Java 学习笔记之一

一、环境变量设置

1、JDK 下载链接地址：

http://www.oracle.com/technetwork/java/javase/downloads/jdk-6u29-download-513648.html

下载完毕后，进行安装，比如安装到 C:\Program Files\Java\jdk1.6.0_12 目录下，安装完毕后设置环境变量，需要设置的项如下：

（1） JAVA_HOME: C:\Program Files\Java\jdk1.6.0_12

（2） PATH:C:\Program Files\Java\jdk1.6.0_12\bin

（3） CLASSPATH: .;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar;

//注意：点号、分号一定不能丢掉了。

2、开发工具 IDE 下载地址：

（1）Eclipse：http://www.eclipse.org/downloads/

（2）Netbeans: http://netbeans.org/

（3）Jcreator：http://www.jcreator.com/

3、开发工具 IDE-Eclipse 常用快捷键：

http://pinefantasy-126-com.iteye.com/blog/1290525

二、Java 基本介绍

1、Java 历史特点：

Java 自 1995 年诞生，至今有 16 年的历史，从诞生之日起，逐渐被广泛接受并成为推动了 web 迅速发展。Java 的三种核心机制为虚拟机机制、代码安全机制、垃圾回收机制。

Java 最大的特点是平台无关性（可移植性好），通过不同的虚拟机与各类操作系统无缝结合。虚拟机实例是指：执行一个应用程序的时候，虚拟机产生一个实例对象来支撑应用程序的运行。

2、类和对象：

类是对现实世界某一类事物的抽象和提取，比如现实世界中有许多各式各样的人，同时每一个具体的个体的名字、性别、身高都是差异化的，更进一步地将这些实例对象抽象成具有某些特征的一类事物，这就是类。类中包括属性、方法等。

3、面向对象语言的三大特征：

封装、继承、多态是面向对象语言的三大特征。

封装：将类的成员变量、构造方法或者成员方法私有化 private，不直接暴露给外部使用者，而是通过提供一些接口供外部调用。

继承：子类一方面通过继承父类的非私有化的成员变量、成员方法来实现自己，并能够在此基础上进一步拓展一把自己。

多态：多态涉及到的是一个类型向上转型和向下转型的问题，通过父类搞出许多子类对象，每个子类对象都有自己的特点。

三、代码实战操作

1、 编写最简单的 helloworld 程序并运行查看效果。

2、 程序的注释方法、类名、变量名、方法名命名规则学习等。

3、 Java 语言的基本数据类型（四类八种）引用数据类型：数组、接口、类。

4、 Java 语言的表达式、运算符、判断和循环语句等。

5、 数组的定义、使用，方法的声明和使用演练等操作。

四、面向对象语言基础部分

面向对象语言的三大特征是：封装、继承、多态

下面先介绍封装和继承，然后再介绍多态性。

父类：人类 Person
子类：学生类 Student
子类向父类转型为向上转型，父类向子类转型为向下转型也叫做强制转换。
代码如下：

```
/**
 * @author Administrator
 *
 * @description 父类，基类
 * @history
 */
class Person{
    private String name; // private进行封装操作
    private int age;
    // 覆写toString方法
    @Override
    public String toString() {
        return "Person[name=" + name + ",age=" + age + "]";
    }
    public Person(String name,int age){ // 全参数构造方法
        this.name = name;
        this.age = age;
    }
    public Person(){ // 无参数构造方法
        //do nothing
    }
    // getter/setter方法
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}

/**
 * @author Administrator
 *
```

```
 * @description 子类，继承自父类person
 * @history
 */
class Student extends Person{
    private String school; // 子类拓展的属性，学校
    // 覆写toString方法
    @Override
    public String toString() {
        return "Student[name =" + super.getName() + ",age = " + super.getAge()
                + ",school=" + school + "]";
    }
    public Student(String name,int age,String school){
        super(name,age); // 调用父类带两个参数的构造方法
        this.school = school;
    }
    public Student(String name){
        super(); // 调用父类无参数构造方法
        this.school = name;
    }
    // getter/setter方法
    public String getSchool() {
        return school;
    }
    public void setSchool(String school) {
        this.school = school;
    }
}

public class MyDemoTest{
    /**
     *@description
     *@param args
     */
    public static void main(String[] args){
        // 定义并实例化父类对象
        Person p = new Person("java",16);
        System.out.println(p); // 调用toString方法
        // 定义并实例化子类对象
        Student s = new Student("java",16,"sun");
        System.out.println(s); // 调用toString方法
        Person p1 = new Student("eclipse",12,"ibm"); // 实例化子类对象赋给父类对象
        Student s1 = (Student)p1; // 父类向子类转换，强制类型转换-向下转型
        System.out.println(s1);
    }
```

```
}
```

```
/**
 * @author Administrator
 *
 * @description
 * @history
 */
public class MyFirstDemo {
    // 类名、方法名、变量名等命名规则
    // 一律做到简单明了，容易明白——简单就是美
    // 1、类名首字母采用大写形式，如果是多个单词拼接采用驼峰形式
    // 2、方法变量名首字母采用小写形式，方法名一般是动+名，变量命名做到见名知意
    /**
     *@description main方法程序的入口
     *@param args 参数
     */
    public static void main(String[] args) {
        System.out.println("hello world"); // 最简单的hello world程序

        // 基本数据类型——四类八种   // 引用数据类型——接口、数组、类
        // byte,short,int,long,float,double,boolean,char
        byte b = Byte.MAX_VALUE; // 基本数据类型的包装类
        short s = Short.MAX_VALUE; // short类型的最大值
        int i = Integer.MIN_VALUE; // int类型的最小值
        long l = Long.MAX_VALUE; // long类型的最大值
        float f = 1.50f; // 定义float类型变量需要设置f
        double d = 1.50d; //定义double类型变量需要设置d
        char c = 'c'; // 定义char类型字符
        boolean boo = Boolean.TRUE; // 定义boolean类型值

        // 遵循变量先声明后使用原则
        // 类型转换，精度高的向精度低的转换为强制类型转换
        byte b1 = 25;
        int i1 = b1; // 自动转换，提高精度
        int i2 = 350;
        byte b2 = (byte)i2; // 强制类型转换，精度丢失

        int sum = 0; // 定义变量时候养成初始化变量的习惯
        byte s1 = 1;
        int s2 = 125;
        sum = s1 + s2; // byte类型会自动转换成int类型，类型提升

        // 设计模式-襄垣模式
```

```
        Integer in1 = 125;
        Integer in2 = 125; // in1==in2 true
        // 注释方式   // /* 注释部分，代码不生效*/  /** 文档注释**/
        // 表达式、运算符、判断循环分支语句和C语言类似
        // if for while do while break continue switch case以及+ - * % /等等

        // 数组的定义、使用以及方法的定义和使用
        int[] array = new int[10]; // 定义整型数组并且数组长度为10
        for (int j = 0; j < array.length; j++) {
            array[j] = j + 1; // 初始化数组元素
        }
        int tmp = 0;
        while (tmp < array.length) {
            System.out.println(array[tmp++]); // 使用输出数组元素
        }
        sayHelloWorld(); // 调用静态方法sayHelloWorld()
    }
    // 定义方法private该类内部才能够调用的方法
    // static静态方法、方法名称动+名，首字母采用小写
    private static void sayHelloWorld() {
        System.out.println("hello world");
    }
}
```

Java 学习笔记之二

一、面向对象基础部分

1、this、super 关键字

```
基类：BaseTest
public class BaseTest {
    protected String pwd; // 定义protected属性pwd
    public BaseTest(){
        // 无参数构造方法

    }
    public String sayHelloWorld(){
        return "hello world";
    }
}
/**
 * @author Administrator
 *
 * @description this,super关键字测试类
 * @history
```

```
 */
public class MyTestDemo extends BaseTest{
    // notes:测试使用而已，代码功能模拟而已
    private String username; // 定义用户昵称私有属性private修饰
    private String password;
    public MyTestDemo(String username){ // 构造方法
        this();// this()调用无参数构造方法
        this.username = username; // this.属性访问当前username属性
        // super.属性调用父类属性
        this.password = super.pwd;
    }
    public MyTestDemo(){
        // 无参数构造方法
        super(); // 调用父类无参数构造方法
    }
    public void sayHello(){
        this.sayWord(); // this.方法调用当前对象的sayWord方法
        super.sayHelloWorld(); // super.方法调用父类sayHelloWorld方法
    }
    public String sayWord(){
        return "hello this";
    }
    // this表示当前对象
    public boolean equals(Object obj){
        MyTestDemo other = (MyTestDemo)obj; // 传入的比较对象，强制转换操作
        // 简单模拟，如果对象的username属性值相同那么就认为相同
        if(this == other){
            return true;
        }
        if(other instanceof MyTestDemo){
            if(this.username.equals(other.username)){
                return true;
            }
        }
        return false;
    }
    /**
     *@description
     *@param args
     */
    public static void main(String[] args) {
        // this关键字绑定了当前该对象，通过this.属性、this.方法
        // super关键字用在子类要调用父类的属性或者方法中，通过super.属性、super.方法
        MyTestDemo obj1 = new MyTestDemo("hello-java"); // 定义并实例化对象
```

```
        obj1.sayHello(); // 调用对象的sayHello方法
        MyTestDemo obj2 = new MyTestDemo("hello-java");
        // equals方法传入的参数类型为object，传入的obj2会自动向上转型处理
        boolean result = obj1.equals(obj2);
        System.out.println(result); // true
    }

}
```

2、static、final 关键字

```
/**
 * @author Administrator
 *
 * @description final和static关键字学习
 * @history
 */
public class FinalAndStaticTestDemo {
    /**
     *@description
     *@param args
     */
    public static void main(String[] args) {
        // final关键字主要有三种场景
        // 1、final定义的变量为常量，比如基本数据类型的值一旦初始化了不能改变
        // 2、final定义的类为终结类，不能再被继承，比如string类型
        // 3、final定义的方法为终结方法，不能被覆写，否则编译器会报错
        final String str = "helloworld"; // final修饰变为常量不能改变
        //str = "hellojava"; // eclipse提示:The final local variable str cannot be
assigned.

        // 进一步思考下面两个问题
        String s1 = "hello";
        String s2 = "hello1";
        final String s = s1;
        s1 = s2;
        System.out.println(s); // hello还是hello1呢

        // 引用不可变，内容可变是这个意思吗？
        StringBuffer sb = new StringBuffer();
        sb.append("hello");
        final String fs = sb.toString();
        final StringBuffer fsb = sb;
        sb.append("world");
        System.out.println(fs);// hello还是helloworld呢
```

```
        System.out.println(fsb.toString()); // hello还是helloworld呢

        // static关键字比较类似也有下面三种场景
        // static修饰的变量为静态变量，它不同于对象/实例变量，是关联在某个类上的
        // static修饰的类为静态类
        // static修饰的方法为静态方法，它也不同于对象/实例方法，是关联在某个类上的
        // static int i = 100;
        // static class HelloWorld{ //... }
        // public static void sayHeloWorld(){ //... }
    }
    // 静态变量和对象变量
    static int i = 100; // 定义静态变量i
    int j = 100; // 定义成员变量/实例变量
    public static void sayHelloWorld(){ // 定义静态方法sayhelloworld
        System.out.println("hello world");
    }
    static class HelloWorld{ // 定义内部类
        // ...
    }

}
```

3、引用传递介绍和应用

```
/**
 * @author Administrator
 *
 * @description 引用传递的简单学习
 * @history
 */
class DemoTest{
    String ide = "eclipse"; // 为了方便访问暂不封装
}
public class MyDemoTest{
    /**
     *@description
     *@param args
     */
    public static void main(String[] args){
        // 对象属性值访问和变更
        DemoTest dt1 = new DemoTest();
        System.out.println(dt1.ide); // eclipse
        dt1.ide = "netbeans"; // 改变ide属性值
        System.out.println(dt1.ide); // netbeans
```

```
        function(dt1); // 调用方法function
        System.out.println(dt1.ide); // jcreator

        // String类赋值和变更
        String ide = "eclipse";
        function(ide);
        System.out.println(ide); // eclipse而不是netbeans
    }
    private static void function(String ide) {
        ide = "netbeans"; // 赋值为netbeans
    }
    // 测试方法function
    private static void function(DemoTest dt2) {
        dt2.ide = "jcreator"; // 改变ide属性值
    }
}
```

4、静态块、代码块、同步块

```
/**
 * @author Administrator
 *
 * @description 静态块、代码块、同步块学习测试类
 * @history
 */
class BlockDemo {
    static {
        // 在类中定义的static{ //...}为静态代码块
        // 静态代码块的程序只会被初始化一次
        System.out.println("BlockDemo->static");
    }
    {
        // 在类中定义的{ //...}为构造代码块
        // 构造代码块和对象一起初始化的
        System.out.println("BlockDemo->{}");
    }

    public BlockDemo() {
        System.out.println("BlockDemo->BlockDemo");
    }
    // 构造方法、构造代码块和静态代码块初始化顺序呢？
    // 静态块、构造块、构造方法 // 表面初步分析，具体有待研究
}
public class MyDemo {
    /**
```

```
     * @description
     * @param args
     */
    public static void main(String[] args) {
        // 静态块、代码块、同步块
        {
            // 在方法中用{ //... }定义的为普通代码块
            int x = 100;
            System.out.println(x); // 100 相当于局部变量
        }
        int x = 200;
        System.out.println(x); // 200

        new BlockDemo();
        new BlockDemo();
        // BlockDemo->static static代码块只会被初始化一次
        // BlockDemo->{}
        // BlockDemo->BlockDemo
        // BlockDemo->{}
        // BlockDemo->BlockDemo
    }
}
```

5、构造方法、匿名对象、单例定义以及实现

```
/**
 * @author Administrator
 *
 * @description 构造方法、匿名对象、单例学习测试类
 * @history
 */
public class MyDemoTest {
    private String username; // 定义私有属性username

    // 构造方法的作用是初始化操作
    // 构造方法是一种特殊的方法，构造方法也有重载
    public MyDemoTest(){
        // 无参数构造方法
    }
    public MyDemoTest(String username) { // 定义构造方法
        this.username = username; // 对属性初始化赋值操作
    }
    /**
     *@description
     *@param args
```

```
     */
    public static void main(String[] args) {
        // 实例化一个匿名对象
        // 匿名对象不在栈中开辟空间赋地址值
        new MyDemoTest("eclipse");
    }

}
/**
 * @author Administrator
 *
 * @description 单例学习测试类
 * @history
 */
class Singleton {
    private Singleton() {
        // 将构造方法私有化，外部不能直接调用
        System.out.println("private Singleton(){//...}");
    }
    // 饿汉式获取单例代码
    private static Singleton instance = new Singleton();
    // 对外提供一个getInstance方法获取单例
    public static Singleton getIntance() {
        return instance;
    }
}
class Singleton2{
    private Singleton2(){
        // 构造方法私有化，外部不能直接调用
        System.out.println("private Singleton2(){//...}");
    }
    // 懒汉式获取单例代码
    private static Singleton2 instance = null;
    private static Object lock = new Object();

    // 获取单例方法，效率比较低
    // 每次进入都要进行锁判断，而实际情况我们是第一次null比较特殊
    /*public static Singleton2 getIntance(){
        synchronized (lock) {
            if (instance == null) { // 如果为null进行实例化对象操作
                instance = new Singleton2();
            }
        }
        return instance;
```

```
    }*/
    // 上面同步代码的另外一种写法如下，效率比较低
    // public static synchronized getInstance(){ //...}

    // 对同步代码进行改写
    public static Singleton2 getIntance(){
        if(instance == null){
            synchronized(lock){
                if(instance == null){ // 进行双重检验操作
                    instance = new Singleton2();
                }
            }
            // 如果换为下面这种形式那么就可能不是产生一个实例对象了
            /*synchronized(lock){
                instance = new Singleton2();
            }*/
        }
        return instance;
    }
}
```

6、内部类、匿名内部类应用

```
class Outer{ // 定义外部类
    private String info = "helloworld" ; // 定义外部类属性
    private static String info2 = "helloeclipse"; // 定义静态变量
    class Inner{  // 定义内部类
        public void print(){
            // 内部类的好处之一：直接访问外部类属性
            System.out.println(info) ; // 直接访问外部类属性
        }
    }
    static class StaticInner{ // 通过static定义的内部类为外部类
        public void print(){
            System.out.println(info2);
        }
    }
    public void fun(){
        new Inner().print() ; // 通过内部类的实例化对象调用方法
    }
}
/**
 * @author Administrator
 *
 * @description 内部类以及匿名内部类学习测试类
 * @history
```

```
 */
public class InnerClassDemo{
    /**
     *@description
     *@param args
     */
    public static void main(String[] args){
        new Outer().fun() ; // 调用外部类的fun()方法
        // 另外一种实例化方式
        Outer out = new Outer();
        Outer.Inner in = out.new Inner();
        in.print();

        // 通过外部类.内部类实例化内部类对象
        //StaticInner sin = new Outer.StaticInner(); //需要导入外部类所在的包
        //sin.print();
    }
}
```

二、面向对象高级部分

1、继承的进一步讨论

```
class Base{
    // 定义基类
    void print(){
        System.out.println("Base->print()");
    }
    public void print1(){
        // 父类方法的访问权限，子类覆写该方法的时候不能降低权限
    }
}
class ZiLei extends Base{
    // 定义子类覆写print方法
    void print(){
        // 子类访问父类中的方法，前面super关键字内容
        // super.print();
        System.out.println("Super->print()");
    }
    /*void print1(){
        // Cannot reduce the visibility of the inherited method from Base
    }*/
    // 覆写override和重载overload区别
    // 覆写是父类子类之间的关系，重载是同一个类之间的关系
}
/**
```

```
 * @author Administrator
 *
 * @description 继承中的方法覆写
 * @history
 */
public class OverrideDemo{
    public static void main(String args[]){
        new ZiLei().print(); // 调用的内容是覆写后的方法实现
        // 另外java中只能够单继承，不同于C++，但是可以多重继承
        /*class A{
            // ...
        }
        class B extends A{  //class C extends A,B{ // ...}编译错误
            // ...
        }
        class C extends B{
            // ...
        }*/
    }
}
```

2、接口和抽象类进一步讨论

```
/**
 * @author Administrator
 *
 * @description 接口测试类
 * @history
 */
interface Interface{
    public static final String INFO = "helloeclipse";
    public void print(); // 定义方法
    public abstract void printInfo();
}
class I implements Interface{
    public void print() {
        // 方法实现
        System.out.println(INFO); // helloeclipse
    }
    public void printInfo() {
        // ...
    }
}
public class InterfaceTestDemo {
    /**
```

```
     *@description
     *@param args
     */
    public static void main(String[] args) {
        // 接口interface
        // 接口interface和抽象类abstract一样不能直接实例化必须通过子类来实现
        new Interface(){
            public void print() {
                // 匿名内部类，方法实现
                System.out.println(INFO); // helloeclipse
            }
            public void printInfo() {
                // ...
            }
        };
        // 接口和抽象类对比，接口更像一种规范、抽象类更像一种模板
        // 抽象类定义一些公共的实现，以及一些未实现的方法交给子类具体实现特定的功能-模板方法
    }
}
/**
 * @author Administrator
 *
 * @description 抽象类学习测试类
 * @history
 */
abstract class Abstract{ // 定义抽象类
    public final static String INFO = "helloeclipse"; // 定义全局变量/常量
    abstract public void print(); // 定义抽象方法
    // 抽象类不能直接实例化，抽象类的子类如果不是抽象类那么一定要实现所有的抽象方法
}
class AbstractZiLei extends Abstract{ // 定义抽象子类
    public void print() { // 实现子类方法
        System.out.println(INFO);
    }
}
public class AbstractTestDemo {
    /**
     *@description
     *@param args
     */
    public static void main(String[] args) {
        // 接口和抽象类interface,abstract
        // 抽象类：含有抽象方法的类叫做抽象类用abstract修饰的
        new Abstract(){
```

```
            public void print() {
                System.out.println(INFO); // 匿名抽象类实现
            }
        };
        // 抽象类能不能被final修饰呢?final定义的类为终结类，抽象类是需要子类来实现的
        // final abstract class Abstract{ //... } // 编译报错
    }
}
```

3、object 类和包装类进一步讨论

```
/**
 * @author Administrator
 *
 * @description Object类学习测试类
 * @history
 */
public class ObjectTestDemo {
    /**
     *@description
     *@param args
     * @throws ClassNotFoundException
     */
    public static void main(String[] args) throws ClassNotFoundException {
        // Object类学习测试代码
        // 1、toString方法
        // 2、equals和hashCode方法
        // 3、getClass方法
        ObjectTestDemo otd = new ObjectTestDemo();
        System.out.println(otd);
        System.out.println(otd.toString());// 现实调用toString方法
        /*
         * Object类中toString方法
         * public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
        }
        * Object类中equal方法
        * public boolean equals(Object obj) {
        return (this == obj);
        }*/
        // 定义昵称和年龄属性，如果昵称和年龄相同则equal方法返回true
        ObjectTestDemo otd1 = new ObjectTestDemo();
        otd1.nickname = "eclipse";
        otd1.age = 20;
        ObjectTestDemo otd2 = new ObjectTestDemo();
```

```
        otd2.nickname = "eclipse";
        otd2.age = 20;
        System.out.println(otd1==otd2); // false
        System.out.println(otd1.equals(otd2)); // 如果不覆写的话返回的一直是false

        // hashCode方法和equals方法的关系，没有绝对的关系
        // 具体参看源代码的注释说明，在实际用中尽量同时覆写这两个方法，保持一致性

        // public final native Class<?> getClass();
        // getClass方法是一个本地方法，调用底层代码，返回当前对象对应的一个类对象实例
        // 类对象实例，理解为内存中对应的那份字节码对象，java反射内容初步学习
        // 通过字节码对象构造出一个个的对象实例
        Class claz = otd.getClass();
        Class claz1 = ObjectTestDemo.class;
        Class claz2 = Class.forName("ObjectTestDemo"); // 类的完整路径
        System.out.println(otd.getClass()); // class ObjectTestDemo
        // VM内存中只会产生一份字节码对象
        System.out.println(claz == claz1); // true

    }
    // 方法模拟，添加两个属性昵称和年龄
    private String nickname;
    private int age;

    // 注意参数类型不要写成了ObjectTestDemo
    public boolean equals(Object obj) {
        ObjectTestDemo otd = (ObjectTestDemo) obj; // 向上转型，强制转换
        if (this == otd) {
            return true;
        }
        if (otd instanceof ObjectTestDemo) {
            if (this.nickname.equals(otd.nickname) && this.age == otd.age) {
                return true;
            }
        }
        return false;
    }
    public String toString(){
        return "helloworld"; //覆写toString方法，返回helloworld
    }

}
```

一、Java 异常处理机制

```
/**
 * @author Administrator
 *
 * @description 异常学习测试类
 * @history
 */
public class ExceptionDemo {
    /**
     *@description
     *@param args
     */
    public static void main(String[] args) {
        // Throable类是所有错误和异常的根基类
        // Throable类下两个重要的子类Exception和Error

        // 1、编写一个常见的异常例子
        try {
            int i = 1;
            int j = 0;
            int r = i / j;
        } catch (ArithmeticException ae) {
            ae.printStackTrace();
            // Exception in thread "main" java.lang.ArithmeticException: / by zero
        }

        // 2、catch多个种类的异常例子
        String s1 = "1";
        String s2 = "0"; // String s2 = "eclipse";
        try{
            int i1 = Integer.parseInt(s1); // 字符串解析成数字
            int i2 = Integer.parseInt(s2);
            int temp = i1/i2;
        } catch(ArithmeticException ae){
            ae.printStackTrace(); // 分母为0异常捕获
        } catch(NumberFormatException nfe){
            nfe.printStackTrace(); // 数字格式转换异常捕获
        }

        // 3、异常可以往上抛出去例子
        int[] array = new int[5];
        try{
            int i3 = array[5]; // 数组越界异常
        } catch(ArrayIndexOutOfBoundsException aiobe){
```

```
            // 捕获异常后往外抛出
            // 在某一层统一处理掉这些异常，比如可以采用拦截器
            throw aiobe;
        }

        // 4、自定义异常类，继承自Exception类
        try{
            // 实例化内部类对象比较特殊点
            ExceptionDemo out = new ExceptionDemo(); // 先实例化外部类对象
            throw out.new MyException("hello-exception"); // 再实例化内部类对象
        } catch(MyException me){
            // hello-exception
            System.out.println(me.getLocalizedMessage());
        }
    }
    // 自定义异常类
    class MyException extends Exception{
        public MyException(String msg){
            super(msg);
        }
    }

}
```

二、Java 多线程编程

```
/**
 * @author Administrator
 *
 * @description Java多线程编程入门测试类
 * @history
 */
// 方法一、继承线程类Thread
class MyThread extends Thread{
    public MyThread(String threadName){ // 设置当前线程的名称
        currentThread().setName(threadName);
    }
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
}
// 方法二、实现Runnable接口
class MyThread2 implements Runnable{
    public void run() {
```

```
        System.out.println(Thread.currentThread().getName());
    }
}
public class SynTestDemo {
    /**
     *@description
     *@param args
     */
    public static void main(String[] args) {
        // 1、线程的定义、线程和进程的区别、为什么要引入线程等
        // 2、Java实现多线程的方法主要有两种：继承Thread类和实现Runnable接口
        // 3、多个线程并发同时访问公共的临界资源的时候需要进行同步处理，过多的同步不当会造成死锁
问题

        MyThread t1 = new MyThread("hello-thread");
        t1.start(); // 启动线程1

        MyThread2 t2 = new MyThread2();
        new Thread(t2).start(); // 启动线程2
    }

}
/**
 * @author Administrator
 *
 * @description 多线程编程演练例子
 * @history
 */
public class SynABCTest {
    /**
     * @description
     * @param args
     */
    public static void main(String[] args) {
        // 通过具体的例子来加深对多线程的认识
        // 问题为：循环打印10遍ABCABC...ABC
        PrintThread p1 = new PrintThread(0, "A"); // 参数为线程标志和打印的内容
        PrintThread p2 = new PrintThread(1, "B");
        PrintThread p3 = new PrintThread(2, "C");
        // 启动线程A B C
        new Thread(p1).start();
        new Thread(p2).start();
        new Thread(p3).start();
    }
```

```
}
// 采用实现接口的方式定义线程类
class PrintThread implements Runnable {
    // 标记执行当前应该执行的线程0、1、2依次表示A B C
    // 定义成静态变量，因为线程各自使用独立的栈
    private static int index = 0;
    private static Object lock = new Object();
    private int key = 0; // 线程标志
    private int print = 0; // 打印的次数
    private String name; // 打印的内容

    public PrintThread(int key, String name) {
        this.key = key;
        this.name = name;
    }

    public void run() {
        while (this.print < 10) { // 打印的次数
            synchronized (lock) {
                while (!(this.key == index % 3)) { // 从0开始执行
                    try {
                        lock.wait(); // 阻塞掉
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.print(this.name); // 打印出内容
                this.print++; // 当前线程打印次数++
                index++; // 线程切换下一个
                lock.notifyAll(); // 唤醒其他等待的线程
            }
        }
    }
}
/**
 * @author Administrator
 *
 * @description 死锁模拟测试类
 * @history
 */
public class DeadLockTest {
    /**
     *@description
     *@param args
```

```
     */
    public static void main(String[] args) {
        // 过多的同步操作可能会造成死锁问题，死锁产生的原因是形成了环路等待
        // 通过两个线程对象进行模拟，线程A完成一个操作需要资源1和资源2，线程B也是一样
        // 在资源分配的过程中，线程A占用了资源1，等待资源2，此时此刻线程B占用了资源2，等待资源1
        DeadThread dt1 = new DeadThread("1", true);
        DeadThread dt2 = new DeadThread("2", false);
        new Thread(dt1).start(); // 启动线程1
        new Thread(dt2).start(); // 启动线程2
    }
    // 定义静态内部类、类似外部类了
    static class DeadThread implements Runnable{
        /**
         * 定义资源1和资源2 lock1和lock2
         */
        private static Object lock1 = new Object();
        private static Object lock2 = new Object();
        private String name; // 线程名称
        private boolean run; // 执行顺序标记
        public DeadThread(String name,boolean run){
            this.name = name;
            this.run = run;
        }
        @Override
        public void run() {
            if(this.run){
                // 线程1先占用资源1
                synchronized(lock1){
                    try {
                        System.out.println("thread1 used lock1");
                        Thread.sleep(3000); // 暂时休眠3秒
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    // 线程1再去申请资源2，此时资源2已经被线程2占用着不放了
                    synchronized(lock2){
                        System.out.println("hello dead-lock");
                    }
                }
            }else{
                // 线程2先占用资源2
                synchronized(lock2){
                    try {
                        System.out.println("thread2 used lock2");
```

```
                    Thread.sleep(3000); // 线程2暂时休眠3秒
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // 线程2再去申请资源1，此时资源1已经被线程1占用着不放了
                synchronized(lock1){
                    System.out.println("hello dead-lock");
                }
            }
        }
    }

}
class MyThread1 implements Runnable {
    private boolean flag = true; // 定义标志位
    public void run() {
        int i = 0;
        while (this.flag) {
            System.out.println(Thread.currentThread().getName() + "运行，i = "
                    + (i++));
        }
    }
    public void stop() {
        this.flag = false; // 修改标志位
    }
}

/**
 * @author Administrator
 *
 * @description 通过修改标记位停止线程
 * @history
 */
public class ThreadStopDemo {
    public static void main(String[] args) {
        MyThread1 my = new MyThread1();
        Thread t = new Thread(my, "线程"); // 建立线程对象
        t.start(); // 启动线程
        try {
            Thread.sleep(50); // 适当地延迟下
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
        my.stop(); // 修改标志位，停止运行
    }
}
```

三、Java 常用类库

```
/**
 * @author Administrator
 *
 * @description Java常用类-StringBuffer学习
 * @history
 */
public class StringBufferTest {
    /**
     *@description
     *@param args
     */
    public static void main(String[] args) {
        // StringBuffer类在处理字符串时候比较常用
        // 1、StringBuffer类的append方法
        // 具体的方法参数个数和类型，请参看JDK的API即可
        StringBuffer sb = new StringBuffer();
        sb.append("helloworld"); // string类型
        sb.append("\n"); // 特殊符号，换行符
        sb.append(false); // boolean类型
        sb.append('j'); // char类型
        sb.append(1.50d); // double类型
        // ... 等等
        sb.insert(0, "eclipse"); // 在index出插入值
        sb.reverse(); // 反转操作
        sb.replace(1, 3, "helloeclipse"); // 替换操作
        sb.substring(1, 5); // 字符串截取操作
        sb.delete(0, 1); // 删除操作
        sb.indexOf("hello"); // index出现的位置

        // 2、StringBuffer类的引用传递
        StringBuffer sb1 = new StringBuffer();
        sb1.append("hello");
        fun(sb1);
        System.out.println(sb1.toString()); // helloworld

        // 3、StringBuffer类和String类的区别
        // 一个可变、一个不可变，具体选择根据具体的场景
    }
    private static void fun(StringBuffer sb1) {
```

```
        sb1.append("world"); // 改变对应的值
    }
}
import java.io.IOException;

/**
 * @author Administrator
 *
 * @description Runtime类学习测试
 * @history
 */
public class RuntimeTest {
    /**
     * @description
     * @param args
     */
    public static void main(String[] args) {
        // Runtime类是一个封装了JVM进程的类，每一个java应用程序都有一个JVM实例来支持
        // 因此每个JVM进程对应一个Runtime类的实例对象，由java虚拟机来实例化对象
        // 查看源代码发现，其构造方法被私有化了，类似单例模式
        /*
         * public class Runtime {
          private static Runtime currentRuntime = new Runtime();
          public static Runtime getRuntime() {
              return currentRuntime;
          }
           private Runtime() {} }
         */
        Runtime run = Runtime.getRuntime(); // 通过静态方法获取实例对象
        // 1、查看一些JVM内存级别的参数
        System.out.println(run.maxMemory()); // 最大内存空间
        System.out.println(run.freeMemory()); // 空间的内存空间
        String str = "";
        for (int i = 0; i < 10000; i++) {
            str += i;
        }
        System.out.println(run.freeMemory()); // 空间的内存空间
        run.gc(); // 进行垃圾回收处理
        System.out.println(run.freeMemory());

        // 2、Runtime类一般和Process类一起使用，可以打开本机的一些进程
        Process p = null; // 定义进程变量
        try {
            p = run.exec("notepad.exe"); // 打开记事本程序
```

```
            /*public Process exec(String command) throws IOException {
            return exec(command, null, null);
            }*/
            // 底层有个ProcessBuilder类处理执行这些命令
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            Thread.sleep(5000); // 让记事本程序执行5秒后关闭掉
        } catch (Exception e) {
        }
        p.destroy(); // 结束此进程
    }

}
import java.io.PrintStream;

class Demo{
    public Demo(){}
    // 覆写该方法，测试System调用gc方法的过程
    protected void finalize() throws Throwable {
        System.out.println("hello-finalize");
    }
}
import java.io.PrintStream;

class Demo{
    public Demo(){}
    // 覆写该方法，测试System调用gc方法的过程
    protected void finalize() throws Throwable {
        System.out.println("hello-finalize");
    }
}
public class SystemTest {
    /**
     *@description
     *@param args
     */
    public static void main(String[] args) {
        // System类和上面说到的Runtime类一样也是比较靠近虚拟机实例的类
        // 1、我们经常使用的方式是打印输出操作System.out.println("hello world");
        // PrintStream extends FilterOutputStream extends OutputStream
        PrintStream out = System.out; // 获取打印流
        out.println("helloworld"); // 打印输出
```

```
        // 2、通过该类获取系统时间操作
        // public static native long currentTimeMillis();
        long current = System.currentTimeMillis();
        System.out.println(current); // 毫秒级别

        // 3、查看系统的属性值情况
        System.getProperties().list(out);
        // java.runtime.name=Java(TM) SE Runtime Environment
        // sun.boot.library.path=C:\Program Files\Java\jre6\bin
        // java.vm.version=20.1-b02
        // java.vm.vendor=Sun Microsystems Inc.
        // java.vendor.url=http://java.sun.com/
        // ...等等系统值

        // Properties extends Hashtable<Object,Object>
        // Key-Value的键值对形式，实现了Map接口的类
        System.out.println(System.getProperty("os.name")); // Windows 7
        /*public static String getProperty(String key) {
        checkKey(key);
        SecurityManager sm = getSecurityManager();
               if (sm != null) {
            sm.checkPropertyAccess(key);
        }
        return props.getProperty(key);
        }*/

        // 4、释放内存空间方法调用
        // 在之前的笔记中说到了Object类，其中有个finalize方法
        // protected void finalize() throws Throwable { }
        Demo demo = new Demo();
        demo = null; // 断开引用设置为null
        System.gc(); // 强制性/显示释放内存空间，打印输出hello-finalize
        // 调用的是Runtime类的释放方法
        /*public static void gc() {
        Runtime.getRuntime().gc();
        }*/
    }
}
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
```

```
 * @author Administrator
 *
 * @description 时间帮助工具类
 * @history
 */
public class DateHelperUtil {
    /**
     * @description 获取当前时间的字符串格式
     * @return
     */
    public static String getNowDate() {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
        return sdf.format(new Date());
    }

    /**
     *@description 对传入的date类型时间转换成字符串格式的时间
     *@param date
     *@return 返回字符串格式时间
     */
    public static String formatDate(Date date){
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
        return sdf.format(date);
    }

    /**
     *@description 对传入的date类型时间转换成字符串格式的时间
     *@param date
     *@param formatStr 格式模板
     *@return 返回字符串格式时间
     */
    public static String formatDate(Date date,String formatStr){
        SimpleDateFormat sdf = new SimpleDateFormat(formatStr);
        return sdf.format(date);
    }

    /**
     *@description 对传入的字符串格式的时间进行解析处理成date类型
     *@param dateStr
     *@return
     * @throws ParseException 解析错误异常
     */
    public static Date parseDate(String dateStr) throws ParseException{
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
```

```
        return sdf.parse(dateStr);
    }

    /**
     *@description 对传入的字符串格式的时间进行解析处理成date类型
     *@param dateStr
     *@param formatStr 解析字符串的时间模板
     *@return
     *@throws ParseException 解析错误异常
     */
    public static Date parseDate(String dateStr, String formatStr) throws
ParseException {
        SimpleDateFormat sdf = new SimpleDateFormat(formatStr);
        return sdf.parse(dateStr);
    }

    /**
     *@description 获取当前时间的时间戳
     *@return
     */
    public static String getTimeStamp() {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMddHHmmssSSS");
        return sdf.format(new Date());
    }

}
import java.math.BigInteger;

/**
 * @author Administrator
 *
 * @description BigInteger大数学习测试类
 * @history
 */
public class BigIntegerDemo01 {
    /**
     *@description
     *@param args
     */
    public static void main(String args[]) {
        BigInteger bi1 = new BigInteger("123456789"); // 声明BigInteger对象
        BigInteger bi2 = new BigInteger("987654321"); // 声明BigInteger对象
        System.out.println("加法操作: " + bi2.add(bi1)); // 加法操作
        System.out.println("减法操作: " + bi2.subtract(bi1)); // 减法操作
```

```
        System.out.println("乘法操作：" + bi2.multiply(bi1)); // 乘法操作
        System.out.println("除法操作：" + bi2.divide(bi1)); // 除法操作
        System.out.println("最大数：" + bi2.max(bi1)); // 求出最大数
        System.out.println("最小数：" + bi2.min(bi1)); // 求出最小数
        BigInteger result[] = bi2.divideAndRemainder(bi1); // 求出余数的除法操作
        System.out.println("商是：" + result[0] + "；余数是：" + result[1]);
    }
}
class Student implements Comparable<Student> { // 指定类型为Student
    private String name;
    private int age;
    private float score;

    public Student(String name, int age, float score) {
        this.name = name;
        this.age = age;
        this.score = score;
    }

    public String toString() {
        return name + "\t\t" + this.age + "\t\t" + this.score;
    }

    public int compareTo(Student stu) { // 覆写compareTo()方法，实现排序规则的应用
        if (this.score > stu.score) {
            return -1;
        } else if (this.score < stu.score) {
            return 1;
        } else {
            if (this.age > stu.age) {
                return 1;
            } else if (this.age < stu.age) {
                return -1;
            } else {
                return 0;
            }
        }
    }
}

public class ComparableDemo01 {
    public static void main(String[] args) {
        Student stu[] = { new Student("张三", 20, 90.0f),
                new Student("李四", 22, 90.0f), new Student("王五", 20, 99.0f),
```

```
                new Student("赵六", 20, 70.0f), new Student("孙七", 22, 100.0f) };
        java.util.Arrays.sort(stu); // 进行排序操作
        for (int i = 0; i < stu.length; i++) { // 循环输出数组中的内容
            System.out.println(stu[i]);
        }
    }
}
class BinaryTree {
    class Node { // 声明一个节点类
        private Comparable data; // 保存具体的内容
        private Node left; // 保存左子树
        private Node right; // 保存右子树

        public Node(Comparable data) {
            this.data = data;
        }

        public void addNode(Node newNode) {
            // 确定是放在左子树还是右子树
            if (newNode.data.compareTo(this.data) < 0) { // 内容小，放在左子树
                if (this.left == null) {
                    this.left = newNode; // 直接将新的节点设置成左子树
                } else {
                    this.left.addNode(newNode); // 继续向下判断
                }
            }
            if (newNode.data.compareTo(this.data) >= 0) { // 放在右子树
                if (this.right == null) {
                    this.right = newNode; // 没有右子树则将此节点设置成右子树
                } else {
                    this.right.addNode(newNode); // 继续向下判断
                }
            }
        }

        public void printNode() { // 输出的时候采用中序遍历
            if (this.left != null) {
                this.left.printNode(); // 输出左子树
            }
            System.out.print(this.data + "\t");
            if (this.right != null) {
                this.right.printNode();
            }
        }
```

```
    }

    private Node root; // 根元素

    public void add(Comparable data) { // 加入元素
        Node newNode = new Node(data); // 定义新的节点
        if (root == null) { // 没有根节点
            root = newNode; // 第一个元素作为根节点
        } else {
            root.addNode(newNode); // 确定是放在左子树还是放在右子树
        }
    }

    public void print() {
        this.root.printNode(); // 通过根节点输出
    }
}

public class ComparableDemo02 {
    public static void main(String[] args) {
        BinaryTree bt = new BinaryTree();
        bt.add(8);
        bt.add(3);
        bt.add(3);
        bt.add(10);
        bt.add(9);
        bt.add(1);
        bt.add(5);
        bt.add(5);
        System.out.println("排序之后的结果：");
        bt.print();
    }
}
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;

/**
 * @author Administrator
 *
 * @description 任务调度程序学习类
 * @history
 */
```

```
class MyTask extends TimerTask{ // 任务调度类都要继承TimerTask
    public void run(){
        SimpleDateFormat sdf = null ;
        sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS") ;
        System.out.println("当前系统时间为: " + sdf.format(new Date())) ;
    }
}

public class TashTestDemo {
    /**
     * @description
     * @param args
     */
    public static void main(String args[]) {
        Timer t = new Timer(); // 建立Timer类对象
        MyTask mytask = new MyTask(); // 定义任务
        t.schedule(mytask, 1000, 2000); // 设置任务的执行，1秒后开始，每2秒重复
    }
}
```

Java 学习笔记之四

一、Java IO 编程

1、基本概念

Java 中对文件的操作是以流的方式进行的，流是 Java 内存中一组有序数据序列。Java 将数据从源（文件、内存、键盘、网络）读入到内存中，形成了流，然后还可以将这些流写到另外的目的地（文件、内存、控制台、网络）之所以叫做流，是因为这个数据序列在不同时刻所操作的是源的不同部分。

2、流的分类

流的分类方式一般有以下三种：

（1） 输入的方向分：输入流和输出流，输入和输出的参照对象是 Java 程序。

（2） 处理数据的单位分：字节流和字符流，字节流读取的最小单位是一个字节。

（3） 功能的不同分：节点流和处理流，一个是直接一个是包装的。

3、流分类的关系

流分类的根源来自四个基本的类，这四个类的关系如下：

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

4、其他知识补充

（1）什么是 IO

IO（Input/Output）是计算机输入/输出的接口，Java 的核心库 java.io 提供了全面的 IO 接口，包括：文件读写、标准设备输出等等。Java 中的 IO 是以流为基础进行输入输出的，所有数据被串行化写入输出流，或者从输入流读入。

（2）流 IO 和块 IO

此外，Java 也对块传输提供支持，在核心库 java.nio 中采用的便是块 IO，流 IO 和块

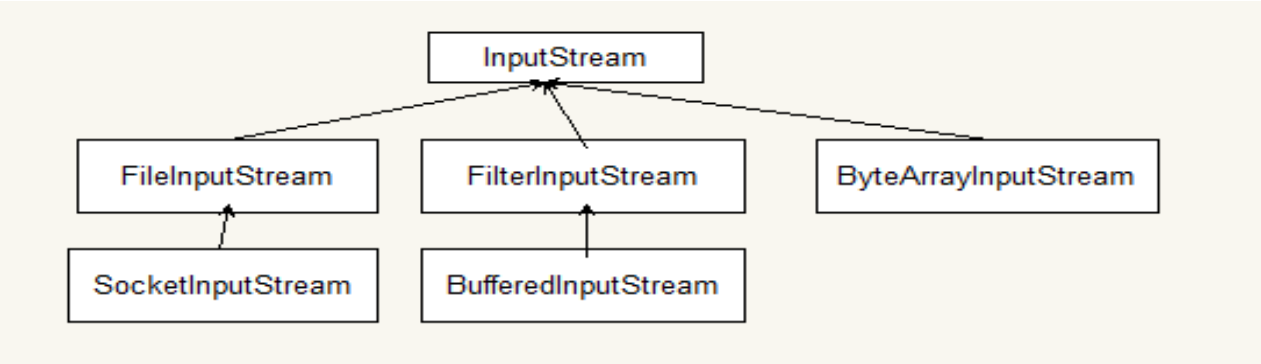
IO 对比而言，流 IO 的好处是简单易用，缺点是效率不如块 IO；相反块 IO 是效率比较高但是编程比较复杂。Java 的 IO 模型设计非常优秀，它使用了 Decorator 模式，按照功能进行划分 stream，编程过程中可以动态地装配这些 stream，以便获取所需要的功能。

备注：以上资料提取自百度文库，链接地址如下：

http://wenku.baidu.com/view/9aa0ec35eefdc8d376ee3280.html

5、代码模拟实战

类结构图示：

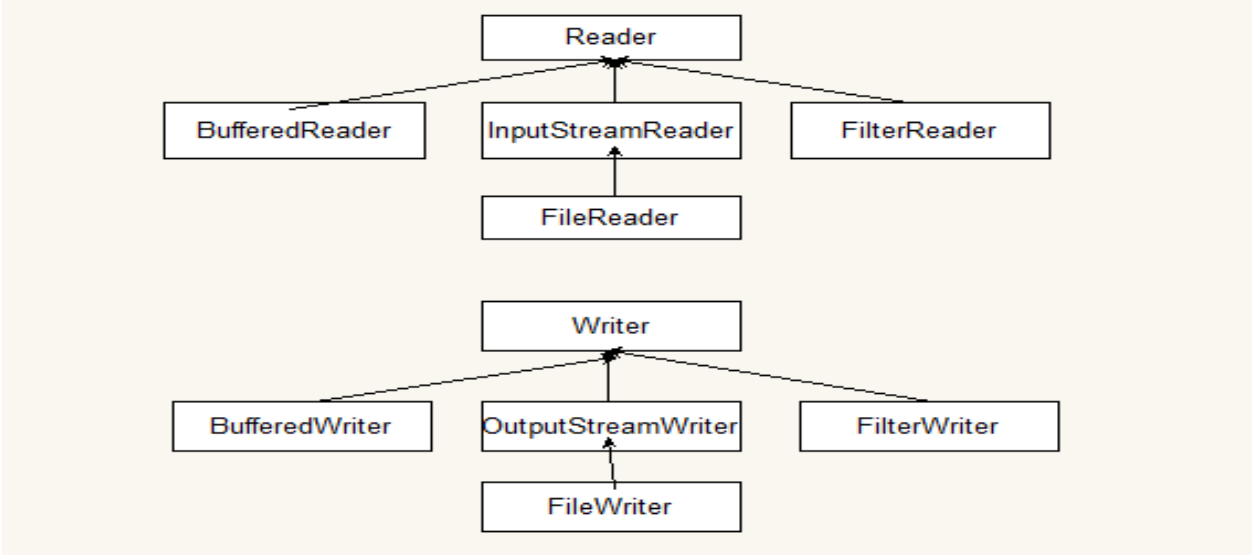


基础类 File

```
import java.io.File;
import java.io.IOException;

/**
 * @author Administrator
 *
 * @description 基础File类操作
 * @history
 */
public class FileDemoTest {
    /**
     * @description
     * @param args
     * @throws IOException IO异常处理
     */
    public static void main(String args[]) throws IOException {
        // Windows系统下的文件目录格式，这个和Unix系统等不同
        // String pathSeparator = File.pathSeparator; // ;
        // String separator = File.separator; // \

        // 1、创建和删除文件
        File file = new File("d:\\helloworld.txt");
        // File file = new File("d:"+File.separator+"helloworld.txt");
        file.createNewFile(); // 创建文件
        if (file.exists()) { // 如果文件存在，则删除文件
            file.delete();
        }
```

```
        // 2、创建文件夹操作
        File file2 = new File("d:\\helloworld");
        // File file2 = new File("d:"+File.separator+"helloworld");
        file2.mkdir(); // 建立文件夹
        if (file2.isDirectory()) {
            System.out.println("hello-directory"); // 判断是否是目录
        }

        // 3、遍历文件或者文件夹操作
        File file3 = new File("d:\\");
        // File file3 = new File("d:"+File.separator);
        File files[] = file3.listFiles(); // 列出全部内容
        for (int i = 0; i < files.length; i++) {
            System.out.println(files[i]);
        }
    }
}
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.FileInputStream;

/**
 * @author Administrator
 *
 * @description InputStream类学习测试代码
 * @history
 */
public class InputStreamDemo {
    /**
     * @description 字节流：输入流InputStream
     * @param args
     * @throws IOException
     */
    public static void main(String args[]) throws IOException {
        // 字节流的输入流和输出流过程步骤比较固定
        // 第1步、使用File类找到一个文件
        File f = new File("d:" + File.separator + "helloworld.txt");
        // 第2步、通过子类实例化父类对象(InputStream为抽象类，本身不能直接实例化)
        InputStream input = new FileInputStream(f);
        // 第3步、进行读操作
        byte b[] = new byte[(int)f.length()]; // 数组大小由文件大小来确定
        for (int i = 0; i < b.length; i++) {
```

```
            b[i] = (byte) input.read(); // 读取内容
        }
        // 第4步、关闭输出流
        input.close();
    }
}
/**
 * @author Administrator
 *
 * @description
 * @history
 */
public class OutputStreamDemo {
    /**
     *@description 字节流：输出流OutputStream类
     *@param args
     *@throws Exception
     */
    public static void main(String args[]) throws IOException {
        // 输入和输出流参考的是Java程序，输出流操作步骤也比较固定
        // 第1步、使用File类找到一个文件
        File f = new File("d:" + File.separator + "helloworld.txt");
        // 第2步、通过子类实例化父类对象
        OutputStream out = new FileOutputStream(f);
        // 第3步、进行写操作
        String str = "say hello world!!!";
        byte b[] = str.getBytes();
        for(int i=0;i<b.length;i++){
            out.write(b[i]); // 单个写入
        }
        // out.write(b);
        // 重载方法write
        // public abstract void write(int b) throws IOException;
        // public void write(byte b[]) throws IOException {}
        // 第4步、关闭输出流
        out.close(); // 关闭输出流
    }
}
```




```
import java.io.File;
import java.io.IOException;
import java.io.Reader;
import java.io.FileReader;

/**
 * @author Administrator
 *
 * @description 字符流：输入流Reader
 * @history
 */
public class ReaderDemo {
    /**
     *@description 字节流和字符流按照处理数据的单位划分
     *@param args
     *@throws IOException IO异常处理
     */
    public static void main(String args[]) throws IOException {
        // 第1步、使用File类找到一个文件
        File f = new File("d:" + File.separator + "helloworld.txt");
        // 第2步、通过子类实例化父类对象
        Reader input = new FileReader(f);
        // 第3步、进行读操作
        char c[] = new char[1024];
        int temp = 0;
        int len = 0;
        while ((temp = input.read()) != -1) {
            // 如果不是-1就表示还有内容，可以继续读取
            c[len] = (char) temp;
            len++;
```

```
        }
        // 第4步、关闭输出流
        input.close();
    }
}
import java.io.File;
import java.io.IOException;
import java.io.Writer;
import java.io.FileWriter;

/**
 * @author Administrator
 *
 * @description
 * @history
 */
public class WriterDemo {
    /**
     *@description 通过代码学习发现，基本上步骤一致的
     *@param args
     *@throws IOException
     */
    public static void main(String args[]) throws IOException {
        // 第1步、使用File类找到一个文件
        File f = new File("d:" + File.separator + "helloworld.txt");
        // 第2步、通过子类实例化父类对象
        Writer out = new FileWriter(f); // 通过对象多态性，进行实例化
        // 第3步、进行写操作
        String str = "\nsay hello\tworld";
        out.write(str);
        // 第4步、关闭输出流
        // out.flush(); // 清空缓冲
        out.close(); // 关闭输出流
    }
}
import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.Reader;
```

```java
import java.io.Writer;

public class ByteAndCharStreamTest {
    /**
     *@description  字节流：InputStream,OutputStream
     *              字符流：Reader,Writer
     *             // 字节流和字符流相互转换测试
     *@param args
     * @throws IOException 文件操作异常处理
     */
    public static void main(String[] args) throws IOException {
        // 1、输入流-字节流转换成字符流
        // 通过InputStreamReader类来进行转换
        InputStream is = new FileInputStream("d:\\helloworld.txt");
        Reader reader = new InputStreamReader(is); // 将字节流转换成字符流
        char c[] = new char[1024];
        int len = reader.read(c); // 读取操作，保存在字符数组中
        reader.close(); // 关闭操作
        System.out.println(new String(c,0,len)); // 字符数组转换成String类实例

        // 2、输出流-字节流转换成字符流
        // 通过OutputStreamWriter类来进行转换
        File f = new File("d:" + File.separator + "helloworld.txt");
        Writer out = new OutputStreamWriter(new FileOutputStream(f)); // 字节流变为
字符流
        out.write("hello world!!!"); // 使用字符流输出
        out.close();

        // 3、从字符流到字节流转换可以采用String类提供的操作
        // 从字符流中获取char[]，转换成String实例然后再调用String API的getBytes()方法
        // 接1中的代码如下，最后通过ByteArrayInputStream即可完成操作
        String str = new String(c,0,len);
        byte b[] = str.getBytes();
        InputStream is2 = new ByteArrayInputStream(b);
        is2.close();

        // 4、其他常见的流
        // 内存操作流ByteArrayInputStream,ByteArrayOutputStream
        // 管道流PipedOutputStream,PipedInputStream
        // 打印流PrintStream
        // 缓存流BufferedReader
        // ...等等
    }
}
```

字符编码学习代码

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;

public class CharSetDemo {
    /**
     * @description 字符编码学习测试方法
     * @param args
     * @throws IOException 文件IO异常处理
     */
    public static void main(String[] args) throws IOException {
        // 1、字符编码，通过system类来获取
        String fe = System.getProperty("file.encoding");
        System.out.println(fe); // GBK

        // 2、进行转码操作
        OutputStream out = new FileOutputStream("d:\\helloworld.txt");
        byte b[] = "Java，你好！！！".getBytes("ISO8859-1"); // 转码操作
        out.write(b); // 保存
        out.close(); // 关闭

    }
}
```

对象序列化学习代码

对象序列化：将对象转换成二进制的字节流，反序列化刚好相反。

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.OutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Demo implements Serializable{ // 实现序列化接口
    // 序列化方式之一，Java自身提供的序列化支持
    // 其他序列化方式，待以后有需要进一步学习
    private static final long serialVersionUID = 1L;
    private String info; // 定义私有属性

    // 如果某个属性不想被序列化，采用transient来标识
    //private transient String noser;

    public Demo(String info){
```

```
        this.info = info;
    }
    public String getInfo() {
        return info;
    }
    public void setInfo(String info) {
        this.info = info;
    }
    public String toString() {
        return "info = " + this.info;
    }
}
public class SerializedDemo {
    /**
     *@description 对象序列化、反序列化操作
     *@param args
     *@throws Exception
     */
    public static void main(String[] args) throws Exception {
        // 1、对象序列化是将内存中的对象转换成二进制形式的数据流
        // 2、对象反序列化刚好和对象序列化方向相反，将二进制数据流转换成对象

        // a、对象序列化采用ObjectOutputStream类
        ObjectOutputStream oos = null; // 声明对象输出流
        OutputStream out = new FileOutputStream("d:\\helloworld.txt");
        oos = new ObjectOutputStream(out);
        oos.writeObject(new Demo("helloworld"));
        oos.close();

        // b、对象反序列化采用ObjectInputStream类
        ObjectInputStream ois = null; // 声明对象输入流
        InputStream input = new FileInputStream("d:\\helloworld.txt");
        ois = new ObjectInputStream(input); // 实例化对象输入流
        Object obj = ois.readObject(); // 读取对象
        ois.close();
        System.out.println(obj);
    }
}
```

二、Java 类集合框架

比较详细的笔记：

http://wenku.baidu.com/view/52cf133f5727a5e9856a6186.html

一个思考题目：如何实现一个 stack，使得增加、删除、获取最大值、最小值以及中值的时间操作效率相当。

```
/**
```

```
 * @author Administrator
 *
 * @description 如何从java类集框架中选取适当的数据结构呢???
 * @history
 */
public interface Stack<T> {
    public void add(T t); // 添加元素
    public void delete(T t); // 删除元素
    public T getMax(); // 获取最大值
    public T getMin(); // 获取最小值
    public T getMiddle(); // 获取第中间大值
}
```