派生类处理。
- 10.2.1 字节输入流
  - **DataInputStream**类：
  - 允许应用程序以与机器无关的方式从基本输入流中以字节的方式读取**Java**基本类型数据。实现**DataInput**接口，其中定义了读取不同数据类型的抽象方法。

## 10.2.1 字节输入流

---

### ■ 1.构造方法

- `DataInputStream(InputStream in)`:创建一个字节流对象，并从基本输入流in对象中读取数据。

### ■ 2.常用方法 （P224）

- 各种类型数据的读取方法。

## 10.2.2 字节输出流

### ■ DataOutputStream类:

- 允许应用程序以适当的方式将Java基本类型数据以字节的方式写入到基本输出流中。实现DataOutput接口，其中定义了写入不同数据类型的抽象方法。
- 构造方法：
  - DataOutputStream(OutputStream out):创建一个对象，将数据写入到基本输出流中。
- 常用方法：P225
  - 各种数据类型的写入方法。
  - void flush()
  - void close()

## 10.2.3 字节过滤流

---

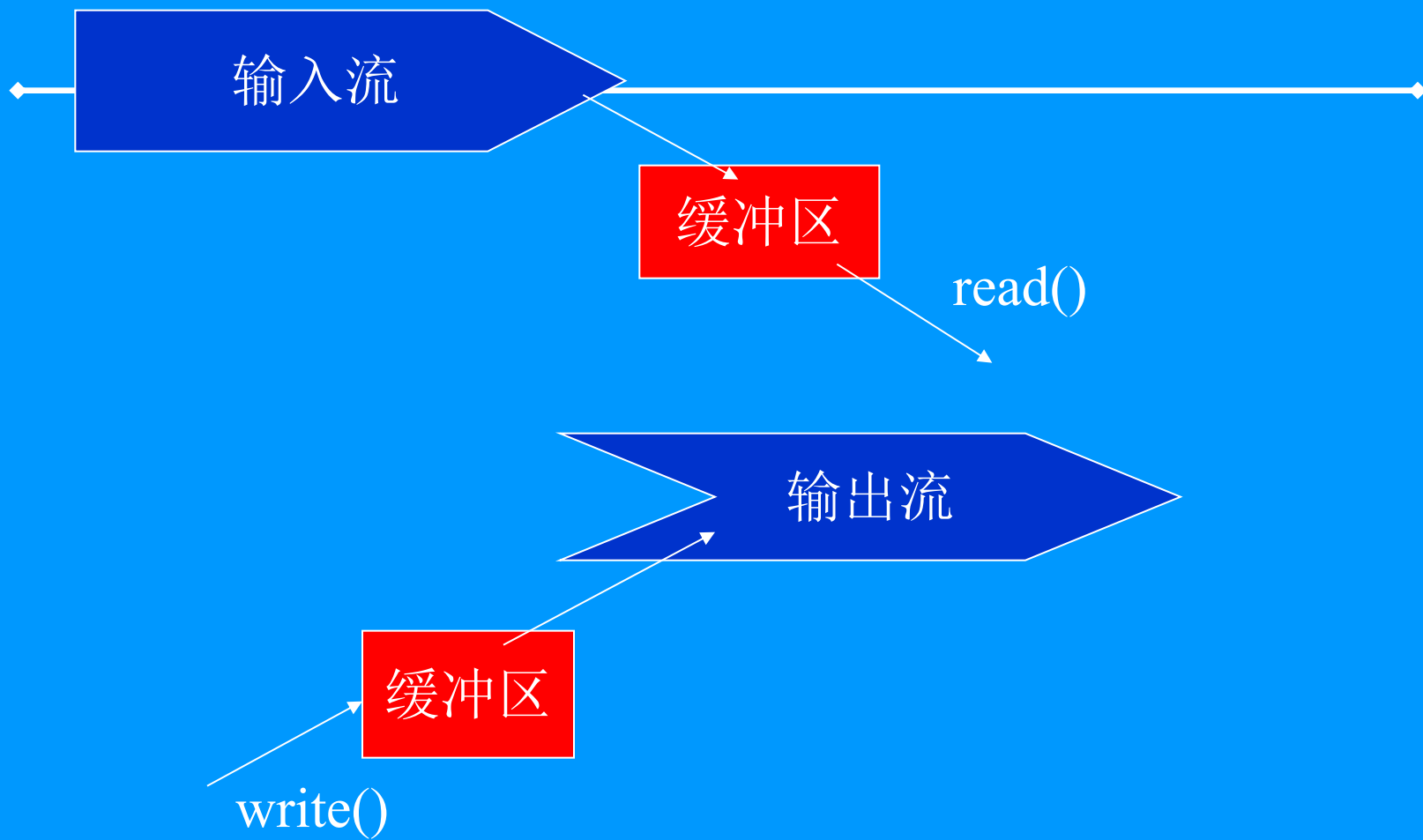
- 过滤：提供更多功能。
- 过滤流与I/O流配合使用,在读写数据的同时对数据进行处理。
- 使用过滤流，须首先把过滤流连接到某个I/O流上，通常在构造方法的参数中指定所要连接的I/O流。
- 常用过滤流都继承于FilterInputStream和FilterOutputStream类。



# 1. BufferedInputStream类

---

- BufferedInputStream类实现了带缓冲的过滤流，它提供了缓冲机制，把任意的输入流“捆绑”到缓冲流上，可以提高该输入流的读取效率。
- 创建BufferedInputStream对象即创建一个带内部缓冲区数组。



# 1. BufferedInputStream类

## ■ 1.构造方法:

- `BufferedInputStream(InputStream in)`: 创建一个带有缓冲区字节过滤输入流对象，连接到基本输入流 `in` 对象上。
- `BufferedInputStream(InputStream in, int size)`: 除了要指定所连接的I/O流之外，还可以指定缓冲区的大小。缺省时是用32字节大小的缓冲区。

## ■ 2.常用方法: P227

- `mark(int readlimit)`: 在此输入流中标志当前位置。
- `reset()`: 重新定位到此输入流最后调用`mark()`方法时的位置。

## 2. BufferedOutputStream类

- 实现带缓冲的输出流。
- 1.构造方法：
  - BufferedOutputStream(OutputStream out):
    - 创建一个带有缓冲区的输出流对象，并将数据写入到指定的基本输出流。
  - BufferedOutputStream(OutputStream out, int size):
    - 创建一个带有指定大小缓冲区的输出流对象。

### 2.常用方法:



## 10.3 字符输入/输出流

- Java语言中字符是采用Unicode字符编码。汉字、英文字母和其他字符等都采用两个字节编码。
- 使用字节流处理汉字时，有时可能因为只读了汉字的一个字节编码，使文件出现乱码。应使用字符流处理。
- 字符I/O流派生于Reader和Writer抽象类。

## 10.3.1 字符输入流

- 派生于Reader抽象类。结构--图10.4。
- 1、常用字符输入流
  - **BufferedReader**: 从字符输入流中读取文本
  - **InputStreamReader**: 将字节流转换为字符流。
- 2、字符输入流常用方法
  - **int read()**: 读取一个字符，返回字符的整数编码。
  - **int read(char b[])**: 读取字符存储到数组b中，返回实际读取字符数量。
  - 其他，P229。

## 10.3.1 字符输入流

### ■ 3、InputStreamReader类

- 功能：将字节流转化为字符流，将读取的字节数据解释为字符编码，可以指定字符集。
- 构造方法：  
`InputStreamReader(InputStream in)`: 创建一个使用默认字符集的字符输入流对象，in为指定要转换的字节流。
- 例10.7， P230

## 10.3.2 字符输出流

- 派生于Writer抽象类。结构--图10.5。
- 1、常用字符输出流
  - BufferedWriter: 将文本写入字符流
  - OutputStreamWriter: 将字节输出流转换为字符流。
- 2、字符输出流类常用方法
  - void write(int b): 将参数b的低16位值写入输出流中。
  - void write(char b[ ]):将字符数组b中的值写入输出流中。
  - void write(String str): 将字符串str中的字符写入输出流。

## 10.3.2 字符输出流

### ■ 3、OutputStreamWriter类

- 功能：将字节流转化为字符流，可以实现各种数据的输出。
- 构造方法：  
`OutputStreamWriter(OutputStream out)`: 创建一个字符输出流对象，与out输出流对象连接。
- 例10.8， P232

## 10.3.3 字符过滤流

### ■ 1、BufferedReader类

- 功能：从字符输入流中读取文本，缓冲各个字符，从而实现字符、数组和行的高效读取。

- 构造方法：

**BufferedReader(Reader in):**创建一个使用默认大小输入缓冲区的缓冲字符输入流。

**BufferedReader(Reader in, int sz):**创建一个使用指定大小输入缓冲区的缓冲字符输入流。

- 常用方法：**readLine()**:读取一个文本行

```
■ BufferedReader in=new BufferedReader(  
    new InputStreamReader( System.in ));  
String s=in.readLine();
```

## 10.3.3 字符过滤流

### ■ 2、BufferedWriter类

- 功能：将文本写入字符输出流，缓冲各个字符，从而提供单个字符、数组和字符串的高效写入。

- 构造方法：

**BufferedWriter(Writer out):**创建一个使用默认大小输出缓冲区的缓冲字符输出流。

**BufferedWriter(Writer out, int sz):**创建一个使用给定大小输出缓冲区的新缓冲字符输出流。

- 常用方法：

**newLine() :**写入一个行分隔符。行分隔符字符串由系统属性 **line.separator** 定义，并且不一定是单个新行 ('\n') 符。

## 10.4 文件输入/输出流

- 文件操作:向文件中写入数据或从文件中读取数据,靠文件输入输出流类来完成。
- 字节文件I/O流类:FileInputStream和FileOutputStream
- 字符文件I/O流类:FileReader和FileWriter
- 随机访问文件类: RandomAccessFile
- 目录管理类: File;
- 注:
  - 字节文件I/O流类可以处理所有文件, 包括字符文件;
  - 字符文件I/O流类不能处理非字符文件, 如图片等二进制字节文件。



## 10.4.1 字节文件输入/输出流

- 在向文件中写数据或从文件中读数据时采用字节方式处理。

- 1、**FileInputStream**类

- 功能：从文件系统中的某个文件中获得输入字节。主要用于读取诸如图像数据之类的原始字节流。
- 构造方法：

**FileInputStream(File file)**: 通过打开一个到实际文件的连接来创建一个 **FileInputStream**对象，该文件通过文件系统中的 **File** 对象 **file** 指定。

**FileInputStream(String name)**: 通过打开一个到实际文件的连接来创建一个 **FileInputStream**对象，该文件通过文件系统中的路径名 **name** 指定。

- 常用方法，P235。

# 10.4.1 字节文件输入/输出流

## ■ 2、FileOutputStream类

- 功能：向文件系统中的某个文件中写入字节。主要用于写入诸如图像数据之类的原始字节流。
- 构造方法：
  - `FileOutputStream(File file)` ;
  - `FileOutputStream(File file, boolean append)`: 创建一个向指定 `File` 对象表示的文件中写入数据的文件输出流。`append` 如果为 `true`，则将字节写入文件末尾处，而不是写入文件开始处。
  - `FileOutputStream(String name)`;
  - `FileOutputStream(String name, boolean append)`: 创建一个向具有指定 `name` 的文件中写入数据的输出文件流。`append` : 追加 (`true`) 或重写 (`false`)。
- 注：指定的文件对象可以不是实际存在的文件，如果不存在会自动按指定文件名新建一个空文件。
- 常用方法：P236。

## 10.4.2 字符文件输入/输出流

- 在向文件中写数据或从文件中读数据时采用字符方式处理。
- 1、**FileReader**类
  - 功能：读取字符文件，使用默认字符编码和默认字符缓冲区大小。
  - 构造方法：
    - **FileReader(File file)**:创建一个从给定File对象中读取数据的字符文件输入流对象。
    - **FileReader(String fileName)**:创建一个从以fileName文件名指定的文件中读取数据的对象。

## 10.4.2 字符文件输入/输出流

### ■ 2、FileWriter类

- 功能：写入字符文件，使用默认字符编码和字节缓冲区大小。
- 构造方法：
  - `FileWriter(File file);`
  - `FileWriter(File file, boolean append);`根据给定的 `File` 对象构造一个 `FileWriter` 对象。`append`: 追加(`true`)或重写(`false`)
  - `FileWriter(String fileName);`
  - `FileWriter(String fileName, boolean append);`根据给定的文件名以及指示是否附加写入数据的 `boolean` 值来构造 `FileWriter` 对象。

## 10.4.3 文件的访问


- **File**类提供对操作系统目录管理的功能，主要用于文件命名、文件属性查询、处理文件目录、创建目录等。不能对文件内容进行读写。
- 1、构造方法
  - **File(File parent, String child)**:根据 **parent** 抽象路径名和 **child** 路径名字符串创建一个新 **File** 实例。
  - **File(String pathname)**:通过将给定路径名字符串转换为抽象路径名来创建一个新 **File** 实例。
- 2、常用方法,P238.
- 3、注：
  - “C:\\java”: 表示C盘根目录下java文件夹;
  - “..\\java”: 表示当前子目录的上一级目录下的java文件夹。

# 作业IV:

---

- 编写一个猜数字游戏程序，要求：
  - 1、图形化界面，基于Swing或AWT;
  - 2、随机产生一个三位整数（三个数字不能重复，即0~9只能用一次），让用户猜;
  - 3、判断用户填入的三位整数是不是符合产生的随机数：如果数字和位置都对，游戏结束；如果没猜对：给出提示信息（猜对了几位数，有几个数位置正确，等等。如156，用户猜135，则提示全对1个(1)，半对1个(5:数字对，位置错)），继续猜，直到用户猜对为止。
  - 4、将游戏结果或过程存入一个存档文件，用户可以查看游戏记录。

# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)



# 第11章 Java的多线程

---

- 11.1 多线程概述
- 11.2 线程的创建
- 11.3 多线程操作



# 多线程

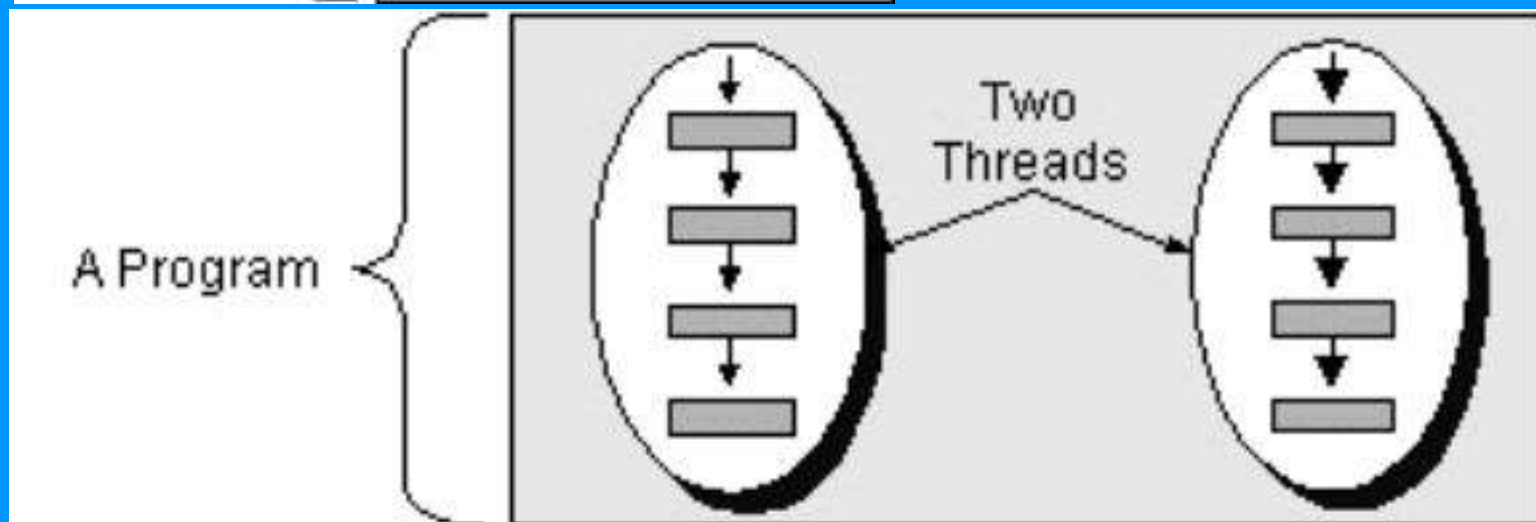
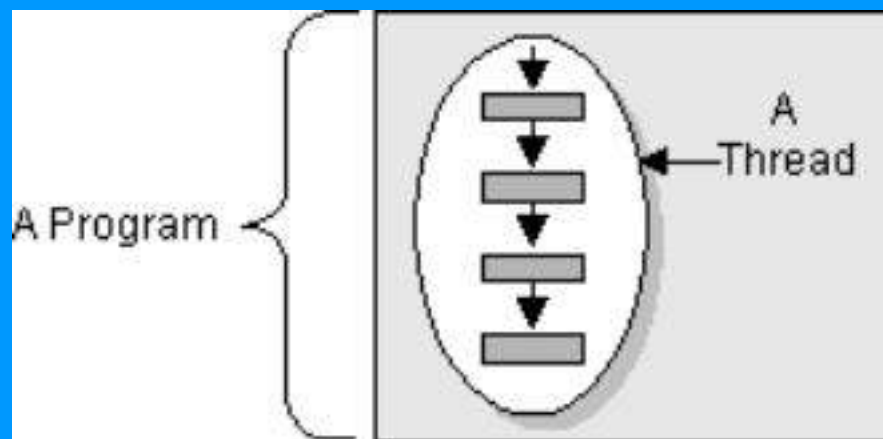
---

- 并发现象在现实生活中大量存在
  - 人体（消化、运动）
  - 计算机（同时运行多中程序）
- 多线程——在一个程序中实现并发
  - 计算机可在同一时间完成不同任务
  - 加速程序运行，提高CPU利用率
  - 网络编程需要设计并发执行

# 11.1 多线程概述

- 以前所编写的程序，每个程序都有一个入口、一个出口以及一个顺序执行的序列，在程序执行过程中的任何指定时刻，都只有一个单独的执行点，即单线程。
- 一个线程是一个程序内部的一个顺序控制流。
- 线程并不是程序，它自己本身并不能运行，必须在程序中运行。在一个程序中可以实现多个线程，这些线程同时运行，完成不同的功能，这就是所谓的多线程。

# 多线程概述示意图



## 11.1.1 与线程有关的概念

- **程序(program)**是对数据描述与操作的一段静态代码的集合，是应用程序执行的依据。
- **进程(process)**是程序的一次动态执行过程，是系统运行程序的基本单位。
- **线程(Thread)**是比进程更小的执行单位，是一个程序内部的顺序控制流。
- **多线程**：一个程序中可以同时运行多个不同的线程，执行不同的任务。

## 11.1.1 与线程有关的概念

- **多线程意义**：一个程序的多行语句在多台处理器下可以同时运行，而在单处理器下分时运行，提高程序的运行效率。
- 多线程与多进程的区别：
  - 内存空间与系统资源共享
  - 相互切换时，系统负担小
- 多线程编程的目的：最大限度地利用CPU资源。
  - CPU发展：速度更快，多核。

## 11.1.2 Java中的线程模型

- **多线程的优势：**
- 多线程编程简单，效率高（能直接共享数据和资源，多进程不能）
- 适合于开发服务程序（如**Web**服务，聊天服务等）
- 适合于开发有多种交互接口的程序（如聊天程序的客户端，网络下载工具）
- 适合于有人机交互又有计算量的程序（如字处理程序**Word**，**Excel**）
  - 减轻编写交互频繁、涉及面多的程序的困难（如监听网络端口）
  - 程序的吞吐量会得到改善（同时监听多种设备，如网络端口、串口、并口以及其他外设）
  - 有多个处理器的系统，可以并发运行不同的线程（否则，任何时刻只有一个线程在运行）

## 11.1.2 Java中的线程模型

- 对多线程的综合支持是Java语言的一个重要特色，它提供了Thread类来实现多线程。在Java中，线程可以认为是由三部分组成的：
  - 虚拟CPU，封装在java.lang.Thread类中，它控制着整个线程的运行；
  - 执行的代码，传递给Thread类，由Thread类控制顺序执行；
  - 处理的数据，传递给Thread类，是在代码执行过程中所要处理的数据。

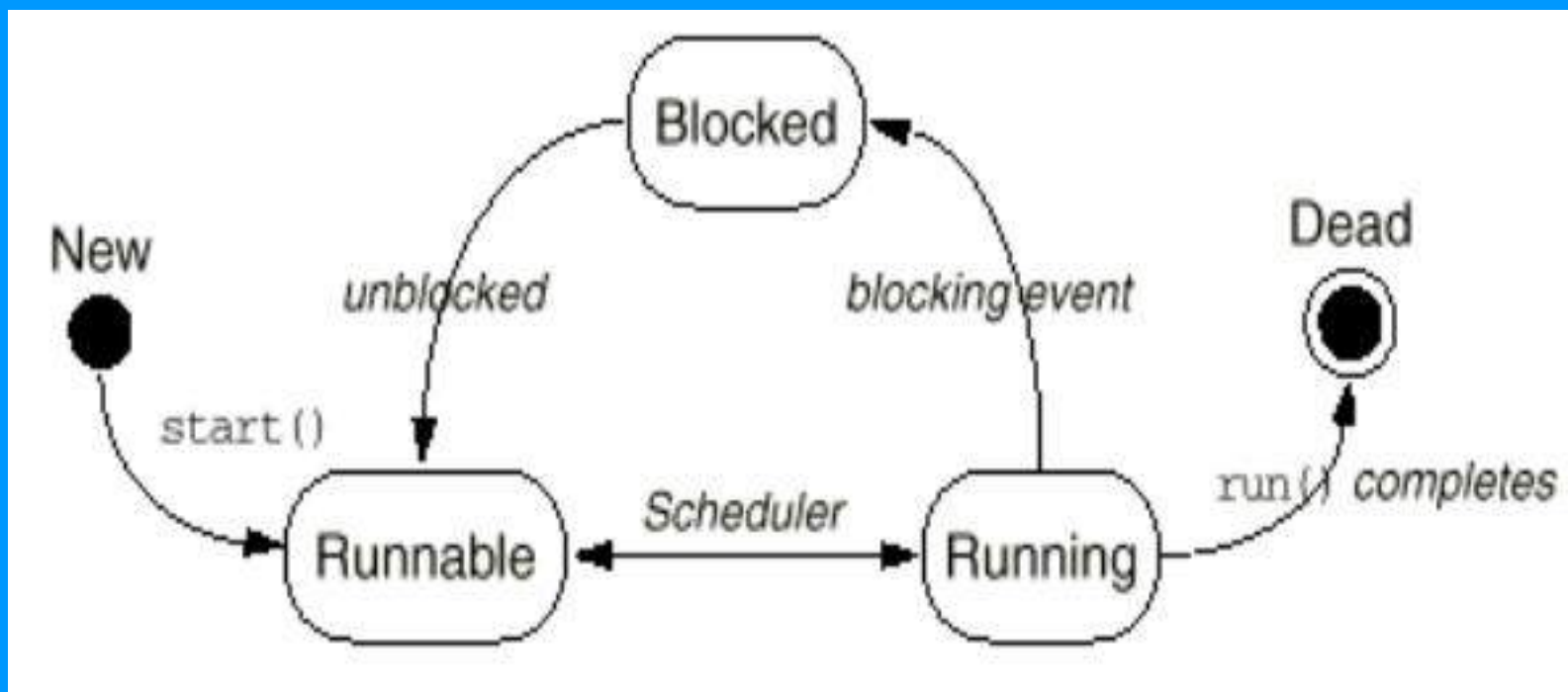
## 11.1.2 Java中的线程模型

- Java的线程是通过Java的软件包java.lang中定义的类Thread来实现的。当生成一个Thread类的对象之后，就产生了一个线程，通过该对象实例，可以启动线程、终止线程、或者暂时挂起线程等。



## 11.1.3 线程的状态和生命周期

- 线程的生命周期：一个线程从创建到消亡的过程。
- 线程的生命周期可分为5个状态：
  1. 新建状态
  2. 可运行状态
  3. 运行状态
  4. 不可运行状态
  5. 死亡状态



## 11.1.3 线程的状态和生命周期

- 新建状态（new Thread）
- 当创建了一个新的线程时（**`myThread thd = new myThread();`**），它就处于创建状态，此时它仅仅是一个空的线程对象，系统不为其分配资源。处于这种状态时只能启动或终止该线程，调用除这两种以外的其它线程状态相关的方法都会失败并且会引起非法状态例外 `illegalThreadStateException`（对于其它状态，若所调用的方法与状态不符，都会引起非法状态例外）。

## 11.1.3 线程的状态和生命周期

- 可运行状态 (Runnable)
- 当线程处于创建状态时，可以调用`start()`方法（`thd.start();`）来启动它，产生运行这个线程所需的系统资源，安排其运行，并调用线程体`run()`方法，这样就使得该线程处于可运行 (Runnable) 状态。
- 需要注意的是这一状态并不是运行中状态 (Running)，因为线程也许实际上并未真正运行 (Ready)。



## 11.1.3 线程的状态和生命周期

- **运行状态(Running)**
- 当可运行状态的线程获得CPU资源时，即实际被CPU运行，这是线程的状态被称为运行状态。线程进入运行状态就会调用该线程对象的RUN()方法，RUN()方法定义该线程所要完成的操作和功能。

## 11.1.3 线程的状态和生命周期

- 不可运行状态（Not Runnable）
- 线程处于可运行状态时，当下面四种情况发生，线程就进入不可运行状态：
  - 调用了`sleep()`方法，睡眠；
  - 调用了`suspend()`方法，挂起；
  - 为等候一个条件变量，线程调用`wait()`方法；
  - 输入输出流中发生线程阻塞，费时操作。

## 11.1.3 线程的状态和生命周期

- 死亡状态 (Dead)
- 线程的终止一般可通过两种方法实现：  
自然撤消或是被停止。
  - 自然撤消是指从线程的run()方法正常退出；
  - 被停止：调用线程的实例方法stop()或destroy()则可以强制停止当前线程,前者会产生异常(SecurityException),后者是强制终止。

## 11.1.4 线程的调度和优先级

- Java提供一个**线程调度器**来监控程序中启动后进入可运行状态的所有线程。
- 线程调度器依据**优先级基础上的先到先服务**原则：按照线程的**优先级**决定调度哪些线程来执行，具有高优先级的线程会在较低优先级的线程之前得到执行。
- 同时线程的调度是**抢先式**的，即如果当前线程在执行过程中，一个具有更高优先级的线程进入可执行状态，则该高优先级的线程会被立即调度执行。
- 多个线程运行时，若线程的优先级相同，按**先到先服务**原则，由操作系统按**分时方式**(时间片轮转)或**独占方式**来分配线程的执行时间。

## 11.2 线程的创建

- 在Java中通过run方法为线程指明要完成的任务，有两种技术来为线程提供run方法。
  1. 继承Thread类并重载run方法。
  2. 通过定义实现Runnable接口的类进而实现run方法。



# 1.Thread类

- 继承Thread类并重载run方法。
- Thread类：是专门用来创建线程和对线程进行操作的类。代表Java程序的单个线程。Thread中定义了许多方法对线程进行操作。在java.lang包。
- (1) 构造方法：

**public Thread( ThreadGroup group, Runnable target, String name)** 创建一个新的线程对象。

- group指明了线程所属的线程组；
- target是线程体run()方法所在的对象；
- name是线程的名称。
- 以上3个参数可任意组合，或者都不写，实现构造方法的多个重载。

# 1.Thread类

## ■ (2)静态方法

- `void sleep(long millis)`: 让正在运行的线程休息`millis`毫秒再运行。产生`InterruptedException`。

## ■ (3)常用方法

- `void start()`: 启动已创建的线程对象。
- `void run()`:最重要的方法，线程执行的起点，线程执行的具体操作都在此方法中编码。
- `void destroy()`:销毁线程
- `void stop()`:终止线程的执行
- `void suspend()` / `void resume()`:挂起/恢复线程

## ■ (4)静态常量

- 优先级表示: `setPriority(int )`;
- `Priority` , 1~10 , `MIN` / `NORM` / `MAX_PRIORITY`



## 2. Runnable接口

---

- 建立支持多线程的类，需实现Runnable接口。Thread类本身也实现Runnable接口。
- 实现Runnable接口只提供一个run()方法，此方法中可编写控制线程的代码。任何实现Runnable接口的类都可以作为一个线程Thread的目标对象。
- 实现Runnable接口位于java.lang包。

## 11.2.1 直接方式创建线程

- 定义一个线程类，继承于Thread类并重写其中的方法run()。由于Java只支持单继承，用这种方法定义的类不能再继承其他类。创建一个线程对象后，在程序中调用start()方法即可启动线程。
- 步骤：
  - (1) 定义Thread类的一个子类；
  - (2) 在该类中覆盖Thread类的run()方法，编写操作代码。
  - (3) 创建该类的一个线程对象。
  - (4) 通过start()方法启动线程，执行run()方法。

## 11.2.1 直接方式创建线程

总体结构如下：

```
public class MyThread extends
    Thread {
    public void run() {
        ... ..
    }
}
.....
MyThread t = new MyThread();
t.start();
```

## 11.2.2 间接方式创建线程

- 通过建立一个实现了Runnable接口的对象，并以它作为线程的目标对象来创建一个线程。
- Runnable接口：定义了一个抽象方法run()。定义如下：

```
public interface java.lang.Runnable{  
    public abstract void run();  
}
```
- 步骤：
  - (1)定义一个实现Runnable接口的类。
  - (2)实现run()方法。
  - (3)创建该类的对象，将该对象做参数，传递给Thread类的构造方法，构造一个Thread对象。
  - (4)调用start(),启动线程。

## 11.2.2 间接方式创建线程

创建的总体框架如下：

```
class MyRunner implements Runnable
{
    public void run() {
        ...
    }
}
```

```
MyRunner my = new MyRunner();
```

```
Thread tt = new Thread( my, ... );
```

```
tt.start();
```

# 11.3 多线程操作

## ■ 11.3.1 多线程的互斥

- 在某一时刻只能有一个线程执行某个执行单元的机制就叫互斥控制或共享互斥。

## ■ 11.3.2 多线程的同步

- 当两个或以上的线程需要共享资源时，它们需要某种方法来确定资源在某一时刻只能被一个线程占用，达到此目的的过程称为多线程的同步。
- 封锁技术：互斥锁—**Synchronized**
- 解决方法：在线程使用一个资源时为其加锁即可。访问资源的第一个线程为其加上锁以后，其他线程便不能再使用那个资源，除非被解锁。



# 11.3 多线程操作

## ■ 11.3 线程的死锁


- 当一个线程需要一个资源而另一个线程持有该资源而不释放时，就会发生死锁。
- **Java**技术既不能发现死锁，又不能避免死锁。注意死锁，尽量避免。
- 死锁是资源的无序使用而带来得，解决死锁问题的方法就是给资源施加排序。



# 线程与进程：

- 从逻辑的观点来看，多线程意味着一个程序的多行语句同时执行，但是多线程并不等于多次启动一个程序，操作系统也不会把每个线程当作独立的进程来对待：
  - 两者的粒度不同，是两个不同层次上的概念。进程是由操作系统来管理的，而线程则是在一个程序（进程）内。
  - 不同进程的代码、内部数据和状态都是完全独立的，而一个程序内的多线程是共享同一块内存空间和同一组系统资源，有可能互相影响。
  - 线程本身的数据通常只有寄存器数据，以及一个程序执行时使用的堆栈，所以线程的切换比进程切换的负担要小。

# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)



# 第12章 Java的网络编程技术

---

- 12.1 网络基础
- 12.2 URL统一资源定位符
- 12.3 TCP Socket通信
- 12.4 UDP Socket通信

# 12.1 网络基础

- 12.1.1 -12.1.3 参看计算机网络课程
- 12.1.4 Socket模式
  - Socket: Berkeley Socket, 一个应用程序接口, 用户开发的应用程序可以通过此接口在网络上与其他应用程序进行通信。
  - Java网络编程开发的基础。
- Java语言在网络编程方面提供了许多方便, 其他语言往往需要数页代码才能完成的事情, 在Java中可能只需要一条语句就可以。

## 12.2 URL

- URL: Uniform Resource Locator 统一资源定位符，表示Internet上某一资源的地址。
- 通过URL，可以用一种统一的格式来描述各种信息资源。

- URL的格式:

`http://www.126.com/index/index.htm`

- 协议: http,ftp,file等
- 主机地址: 域名或IP地址（有时包括端口如:8008），与协议用” ://”隔开；
- 资源具体地址: 目录或文件名，与主机用” /”隔开。

## 12.2 URL

- URL类：在`java.net`包中，对URL地址的抽象。
- URL对象：存放一个具体的资源引用地址；
- Java虚拟机中有些协议不支持，常见协议HTTP、FTP、File等都支持。
  - 构造方法：`public URL(String url)`:通过一个表示URL地址的字符串创建一个URL对象。
  - 注意可能抛出`MalformedURLException`异常。
  - 常用方法：
    - `InputStream openStream()`:返回一个指向URL对象所包含的资源的输入流。
    - `String getHost()`: 返回主机信息
    - `int getPort()`: 返回端口信息
    - `String getProtocol`: 返回协议信息

# 例：简易浏览器

---

## ■ 关键代码：

.....

```
JEditorPane htm;
```

```
try{
```

```
    URL url = new URL("http://www.126.com");
```

```
    htm.setPage(url); }
```

```
catch(IOException e){ }
```

.....



## 12.3 TCP Socket通信

- TCP协议和UDP协议--各有各的用处:
  - 面向连接的通信协议，当对所传输的数据具有时序性和可靠性等要求时，应使用TCP协议；
  - 面向无连接的数据报通信协议，当传输的数据比较简单、对时序等无要求时，UDP协议能发挥更好的作用，如ping、发送时间数据等。
- TCP: Java提供两种套接字类:
  - java.net.Socket: 编写客户端程序用；
  - java.net.ServerSocket: 编写服务器端软件；

## 12.3 TCP Socket通信

- 无论一个Socket通信程序的功能多么齐全、程序多么复杂，其基本结构都是一样的，都包括以下四个基本步骤：
  - 1、在客户方和服务器方创建Socket/ServerSocket实例。
  - 2、打开连接到Socket的输入/输出I/O流。
  - 3、利用I/O流，按照一定的协议对Socket进行读/写操作。
  - 4、关闭输入/输出流和Socket。
- 通常，程序员的主要工作是针对所要完成的功能在第3步进行编程。

## 12.3.1 TCP客户端


### 创建TCP客户端程序的步骤：

- (1)创建Socket对象，连接服务器。
  - Socket类构造方法：
    - public Socket(InetAddress,String host,int port):
      - InetAddress:主机IP地址;
      - host:主机名;
      - port:服务器端口号;
      - 如本节测试: **Socket("127.0.0.1",4444);**
- (2)利用getInputStream()或getOutputStream()方法，收发信息；(即打开连接到Socket的I/O流).
- (3)处理I/O流，read()或write()方法;
- (4)关闭I/O流和Socket对象，close()方法;

## 12.3.2 TCP服务器端

- 创建TCP服务器端程序的步骤：
  - (1)创建ServerSocket对象，构造方法：
    - `public ServerSocket(int port, int maxcount):`
      - `port`:所使用端口号；
      - `maxcount`:最大客户端数目；
  - (2)与客户端建立通信连接。
    - `Socket accept()`方法：等待客户端服务请求，返回值；
  - (3)与客户端通信。
    - 利用`getInputStream()`或`getOutputStream()`方法，收发信息；
  - (4)处理I/O流，`read()`或`write()`方法；
  - (5)关闭I/O流和ServerSocket对象，`close()`方法；

# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)



# 第十三章 Java数据库编程

---

- 13.1 数据库简介
- 13.2 JDBC技术应用
- 13.3 数据库编程实例



# 13.1 数据库简介

---

- 13.1.1 关系数据库
- 13.1.2 SQL简介

注：参阅《数据库系统概论》课本。

# 13.2 JDBC技术应用

## ■ 13.2.1 JDBC简介

- JDBC(Java Database Connectivity)是一个独立于特定数据库管理系统的、通用的SQL数据库存取和操作的公共接口（一组API），定义了用来访问数据库的标准Java类库，使用这个类库可以以一种标准的方法、方便地访问数据库资源（在java.sql类包中）。
- JDBC为访问不同的数据库提供了一种统一的途径，象ODBC(Open Database Connectivity)一样，JDBC对开发者屏蔽了一些细节问题。
- JDBC的目标是使应用程序开发人员使用JDBC可以连接任何提供了JDBC驱动程序的数据系统，无需对特定的数据库系统的特点有过多的了解



## 13.2.1 JDBC简介

### ■ JDBC支持四种类型的驱动程序：

- JDBC-ODBC Bridge, plus ODBC driver
  - 由 Sun的Java2 JDK提供(sun.jdbc.odbc.JdbcOdbcDriver)
  - 通过ODBC驱动程序来获得对数据库的JDBC访问；
  - 必须先安装ODBC驱动程序和配置ODBC数据源；
  - 仅当特定的数据库系统没有相应的JDBC驱动程序时使用。
- Native-API, partly Java driver
- JDBC-net, pure Java driver
- Native-protocol, pure Java driver

## 13.2.1 JDBC简介

---

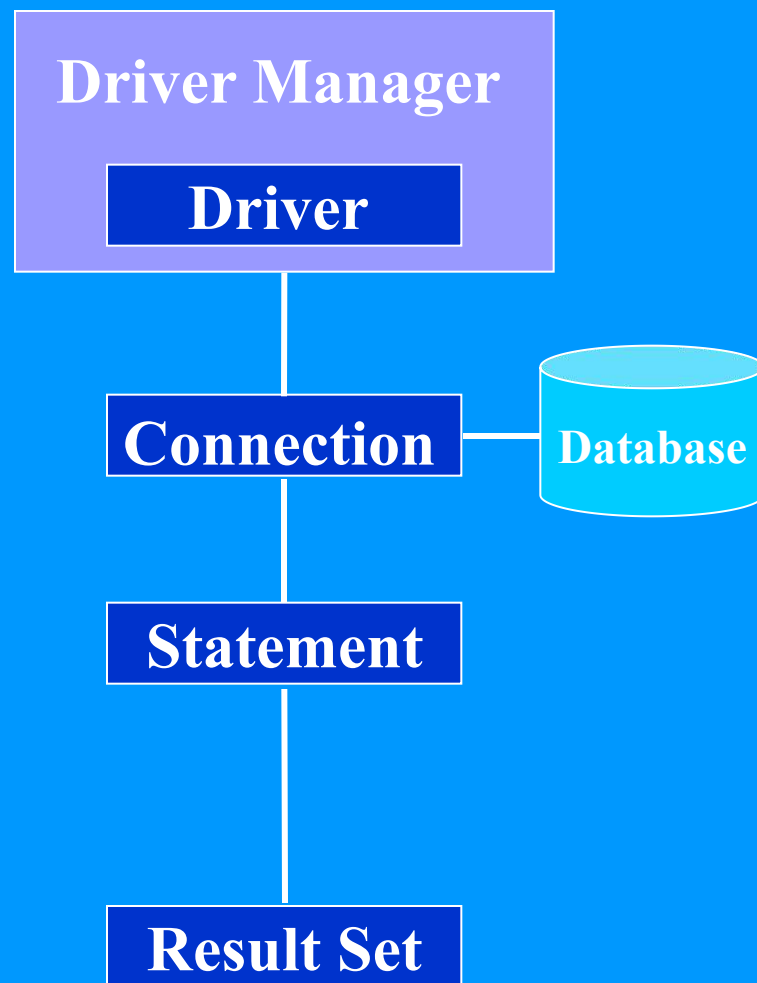
- JDBC与主流数据库的连接方法：
  - 1、创建指定数据库的URL对象。
  - 2、通过URL对象，利用DriverManager的getConnection方法建立连接。
- 数据库URL对象，构成格式。
- 常见数据库连接属性（SQL Server）
- JDBC提供的主要接口和类。

# JDBC提供的主要接口和类

- **DriverManager (java.sql.DriverManager)**
  - 装载驱动程序，管理应用程序与驱动程序之间的连接。
- **Connection (java.sql.Connection)**
  - 将应用程序连接到特定的数据库
- **Statement (java.sql.Statement)**
  - 在一个给定的连接中，用于执行一个静态的数据库SQL语句。
- **ResultSet (java.sql.ResultSet)**
  - SQL语句中心完后，返回的数据结果集（包括行、列）。

## 13.2.2 JDBC工作机制

- 任何一个JDBC应用程序，都需要四个步骤：
  - 加载JDBC驱动程序
  - 建立与数据库的连接
  - 进行数据库操作
  - 关闭相关连接



# 加载JDBC驱动程序

- 在应用程序中，有三种方法可以加载驱动程序：

- 利用System类的静态方法setProperty()

**System.setProperty(**

**“jdbc.drivers”, “sun.jdbc.odbc.JdbcOdbcDriver”);**

- 利用Class类的静态方法forName()

**Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);**

**Class.forName(“oracle.jdbc.driver.OracleDriver”);**

- 直接创建一个驱动程序对象

**new sun.jdbc.odbc.JdbcOdbcDriver();**

# 建立与数据库的连接

- 利用DriverManager类的静态方法getConnection()来获得与特定数据库的连接实例（Connection实例）。

```
Connection conn =  
    DriverManager.getConnection(source);
```

```
Connection conn =  
    DriverManager.getConnection(source, user,  
    pass);
```

- 这三个参数都是String类型的，使用不同的驱动程序与不同的数据库建立连接时，source的内容是不同的，但其格式是一致的，都包括三个部分：

*jdbc:driverType:dataSource*

- 对于JDBC-ODBC Bridge，driverType为odbc，dataSource则为ODBC数据源：“jdbc:odbc:myDSN”。
- 对于其他类型的驱动程序，根据数据库系统的不同driverType和dataSource有不同的格式和内容。

# 进行数据库操作1

- 每执行一条SQL语句，都需要利用Connection实例的createStatement()方法来创建一个Statement实例。Statement的常用方法包括：
  - 执行SQL INSERT, UPDATE 或 DELETE 等语句
    - int executeUpdate(String sql)
  - 执行SQL SELECT语句
    - ResultSet executeQuery(String sql)
  - 执行一个可能返回多个结果的SQL语句
    - boolean execute(String sql) (与其他方法结合起来来获得结果)
- Statement 中还有其它方法执行SQL语句。

# 进行数据库操作2

- 通过ResultSet来获得查询结果：
  - ResultSet实例最初定位在结果集的第一行（记录）
  - ResultSet提供了一些在结果集中定位的方法，如next()等。
  - ResultSet提供了一些方法来获得当前行中的不同字段的值，getXXX()。
- ResultSet中还提供了有关方法，来修改结果集，并提交到数据库中去。



# *ResultSet*常用getXXX方法

返回值类型	方法名称
boolean	getBoolean()
byte	getByte()
byte[]	getBytes()
java.sql.Date	getDate()
double	getDouble()
float	getFloat()
int	getInt()
long	getLong()
Object	getObject()
short	getShort()
java.lang.String	getString()
java.sql.Time	getTime()

- 参数: **int colIndex** 或 **String colName**




# 关闭相关连接

---

- 依次关闭ResultSet、Statement、Connection对象，使用close()方法。

# JAVA程序设计与 实训教程



梁旭

淄博师专.信息科学系

email: [liangxu66@126.com](mailto:liangxu66@126.com)



# 几个Java编程技巧

---

- 1 Java所支持的.wav文件的转换.
- 2 在没有JCreator情况下,Java的编译和运行.
- 3 把Java程序打包成可执行文件.
- 4 Java图形用户界面绘图技巧.



# 1 Java所支持的.wav文件的转换

- Java支持的声音文件格式主要有：
  - .au; .aif; .mid; .midi; .rfm; .wav; 等。
- ImTOOAudioMaker软件的使用。
- 把.mp3转换为.wav。
- 同理,图片文件的转换也是如此,可用“画图板”转换.

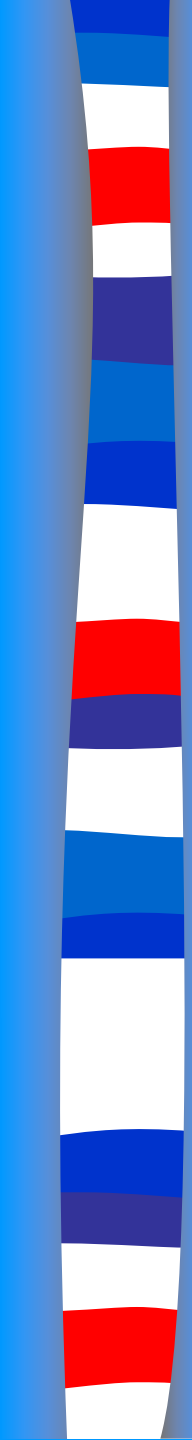
# 在没有JCreator情况下编译和运行

- 1.命令行界面；（先用记事本写好代码）
- 一般是在C:\JDK\*\*\*\BIN目录下。
- 2.java程序编译命令字；
- `javac ***.java <回车>`
- 3.java程序解释执行命令字。
- `Java *** <回车>`

# 把Java程序打包成可执行文件

- 编译好的java项目，在WINDOWS下可以这样打包与执行：
- 1、工程目录设置为 `$\proj`
- 2、代码文件放到`$\proj\filename`(代码所在文件夹,代码文件.java中要有 `package **`语句)。
- 3、用记事本创建并设置manifest.mf文件,放到目录: `$\proj\manifest.mf`
  - 内容如下: `Main-Class: filename.Frame1`<回车>
  - 格式为:`Main-Class: 可执行主类全名(包含包名)`
- 4、创建并设置批处理文件(用记事本)，放到目录: `$\proj\play.bat`先编译，后打包，再运行。  
内容如下：  

```
@echo off
javac filename \*.java
jar cvfm filename.jar manifest.mf filename
java -jar filename.jar
@echo on
```
- 5、把得到的filename.jar拷贝到任何地方都可以双击运行了。



# Java图形用户界面绘图技巧

- 常用两种实现方法:
- 1.直接在JPanel上的paintComponent方法中绘图.
- 2.用带有所绘图形的Jpanel放到当前Jpanel中实现绘图.