

1.1 Java 概述

Java 语言特征

1. 简单性

Java 去掉指针，取消多重继承和运算符重载。

Java 设立自动内存分配与回收机制。

2. 面向对象

JAVA 对象有模块化性质和信息隐藏能力，满足面向对象的封装要求；

支持继承；

通过抽象类与接口支持多态。

3. 分布式

Java 是面向网络的语言。通过它提供的类库可以处理 TCP/IP 协议,用户可以通过 URL 地址在网络上很方便地访问其它对象，Java 的网络编程就犹如从文件中发送和接收数据般简单。

4. 鲁棒性

通过集成的面向对象的异常处理机制，在编译时，Java 提示出可能出现但未被处理的异常。它提供自动垃圾收集来进行内存管理，防止程序员在管理内存时容易产生的错误。

5. 安全性

Java 不支持指针，一切对内存的访问都必须通过对象的实例变量来实现。

JVM 采用的是“沙箱”运行模式，即把 java 程序的代码和数据都限制在一定内存空间里执行，不允许程序访问该内存空间外的内存。

6. 体系结构中立(平台无关性)

只要安装了 Java 虚拟机(Java Virtual Machine ,JVM),编写的程序就可以在不同的平台上运行。

7. 可移植性

与平台无关的特性使 Java 程序可以方便地被移植到网络上的不同机器，而不必重新编译。

Java 的类库中也实现了与不同平台的接口，使这些类库可以移植。

Java 编译器是由 Java 语言实现的，Java 运行时系统由标准 C 实现，这使得 Java 系统本身也具有可移植性。

8. 解释执行

Java 程序被编译成 JVM 字节码(bytecode)。字节码不依赖于机器硬件配置，并可运行于任意安装了 Java 解释器的机器上，其中 Java 解释器是 Java 虚拟机的重要组成部分。Java 解释器直接对 Java 字节码进行解释执行。

9. 高性能

10. 多线程

多线程编程集成于 Java 环境中，而很多语言都是需要调用操作系统级别的特定过程才能支持多线程。

11. 动态性

在类库中可以自由地加入新的方法和实例变量而不会影响用户程序的执行。

Java 通过接口来支持多重继承，使之比严格的类继承具有更灵活的方式和扩展性。

java 与 c, c++的区别

(1)Java 语言中不允许在类之外定义全局变量，而只能通过类中定义静态变量来实现；

- (2)Java 语言中没有 goto 语句;
- (3)Java 语言中没有指针型变量;
- (4)内存管理实现了自动化, 去掉了 C 语言中的 malloc()、free();
- (5)Java 语言对于不同的数据类型定义统一的规格, 保证了平台无关性;
- (6)Java 语言中不允许像 C 和 C++中那样任意进行类型转换;
- (7)Java 语言中无头文件;
- (8)Java 语言中无结构体和联合;
- (9)Java 语言中无预处理和宏定义。

1.2 Java 程序与运行机制

The diagram shows a Java program snippet with several annotations in Chinese boxes:

- 注释行** (Comment line) points to the first two lines of the code.
- 主类** (Main class) points to the `public class SimpleApp` line.
- 主方法 程序入口** (Main method, Program entry) points to the `public static void main` line.
- 应用类** (Application class) points to the `main` method block.
- 结束主方法** (End main method) points to the closing brace of the `main` method.
- 结束主类** (End main class) points to the closing brace of the `SimpleApp` class.

Below the code snippet, the compilation and execution commands are shown:

```
>javac SimpleApp.java
>java SimpleApp
```

The output of the program is displayed in yellow text on a black background:

```
this s a simple program!
```

38

package 语句;

//该部分至多只有一句, 必须放在源程序的第一句

import 语句;

/*该部分可以有若干 import 语句或者没有, 必须放在所有的类定义之前*/

public class Definition;

//公共类定义部分, 至多只有一个公共类的定义

//java 语言规定该 java 源程序的文件名必须与该公共类名完全一致

class Definition;

//类定义部分, 可以有 0 个或者多个类定义

interface Definition;

//接口定义部分, 可以有 0 个或者多个接口定义

Java Application 程序的结构

一个程序可以分成若干个文件，一个文件中可以含有若干个类，每个类中包含若干个方法和变量，每个方法中包含若干执行语句，还可以包含若干变量。

当源程序被存储时，如果文件中含有主类，按 Java 语言的规定必须以主类名作文件基本名，以 java 为扩展名。Java 字节码文件的基本名与源程序文件基本名相同，以 class 为扩展名。

```
public class SimpleApp1
{
    public static void main(String args[])
    {
        SimpleApp1 simpleapp = new SimpleApp1();
        simpleapp.sayASentence();
        simpleapp.sayAnotherSentence();
    }
    public void sayASentence()
    {
        System.out.println("this is a simple program!");
    }
    public void sayAnotherSentence()
    {
        System.out.println("it is very easy to learn!");
    }
}
```

Java 程序运行的五个阶段：

编辑、编译、加载、验证、执行

2.1 标识符、关键字和数据类型

1. 字符集和标识符

Java 语言采用 Unicode 字符集，Java 语言的标识符是以字母、下划线_或\$符号开头的后面含有字母、下划线_、\$符号和数字的字符串，标识符的长度没有限制，但 Java 系统最多可以识别前 255 个字符；区分大小写。

Java 标识符使用惯例：

类和接口——类名和接口名通常用名词，且每个单词的首字母大写：SimpleApp

方法——方法名用动词开头的单词序列，首单词全部小写，后面的每个单词首字母大写；processResult

常量——常量名全部用大写字母；PI

变量——所有的对象实例名和全局变量名都使用首单词全部小写，后面的每个单词首字母大写的格式；outputResult

2. 关键字

在 Java 中保留但已经不再使用的 2 个关键字：const，goto

2. 数据类型

Java



8

数据类型	数据长度	取值范围
byte	8位	$-2^7 \sim 2^7 - 1$
short	16位	$-2^{15} \sim 2^{15} - 1$
int	32位	$-2^{31} \sim 2^{31} - 1$
long	64位	$-2^{63} \sim 2^{63} - 1$

数据类型	数据长度
float	32位
double	64位

转义字符

\b	退格	backspace	\u0008
\t	水平制表	tab	\u0009
\n	换行	linefeed	\u000a
\r	回车	carriage return	\u000d
\"	双引号	double quote	\u0022
\'	单引号	single quote	\u0027
\\	反斜线	backslash	\u005c

(4) 逻辑类型：boolean

两种取值：“true”和“false”。

注意：在Java语言中，逻辑类型与整数类型不能进行直接转换。这与C和C++语言有明显的不同。

3.常量

用“final”修饰的Java语言标识符为符号常量，其值在赋值之后将不能再作改动。

```
static final PI=3.14159;
```

4.变量

任何变量、数组、对象实例在使用之前必须经过声明、创建和初始化，否则将无法完成任何操作。

变量的声明是要把代表变量的标识符作出说明

变量的创建是为其分配存储空间

变量的初始化：

方法体外声明的变量，系统可自动赋初值；（类成员变量）

方法体内声明的变量，需由语句赋初值。（局部变量）

5.操作符

Java语言取消了结构体成员操作符“->”，指针操作符“*”和“&”，长度操作符“sizeof”。

若想对比两个对象的实际内容是否相同，必须使用所有对象都适用的特殊方法equals()。但

这个方法不适用于“基本类型”，基本类型直接使用==和!=即可

逻辑操作符

在 C 和 C++ 中，用整型数据替代逻辑型数据；而 JAVA 中整型数据与逻辑型数据不存在互换关系。故逻辑操作符：`!` `&` `|` `^` `&&` `||` 仅适用于 `bool` 型变量

需要提醒，“`&&`”和“`||`”在计算逻辑值时，如果仅靠左运算数即可判定运算结果时，右运算数的值将不被计算，所以它们也被称为“条件与”和“条件或”。而“`&`”和“`|`”在计算逻辑值时，总是将两个运算数的值都计算出来之后再作逻辑运算，这是它们的差别所在。

字符串操作符

操作符“`+`”可以实现两个或多个字符串的连接，也可实现字符串与其他类对象的连接，在连接时，其他类对象会被转换成字符串。

类型转换操作符

在一个表达式中可能有不同类型的数据进行混合运算，这是允许的，但在运算时，Java 将不同类型的数据转换成相同类型，再进行运算。

不同类型数据之间的转换规则分自动类型转换和强制类型转换。

自动类型转换（低→高）

强制类型转换（高→低）`(type) expression`

2.3 表达式与语句

Java 中的每条语句均以分号（`;`）结束。

语句块是以左大括号和右大括号为边界的语句集合。块在语法上等价于单个语句，能用在单个语句可以出现的任何地方。

分支结构：

```
switch(expression){
    case  const1:statements（多条语句）;break;
    case  const2:statements;break;
    .....
    default:statements;
}
```

其中 `expression` 只能是整数类型或字符型或 `enum` 类型，不能是浮点类型。

带 `label` 的 `break` 语句的格式为

`break label`

`label` 为一个标识符，标定一条语句，带 `label` 的 `break` 语句的用法的作用是跳出 `label` 所标定的块。

带 `label` 的 `continue` 语句的格式为


`continue label`

这里的 `label` 仍然是一个标定语句的标识符，带 `label` 的 `continue` 语句的作用是结束标号指定的循环中的一个周期的剩余部分，开始下一个循环。


```

import javax.swing.JOptionPane;
public class ContinueAndBreakWithLabel
{
    public static void main(String[] args)
    {
        String output = "";
        L01:{
            for(int i=1;i<=10;i++)    //计划进行十次外部循环
            {
                output = "\n第";
                output +=i;
                output += "次外部循环开始 ";
                for(int j=1;j<=8;j++){ //计划进行八次内部循环
                    if(i==4&j==6)
                        break L01;
                    output += "## ";
                } //结束内部循环
                output += "第";
                output +=i;
                output += "次外部循环结束";
            } //结束外部循环
        } //结束L01
        output += "\n结束break语句的输出工作。";
    }
}

```



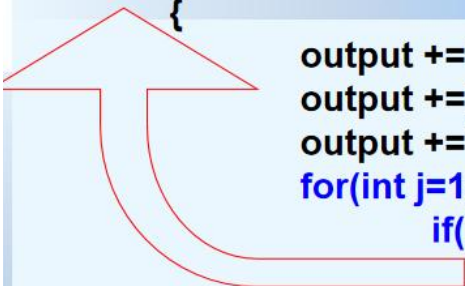
61

```

L02:for(int i=1;i<=5;i++)    //计划进行五次外部循环
{
    output += "\n第";
    output +=i;
    output += "次外部循环开始 ";
    for(int j=1;j<=6;j++){ //计划进行六次内部循环
        if(i==2||i+j==7)
            continue L02;
        output += "$$ ";
    } //结束内部循环
    output += "第";
    output +=i;
    output += "次外部循环结束";
} //结束外部循环
output += "\n结束continue语句的输出工作。";

JOptionPane.showMessageDialog(null,output);
System.exit(0);
}
}

```



3.1 Java 类

类是若干对象所具有的共性。在类中定义所有对象所共有的内容,将其实例化即生成了对象。

```
class ATypeName { /*类主体置于这里*/ }
```

```
    ATypeName a = new ATypeName();
```

可以说,类是对象的模板,是抽象的模型;对象是类的实例化。

一个对象也可能包含了另一个对象,而另一个对象里则包含了我们想修改的数据。

在类体中声明的变量成员和方法成员的作用域是整个类。

变量成员

变量成员的基本声明格式为

```
[public|protected|private][static][final][transient][volatile] type variableName;
```

其名字在类成员中必须是唯一的,不能与其他变量同名,但可以和类内的某个方法使用同一个名字。

变量成员的作用域为整个类。

变量成员的访问方式: `ATypeName a = new ATypeName(); a.变量成员名=18;`

方法成员

方法成员的基本声明格式为

```
[public|protected|private][static] returnType methodName([paramList]){methodbody}
```

其中 `returnType` 可以是基本数据类型,也可以是引用数据类型,或者当无返回值时为 `void`。

参数列表中的参数也可以是各种数据类型。

方法体是方法的实现,其中可以包含局部变量的声明和所有合法的 Java 语句。事实上这是 Java 语言程序的实质部分,程序中所有的执行动作都是在这里实现的。

在 Java 中,方法只能作为类的一部分来创建, [方法只有通过对象才能调用](#),且这个对象必须能够执行这个方法调用。

```
int x = a.f();
```

static 属性

通常来说,当创建类时,就是在描述这个类的对象的属性与行为。除非用 `new` 创建这个类的对象,否则并未获得任何对象。执行 `new` 来创建对象时,数据存储空间才被分配,其方法才供外界调用。

static 变量在类第一次被引用的时候分配内存,然后一直到程序退出才会被回收

类变量与实例变量

用 `static` 声明的变量成员为类变量,不用 `static` 声明的变量成员为实例变量。类变量由多个对象实例共享一个内存区,共享同一个值。实例变量各自有各自的内存区,对于不同的对象实例而言,其对应的变量成员有不同的值。

实例变量必须在生成对象实例后通过对象实例名来访问;类变量既可以通过对象实例名访问,也可以通过类名直接访问。

类方法与实例方法

用 `static` 声明的方法成员为类方法,不用 `static` 声明的方法成员为实例方法。

在类方法中不能使用 `this` 或 `super`。

可以不生成类对象而直接通过类名访问类变量和类方法。这就是为什么将其称为“类”成员。

[类方法不能访问普通的实例变量](#),而仅能访问类变量。实例方法可以访问类变量和实例变量。

访问控制符

可以对所有的成员分别设定访问权限，以限定其他对象对它的访问，Java 语言设定以下四种访问权限：

public

protected

默认（包访问权限）

private

public: 使用关键字 **public**，就意味着 **public** 之后紧跟着的成员，对每个人都是可用的，可以被任何其他类访问。

protected: **protected** 控制符修饰的成员能在它自己的类中和继承它的子类中被访问，也可以被同一包中的类访问。

默认: 也叫包访问权限，友元 **friendly**。如果没有使用可见性的控制符，那么默认为类、方法和变量是可以被同一个包中任何一个类访问。

private: **private** 控制符修饰的成员只能在它自己的类中被访问。

	同一个类中	同一个包中	不同包中的子类	不同包中的非子类
public	√	√	√	√
protected	√	√	√	
默认	√	√		
private	√			

构造方法

如果在定义类时没有定义构造方法，则 Java 系统会自动提供默认的无参数构造方法。所以，任何类都有构造方法。

构造方法可以重载。

构造方法只能用 **new** 关键字调用。

若构造方法体定义为空，则自动调动其父类的构造方法。

```
class Rock{
    Rock(){
        System.out.print("Rock ");
    }
}
```

3.2 Java 对象

对象的创建

Java 中一切都是对象，类是创建对象的模板。

对象是类在程序中的实例化。对象实例在程序中包括生成、使用和清除三个阶段。

对象实例的生成包括声明、实例化和初始化三个步骤

```
type  objectName = new  type([paramList]);
```

这里的 `type objectName` 声明了一个对象实例的引用，`new` 运算符为其分配内存空间，完成了实例化，`type` 构造方法执行初始化工作。

对象初始化

若类的变量成员是基本类型，编辑器会在没有显式初始化的时候给定默认初始值。

最先初始化 `statics` 成员，静态成员只在第一次访问时初始化一次。

在类的内部，变量定义的先后顺序决定了初始化的顺序。即使变量定义散布于方法定义之间，它们仍旧会在任何方法被调用之前得到初始化。

创建对象时，[计算机先初始化它的变量成员，然后再调用构造函数](#)

内存分配

寄存器：这是最快的存储区，因为它位于不同于其他存储区域的地方--处理器内部。我们在程序中无法控制。

栈(Stack)：驻留于 RAM（随机访问存储器）区域，但通过堆栈指针可以从处理器获得直接支持。存放基本类型的数据和对象的引用。

堆(Heap)：一种通用的内存池（也位于 RAM 区），用于存放所有的 Java 对象，即用 `new` 产生的数据。

静态域：存放在类中用 `static` 定义的静态成员。

常量池：常量值通常直接存放在代码内部，这样做是安全的，因为它们永远不会被改变。

非 RAM 存储：数据完全存活于程序之外。硬盘等。

```
String str=new String(“abc”);
```

栈 堆

垃圾回收机制

垃圾回收机制 GC(Garbage Collection)是 Java 语言的核心技术之一。

不需要程序员手动释放对象占用的内存资源。

GC 通过确定对象是否被活动对象引用来确定是否收集该对象。

3.3 包：库单元

命名空间

Java 引入了包的机制，以提供类的多重命名空间，同时也负责类名空间管理。把因不同目的和不同工作而开发的类放在不同的包中，即使在出现相同的类名的时候，也可以很好地管理。

Package 语句

package 语句放在 Java 源程序文件的第一行，指明该文件中定义的类存放到哪个包中。程序中可以有 package 语句，此时类将存放到默认包中。

```
package pkg1.[pkg2.[pkg3.···]];
```

```
package access.mypackage;
```

package 语句最多只能有一条。

```
package access.mypackage;
```

```
public class MyClass {  
    //...  
}
```

- MyClass 是命名在 access.mypackage 包中的，要想使用这个类，需要用 import 语句来导入该类。

Import 语句

import 语句的作用是引入所需的类，以供在程序中使用类库中现有的类。

```
import pkg1[.pkg2···].classname*;    引入 pkg1[.pkg2···] 中的 classname 类
```

import 语句在程序中放在 package 语句之后，类定义之前。一个 Java 源程序中可以有 multiple import 语句。

例如：

要用 java.util 中的其他类

```
import java.util.*    引入 java.util 里的所有类
```

3.4 Java 标准类库

java.lang 包：提供了 Java 程序设计中最基础的类。包含基本数据类型、基本数学函数、字符串处理、线程、异常处理类等。只有这个包不需要引入就能用。

java.lang.Object

Object 类是所有 Java 类的基类，它处于 Java 开发环境的类层次树的根部。

如果一个类在定义的时候没有包含 extends 关键字，编译器会将其作为 Object 类的直接子类。

Object 类定义了所有对象的最基本的一些状态和行为，它提供的方法会被 Java 中的每个类所继承。

可以使用类型为 Object 的变量引用任意类型的对象。

```
Object obj = new MyObject();
```

```
MyObject x = (MyObject)obj;
```

java.lang.System

System 不能被实例化，所有的方法都可以直接引用。

System 类提供了标准输入（in）输出（out）和错误（error）流。

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

```

        System.out.println("Hello System.out!");
        System.err.println("Hello System.err!");    System.err 打印出来的是红色的字体
    }

```

System 还提供了系统属性的获取和设置功能。

currentTimeMillis()方法来获取当前时间；

exit()方法使程序退出系统；

java.lang.Math

包含完成基本数学函数所需的方法。

静态方法，不用声明具体的实例。

主要包括三角函数方法、指数方法、服务方法、随机数方法。

Math 中还声明了两个 double 型的常量，PI 和 E 可以在任何程序中用 Math.PI 和 Math.E。

java.lang.String

类 String 用来处理不变的字符串常量。

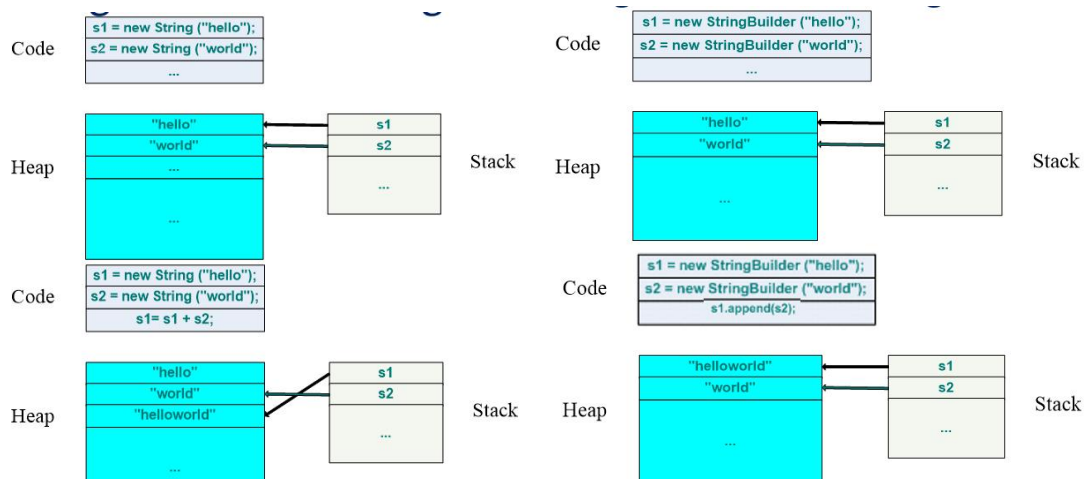
当使用“CPU = ” + CPU 语句时，我们将得到一个 String 对象，而 CPU 是一个自己定义对象。因此，编译器就会调用 CPU 所属 CentralProcessingUnit 类中的 toString()方法。如果我们这种使用方法，就必须在定义类时重写 toString()方法。

否则，在将对象输出为字符时，使用 object 类中的 toString()方法，输出的是该对象的内存地址。

Java.lang.StringBuilder / Java.lang.StringBuffer

比 String 类更灵活，可以给一个 StringBuilder 或 StringBuffer 中删除、插入或追加新的内容。

而 String 对象一旦创建，它的值就确定了。优先采用 StringBuilder 类



java.util 包: 包括许多具有特定功能的类，提供了一些实用的方法和数据结构。如日期(Data)类、日历(Calendar)类来产生和获取日期及时间，提供随机数(Random)类产生各种类型的随机数，还提供了堆栈(Stack)、向量(Vector)、位集合(Bitset)以及哈希表(Hashtable)等类来表示相应的数据结构。

java.io 包: 主要包含与输入输出相关的类，这些类提供了对不同的输入和输出设备读写数据的支持。

javax.swing 包: 提供了创建图形用户界面元素的类。该包中包含定义窗口、对话框、按钮、复选框、列表、菜单、滚动条及文本域类。

java.net 包：包含与网络操作相关的类，如 URL、InetAddress 和 Socket 等。

4.1 类的复用

可重用性可以从类定义的角度来体现：

类的概念封装了类,类的对象也就起到了代码复用的效果。

还可以从类间关系的角度来考虑可重用性：

组合 (横向关系): has a

继承 (纵向关系): is a

组合

合成：A 类中包含 B 类的一个引用 b，当 A 类的一个对象消亡时，b 这个引用所指向的对象也同时消亡。

聚合：反之 b 所指向的对象还会有另外的引用指向它。如人与电脑。

继承

被继承的类叫超类（superclass）、父类、基类。通过继承能够创建一个通用类，它可以被更具体的类继承。

继承超类的类叫子类（subclass）、派生类、扩展类。子类是超类的具体化，是超类的一个特定用途的类。

声明一个继承超类的类的通常形式如下：

```
class subclass-name extends superclass-name
{
    // body of class
}
```

派生出来的子类包括超类的所有成员，除了超类中被声明成 `private` 的成员；可以定义属于自己的新方法成员，新变量成员。

一个子类仅能继承一个父类

构造方法的执行按先超类后子类的层次顺序，所以超类会在子类访问它之前得到正确的初始化。如果调用超类中含有参数的构造方法，必须明确地编写对超类的调用代码，用 `super(参数表)`调用。

重写

有时子类并不想原封不动地继承父类的方法，而是需要作一定的修改，子类修改超类已有的方法叫重写，但参数列表和返回类型必须相同，被子类重写的方法不能拥有比超类方法更加严格的访问权限。

超类的访问权限修饰符的限制一定要大于被子类重写方法的访问权限修饰符，`private` 权限最小。如果某一个方法在超类中的访问权限是 `private`，那么就不能在子类中对其进行重写。

重载(Overloading)

在一个类中，多个方法的方法名相同，但是参数列表不同。可以有不同的访问修饰符、不同的返回类型、不同的异常抛出。在使用重载时只能通过不同的参数列表辨别。

区别点	重写（覆写）	重载
英文	Overriding	Overloading
定义	方法名称、参数类型、返回值类型全部相同	方法名称相同，参数的类型或个数不同
	被重写的方法不能拥有更严格的权限	对权限没有要求
范围	发生在继承类中	发生在同一个类中

abstract

Abstract 修饰符表示所修饰的成分没有完全实现，还不能实例化。

如果在类的方法声明中使用 abstract 修饰符，表明该方法是一个抽象方法，它需要在子类实现。

如果一个类包含抽象方法，则这个类也是抽象类，必须使用 abstract 修饰符，并且不能实例化。抽象类里面可以有非抽象方法，即已实现的方法

例：

因为包含一个抽象方法 test，类 a1 必须被声明为抽象类。它的子类 a2 继承了抽象方法 test，但没有实现它，所以它也必须声明为抽象类。

构造方法、static 方法，final 方法不能使用 abstract 修饰符;接口默认为 abstract。

final

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。故而将方法声明为 final，是为方法“上锁”，防止他被继承

用 final 修饰的成员变量表示常量，值一旦给定就无法改变。final 修饰的变量有三种：静态变量、实例变量和局部变量，分别表示三种类型的常量。在默认情况下，所有的成员方法和实例变量都可以被覆盖。如果你希望你的变量或成员方法不再被子类覆盖，可以把它们声明为 final。

this

this 的两种用法

this(有参数/无参数)： 用于调用本类相应的构造方法

this.后跟变量或方法： 使用本类的变量成员或方法成员

super

super 的两种用法:

super(有参数/无参数) : 用于调用父类相应的构造方法

super. 后跟变量或方法(父类中指明为 public 或 protected 的): 使用超类的变量或方法
多用于在子类中调用被本子类重写方法后覆盖的父类的原方法或父类中的被本子类覆盖的原变量成员

4.2 多态

1.向上转型

超类类型的引用指向一个子类的对象。

超类类型的引用可以调用超类中定义的所有变量和方法,无法使用子类中定义而超类中没有定义的方法。如果子类中重写了超类中的方法,那么超类类型的引用将会调用子类中的这个方法,这就是动态绑定。

```
Father e=new Father();
```

```
Son m=new Son();
```

```
e=m;
```

或

```
Father sample = new Son();
```

上转型对象只能访问父类中声明的成员变量和成员方法,不可以访问子类新增的特有的成员变量和成员方法。

如果子类重写了父类的方法,则上转型对象调用该方法时,必定是调用重写的方法。

如果子类重新定义了父类的同名变量,则上转型对象应用该变量时是父类中的定义的变量,而不是子类中定义的变量

2.多态性

当超类变量引用子类对象时,由被引用对象的类型而不是引用变量的类型,来决定调用谁的成员方法。

每一个实例对象都自带一个虚拟函数表(virtualtable),这个表中存储的是指向虚函数的指针,实例对象通过这个表来调用虚函数,以实现多态。

3.运行时绑定(动态绑定)

将一个方法调用和一个方法主体连接到一起称为绑定(Binding),运行时绑定也叫动态绑定。

如果在程序运行之前进行绑定(由编译器和链接程序完成),称为早期绑定,或静态绑定、编译时绑定。

如果在程序运行期间进行绑定,称为后期绑定,或动态绑定、运行时绑定。

Java 的方法调用过程

编译器查看对象变量的声明类型和方法名

通过声明类型找到虚拟函数表

编译器查看调用方法时提供的权限类型

如果方法是 private、static、final 或者构造函数,编译器就可以确定调用那个方法,这是静态绑定;

如果不是上述情况，就要使用运行时（动态）绑定。在程序运行时，采用动态绑定意味着：虚拟机将调用对象实际类型所限定的方法。

运行时（动态）绑定的过程

虚拟机提取对象的实际类型的虚拟函数表

虚拟机搜索匹配的方法声明

最后调用该方法。

注意，这里说的是对象的实际类型。即在多态的情况下，虚拟机可以找到所运行对象的真正类型。

向上转型时，只有方法会动态绑定，变量仍然是静态绑定，即只调用父类里的变量。（所以本质上向上转型明显就不是为了得到子类，而是为了修改父类得到一个不一样的父类嘛）

若想要试图调用子类的成员变量，最简单的办法是将该成员变量封装成方法形式，通过动态绑定方法来实现。

```
class Father3 {
    protected String name="父亲属性";
    public String getName() {return name; }
    public void method() {
        System.out.println("超类方法，对象类型：" + this.getClass());
    }
}

public class Son3 extends Father3 {
    protected String name="儿子属性";
    public String getName() {return name; }
    public void method() {
        System.out.println("子类方法，对象类型：" + this.getClass());
    }
}

public static void main(String[] args) {
    Father3 sample = new Son3(); //向上转型
    System.out.println("调用的成员："+sample.getName());
}
```

4.instanceof 对象类型判定运算符

对象类型判定运算符 “instanceof”，其使用格式为：

instance instanceof classname

其中，instance 是某个对象实例的引用标识符，classname 是某个已经定义的类的类名，其运算结果为逻辑类型。表达式的含义为测试 instance 是否为 classname 类的对象实例，答案为是则返回逻辑值 true，答案为否则返回逻辑值 false。

5.多态的实现

继承实现的多态

方法的重写:派生类（子类）将继承超类（父类）所有的变量和方法，可根据需要来重新定义超类的变量和方法，甚至增加或者修改部分内容，以提供不同形式的实现。

方法的重载:同一类中定义同名方法。

抽象类实现的多态

使用 abstract 类型修饰符的抽象方法或类，为其它子孙类用抽象机制实现多态性提供统一的

界面。

接口实现的多态

“interface”（接口）是一个“纯”抽象类。

接口中只有常量和抽象方法的声明，实现接口时没有成员变量名字冲突问题，也没有对超类方法的重写问题，也不存在多重继承的问题，比一般类的多态继承简单。

4.3 接口

子类不能同时继承多个父类，Java 通过一个类可以继承多个接口实现了多重继承的功能。

1. 接口的定义

接口体由常量定义和方法定义两部分组成，基本格式如下：

[public] interface 接口名

{

 [public] [static] [final] 常量; #成员变量默认都是 public,static,final 类型

 [public] [abstract] 方法; #方法默认都是 public,abstract 类型

}

例：

public interface Calculate {

 final float PI=3.14159f; //定义用于表示圆周率的常量 PI

 float getArea(float r); //定义一个用于计算面积的方法

 float getCircumference(float r); //定义一个用于计算周长的方法

}

Ps:

直接写函数体，则 float getArea(float r){。。。}花括号后无需跟 ; 号。而方法定义则不需要花括号，但是需要 ;

接口可以继承，并且支持多重继承，即可以在多个父接口的基础上定义子接口。子接口继承所有的父接口的成员，其成员集合是各个父接口成员集合的并集，并且还可以在此基础上声明新的成员。

基本格式如下：

[public] interface 接口名 extends 父接口名列表

{

 [public] [static] [final] 常量;

 [public] [abstract] 方法;

}

接口可以在类中实现，类实现接口的过程就是给出接口的方法实现的过程，所要实现的方法是接口列表中所有接口中的方法。

基本格式如下：

[public] class 类名 implements 接口名列表

{

 变量成员定义;

 方法成员定义和方法实现{};

}

必须全部实现接口中的方法，如果不想实现可以{}内为空，但必须全部物理上实现

2. Java 接口的特征

成员变量必须被显示初始化,即接口中的成员变量为常量;方法没有方法体,在接口中不能被实例化

接口中没有构造方法,不能被直接实例化

一个接口不能实现(implements)另一个接口,但它可以继承(extends)多个其它的接口

一个类只能继承一个直接的超类,但可以实现多个接口,间接地实现了多重继承

`public class A extends B implements C, D {...}` //B 为 class,C,D 为 interface

不允许创建接口的实例(实例化),但允许定义接口类型的引用变量,该引用变量引用实现了这个接口的类的实例

使用时可以直接用 接口类名.常量名 这样的形式来引用其中的常量。

我们还可以用下面这种方法来创建初始值不确定的常量。

```
public interface RandomColor {  
    int red = Math.random() * 255;  
    int green = Math.random() * 255;  
    int blue = Math.random() * 255;  
}
```

其中 red、green 和 blue 的值会在第一次被访问时建立,然后保持不变。

3. 接口与抽象类的对比

抽象类是部分抽象的,接口是纯抽象的

都不能被实例化

都包含抽象方法

抽象类可以提供某些方法的部分实现,而接口不可以。可以向抽象类里加入一个新的具体方法,所有的子类都自动得到这个方法;但接口里加入一个新方法,必需手动给每个实现了该接口的类加上该方法的实现。

抽象类不能被多重继承,接口可以

4.4 内部类

1.内部类的定义

内部类,一个类的定义放在另一个类的内部,这个放在内部的类就叫做内部类。

它有两种形式:

嵌套类(nested class)(静态内部类):静态的,使用很少;

内部类(inner class):非静态,较重要。

内部类可以像方法一样被修饰为 public, default, protected 和 private,也可以是静态 static 的,即嵌套类。



创建内部类对象:

outerObject=new outerClass(Constructor Parameters);

outerClass.innerClass innerObject=

outerObject.new innerClass(Constructor Parameters);

需要先创建起相应的外部类对象，再创建内部类对象。

内部类的作用:

内部类允许有 `private` 与 `protected` 权限。内部类作用是更小层次的封装，把一个类隐藏在另一个类的内部，只让它的外部类看得到它，能更方便地在内部类中访问外部类的私有成员。由于内部类对外部类的所有内容都是可访问的，我们就可以考虑使用内部类来实现接口，可以避免接口和同一个类中两种同名方法的调用。Java 中的内部类与接口联合应用，更好地实现多重继承。

2.其他类型内部类

静态内部类:

创建静态内部类（嵌套类）的对象时，不需要创建其外部类的对象。

outerClass.innerClass innerObject=

outerClass.new innerClass(Constructor Parameters);

静态内部类不能访问外部类的非静态成员。

局部内部类:

Java 内部类也可以是局部的，它可以定义在一个方法甚至一个代码块之内。

访问局部 内部类必须有外部类对象;

局部内部类只能访问外部类的 `final` 类型局部变量。

局部内部类不能定义静态成员变量

```
// 如果内部类中有与外部类同名的变量，直接用变量名访问的是内部类的变量
System.out.println(s);
// 用this.变量名访问的也是内部类变量
System.out.println(this.s);
// 用外部类名.this.内部类变量名访问的是外部类变量
System.out.println(Outer1.this.s);
```

匿名内部类:

当你只需要创建一个类的对象而且用不上它的名字时，使用匿名内部类可以使代码看上去简

洁清楚。

```
new interfacename(){.....};
```

```
或 new superclassname(){.....};
```

例如：

```
public class Goods3 {  
    public Contents cont() {  
        return new Contents() {  
            private int i = 11;  
            public int value() {  
                return i;  
            }  
        };  
    }  
}
```

有一点需要注意的是，匿名内部类由于没有名字，所以它没有构造函数（但是如果这个匿名内部类继承了一个只含有带参数构造函数的超类，创建它的时候必须带上这些参数，并在实现的过程中使用 `super` 关键字调用相应的内容）。

3. 内部类与外部类

内部类生成的 `class` 文件

内部类的 `class` 文件名是由外部类的类名与内部类的类名用 “\$” 连起来形成的，如 `OutClassName$InnerClassName.class`

匿名内部类的类名为 `OutClassName$#.class`，其中的 “#” 为数字，从 1 开始，每生成一个匿名内部类，数字递增 1

内部类访问外部类

内部类对象与创造它的外部类对象存在着固有联系，一旦内部类对象创建，就可以在不需任何特殊条件的情况下访问外部类的所有成员。

使用关键字 `.this` 与 `.new`

我们要得到对外部类的引用，可以在内部类中使用 `.this` 关键字。


```
public class Outer {
    private int num;

    public Outer() {
    }

    public Outer(int num) {
        this.num = num;
    }

    private class Inner {
        public Outer getOuter() {
            return Outer.this;
        }

        public Outer newOuter() {
            return new Outer();
        }
    }

    public static void main(String[] args) {
        Outer test = new Outer(5);
        Outer.Inner inner = test.new Inner();
        Outer test2 = inner.getOuter();
        Outer test3 = inner.newOuter();
        System.out.println(test2.num);
        System.out.println(test3.num);
    }
}
```

5
0

我们要仔细注意this和new在这里的不同，内部类Inner内方法getOuter()使用.this得到是创建该内部类时使用的 外部类对象的引用，而newOuter()方法使用new创建了一个新的引用。

OutClass.InnerClass obj = outClassInstance.new InnerClass();

实用外部类对象调用内部类的构造函数，返回内部类对象（也就是说每个内部类对象必须依附于一个外部类对象而存在的）

再使用内部类对象调用内部类的方法

内部类的方法可以创建外部类对象

在本例中是：Outer.Inner inner = test.new Inner();

普遍的形式：

OutClass.InnerClass obj = outClassInstance.new InnerClass();

注意：可以是外部类对象.new，而不能是外部类.new

内部类与向上转型

将内部类向上转型为超类型，尤其是接口时，内部类就有了用武之地。

```

interface Shape {
    public void paint();
}

public class Painter {

    private class InnerShape implements Shape {
        public void paint() {
            System.out.println("painter paint() method");
        }
    }

    public Shape getShape() {
        return new InnerShape();
    }

    public static void main(String[] args) {
        Painter painter = new Painter();
        Shape shape = painter.getShape();
        shape.paint();
    }
}

```

painter paint() method

81

内部类的重载问题

子类和父类内涵同名内部类

```

class Egg2 {
    protected class Yolk {
        public Yolk() {
            System.out.println("Egg2.Yolk()");
        }
        public void f() {
            System.out.println("Egg2.Yolk.f()");
        }
    }
    private Yolk y = new Yolk();
    public Egg2() {
        System.out.println("New Egg2()");
    }
    public void insertYolk(Yolk yy) {
        y = yy;
    }
    public void g() {
        y.f();
    }
}

public class BigEgg2 extends Egg2 {
    public class Yolk extends Egg2.Yolk {
        public Yolk() {
            System.out.println("BigEgg2.Yolk()");
        }
        public void f() {
            System.out.println("BigEgg2.Yolk.f()");
        }
    }
    public BigEgg2() {
        insertYolk(new Yolk());
    }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
    }
}

```

内部类的构造函数

内部类的成员函数

成员变量

构造函数

外部类的函数

使用内部类型的成员变量调用内部类的函数

创建bigegg2类的对象，则先创建其父类egg2，而在创建其父类时，先创建其父类的成员变量，此时在父类的空间内，创建的yolk类型自然是父类的内部类yolk，而后调用其父类的构造函数。

父类对象创建好后，开始调用子类构造函数，进入子类的空间内，此时再new yolk则是创建子类的内部类yolk的对象，只是此时子类中的yolk又是父类中的yolk的子类，故而又要先创建父类中的yolk再继续子类的yolk，最后的y是bigeggs对象，故而调用f()必然是bigeggs的方法

83

```

class Egg {
    private Yolk y;

    protected class Yolk {
        public Yolk() {
            System.out.println("Egg.Yolk()");
        }
    }

    public Egg() {
        System.out.println("New Egg()");
        y = new Yolk();
    }
}

```

这种，成员变量 y 没有指定初始化的，就不会先于构造函数而调用 Yolk 构造函数

内部类的继承问题

创建外部类对象不会调用内部类的构造函数

如果内部类不是 static，有一个类要 extend 这个内部类，则这个类的构造函数中必有外部类的构造函数

例：inheritinner 做了 withinner 的内部类 inner 的子类，故而在创建 inheritinner 对象时，会先执行其父类 inner 的初始化

完整的 Java 语言 application 程序的基本结构

一、完整的 Java 语言程序文件的格式如下

```

package packageName;
//指定文件中定义的类所存的包，0 个或 1 个

import packageName.(className|*);
//指定引入的类，0 个、1 个或多个

public classDefinition
    //主类定义，0 个或 1 个
interfaceDefinition and classDefinition
    //接口或类定义，0 个、1 个或多个

其中，完整的类定义为
[public][abstract|final] class className [extends superclassName]
[implements interfaceNameList]
{
    [public|protected|private][static][final][transient][volatile]type variableName;
        //变量成员定义，0 个、1 个或多个
    [public|protected|private][static][final|abstract][native][synchronized]
    returnType methodName ([paramList]) [throws exceptionList] {
        statements
    }
        //方法成员定义，0 个、1 个或多个
}

```

二、有三个特别的类方法成员

1.构造方法：

```
className([paramList]){...}
```

2.main()方法:

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

3.finalize()方法:

```
protected void finalize() throws Throwable{...}
```

三、完整的接口定义为

```
[public]interface interfaceName [extends superInterfaceList]{  
    type constantName = Value;  
        //常量成员定义, 0 个、1 个或多个  
    returnType methodName([paramList]);  
        //方法成员声明, 0 个、1 个或多个  
}
```

5.1 图形化用户界面（GUI）概述

Swing 组件:

Swing 组件从 API 类使用上可分为三大类:

(1) 容器类 (又分顶层容器和中间容器)

字面意思就是用来包含其它组件的一个容器。例如: JFrame、JApplet、JDialog、JPanel。

(2) 组件类

都是 JComponent 类(抽象类)的子类。例如: JButton、JTextField, JTextArea, JComboBox, JList, JRadioButton, JMenu。

(3) 辅助类

是描述和绘制容器类和组件类属性和放置的, 如图形环境、颜色、字体、大小以及摆放位置等等, 例如: Graphics, Color, Font, FontMetrics, Dimension, LayoutManager。

5.2 常用组件

1. 窗口 JFrame

构造方法:

```
public JFrame()
```

```
public JFrame(String title)
```

成员方法:

setSize()窗口大小、setVisible()窗口是否可见、setLocation()窗口在输出设备上的位置 (左上为 0 点)、setDefaultCloseOperation()默认关闭窗口时对程序的操作、getContentPane()、setTitle()
public void setDefaultCloseOperation(int operation)中参数 operation 的 4 种取值:

DO_NOTHING_ON_CLOSE: 不执行任何操作; 要求程序在已注册的 WindowListener 对象的 windowClosing 方法中处理该操作。

HIDE_ON_CLOSE: 调用任意已注册的 WindowListener 对象后自动隐藏该窗体。

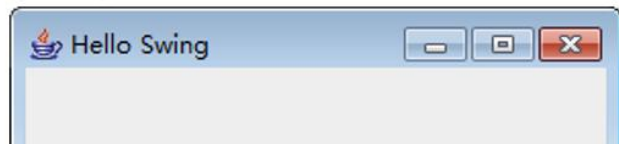
DISPOSE_ON_CLOSE: 调用任意已注册 WindowListener 的对象后自动隐藏并释放该窗体。

EXIT_ON_CLOSE: 使用 System exit 方法退出应用程序。仅在应用程序中使用。

```

import javax.swing.*;
public class HelloSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello Swing");
        //设置窗口标题
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //设置窗口关闭时动作（使用类名直接调用JFrame类的变量成员）
        frame.setSize(300, 100); //设置窗口的大小
        frame.setVisible(true); //设置窗口可见性
    }
}

```



2.按钮 JButton

构造方法:

`public JButton()`

`public JButton(String text)` (按钮上显示文字)

`public JButton(Icon icon)` (按钮上显示图片)

`public JButton(String text, Icon icon)`

成员方法:

`public void setText(String text)`

`public setIcon(Icon icon)`

`public void setBounds(int height, int width, int x, int y)` (按钮的大小和坐标)

`public void addActionListener(ActionListener l)` (设定动作监听器, l 是一个动作监听器, 用于监听按钮上的动作事件)

```

import javax.swing.*;
import java.awt.*;
public class Button1 extends JFrame {
    private JButton (两个按钮变量成员)
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    public Button1() { (构造函数)
        setLayout(new FlowLayout()); (为按钮设置布局管理器: new 创建无名对象管理器来作为布局管理器)
        add(b1); 往布局管理器中添加组件
        add(b2);
    }
    public static void main(String[] args) {
        run(new Button1(), 200, 100);
    }
}

```



3. 图标 Icon

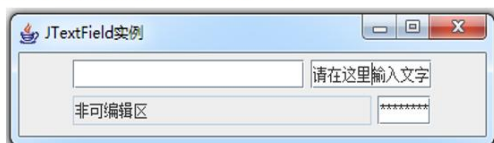
●和JLabel一起使用Icon

```
Icon bug = new ImageIcon( "img.jpg" );  
label2 = new JLabel ( "Label以文字和图片形式存在", bug,  
SwingConstants.LEFT );
```



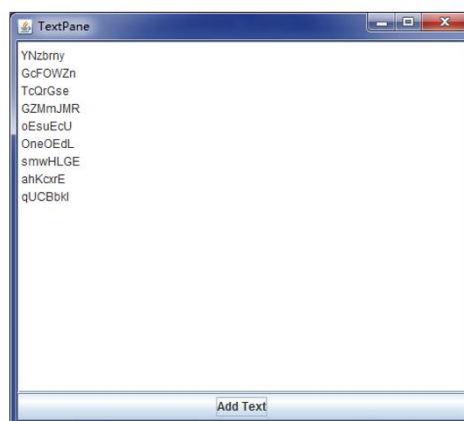
4. 文本域 JTextField

```
Container container = getContentPane();  
container.setLayout( new FlowLayout() );  
textField1 = new JTextField( 17 );  
container.add( textField1 );  
textField2 = new JTextField( "请在这里输入文字" );  
container.add( textField2 );  
textField3 = new JTextField( "非 可 编 辑 区", 22 );  
textField3.setEditable( false );  
container.add( textField3 );  
passwordField = new JPasswordField( "这里是密码输入框" );  
container.add( passwordField );
```

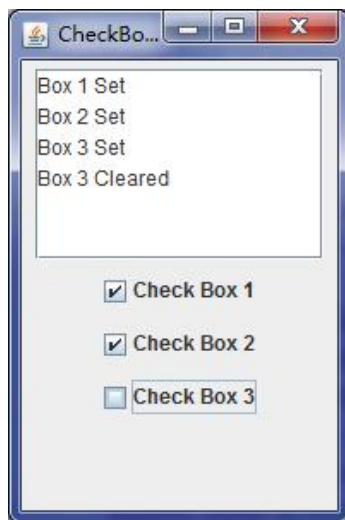


5. 文本面板 JTextPane

• 提供文本编辑功能



6. 复选框 JCheckBox

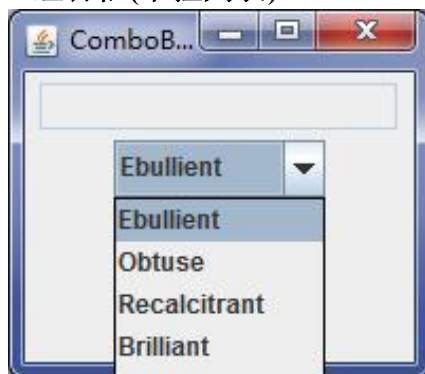


7. 单选按钮 JRadioButton

- JRadioButton
- ButtonGroup



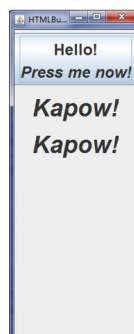
8. 组合框(下拉列表)JComboBox



13. Swing 组件上的 HTML

- 任何能接受文本的组件都可以接受HTML文本，且能根据HTML的规则来重新格式化文本。

```
Private JButton b=new JButton(  
"<html><b><font size=+2>"+  
"<center>Hello!<br><i>Press me now!")");
```



5.3 布局管理

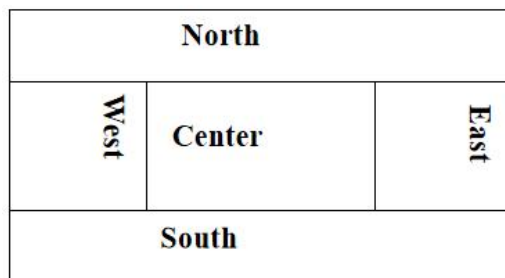
在界面设计中，一个容器中是要添加若干组件的。为组件安排在容器中的位置需要使用布局管理器。

在容器中有一个函数叫 `setLayout()`，可以选择不同的布局方式 `BorderLayout`、`FlowLayout`、`GridLayout`

1. 边界布局 `BorderLayout`

`BorderLayout` 是 `JFrame` 的默认布局

`BorderLayout` 将组件分置五个区域：北、南、东、西、中



使用 `add()` 加入的组件，把它放置在中央，然后把组件向各个方向拉伸，直至其他组件或边框

2. 顺序布局 `FlowLayout`

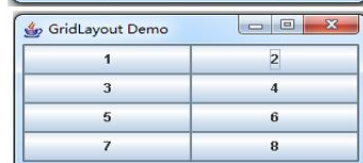
将组件从左到右，直到占满上方空间，再向下移动一行继续放置

所有的组件将被压缩到它们的最小尺寸，按“合适”的大小呈现



3. 网格布局 `GridLayout`

`GridLayout` 允许建立一个组件的网格，在网格里从左到右从上到下布局，在构造器中，我们确定行列数后，他们将以相同的长宽比排列。



container 容器，panel 是 container 的子类

frame 是顶层容器

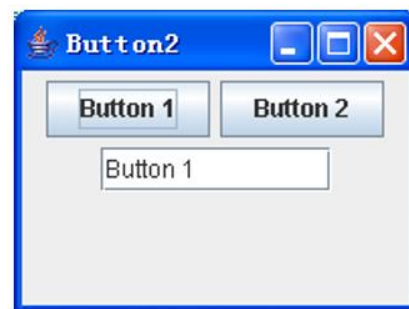
容器包括两个方面要设置，一个是容器里放什么（add），一个是怎么放（布局管理器 layout）

5.4 事件处理

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Button2 extends JFrame {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    private JTextField txt = new JTextField(10);
    class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String name = ((JButton)e.getSource()).getText();
            txt.setText(name); } }

    private ButtonListener bl = new ButtonListener();
    public Button2() {
        b1.addActionListener(bl); b2.addActionListener(bl);
        setLayout(new FlowLayout());
        add(b1); add(b2); add(txt);
    }
    public static void main(String[] args) {
        run(new Button2(), 200, 150); } } //:~
```



1. 事件处理机制

三类重要的对象:

事件(Event)

例如:KeyEvent,(ActionEvent)

事件源(Event Source)，例如: Button

事件处理器,事件监听器(Event handler)

事件监听器是对监听器接口的一个实现

用 addXXXListener()/removeXXXListener()添加和除去监听器

实现监听器接口，定义监听器类，在类的定义中完成事件的处理(actionPerformed 方法);

通过 new 创建事件监听器类对象;



实现监听器类的方法

1. 直接实现：编写的类直接继承监听器

到时候实例化的时候，一个对象内必然包含且仅包含一个 `actionlistener`

```
public class ButtonAct extends Frame implements ActionListener {
    private JButton b1;
    ... ..
    public ButtonAct() {
        ... .. //为 b1 注册事件监听者 B1
        b1.addActionListener(this);
        add(b1);
        ... ..
    }
    //利用 actionPerformed 方法进行事件处理
    public void actionPerformed(ActionEvent e){ #直接重写对象 this 的 actionPerformed
        who.setText("Button 1");
    }
    ... ..
}
```

2. 内部类实现(1)：定义内部类继承监听器

普普通通在类里面写个监听器类，然后调用

```
public class ButtonAct extends Frame {
    private JButton b1;
    ... ..
    public ButtonAct() {
        ... .. //为 b1 注册事件监听者 B1
        b1.addActionListener(new B1());
        add(b1);
        ... ..
    }
    //利用内部结构定义监听器类
    class B1 implements ActionListener {
        //利用 actionPerformed 方法进行事件处理
        public void actionPerformed(ActionEvent e){
            who.setText("Button 1");
        }
    }
    ... ..
}
```

3.内部类实现(2)：和(1)的区别也就是一个监听器有名儿，一个没名儿而已

```
public class ButtonAct extends Frame {
    private JButton b1;
    ... ..
    public ButtonAct() {
        ... .. //为 b1 注册事件监听者 B1
        b1.addActionListener(b1);
    }
}
```

```

        add(b1);
        ... ..
    }
    private B1 b1 = new B1();
    //利用内部结构定义监听器类
    class B1 implements ActionListener {
    //利用 actionPerformed 方法进行事件处理
        public void actionPerformed(ActionEvent e){
            who.setText("Button 1");
        }
    }
    ... ..
}

```

3. 利用匿名内部类(1): 在 addActionListener 函数的参数栏里重写监听器类
这个不是定义出来监听类只能当下用嘛，下面那个法(2)就能多次用了

```

public class ButtonAct extends Frame {
    private JButton b1;
    ... ..
    public ButtonAct() {
        ... .. //为 b1 注册事件监听者 B1
        b1.addActionListener(
            //利用匿名内部类结构定义监听器类
            new ActionListener (){
                //利用 actionPerformed 方法进行事件处理
                public void actionPerformed(ActionEvent e){
                    who.setText("Button 1");
                }
            });
        add(b1);
        ... ..
    }
    ... ..
}

```

5.利用匿名内部类(2): 在创建监听器对象时定义监听器类

```

public class ButtonAct extends Frame {
    private JButton b1;
    ... ..
    public ButtonAct() {
        ... .. //为 b1 注册事件监听者 B1
        b1.addActionListener(b1);
        add(b1);
        ... ..
    }
    //利用匿名内部类结构定义监听器类

```

```
private ActionListener bl = new ActionListener () {
//利用 actionPerformed 方法进行事件处理
    public void actionPerformed(ActionEvent e){
        who.setText("Button 1");
    }
... ..
}
```

2. 事件类与事件监听接口

自己看 ppt 去

6.1 异常概述

1.异常和错误

Java 语言中的异常事件分为两种：

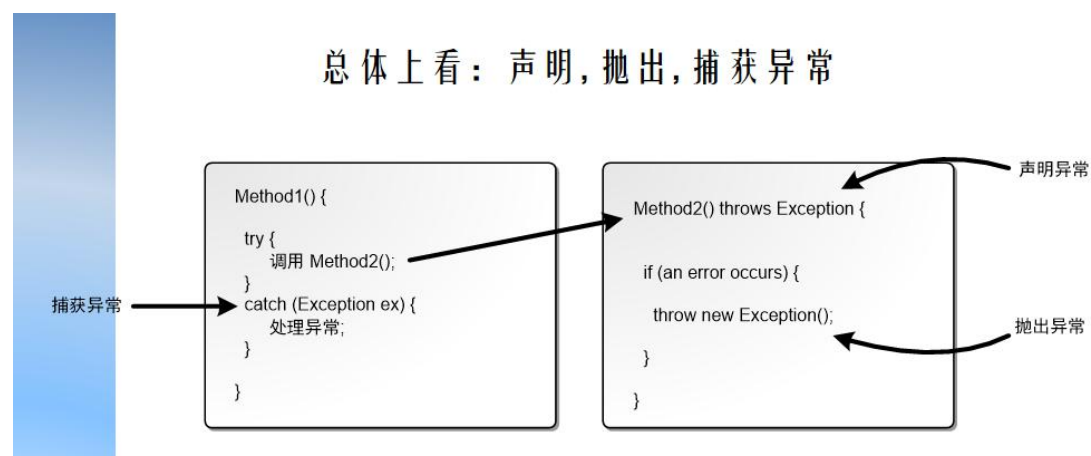
Exception 是指非致命的（程序改改还能用的），如：输入输出异常、运行时异常……

异常描述那些程序或外部环境引起的错误。这些错误可以捕获和处理。

Error 是指致命的（改程序也解决不了的），如：虚拟机错误、内存溢出错误……

错误是由 JVM 抛出，来描述系统内部错误的。这种错误很少发生，但是一旦发生除了提醒用户或者终止程序你没有办法来处理。

6.2 异常处理方法



两种处理异常的方法：

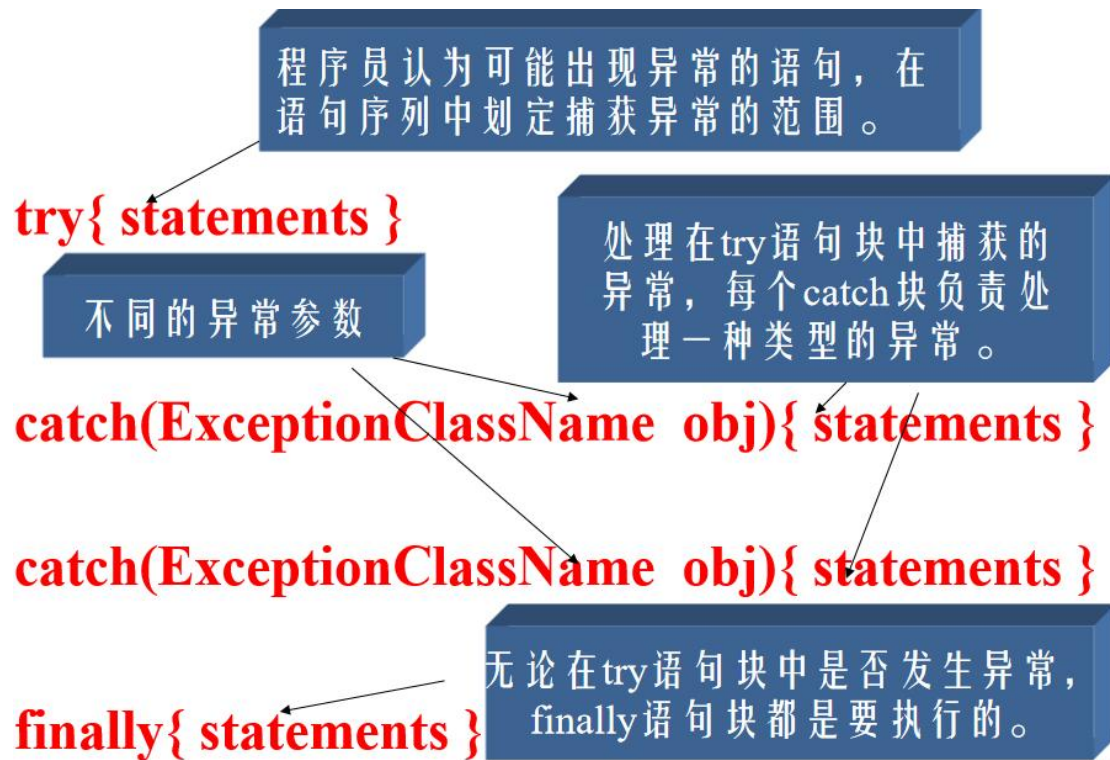
- 1、使用 try-catch-finally 语句块结构在程序代码中捕获和处理异常；
- 2、把异常对象通过层层向上抛出直至转交给 JVM 处理。Java 语言称产生异常和转交异常的过程为抛出异常。

1.捕获和处理异常

捕获和处理异常是通过 try-catch-finally 语句块实现的。

程序在运行过程中对 try 语句块中的语句进行监测，根据其中出现的异常的种类决定是否采用 catch 语句块中的语句以及采用哪个 catch 语句块中的语句处理异常，最后，再运行 finally 语句块中的语句来结束捕获和处理异常的过程。

多个 catch 块时候，只会匹配其中一个异常类并执行 catch 块代码，而不会再执行别的 catch 块，并且匹配 catch 语句的顺序是由上到下。



获取异常有关信息的三个方法：

getMessage() ： 获取错误性质。

toString() ： 给出异常的类型与性质。

printStackTrace() ： 指出异常的类型、性质、栈层次及出现在程序中的位置。

2. 抛出异常

Java 语言也允许指明出现的异常不在当前方法内处理，而是将其抛出，送交到调用它的方法来处理，在调用序列中逐级向上传递，乃至传递到 Java 运行时系统，直至找到一个运行层次可以处理它为止。

声明抛出异常是在一个方法声明中的 throws 子句中给出的。其语法格式为

returnType methodName([paramList]) throws ExceptionList { ... }

例：抛出异常程序

```
public class ThrowsException {
    public void catchThrowException(int str) throws ArrayIndexOutOfBoundsException,
    ArithmeticException, NullPointerException {
        System.out.println(str + " ");
        if (str == 1) {
            int a = 0;
            int b = 6 / a;
        } else if (str == 2) {
```

```

        String s[] = new String[5];
        s[0].toCharArray();
    } else if (str == 3) {
        int[] a = new int[5];
        a[10] = 10;
    } else {
        System.out.println("没有发现异常,系统正常执行");
    }
    //方法结束
}
public static void main(String args[]) {
    ThrowsException te02 = new ThrowsException();
    try {
        te02.catchThrowException(0);
    } catch (Exception e) {
        System.out.println("异常:" + e.getMessage());
    }
    try {
        te02.catchThrowException(1);
    } catch (Exception e) {
        System.out.println("异常:" + e);
    }
    try {
        te02.catchThrowException(2);
    } catch (Exception e) {
        System.out.println("异常:" + e);
    }
    try {
        te02.catchThrowException(3);
    } catch (Exception e) {
        System.out.println("异常:" + e);
    }
} //主方法结束
} //主类结束

```

3.异常传递链

如果在方法中产生了异常，方法会在抛出异常的地方退出；
 如果不想终止方法，那就需要在特定的区域用 try 来捕获异常。

6.3 自定义异常

1.自定义异常的创建

可以在 Java 类库中现有异常类的基础上由用户创建新的异常类，新的异常类必须用 `extends`

子句声明是 Exception 类的子类。

```
public class MyException extends Exception{
    public MyException(String ErrorMessage){
        super(ErrorMessage);
    }
}
```

2.自定义异常的抛出与捕获

自定义异常的抛出：

定义异常类的代码写在程序中，与其他类定义并列成为程序的一部分，在使用时与已有的异常类基本相同，只是在 throw 子句中使用下面的语法：

```
throw new MyException();
```

throws 关键字和 throw 关键字的区别：

throws 关键字是在方法声明时放在方法头中的，作用是声明一个方法可能抛出的所有异常；

throw 关键字则出现在方法体的内部，是一个具体的执行动作，作用是抛出一个具体异常对象。

7.1 数组

数组用来存储[相同类型](#)(对象，基本类型)值的集合。

数组元素的访问：

数组名[索引值]，如：a[10]

a.length 表示数组的长度

1.数组的创建、初始化和使用

(1) 数组的声明

声明数组变量有两种形式：

类型[] 数组名；

类型 数组名[]；

例：int a[]; String[] b;

(2) 数组初始化

静态初始化

```
int a[]={21,34,7,8,10};
```

```
int[] a = new int[] {1, 2, 3, 4};
```

动态初始化

```
int[] a = new int[10];
```

 动态初始化[可以使用变量](#)作为参数

```
int a[] = new int[x];
```

动态初始化后数组元素被自动赋默认值

对象数组的初始化

数组内的元素若是对象，可使用以下语法：

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

例

```
Apple a[] = new Apple[] { new Apple(), new Apple() } ;
```

2. 多维数组

n 维数组中存放着多个 n-1 维数组的引用。

两种常用的声明方式：

```
int[][] a;    int a[][];
```

初始化：

```
int a[][] = { {1,2,3} , {4,5,6} };
```

```
int[][] a = new int[][] { {1,2,3} , {4,5,6} };
```

动态初始化：

```
int a[][] = new int[2][3];
```

数组元素引用的数组长度不等的多维数组称为不规则数组，这是 Java 语言数组的一个特性：

```
int[][] a = new int[2][];
```

```
a[0]=new int[2];    a[1]=new int[3];
```

多维数组的 length 属性：

a.length 指第一维数组的长度

a[0].length 数组中第 1 个元素引用的数组的长度

3. 数组与数组的引用

可以将数组变量作为方法参数达到改变数组元素值的效果

```
public static void changeArrayValue(int[] para) {  
    para[0]=2;  
}
```

4. 数组工具类 java.util.Arrays

方 法	描 述
copyOf()	将一个数组中的值拷贝到新的数组中
sort()	将一个数组中的值进行排序，默认是升序排列
binarySearch()	在已排好序的数组中查找特定值
equals()	判断两个数组是否相等
asList()	将数组重构为列表

15

5. 对象比较接口

当数组为对象类型时，为数组排序必须能判断数组中两个对象之间的大小。

java.lang.Comparable 接口

```
int compareTo(Object o)
```

java.util.Comparator 接口

```
int compare(Object o1, Object o2)
```

```
import java.lang.Comparable;
class Employee implements Comparable {
    int id;
    String name;
    public int compareTo(Object o) {
        Employee e=(Employee)o;
        return this.id-e.id;
    }
}
```

Employee类是可以比较大小的，当调用**sort()**方法对**Employee**类型数组进行排序时，方法将自动调用**Employee**类中的**compareTo()**方法比较大小，并按**id**值从小到大排列。否则调用**sort()**会报异常。

即自创类组成的数组要调用数组排序函数，

必须保证类内重写了 compare 函数

7.2 枚举

有些数据集合，它们的数值是不变化的，而且集合中的元素个数是有限的。

例：季节={春，夏，秋，冬}

创建枚举类型使用“enum”关键字，枚举类型变量只能保存此类声明时给定的某个枚举值。

```
enum Season { SPRING, SUMMER, FALL, WINTER};
```

```
Season s = Season.SPRING;
```

例：

```
public class TestEnum {
    public enum Season { SPRING, SUMMER, FALL, WINTER };
    public static void main(String[] args) {
        seasonOutput(Season.SPRING);
    }
    private static void seasonOutput(Season s){
        switch (s) {
            case SPRING:
                System.out.println("春");    break;
            case SUMMER:
                System.out.println("夏");    break;
            case FALL:
                System.out.println("秋");    break;
            case WINTER:
                System.out.println("冬");    break;
            default :
                System.out.println("default");
        }
    }
}
```

7.3 容器

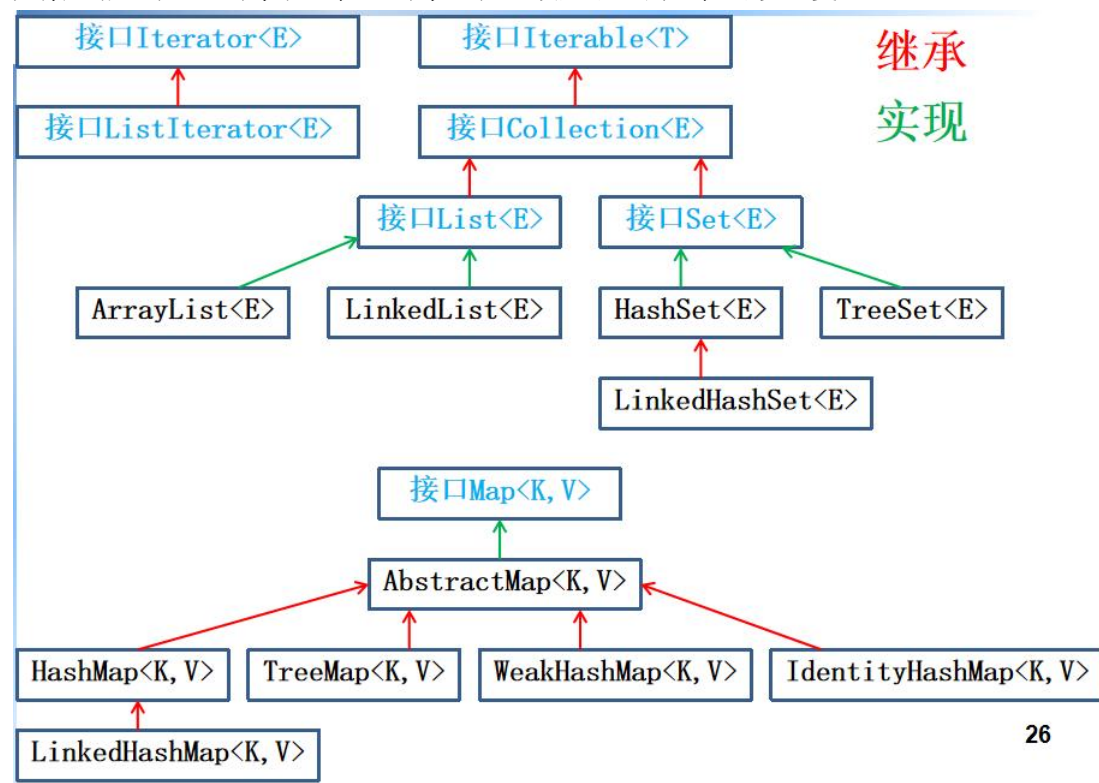
接口 Collection（类集）：一组单独元素集合

List：保存的对象有顺序，按照元素的索引位置检索对象。允许重复元素。

Set：不允许保存重复的元素。元素之间没有顺序。靠元素值检索对象。

接口 Map<K,V>（映射）：一组键值对

元素包括“键”对象和“值”对象。键必须是唯一的，值可以重复。



26

1. List（列表）

List 是 Collection 的子接口，是有序的 Collection

顺序是 List 和其它集合最大的区别，索引位置是其重要属性

<code>add()</code>	重载的add()方法，可在指定位置处插入元素
<code>remove()</code>	重载的remove()方法，可在指定位置处删除元素
<code>get(int)</code>	获取指定位置的元素
<code>set()</code>	重设指定位置的元素的值
<code>indexOf()</code>	获取指定元素的位置
<code>lastIndexOf()</code>	获取指定元素的最后一次出现的位置

List 常用实现类有 ArrayList（线性表）和 LinkedList（链表），两种类型存在性能差异
程序示例：

```
List a = new ArrayList();
//List a = new LinkedList();
a.add( "cat" );
a.add( "dog" );
```



```

a.add( "cat" );
a.remove(0);
for(int i=0; i<a.size(); i++) { // a.size()=2
    System.out.print(a.get(i)+",");
}

```

当使用实现类特有方法时不能使用接口定义，而应该使用具体类定义

```
ArrayList a = new ArrayList();
```

线性表：

◆ 构造方法

<code>ArrayList()</code>	构造一个空的线性表，容量为10
<code>ArrayList(Collection)</code>	根据已有类集构造线性表
<code>ArrayList (int)</code>	构造指定初始容量大小的线性表

链表：

<code>addFirst()</code>	向链表头插入元素
<code>addLast ()</code>	向链表尾插入元素
<code>getFirst ()</code>	获取链表头元素
<code>getLast ()</code>	获取链表尾元素
<code>removeFirst ()</code>	移除链表头元素
<code>removeLast ()</code>	移除链表尾元素

使用 `LinkedList` 可以很简单地构造类似“栈”（stack）、“队列”（queue）这样的数据结构

元素类型可以不一致：

```
List a = new ArrayList();
```

```
a.add( "haha" );           //插入一个 String 类
```

```
a.add( new Cat() ); //插入一个 Cat 类
```

```
a.add( new Dog() );//插入一个 Dog 类
```

```
String s = (String)a.get(0);//取出第一个元素
```

2. 泛型

（1）泛型类

在声明泛型类的变量时，使用尖括号“<>”来指定形式类型参数

```
class 类名<类型参数列表> {类体}
```

例：Class A <X1,X2> {X1 a; X2 b;}

在应用泛型类时，用具体类型填入类型参数，泛型类的具体化

类名<具体类型列表> 变量名=new 类名<具体类型列表>（构造函数的参数列表）；

例：A<String,Integer> m = new A<String,Integer>();

(2) 泛型方法：

```
public <T> T testGen (boolean b, T first, T second) {return b ? first : second;}
String s = testGen(true, "a", "b");
```

(3) 容器的泛型

示例：定义泛型的类+

```
import java.util.LinkedList;
class StackL<T> { //构造 “栈”
    private LinkedList<T> list = new LinkedList();

    public void push(T o) {           //进栈
        list.addFirst(o);
    }
    public T top() {                  //查看栈顶元素
        return list.getFirst();
    }
    public T pop() {                  //出栈
        return list.removeFirst();
    }
}
```

3. Set (集合)

不允许保存重复的元素，元素之间无顺序，无索引

常用的实现类：HashSet 和 TreeSet 以及 LinkedHashSet。

例：

```
Set<String> set = new HashSet<String>();
set.add("cat");
set.add("dog");
set.add("cat");
set.add("cat");//无法插入
set.remove("cat");
System.out.println(set);
```

(1) Iterator (迭代器)

调用方法 iterator() 返回一个迭代器。第一次调用 Iterator 的 next()方法时，它返回序列的第一个元素。

使用 next()获得序列中的下一个元素，每成功调用一次迭代器向后移动一个元素。

使用 hasNext()检查序列中是否还有元素。

使用 remove()将迭代器新返回的元素删除。

Collection 接口都可以使用迭代器：

```
List<String> a = new ArrayList<String>();
...
for(Iterator<String> it= a.iterator(); it.hasNext(); ) {
```

```

...
String s = it.next(); //取出元素
...
}

```

(2) for-each 遍历

for(变量类型 变量名:集合){...}

for-each 只能遍历两种类型的对象:

—数组

—实现了 java.lang.Iterable 接口的类的实例

```
Set<String> set = new HashSet<String>();
```

...

```

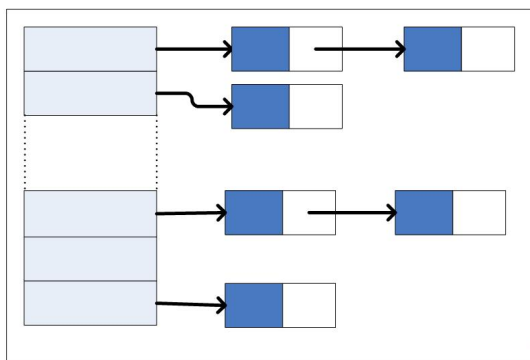
for(String s : set) {
    System.out.println(s);
}

```

(3) HashSet (哈希集)

采用“哈希表”与哈希算法实现快速查找

内部维护一个链表数组



要求对象重写 hashCode()方法与 equals()方法

◆构造方法

HashSet()	构造空的HashSet，容量16，载入因子0.75
HashSet(Collection)	从其它类集构造HashSet
HashSet(int)	指定初始容量，载入因子0.75的HashSet
HashSet(int, float)	指定初始容量与载入因子的HashSet

其中，载入因子为数组重构的条件，当（元素的数目>容量*载入因子）时，扩展哈希集容量。

查找元素 a 过程:

调用 a.hashCode()方法获得哈希码

由哈希码计算出 a 在数组中位置，即插入哪一个链表。

调用 a.equals()方法和此链表中每一个元素比较。

如果有相同的元素，a 被找到。结束。

否则说明未找到 a。

示例：

```
class Employee { //员工类
    int id;
    public Employee(int id) {
        this.id = id;
    }
    public int hashCode() {
        return this.id;
    }
    public boolean equals(Object o) {
        Employee e = (Employee) o;
        if(this.id==e.id) return true;
        else return false;
    }
}
```

系统默认 hashCode()返回对象在内存中的地址

系统默认 equals()方法比较两个对象内存中的地址

HashSet 中的对象需要重写 hashCode()方法与 equals()方法

如果 a.equals(b)返回 true，那么 a.hashCode()==b.hashCode()

内容 equals 的对象 hashcode 一定相等

两个对象的 hashcode 不相等则两个对象一定不 equals

(4) 其它 Set

HashSet 的元素存放无顺序

LinkedHashSet 保持元素的添加顺序

TreeSet 元素按值进行排序存放

4. Map（映射）

键与值都为对象，键值对也为对象

常用的实现类：HashMap 和 TreeMap 以及 LinkedHashMap。

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

```
map.put(1," abc" );
```

(1) Map 遍历

Map 接口没有实现 Iterable 接口，无法使用迭代器

Map 键值对元素无序，无法使用索引

Map 接口提供 3 种集合的视图：

keySet(): 生成键的 Set 型集合

values(): 生成值的 Collection 型集合

entrySet(): 生成键值对的 Set 型集合

方法一（遍历值）：

```
public void byValue(Map<String,Student> map) {
    Collection<Student> c = map.values();
    Iterator it;
```

```

        for (it=c.iterator(); it.hasNext();){
            Student s = (Student)it.next();
        }
    }
}

```

方法二（通过键遍历值）：

```

public void byKey (Map<String, Student> map) {
    Set<String> key = map.keySet();
    Iterator it;
    for (it = key.iterator(); it.hasNext(); ) {
        String s = (String) it.next();
        Student value = map.get(s);
    }
}

```

方法三：Map.Entry:Map 内部定义的一个接口，专门用来保存键值对的内容

```

public static void byEntry(Map<String, Student> map) {
    Set<Map.Entry<String, Student>> set = map.entrySet();
    Iterator<Map.Entry<String, Student>> it;
    for (it= set.iterator(); it.hasNext(); ) {
        Map.Entry<String, Student> entry = it.next();
        System.out.println(entry.getKey() + "--->" + entry.getValue());
    }
}

```

（2）HashMap（哈希映射）

HashSet 内部维护着一个 HashMap

“键”就是我们要存入的对象

“值”是一个常量 PRESENT

例如：

HashSet 的 add()方法调用其内部 HashMap 的 put()方法

5. Collections（工具类）

一个包含可以操作或返回集合的专用静态类 java.util.Collections

常用方法：

排序 sort()：对指定 List 按升序进行排序。列表中的所有元素都必须实现 Comparable 接口

二分法查找 binarySearch()：使用二分法查找元素，List 必须已经排好序

反转 reverse()：将 List 中的元素按逆序排列

复制 copy()：将源 List 的元素复制到目标，并覆盖它的内容

min()、max()：返回给定 Collection 的最值元素

6. 容器类型选择

List 的选择

ArrayList：随机访问（即调用 get()方法）效率最高

LinkedList：在列表中部进行插入和删除操作效率很高

两者的查找效率都相当低

Set 和 Map 的选择

HashSet: “增，删，改，查”这类数据处理上的效率最高。我们大多数情况下会选择使用 HashSet。

TreeSet: 查找速度也是较快的，大概为 $\log(n)$ 级别。不过这个较快也只是相对于 List 来说，还是比不上 HashSet。

LinkedHashSet: 在有高效存取性能要求，同时又要求数据有序的情况下比较适用。

8.1 流与相关类

1. 流的概念

Java.io 包中定义了多个流类型(类或抽象类)来实现输入/输出的功能;屏蔽了处理数据的细节。

可以从不同角度进行分类:

数据流向: 输入流 输出流

(输入流: 从文件中读取数据到程序 输出流: 从程序向文件中写入数据)

处理数据单位: 字节流 字符流

功能不同: 节点流 处理流

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

2. 字节流及其方法

继承自 InputStream 的流都是用于向程序中输入数据，且数据单位为字节（8 bits）

InputStream 的方法(throws IOException)

int read() 读取一个字节并以整数的形式返回

int read(byte[] buffer) 读取一系列字节并存储到一个数组 buffer 中，返回实际读取的字节数

int read(byte[] buffer, int offset, int length) 读取 length 字节并从 offset 位置存储到一个数组 buffer 中

void close() 关闭流释放内存资源

long skip(long n) 跳过 n 个字节不读，返回实际跳过的字节数

OutputStream 的方法(throws IOException)

void write (int b) 向流中写入一个字节数组，该字节数据为参数 b 的低 8 位

void write (byte[] b) 将一个字节类型的数组中的数据写入输出流

void write (byte[] b, int off, int len) 将一个字节数组指定位置 off 开始 len 个字节写入输入流

void close() 关闭流释放内存资源

void flush() 将输出流中缓冲的数据全部写入到目的地

3. 字符流及其方法

继承自 Read 的流都是用于向程序中输入数据，且数据单位为字符（16 bits）

Reader 的方法(throws IOException)

`int read()` 读取一个字符并以整数的形式返回

`int read(char[] cbuf)` 读取一系列字符并存储到一个数组 `cbuf` 中，返回实际读取的字符数

`int read(char[] cbuf, int offset, int length)` 读取 `length` 字符并从 `offset` 位置存储到一个数组 `cbuf` 中

`void close()` 关闭流释放内存资源

`long skip(long n)` 跳过 `n` 个字符不读，返回实际跳过的字符数

Writer 的方法 (throws IOException)

`void write (int c)` 向输出流写入一个字符数据，为参数 `c` 的低 16 位

`void write (char[] cbuf)` 将一个字符数组写入输出流

`void write (char[] cbuf, int offset, int length)` 将字符数组从指定位置 `offset` 开始 `length` 个字符写入到输出流

`void write (String string)` 将一个字符串写入到输入流

`void write (String string, int offset, int length)` 将一个字符串从 `offset` 开始 `length` 个字符写入输入流

`void close()` 关闭释放内存的资源

`void flush()` 将输出流中的缓冲的数据全部写出到目的地

4.其他 IO 流

(1)节点流

节点流 (数据接收流)包括文件，内容和管道，可以从一个特定的数据源（节点）读写数据（如：文件、内存等）。类型包含对文件操作的、在内存里按数组操作的、在内存里按字符串操作的、和对管道进行操作的。

(2)处理流

处理流 (过滤流)结合在其他流之上，可以修改或管理流中数据，并提供额外的功能

(3)缓冲流

带缓冲的输入流从一个类似于缓冲区的内存区域中读取数据，当缓冲区为空时，调用基本的输入 API，同样地，缓冲输出流向缓冲区中写数据，在缓冲区已满时调用基本的输出 API。

`void mark()`

//标记流中的当前位置.

`void reset()`

//尝试将该流重新定位到最近标记的点.

`void readLine()` // 仅适用于 `BufferedReader`

// 读文本中的一行

`void newLine()` // Only for `BufferedWriter`

// 写入一个行分隔符

`void flush()`

// 刷新流

(4)打印流

`PrintStream` 类 (字符)

`PrintWriter` 类(字节)

用于为其他输出流添加功能，使它们能够方便地打印各种数据值表示形式。分别针对字节和

字符，提供了重载的 `print` 和 `println` 方法用于多种数据类型的输出

`PrintWriter` 的示例：

```
import java.io.*;

public class FileOutputShortcut {
    static String file = "FileOutputShortcut.out";
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader( new StringReader(
            BufferedInputFile.read("FileOutputShortcut.java")));
        PrintWriter out = new PrintWriter(file);
        int lineCount = 1;
        String s;
        while((s = in.readLine()) != null )
            out.println(lineCount++ + ": " + s);
        out.close();
        System.out.println(BufferedInputFile.read(file));
    }
}
```

(5)数据流

数据流实现八种基本类型数据的输入/输出。同时，也可实现字符串的输入/输出。需要在 `InputStream` 和 `OutputStream` 类型的流上，也就是字节流之上进行构造。

构造器

- `DataInputStream` (`InputStream in`)

- `DataOutputStream` (`OutputStream out`)

数据流的示例：

```
import java.io.*;

public class TestDataStream {
    public static void main(String[] args) {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos); //将输出流 dos 对象输出
        //位置定为 ByteArrayOutputStream()类对象 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); //输入流读取对象设为 bais
            System.out.println(dis.readDouble());
            System.out.println(dis.readBoolean());
            dos.close();
            dis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
}  
}
```

ps:

ByteArrayOutputStream() 字节数组输出流在内存中创建一个字节数组缓冲区，所有发送到输出流的数据保存在该字节数组缓冲区中。

toByteArray() 创建一个新分配的字节数组。数组的大小和+当前输出流的大小，内容是当前输出流的拷贝

(6)对象流

对象流支持对象的输入输出，包括 `ObjectInputStream` 和 `ObjectOutputStream`

对象流的示例：

```
import java.util.concurrent.*;  
import java.io.*;  
import java.util.*;  
public class Logon implements Serializable {  
    private Date date = new Date();  
    private String username;  
    private transient String password;  
    public Logon(String name, String pwd) {  
        username = name;  
        password = pwd;  
    }  
    public String toString() {  
        return "logon info: \n    username: " + username +  
            "\n    date: " + date + "\n    password: " + password;  
    }  
    public static void main(String[] args) throws Exception {  
        Logon a = new Logon("Hulk", "myLittlePony");  
        System.out.println("logon a = " + a);  
        ObjectOutputStream o = new ObjectOutputStream(new FileOutputStream("Logon.out"));  
        o.writeObject(a);  
        o.close();  
        TimeUnit.SECONDS.sleep(1); // Delay  
        ObjectInputStream in = new ObjectInputStream(new FileInputStream("Logon.out"));  
        System.out.println("Recovering object at " + new Date());  
        a = (Logon)in.readObject();  
        System.out.println("logon a = " + a);  
    }  
}
```

8.2 标准 I/O 流

标准 I/O (与文件 File)

三个标准 I/O 流

- `System.in` : `InputStream` 类的对象实例 `in` 作为标准输入流对象，对应于键盘输入
- `System.out`: `PrintStream` 类的对象实例 `out` 作为标准输出流对象，对应于显示器输出
- `System.err`: `PrintStream` 类的对象实例 `err` 作为标准错误输出流对象，对应于显示器输出

`System.in` 的示例：

```
import java.io.*;
public class Echo {
    public static void main(String[] args)
        throws IOException {
        BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
//定义键盘输入为 bufferedreader 类对象 stdin 的读取对象
// BufferedReader 类从字符输入流中读取文本并缓冲字符，以便有效地读取字符，数组和行
        String s;
        while((s = stdin.readLine()) != null && s.length() != 0)
//s = stdin.readLine() s 就等于从 stdin 的读取路径中读取一行数据
            System.out.println(s);
        }
}
```

`System.out` 转换示例：

```
import java.io.*;
public class ChangeSystemOut {
    public static void main(String[] args) {
        PrintWriter out = new PrintWriter(System.out, true); //输出对象定义为显示器
        out.println("Hello, world");
    }
}
//Output:
Hello, world
```

重新定义标准 I/O 的示例：

```
import java.io.*;
public class Redirecting {
    public static void main(String[] args) throws IOException {
        PrintStream console = System.out;
先保存打印流类的显示器对象为 console
        BufferedInputStream in = new BufferedInputStream(
            new FileInputStream("Redirecting.java"));
输入流对象 in 从文件"Redirecting.java"读取
        PrintStream out = new PrintStream(
            new BufferedOutputStream(new FileOutputStream("test.out")));
打印流类打印位置为 test.out 的 out 对象
```

```

        System.setIn(in); 重设 System.in 的输入位置
        System.setOut(out);
        System.setErr(out);
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String s;
        while((s = br.readLine()) != null)
            System.out.println(s);
        out.close(); // Remember this!
        System.setOut(console);
    }
}

```

8.3 文件输入输出流

1. File 类

java.io.File

表示系统文件名，包括某个特定的路径和文件名，或者一系列路径和文件名

File 类的构造器：

public File (String pathname)

//以 pathname 为路径创建 File 对象，如果 pathname 为相对路径则在默认当前路径下进行存储

public File(String parent, String child)

//以 parent 为父路径，child 为子路径创建 File 对象

canRead() 可读？ canWrite() 可写 exists() 存在？

isDirectory() 是路径？ isFile() 是文件？ isHidden() 隐藏的？

lastModified() 最近修改时间 length() 长度

getName() 名称 getPath() 路径

File 类的方法：

public boolean createNewFile() throws IOException

// 如果该文件或路径不存在，则自动创建一个新的空文件或路径

public boolean delete()

// 删除文件或目录

public boolean mkdir()

// 创建一个目录

public boolean mkdirs ()

// 创建一系列目录

File 类的示例：

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

```
public class Test {
```

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

```
String filename = "myfile.txt";
```

```

String directory = " mydir1 " + "/" + "mydir2";
File f = new File(directory, filename);
if (f.exists()) {
    System.out.println("File Name: " + f.getAbsolutePath());
    System.out.println("File Length: " + f.length());
} else {
    f.getParentFile().mkdirs();
    try {
        f.createNewFile() ;
    } catch (IOException e) {
        e.printStackTrace() ;
    }
}
}
}
}

```

2.基于字节的文件流

(1) FileInputStream

访问文件的示例： (FileInputStream)

```

import java.io.*;

public class TestFileInputStream {
    public static void main(String[] args) {
        int b = 0;
        FileInputStream in = null;
        try {
            in = new FileInputStream("./TestFileInputStream.txt");
        } catch (FileNotFoundException e) {
            System.out.println("Can't find file");
            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("read "+num+" Byte");
        } catch (IOException e1) {
            System.out.println("File read error");
            System.exit(-1);
        }
    }
}

```

(2) FileOutputStream

访问文件的例子： (FileOutputStream)

```

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("./TestFileInput.txt ");
        out = new FileOutputStream("./TFI.txt ");
        while((b=in.read())!=-1){
            out.write(b);
        }
        in.close();
        out.close();
    } catch (FileNotFoundException e2) {
        System.out.println("Can't find file");
        System.exit(-1);
    } catch (IOException e1) {
        System.out.println("File copy error");
        System.exit(-1);
    }
    System.out.println("File copy finished");
}
}

```

3.基于字符的文件流

(1) FileReader

访问文件的例子: (FileReader)

```

import java.io.*;
public class TestFileReader {
    public static void main(String[] args) {
        FileReader fr = null;
        int c = 0;
        try {
            fr = new FileReader("./TestFileReader.java");
            int ln = 0;
            while ((c = fr.read()) != -1) {
                System.out.print((char)c);
            }
            fr.close();
        } catch (FileNotFoundException e) {...}
        catch (IOException e) {...}}
    }
}

```

(2) FileWriter

访问文件的例子: (FileWriter)

```

import java.io.*;
public class TestFileWriter {
    public static void main(String[] args) {

```

```

    FileWriter fw = null;
    try {
        fw = new FileWriter("./unicode.dat");
        for(int c=0;c<=50000;c++) fw.write(c);
        fw.close();
    } catch (IOException e1) {
        e1.printStackTrace();
        System.out.println("File write error");
        System.exit(-1); }}}

```

8.4 随机访问文件

适用于由大小已知的记录组成的文件，可以使用 `seek()` 将记录从一处转移到另一处，只有 `RandomAccessFile` 类在文件上支持搜寻方法

构造器和方法：

`RandomAccessFile(File file, String mode)`

`RandomAccessFile(String name, String mode)`

`mode` 为 'r' 表示随机读，'rw' 表示随机读写，没有只随机写

`void seek(long pos)`

// 用于文件内移至新位置

`long getFilePointer()`

// 得到当前的位置

`long length()`

// 判断文件的大小

随机访问文件的示例：

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

```
public class UsingRandomAccessFile {
```

```
    static String file = "rtest.dat";
```

```
    static void display() throws IOException {
```

```
        RandomAccessFile rf = new RandomAccessFile(file, "r");
```

```
        for(int i = 0; i < 7; i++)
```

```
            System.out.println("Value " + i + ": " + rf.readDouble());
```

```
            System.out.println(rf.readUTF());
```

```
        rf.close();    }
```

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

```
    RandomAccessFile rf = new RandomAccessFile(file, "rw");
```

```
    for(int i = 0; i < 7; i++)
```

```
        rf.writeDouble(i*1.414);
```

```
    rf.writeUTF("The end of the file");
```

```
    rf.close();    display();
```

```
    rf = new RandomAccessFile(file, "rw");
```

```
    rf.seek(5*8);
```

```

System.out.println(rf.length());
System.out.println(rf.getFilePointer());
rf.writeDouble(47.0001);
rf.close(); display(); }}

```

8.5 压缩

Java I/O 类库中的类读写压缩格式的数据流，可以用它们对其他的 I/O 类进行封装，以提供压缩功能

这些类不是从 Reader 和 Writer 类派生的，而是属于 InputStream 和 OutputStream

主要有下面这些压缩类

GZIP 比较简单，适合对单个数据流进行压缩

ZIP 功能比较强大适合对多文件进行压缩

压缩类	功能
ZipOutputStream	压缩数据，生成Zip格式文件
GZIPOutputStream	压缩数据，生成GZip格式文件
ZipInputStream	解压缩已经生成的Zip格式文件
GZipInputStream	解压缩已经生成的GZip格式文件

另外两个重要的类用于读取和生成 JAR 压缩文件——一种 Java 技术中常用的压缩文件。分别是 ZipInputStream 和 ZipOutputStream 类的子类，保存于 java.util.jar 包中。

压缩类	功能
JarInputStream	用于从任何输入流读取 JAR 文件内容
JarOutputStream	用于向任何输出流写入 JAR 文件内容

GZIP 的示例：

```

import java.util.zip.*;
import java.io.*;
public class GZIPcompress {
    public static void main(String[] args) throws IOException {
        if(args.length == 0) {
            System.out.println("Usage: \nGZIPcompress file\n"
                +"\tUses GZIP compression to compress "+"the file to test.gz");
            System.exit(1);
        }
        BufferedReader in = new BufferedReader(new FileReader(args[0]));
        BufferedOutputStream out = new BufferedOutputStream(
            new GZIPOutputStream(new FileOutputStream("test.gz"))); //压缩文件为 test.gz
        //不是先有完整文件然后再压缩，而是直接有一个文件为压缩格式，你还可以随时往这个.gz
        压缩文件里写东西
        System.out.println("Writing file"); //显示器上显示
        int c;
        while((c = in.read()) != -1)

```

```

        out.write(c);    //从 args 往 test.gz 里写
    in.close();
    out.close();
    System.out.println("Reading file");
    BufferedReader in2 = new BufferedReader(
        new InputStreamReader(new GZIPInputStream(new FileInputStream("test.gz"))));
//解压文件 test.gz
    String s;
    while((s = in2.readLine()) != null)
        System.out.println(s);
    }
}

```

ZIP 的示例:

```

import java.util.zip.*;
import java.io.*;
import java.util.*;
public class ZipCompress {
    public static void main(String[] args) throws IOException {
        FileOutputStream f = new FileOutputStream("test.zip");
        CheckedOutputStream csum = new CheckedOutputStream(f, new Adler32());
都是套管子的，按它们包装的顺序反过来 close，这种设计是 Filtered，就是一层套一套，
比如 ZipOutputStream 它只负责 zip 压缩，至于内容最终存放在哪里是由它的构造函数
参数 cos 来负责的，而 Cos 只负责按 CRC 32 检验数据，至于数据本身最终又存放在哪
里是由 fileOutputStream 说得算，因此最终数据需要从管道中输出到 fileOutputStream 中
的。

```

```

        ZipOutputStream zos = new ZipOutputStream(csum);
        BufferedOutputStream out = new BufferedOutputStream(zos); //输出对象定为 zip 类对象
zos
        zos.setComment("A test of Java Zipping");
        for(String arg : args) {
            System.out.println("Writing file " + arg);
            BufferedReader in = new BufferedReader(new FileReader(arg));
            zos.putNextEntry(new ZipEntry(arg));
//putNextEntry(ZipEntry e)方法开始编写新的 ZIP 文件条目并将流定位到条目数据的开头。
            int c;
            while((c = in.read()) != -1)
                out.write(c);
            in.close();
            out.flush();
        }
        out.close();
        System.out.println("Checksum: " + csum.getChecksum().getValue());
        System.out.println("Reading file");

```

```

FileInputStream fi = new FileInputStream("test.zip");
CheckedInputStream csumi = new CheckedInputStream(fi, new Adler32());
ZipInputStream in2 = new ZipInputStream(csumi);
BufferedInputStream bis = new BufferedInputStream(in2);
ZipEntry ze;
while((ze = in2.getNextEntry()) != null) {
    System.out.println("Reading file " + ze);
    int x;
    while((x = bis.read()) != -1)
        System.out.write(x);
}
if(args.length == 1)
    System.out.println("Checksum: " + csumi.getChecksum().getValue());
bis.close();
ZipFile zf = new ZipFile("test.zip");
Enumeration e = zf.entries();
while(e.hasMoreElements()) {
    ZipEntry ze2 = (ZipEntry)e.nextElement();
    System.out.println("File: " + ze2);}}}}

```

第九章 并发控制

9.2 基本概念

多线程的作用

- 提高 UI 的响应速度
- 提高硬件资源的利用效率
- 隔离高速硬件和低速硬件
- 提供程序上的抽象

线程的创建:

从根本上讲, 线程是一段程序代码, 定义线程的关键是将这样的代码定义到主程序中。

具体表现为使主程序通过继承 `Thread` 类或者实现 `Runnable` 接口获得 `run()` 方法, 之后重写或者实现 `run()` 方法, 把程序代码写入 `run()` 方法中。

方法一: 实现 `Runnable` 接口

```

class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }
}

```

```

        public void run() {
            // compute primes larger than minPrime
            ...
        }
    }

```

方法二：继承 Thread 类

```

class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        ...
    }
}

```

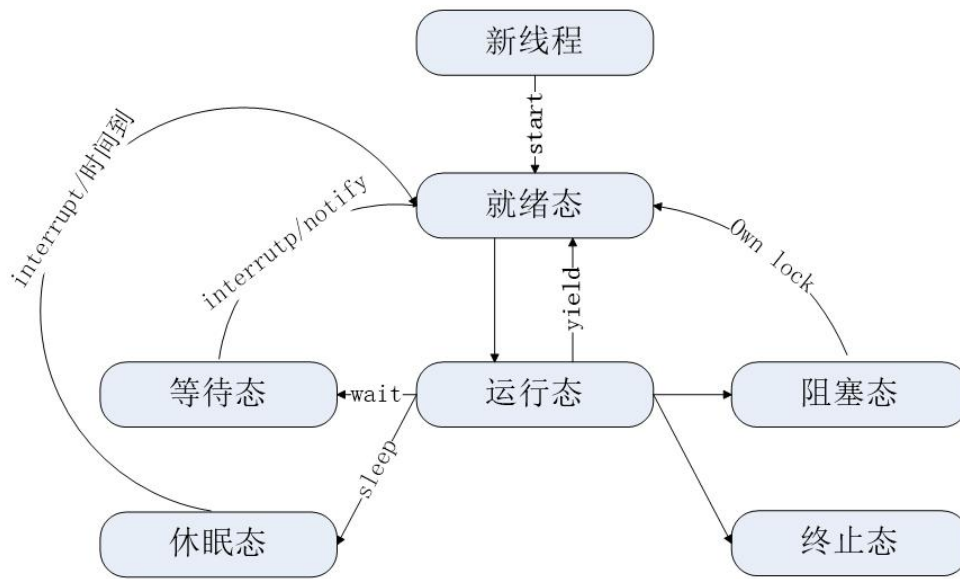
线程体的创建：

- 目标是获取到一个 Thread 的实例
- 调用 start 方法开始线程并分配资源
- 从 run 方法中返回是退出线程唯一正确的方法。

线程的状态：

- Born state / New Thread
- Ready state / Runnable Thread
- Running state
- Sleeping state
- Waiting state
- Blocked state
- Dead state

线程的生命周期：



线程的优先级：

- Java 线程优先级由操作系统实现
- 优先级分为 10 级，最高 10 级，用 MAX_PRIORITY 描述；最低 1 级，用 MIN_PRIORITY 描述；默认为 5 级，用 NORM_PRIORITY 描述。
- 优先级和线程组之间的关系
- 优先级设置的复杂性

Daemon 线程：

线程有一个特殊的属性，属性名为 daemon。如果此值为真，当其他非 Daemon 线程都处于终止态时，整个进程结束。也就是说，daemon 线程是后台监控线程，它有可能在任何运行点被终止。

两个相关的方法：

```
setDaemon  
isDaemon
```

9.3 线程之间的协作

- *部分更新问题
- *读写不一致问题

同步区域：

方法一

```
synchronized ReturnType methodName(parameterList){  
    method statement;  
}
```

方法二

```
synchronized (object){  
    statement;  
}
```

使用了锁（monitor）对象

同一时间只能有一个线程拥有监控器对象，称为获得锁
编程者要根据需要设置监控器。

协作机制：

多线程程序如何协作完成任务？

wait 方法簇与 notify 方法簇

调用 wait 方法后进入等待态

释放监控器的锁，暂停运行

条件满足后，恢复运行

申请监控器的锁，如不能，进入阻塞态。

获取锁，运行 wait 方法后续代码

```
public void put(Object x) throws InterruptedException {  
    synchronized (lock) { //需同步方法一  
        while (count == items.length)  
            lock.wait(); //等待 take 线程取走数据  
        items[putptr] = x;  
        if (++putptr == items.length)  
            putptr = 0;  
        ++count;  
        synchronized (emptyLock) {  
            emptyLock.notify(); //通知 take 线程  
        }  
    }  
}  
  
public Object take() throws InterruptedException {  
    synchronized (lock) { //需同步方法二  
        while (count == 0){  
            synchronized (emptyLock) { //如果为空，等待  
                emptyLock.wait();  
            }  
        }  
        Object x = items[takeptr];  
        if (++takeptr == items.length)  
            takeptr = 0;  
        --count;  
        lock.notify(); //通知 put 线程  
        return x; }  
}
```

```
}
```

死锁的概念

避免死锁的几个设计原则：逻辑隔离、即刻释放、顺序申请

9.4 扩展锁机制

现有机制的缺陷：

这个 `synchronized` 锁队列中获取到锁的顺序是不确定的。

同时苏醒算法不是公平的

查询不到当前的锁的状态。

锁的申请和释放有严格的顺序。

每个锁只能有一个等待队列。

`Lock&Condition` 是 Java 5 的新特性

`Lock` 对应 `synchronized`

`Condition` 上可以调用类 `wait/notify` 方法

`java.util.concurrent.locks.ReentrantLock` 类：

一个可重入的互斥锁定 `Lock`，它具有与使用 `synchronized` 方法和语句所访问的隐式监视器锁定相同的一些基本行为和语义，但功能更强大。

`lock()` 方法获取锁定，`unlock()` 试图释放此锁定。

```
class SimpleLock {  
    private final ReentrantLock lock = new ReentrantLock();    //①  
    public void doSth() {  
        lock.lock();    //②  
        try {  
            // statement  
        } finally {    //③  
            lock.unlock()    //④  
        }  
    }  
}  
  
public void put(Object x) {  
    lock.lock();  
    try {  
        while (count == items.length)  
            notFull.await();  
        items[putptr] = x;  
        if (++putptr == items.length) putptr = 0;  
        ++count;  
        notEmpty.signal();  
    } finally {  
        lock.unlock();  
    }  
}  
  
public Object take() {
```

```
lock.lock();
try {
    while (count == 0)
        notEmpty.await();
    Object x = items[takeptr];
    if (++takeptr == items.length)
        takeptr = 0;
    --count;
    notFull.signal();
    return x; } finally {
    lock.unlock(); }
```

第一部分、Java程序的书写格式

- 格式一：把程序的主要内容写在主方法中

```
public class 主类 {
    public static void main(String args[]) {
        功能语句
    }
}
```

第一部分、Java程序的书写格式

- 格式二：把程序的主要内容写在主类的某个一般方法中

```
public class 主类 {  
    public static void main(String args[]) {  
        主类 实例 = new 主类();  
        实例.一般方法();  
    }  
    public void 一般方法()  
    {  
        功能语句  
    }  
}
```

第一部分、Java程序的书写格式

- 格式三：把程序的主要内容写在主类的构造方法中

```
public class 主类 {  
    public static void main(String args[]) {  
        主类 实例 = new 主类();  
    }  
    public 主类() {  
        功能语句  
    }  
}
```

第一部分、Java程序的书写格式

- 格式四：把程序的主要内容写在一个封装类中，用构造方法或一般方法封装

```
public class 主类 {  
    public static void main(String args[]) {  
        封装类 实例 = new 封装类();  
        实例.功能方法();  
    }  
}  
  
public class 封装类 {  
    功能方法() {  
        功能语句  
    }  
}
```

锁定画面

第一部分、Java程序的书写格式

- 格式五：最通用格式，接口定义，类实现，主类调用

```
public class 主类 {  
    public static void main(String args[]) {  
        封装类 实例 = new 封装类();  
        实例.功能方法();  
    }  
}  
  
public class 封装类 implements 功能接口 {  
    功能方法() {  
        功能语句  
    }  
}  
  
public interface 功能接口 {  
    功能方法();  
}
```

说点什么...

第二部分、监听器接口的实现格式

- 格式一：在主类中实现

```
public class 主类 implements XXXXListener {  
    public static void main(String[] args) {  
    }  
    public 监听方法实现{  
        实现语句  
    }  
}
```

第二部分、监听器接口的实现格式

- 格式一：在主类中实现

加载方式：

```
xxx.addXXXXListener(this);
```

第二部分、监听器接口的实现格式

- 格式二：在单独的类中实现

```
public class 主类 {  
    public static void main(String[] args) {  
    }  
}  
  
public class 封装类 implements  
    XXXXListener {  
    public 监听方法实现{ 实现语句 }  
}
```

说点什么...

第二部分、监听器接口的实现格式

- 格式二：在单独的类中实现

加载方式：

①

```
封装类 实例 = new 封装类();  
xxx.addXXXXListener(实例);
```

②

```
xxx.addXXXXListener(new 封装类());
```


第二部分、监听器接口的实现格式

- 格式三：在主类或功能类的内部类中实现

```
public class 主类 {  
    public static void main(String[] args) {  
    }  
    public class 封装类 implements  
    XXXXListener {  
        public 监听方法实现{ 实现语句 }  
    }  
}
```

第二部分、监听器接口的实现格式

- 格式三：在主类或功能类的内部类中实现

加载方式：

①

```
封装类 实例 = new 封装类();  
xxx.addXXXXListener(实例);
```

②

```
xxx.addXXXXListener(new 封装类());
```

第二部分、监听器接口的实现格式

- 格式四：在主类或功能类的匿名内部类中实现和加载

```
public class 主类 {  
    public static void main(String[] args) {  
        }  
        xxx.addXXXXListener(new XXXXListener(){  
            方法实现(){实现语句}  
        });  
    }  
}
```

第三部分、常见的典型编写错误

- 一、在主类中定义变量成员，然后在主方法中访问

初学者在编写一个程序时经常出现的错误，把在学习C语言时的习惯带到编写Java语言程序中。在主类中定义变量，然后在主方法中访问。

主方法是静态方法，不能访问非静态的变量成员，由此使程序无法通过编译。

第三部分、常见的典型编写错误

- 二、在方法体中重复定义变量成员

变量成员作为类的成员，其声明的作用域是整个类体。如果在方法体中声明了与类的变量成员同名的变量，则变量成员的作用将被覆盖，在方法体中访问到的将是在方法体中定义的局部变量。

这可能会引起数据传递的错误，使得赋值给局部变量的值保存在局部变量中，在类的其他位置访问的是变量成员，其中没有存储方法体计算的结果。

说点什么...

第三部分、常见的典型编写错误

- 三、调用方法时没有严格对应实参与形参

很多Java语言的初学者都是学过C语言的，在编写Java语言程序时，对于Java语言对方法参数的顺序和类型的要求未能深入理解，依然习惯性地忽略实参与形参的严格对应关系。

第三部分、常见的典型编写错误

- 四、GUI程序中出现了多个顶层容器实例
每个Java语言程序的图形用户界面只允许使用一个顶层容器，如果程序员在程序中使用了多个顶层容器，虽然在编译程序时不会显示错误信息，但是当程序运行时，会影响界面的显示和操作效果。所以，要在编写程序时注意避免类似的错误。

第四部分、程序设计步骤

- 一、先理清思路，找到解决方法，完成设计细节

所谓程序，是指令序列，是人对机器下达的执行任务，是完全“可执行的”任务序列。如果人还不知道怎么做，机器怎么能完成任务呢？所以，一定是找到了完善的解决方法之后，再动手录入代码。执行顺序、数据存储等等都事先考虑清楚。

用UML描述对写代码很有帮助。

第四部分、程序设计步骤

- 二、先要有细小的任务的解决方案，再构造大一级的解决方案

程序的组织遵循模块化的规律，“细小的任务”可能就是几条语句，放到大一级的结构里，就可以获得更大的功能。比如例题中“100!”那个题，就先有“辗转相除”功能的语句块，才有统计约数“2”和约数“5”的个数的语句块。

“模块化”设计思路是很值得推荐的。

定画面

第四部分、程序设计步骤

- 三、对问题的描述尽量采用最简单的方式
很多实际问题需要使用逻辑表达方式、选择方式等进行描述，尽量采用最简单、最简洁的方式来表达，可以大幅度降低出错的概率，写到程序代码中也易于检查，从而减少错误发生。

说点什么...