《高级程序设计 A》（JAVA）讲义

1、JAVA 基础概念：

（1）类、对象：以类的方式组织代码、以对象的方式组织数据。同类对象的抽象→类，类是对象的模板。

（2）变量：存储数据的单元，name、type、value 三部分。

Name：标识符的规则、字符集（iso8859-1、BIG5、GB2312、GBK、GB18030、Unicode、UTF-8、UTF-16）。

Type：原始类型（byte、short、int（二进制、八进制、十六进制表示形式）、long、float、double（十进制形式、科学计数法形式）、char、boolean）和引用类型（class、interface、array）、自动类型转换、强制类型转换、类型提升问题（所有二元运算，都存在）。

```
/*
*测试标识符的写法
*/
public class TestIdentifier{
    public static void main(String[] args){
        int   $abc = 3;
        int $ = 5;
        int _123=5;
        //int 123abc = 6;      //标识符不能以数字开头
       //int   abc# = 3;      //标识符不能包含除了字母、数字、下划线、$之外的其他字符
        //int   class = 3;
        int 武汉 = 10;    //java 内部采用了 Unicode 字符集，universal。
    }
}
```

```
//测试整数类型：byte,short,int,long。以及进制之间的转换问题
public class TestDataType {
    public static void main(String[] args){
        int a = 10;
        int a2 = 010;
        int a3 = 0xf;
    //  byte b = 200;
    //  System.out.println(b);
        System.out.println(a);
        System.out.println(a2);
        System.out.println(a3);
        System.out.println(Integer.toBinaryString(a));
        System.out.println(Integer.toOctalString(a));
        System.out.println(Integer.toHexString(a));
int a = 0b0000_0000_0000_0000_0000_0000_0000_0011;
        int b = 1_2312_3131;
        System.out.println(a);
        System.out.println(b);

        int a5 = 10;
        long a6 = 200;
        byte b2 = 100;    //如果数据的大小没有超过 byte/short/char 的表述范围，则可以自动转型。

        long   a7 = 11123213232L;
        long l = 3;
        long l2 = l+3;      //L 问题
    }
}
```

```
//测试浮点数
public class TestFloatType {
    public static void main(String[] args){
        //double d = 3.14;    //浮点数常量默认类型是 double。
        //float f = 6.28F;
        double d2 = 314e-2;    //采用科学计数法的写法
        System.out.println(d2);
        float f = 0.1f;
        double d = 1.0/10;
        System.out.println(f==d);    //false
    }
}
```

```
//测试 char
public class TestCharType {
    public static void main(String[] args){
    /*
        char c1 = 'a';
        char c2 = '方';    //unicode   2: 0-65535
        char c3 = '\n';
        System.out.print(c1);
        System.out.print(c3);
        System.out.print(c2);
    */
    /*
        char c4 = 'a';
        int i = c4 + 2;
        char c5 = (char)i;    //强制转型
        System.out.println(c5);

        //循环打印 a-z
        for(int j=0;j<26;j++){
```

```
            char temp = (char)(c4+j);
            System.out.print(temp);
        }

        //java 里面的字符串，是定义成：String 类了。
        String   str = "abcdefghijklmnopqrstuvwxyz";
        System.out.println("\n"+str);
    */
        boolean b = false;    //false
        if(b){      //if(b==true)..if(b=true)
            System.out.println("true");
        }
    }
}
```

byte->short->int->long/char->int/int->double
float->double///int..>float/long..>float/long..>double

//测试自动转型和强制转型 CAST

```
public class TestCast {
    public static void main(String[] args){
    /*
        byte b = 123;
        //byte b2 = 300;
        //char c = -3;
        char c2 = 'a';
        int i = c2;
        long d01 = 123213;
        float f = d01;

        //测试强制转型
        int i2 = -100;
        char c3 = (char)i2;    //-100 超过 char 的表数范围，所以转换成完全不同的值，无意义的值！
        System.out.println(c3);
    */
    /*
        //表达式中的类型提升问题
        int a = 3;
        long b = 4;
        double d = 5.3;
        int  c  =  (int)(a+b);    //做所有的二元运算符(+-/*%)，都会有类型提升的问题！
        float f = (float)(a + d);
    */
        int money = 1000000000;   //10 亿
        int years = 20;
        long  total  =  (long)money*years;    //返回的是负数//(long)(maney*years)
        System.out.println(total);

        //一个人 70 年心跳多少次
        long times = 70L*60*24*365*70;
        System.out.println(times);
    }
}
```

（3）局部变量、实例变量、类变量（在类中，用 static 声明的成员变量为静态成员变量，它为该类的公用变量、属于类，被该类的所有实例共享，在类被载入时被显示初始化；对于该类的所有对象来说，static 成员变量只有一份；可以使用“对象.类属性”来调用，不过，一般都是用”类名.类属性”；static 变量置于方法区中**）、常量**。局部变量使用前必须要先赋值、实例变量则可以不需要。变量名、常量名、方法名、类名的命名规则。

//实例变量、成员变量、常量

```
public class TestVariable {
    int t;      //实例变量，成员变量，属性
    public static void main(String[] args){
        int a; //局部变量使用之前必须要赋值，而实例变量则有缺省初值
        int b = a+3;
        int x,y,z;
        final int C=34;
        C = 35;
        final int   MAX_SPEED = 120;
    }
}
```

//变量首字母小写，驼峰原则；类名首字母大写，驼峰原则；常量全部大写，单词之间下划线相隔

Type: primitive type(byte,short,int,long,float,double,char, boolean), reference type(class, interface, Array)

（4）运算符：算术运算符、赋值运算符、关系运算符、逻辑运算符（&&、||、!）、位运算符（&（按位与）、|（按位或）、^（按位异或）、~（按位取反）、>>（右移运算符，右移一位相当于除 2 取商）、<<（左移运算符，左移一位相当于乘 2）、>>>（无符号右移、忽略符号位扩展、0 补最高位））、***字符串连接符（+）***、条件运算符、扩展运算符（+=、-=、*=、/=、%=）。短路求值。运算符优先级。

//运算符 Operator

算术运算符：+、-、*、/、%、++、--
赋值运算符：=

关系运算符：>、<、>=、<=、==、!=、instanceof
逻辑运算符：&&、||、!
位运算符：&（按位与）、|（按位或）、^（按位异或）、~（按位取反）、>>（右移运算符，右移一位相当于除 2 取商）、<<（左移运算符，左移一位相当于乘 2）
条件运算符：?:
扩展运算符：+=、-=、*=、/=、%=

(1) 5/2 = 2, 5/2.0=2.5
(2) x=x+1/x+=1/x++ (06slides.pdf)
(3) CONSTANT: private static final double PI = 3.14;
(4) 函数，整形与浮点数的除法运算操作符，操作符优先级，类型转换，常量，布尔数据类型，值的比较，布尔表达式，短路求值 (07slides.pdf)
(5) Average2Integers.java

```
public class TestOperator {
    public static void main(String[] args){
    /*
        double d = 10.2%3;
        System.out.println(d);

        int a = 3;
        int b = a++;    //执行完后,b=3。先给 b 赋值，再自增。
        int c = ++a;    //执行完后,c=5。先自增,再给 b 赋值
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    */
    //  int c = 3/0;

    /*
        boolean c = 1<2&&2>(3/0); //短路运算
        System.out.println(c);
    */
    /*
        //测试位运算
        int m = 8;
        int n = 4;
        System.out.println(m&n);
        System.out.println(m|n);
        System.out.println(~m);
        System.out.println(m^n);

        int a = 3*2*2;
        int b = 3<<3;   //相当于：3*2*2;
        int c = 12/2/2;
        int d = 12>>2;
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);

        boolean b1 = true&false; //没有短路功能
        System.out.println(b1);
    */

    /*
        //扩展运算符
        int  a = 3;
        a +=5;  //a = a+5;
    */
    /*
        //字符串相连：加号两边只要有一个为字符串，则变为字符串连接符，整个结果为字符串。
        System.out.println(4+"5");
    */
        int a=3;
        int b=5;
        String str= "";
        /*
        if(a<b){
            str = "a<b";
        }else{
            str = "a>b";
        }
        */
        str = (a<b)?"a<b":"a>=b";
        System.out.println(str);
    }
}
```

（5）控制语句：顺序、语句块、选择（单选、双选、多选（if else if、switch 语句（case 值可以是 int 或者自动可以转换为 int 的类型 byte、char、short，以及枚举和字符串，理解 case 穿透现象，一般每个 case 后面都要加 break，防止 case 穿透））、循环（for 循环语句（迭代不一定得是 i++，可以是任何的迭代），while 循环语句（while 先判断后执行、dowhile 先执行后判断））。变量的作用域。

```
//控制语句
//顺序、选择（单选、双选、多选）、循环
//语句块，变量的作用域，if 套嵌，switch 语句，for 循
```

环语句，while 循环语句(07slides.pdf)
Checkerboard.java

```
/*
 * 测试 if 语句
 */
public class TestIf {
	public static void main(String[] args) {
		double d = Math.random();
		int e =1+(int)(d*6);
		System.out.println(e);

		if(e>3) {
			System.out.println("大数！");
			System.out.println("大数！!!!");
		}else{
			System.out.println("小数！");
		}

		System.out.println("测试多选择结构");
		if(e==6){
			System.out.println("运气非常好！");
		}else if(e>=4){
			System.out.println("运气不错！");
		}else if(e>=2){
			System.out.println("运气一般吧");
		}else{
			System.out.println("运气不好！");
		}
	}
}

/*
 * 测试 swtich 语句
 */
public class TestSwitch {
	public static void main(String[] args) {
		double d = Math.random();
		int e =1+(int)(d*6);
		System.out.println(e);
		System.out.println("测试多选择结构");
		if(e==6){
			System.out.println("运气非常好！");
		}else if(e==5){
			System.out.println("运气很不错！");
		}else if(e==4){
			System.out.println("运气还行吧");
		}else{	//1,2,3
			System.out.println("运气不好！");
		}

	System.out.println("##########################");
		switch(e){	//int,或者自动可以转为 int 的类型(byte,char,short)。枚举。//JDK7 中可以放置字符串。
		case 6:
			System.out.println("运气非常好！");
			break;	//一般在每个 case 后面都要加 break，防止出现 case 穿透现象。
		case 5:
			System.out.println("运气很不错！");
			break;
		case 4:
			System.out.println("运气还行吧");
			break;
		default:
			System.out.println("运气不好！");
			break;
		}

	System.out.println("**************************
下面例子反过来利用了 case 穿透现象！");
		char	c = 'a';
		int rand =(int) (26*Math.random());
		char c2 = (char)(c+rand);
		System.out.print(c2 + ": ");
		switch (c2) {
		case 'a':
		case 'e':
		case 'i':
		case 'o':
		case 'u':
			System.out.println("元音");
			break;
		case 'y':
		case 'w':
			System.out.println("半元音");
			break;
		default:
			System.out.println("辅音");
		}
	}
}

/**
```

```
 * 测试 JDK7 中的 switch 新特性
 * @author dell
 *
 */
public class TestSwitch02 {
	public static void main(String[] args) {
		String	a = "方";
		switch (a) {	//JDK7 的新特性，表达式结果可以是字符串！！！
		case "何":
			System.out.println("何");
			break;
		case "方":
			System.out.println("方");
			break;
		default:
			System.out.println("大家好！");
			break;
		}
	}
}

/**
 * 测试 while 循环的用法
 * @author dell
 *
 */
public class TestWhile {
	public static void main(String[] args) {
		int a = 1;	//初始化
		while(a<=100){	//条件判断
			System.out.println(a);	//循环体
			a++;	//迭代
		}
		System.out.println("while 循环结束！");
		//计算：1+2+3+...+100
		int b = 1;
		int sum = 0;
		while(b<=100){
			sum += b;	//sum = sum + b;
			b++;
		}
		System.out.println("和为："+sum);
	}
}

//while 先判断后执行，dowhile 先执行后判断

/**
 * 测试 For 循环语句
 * @author dell
 *
 */
public class TestFor {
	public static void main(String[] args) {
		int a = 1;	//初始化
		while(a<=100){	//条件判断
			System.out.println(a);	//循环体
			a+=2;	//迭代
		}
		System.out.println("while 循环结束！");
		for(int i = 1;i<=100;i++){	//初始化//条件判断//迭代//迭代不一定非得是 i++，可以是任何的迭代
			System.out.println(i);	//循环体
		}
		System.out.println("while 循环结束！");
	}
}

/**
 * while 和 for 循环后的练习题目
 * @author dell
 * 计算 100 以内基数和偶数的和，并输出
 */
public class TestWhileFor {
	public static void main(String[] args) {
		int oddSum = 0;	//用来保存奇数的和
		int evenSum = 0;	//用来存放偶数的和
		for(int i=0;i<=100;i++){
			if(i%2!=0){
				oddSum += i;
			}else{
				evenSum += i;
			}

		}
		System.out.println("奇数的和："+oddSum);
		System.out.println("偶数的和："+evenSum);
		System.out.println("##########################");
//输出 1-1000 以内能被 5 整除的数，且每行输出 3 个
		for(int j = 1;j<=1000;j++){
			if(j%5==0){
				System.out.print(j+"\t");
			}
```

```
                if(j%(5*3)==0){
                    System.out.println();
                }
            }
        }
}

/**
 * 测试控制语句练习
 * 99 乘法表
 * @author dell
 *
 */
public class TestWhileFor02 {
    public static void main(String[] args) {
        for (int m = 1; m <= 9; m++) {
            for (int i = 1; i <= m; i++) {
                System.out.print(i + "*" + m + "=" + (i * m) + "\t");
            }
            System.out.println();
        }
    }
}

/**
 * 测试 break 和 continue
 *
 * @author dell
 *
 */
public class TestBreakContinue {
    public static void main(String[] args) {
        int total = 0;
        System.out.println("Begin");
        while (true) {
            total++;
            int i = (int) Math.round(100 * Math.random());
            if (i == 88) {
                break;
            }
        }
        System.out.println("Game over, used " + total + " times.");
    System.out.println("##########################");
        for (int i = 100; i < 150; i++) {
            if (i % 3 == 0) {
                continue;
            }
            System.out.println(i);
        }
    System.out.println("**********************");
        int count = 0;
        outer: for (int i = 101; i < 150; i++) {
            for (int j = 2; j < i / 2; j++) {
                if (i % j == 0)
                    continue outer;
            }
            System.out.print(i + "    ");
        }
    }
}
```

（6）break、continue、带标签的 break 和 continue：break 跳出循环、continue 跳出当次循环。

（7）字符串 String：是一个类，String 是 immutable 的，想做字符串中直接作字符的修改不被允许。

char ch; String str; ch = Character.toUpperCase(ch); str = str.toUpperCase();（由于不变形，新串的地址赋给 str）

if(str == s2){}（比较的是字符串的地址，而非里面的内容，要用 equals 方法。）

2、方法

（0）软件工程中不写重复的代码，保证代码的通用性→将可能多次使用的代码封装成方法，以进行重复利用（如 CD 机的例子，为什么我们不将 CD 唱片和 CD 机制造到一起的原因→因为我们想一次制造出一个对所有 CD 唱片通用的播放硬件）。

（0.1）08-slides.pdf

（0.2）Information hiding / everything in the world is a function or method 一个方法内只能使用自己声明的局部变量、参数、实例变量、类变量，不能使用其它方法内的局部变量。两个方法内的同名局部变量互不影响，哪怕它们同名。

（0.3）09-slides.pdf　RollDices.java

（0.4）MyCounter.java UseCounter.java

```
public class Counter {
    private int counter;
    public Counter(int counter) {
        this.counter = counter;
    }
    public Counter() {
        counter = 1;
    }
```

```
    public int nextValue() {
        int temp = counter;
        counter++;
        return temp;
    }
}
```

（1）visiblity type name (parameters){body→return expression}

（2）Java 使用下述形式调用方法：对象名.方法名（实参列表）<--receiver.name(arguments→parameters)

（3）Java 的方法类似于其它语言的函数，是一段用来完成特定功能的代码片段

（4）包括**形参**（在方法被调用时用于接收外界传进来的数据）、**实参**（调用方法时实际传给方法的数据）、**返回值**（方法在执行完毕返回给调用它的环境的数据）、**返回值类型**（事先约定的返回值的数据类型，如无返回值，必须写上 void）。

（5）实参数目、类型和次序必须和所调用的方法声明的形参列表匹配。基本类型变量传递的是值拷贝、引用类型变量传递的是该引用所指向动态变量的地址。所以其实都是值传递。

（6）return 语句终止方法的运行并指定要返回的数据。

（7） returned sin(x)→parameter

（8）import java.lang.Math; double x = 9.5; double y = Math.sqrt(x); y = Math.pow(x,y); **package（解决类重名、更便于管理类、是类的第一句非注释性语句）、import**

（9）设计方法的原则：方法的本意是功能块，就是实现某个功能的语句块的集合。 我们设计方法的时候，最好保持方法的原子性，就是一个方法只完成 1 个功能，这样利于我们后期的扩展。

（10）main 方法，程序的入口方法。

（11）构造方法，又称为构造器 Constructor，用于构造该类的实例。

- 格式：[修饰符] 类名 (形参列表) {…}
- 通过 new 关键字调用；
- 构造器虽然有返回值，但是不能定义返回类型（返回值的类型肯定是本类），不能在构造器里调用 return；
- 如果我们没有定义构造器，系统自动会定义一个无参的构造器，如果已定义则编译器不会添加；
- 构造器的方法名必须和类名一致。

（12）方法的重载 overload。方法的重载是指一个类中可以定义有相同的名字，但参数不同的多个方法。调用时，会根据不同的参数表选择对应的方法。

- 两同三不同：同一个类，同一个方法名，不同的参数列表（类型、个数、顺序不同）。
- 只有返回值不同不构成方法的重载。
- 只有形参的名称不同，不构成方法的重载。
- 构造方法与普通方法一样也可以重载。

（13）静态方法：用 static 声明的方法为静态方法。不需要对象，就可以 调用（“类名.方法名”）；在调用该方法时，不会将对象的引用传递给它，所以在 static 方法中不可访问非 static 的成员。

（14）静态初始化块 static {}：如果希望加载后，对整个类进行某些初始化操作，可以使用 static 初始化块。

- 是在类初始化时执行，不是在创建对象时执行。
- 静态初始化块中不能访问非 static 成员变量。
- 执行顺序：上溯到 Object 类，先执行 Object 的静态初始化块，再向下执行子类的静态初始化块，直到我们的类的静态初始化块为止。

（15）this（方法的一个隐式参数，还有一个隐式参数叫 super(父类对象的引用)）关键字。

- 普通方法中，this 总是指向调用该方法的对象；
- 构造方法中，this 总是指向正要初始化的对象；
- this 不能用于 static 方法；
- 可以在一个构造方法中通过 this 调用其它构造方法，且必须是构造方法中的第一条语句。

//方法 Methods 08-slides.pdf 09-slides.pdf

(1)Java 的方法类似于其它语言的函数，是一段用来完成特定功能的代码片段 Information hiding / everything in the world is a function or method

(2)包括形参、实参、返回值、返回值类型。

(3)Java 使用下述形式调用方法：对象名.方法名（实参列表）<--receiver.name(arguments→parameters)

(4)实参数目、类型和次序必须和所调用的方法声明的形参列表匹配。

(5)Return 语句终止方法的运行并指定要返回的数据。

(6) returned sin(x)→parameter

(7)import java.lang.Math; double x = 9.5; double y = Math.sqrt(x); y = Math.pow(x,y);

(8)Visiblity type name (parameters){body→return expression}

```
/**
 * 测试方法
 *
 * @author dell
 *
 */
public class TestMethod {

    //设计方法的原则：方法的本意是功能块，就是实
```

现某个功能的语句块的集合。　我们设计方法的时候，最好保持方法的原子性，就是一个方法只完成 1 个功能，这样利于我们后期的扩展。

```
    public static void test01(int a) {
        int oddSum = 0; // 用来保存奇数的和
        int evenSum = 0; // 用来存放偶数的和
        for (int i = 0; i <= a; i++) {
            if (i % 2 != 0) {
                oddSum += i;
            } else {
                evenSum += i;
            }

        }
        System.out.println("奇数的和：" + oddSum);
        System.out.println("偶数的和：" + evenSum);
    }

    public static void test02(int a,int b,int c){
        for (int j = 1; j <= a; j++) {
            if (j % b == 0) {
                System.out.print(j + "\t");
            }
            if (j % (b * c) == 0) {
                System.out.println();
            }
        }
    }

    public static int add(int a,int b){
        int sum = a+b;
        if(a==3){
            return 0;       //return 两个作用：结束方法的运行、返回值。
        }
        System.out.println("输出");
        return sum;
    }

    public static void main(String[] args) {
        test01(1000);
        test02(100,6,3);   //1-100 之间，可以被 6 整除，每行输出 3 个。
        System.out.println("############");
        int s = add(3,5);
        System.out.println(s);
    }
}

/**
 * 测试递归算法
 *
 */
public class TestRecursion {
    static int a = 0;

    public static void test01(){
        a++;
        System.out.println("test01:"+a);
        if(a<=10){   //递归头
            test01();
        }else{          //递归体
            System.out.println("over");
        }
    }

    public static void test02(){
        System.out.println("TestRecursion.test02()");
    }


    public static long factorial(int n){
        if(n==1){
            return 1;
        }else{
            return n*factorial(n-1);
        }
    }

    public static void main(String[] args) {
        test01();
        System.out.println(factorial(10));
    }
}
import java.util.Scanner;
/**
 * 测试 Scanner 类的使用，如何接收键盘的输入。
 * @author dell
 *
 */
public class TestScanner {

    public static void test01(){
        Scanner s = new Scanner(System.in);
```

```
        String str = s.next();  //程序运行到 next 会阻塞，等待键盘的输入！
        System.out.println("刚才键盘输入："+str);
    }

    public static void test02(){
        Scanner s = new Scanner(System.in);
        System.out.println("请输入一个加数：");
        int a = s.nextInt();
        System.out.println("请输入被加数：");
        int b = s.nextInt();
        int sum =a+b;
        System.out.println("计算结果，和为："+sum);
    }


    public static void main(String[] args) {
        test02();
    }
}
```

3、代码的内存分析

Memory(RAM)：bit、byte（8 bits）、word（4 bytes）、Kilobyte、megabyte、gigabyte、terabyte（百万兆字节）、petabyte（十亿兆字节）、Exabyte（万亿兆字节）、Zettabyte（千万亿兆字节）、yottabyte（百亿亿兆字节）。美国图书馆全部藏书 10TB、至今所有印刷品 200PB、人类说的所有的话大约 5Exabytes。

Hexadecimal：十六进制表示地址。内存分一个个单元，每个单元 1byte（8 bits），都是有地址（十六进制表示）的。所以如果你 2G 的内存条，就是 20 亿个位置，需要 20 亿个地址， 如果内存够大，FFFF….的个数会越多。

栈 stack 自动分配连续的空间，后进先出，放置局部变量（local variables/parameters→stack）；

堆 heap 不连续，放置 new 出来的对象（Dynamic variables→heap）；

方法区也是堆，存放类的代码信息、static 变量、static 方法、常量池（字符串常量）。（static variables/constant→special）

方法区和堆依次存放于内存的低区域（这里的地址数值比较小），堆的地址从小往大分配；栈存放于内存的高区域（地址数值大，从 FFF…开始），栈的地址从大往小分配。当互相重叠时，就互相修改数据，程序崩溃。

Point p1 = new Point(1,1);Point p2 = p1;P2.move(10,10);Point p3;p3.move(10,10)(错误);

String s1 = “hello”; String s2 = “hello”; if(s1==s2)（相等）{}

Primitive → passing by value; Objects →passing by object(reference)

Int Integer\double Double\boolean Boolean\char Character 一旦给了初值，就不能改变，同 String 一样（immutable）。

```
package cn.bjsxt.oop;
public class Student {
    //静态的数据
    String name;
    int id;     //学号
    int age;
    String gender;
    int weight;
    Computer computer;
    //动态的行为
    public void study(){
        System.out.println(name+"在学习");
    }
    public void sayHello(String sname){
        System.out.println(name+"向"+sname+"说：你好！");
    }
}
package cn.bjsxt.oop;
public class Computer {
    String brand;
    int cpuSpeed;
}
package cn.bjsxt.oop;
public class Test1 {
    public static void main(String[] args) {
        //通过类加载器 Class Loader 加载 Student 类。加载后，在方法区中就有了 Student 类的信息！
        Student s1 = new Student();
        s1.name = "方";
        s1.study();
        s1.sayHello("何");
        Student s2 = new Student();
        s2.age = 18;
        s2.name="老方";
    }
}
package cn.bjsxt.oop;
public class Test2 {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.name = "方";
```

```
        s1.age = 18;
        Computer c = new Computer();
        c.brand = "神舟";
        c.cpuSpeed = 100;
        s1.computer = c;
        c.brand = "惠普";
        System.out.println(s1.computer.brand);
    }
}
package cn.bjsxt.oop.testThis;
public class Student {
    String name;
    int id;
    public  Student(String name,int id){
        this(name);    //通过 this 调用其他构造方法，必须位于第一句！ Constructor call must be the first statement in a constructor
        this.name = name;
        this.id = id;
    }
    public Student(String name){
        this.name = name;
    }
    public Student(){
        System.out.println("构造一个对象");
    }
    public void setName(String name){
        this.name = name;
    }
    public void study(){
        this.name=  "张三";
        System.out.println(name+"在学习");
    }
    public void sayHello(String sname){
        System.out.println(name+"向"+sname+"说：你好！");
    }
}
package cn.bjsxt.oop.testStatic;
public class Student {
    String name;
    int id;
    static  int  ss;
    public static void printSS(){
        System.out.println(ss);
    }
    public void study(){
        printSS();
        System.out.println(name+"在学习");
    }
    public void sayHello(String sname){
        System.out.println(name+"向"+sname+"说：你好！");
    }
}
package cn.bjsxt.oop.testStatic;
public class Test {
    public static void main(String[] args) {
        Student.ss = 323;
        Student.printSS();
        Student s1 = new Student();
    }
}
```

4、API 文档

生成自己的 API 文档：文档注释/**
解决代码和文档的自动分离。
@author 作者
@version 版本
@param 参数
@return 返回值的含义
@throws 抛出异常描述
@deprecated 废弃，建议用户不使用这个方法
-encoding UTF-8 -charset UTF-8

5、垃圾回收机制（Garbage Collection）

- 使用 new 关键字创建对象时，进行对象空间的分配；
- 当对象没有引用指向时（将对象赋值 null 即可），垃圾回收器将负责回收所有不可达对象的内存空间；
- java 程序员无权调用垃圾回收器；
- 程序员可以通过 system.gc()通知 GC 运行，但是 java 规范并不能保证立刻运行；
- Finalize 方法，是 java 提供给程序员用来释放对象或资源的方法，但是尽量少用。

6、继承（以及继承的内存分析）

（1）从面向对象设计 OOD 的角度来说，类是对对象的抽象、继承是对某一批类的抽象，从而实现对现实世界更好的建模；从面向对象编程 OOP 的角度来说，这种组织方式可以提高代码的复用性。

（2）extends 的意思是“扩展”。子类是父类的扩展。

- human 类（人类）继承 primate 类（灵长类）、primate 类继承 mammal 类（哺乳类）、哺乳类继承 animal

类（动物类）；

- monkey（猴子类）也继承 primate 类。人类和猴子类不仅继承灵长类，也继承哺乳类、动物类；
- reptile（爬行类）和 insect（昆虫类）也继承 animal 类。鳄鱼类（corcodile）和蛇类（snake）也继承爬行类。蝴蝶类（butterfly）也继承昆虫类；
- 动物：digest food（消化食物）、through the blastula stage（胚胎阶段）；
- 哺乳动物：warm blooded（恒温）、mammary glands（乳腺）；
- 灵长动物：five fingers（5 根手指）、opposable thumbs（一对手指）、fingernails（指甲）；
- 人类：highly developed brains（发达的大脑）、erect body carriage（直立姿态）。

（3）例子：(09-code) Student.java、FreshMan.java、UseStudent.java

（4）子类继承父类，可以得到父类的全部属性和方法（除了父类的构造方法）；

（5）在 java（C++-）中类只有单继承，没有像 C++那样的多继承。多继承，就是为了实现代码的复用性，却引入了复杂性，使得系统类之间的关系混乱；

（6）Java 中的多继承，可以通过接口实现 ；

（7）如果定义一个类时，没有调用 extends，则它的父类是 java.lang.Object。Object 类是所有 java 类的根基类。

```
public static void main(String[] args){
    Object obj = new Object();
    Object obj2 = new Object();
    System.out.println(obj.toString());
    System.out.println(obj2.toString());//包名+类名+@+根据对象内存位置生成的唯一的哈希码
    System.out.println(obj==obj2);
    System.out.println(obj.equals(obj2));
}
```

（8）方法的重写（override）

- 在子类中可以根据需要对父类中继承来的方法进行重写。
- 重写方法必须和被重写方法具有相同方法名称、参数列表和返回类型。通过子类去调用该方法，会调用重写方法而不是被重写方法（叫做重写方法覆盖被重写方法）。
- 可以在子类重写方法中调用被重写方法：Super 关键字。
- 重写方法不能使用比被重写方法更严格的访问权限（由于多态）。
- 与重载（overload）没有任何关系。
- 利用重写，既可以重用父类的方法，而且还可以灵活的扩充。
- 对象.方法（）：先在本类内部找是否有该方法，如果没有，到直接父类去找，如果还没有，则一直往上层找，一直找到 Object，如果还没有，则报错。

（9）super 关键字

- super 是直接父类对象的引用。可以通过 super 来访问父类中被子类覆盖的方法或属性。
- super 和 this 一样，只能用于方法内部，是方法的两个隐式参数。
- 普通方法：没有顺序限制，可以随便调用。
- 构造方法：任何类的构造方法中，若是构造函数的第一行代码没有显式调用 super(…)；那么 Java 默认都会调用 super()；作为父类的初始化函数。所以这里的 super()加不加都会无所谓。**（内存分析，wrap:new 对象的时候采用子类包裹父类的结构）**
- 同一个构造方法里面不能同时调用 super()和 this()。
- 在本类构造方法中通过 super()调用，会一直上溯到 Object()这个构造函数，然后按类层级，依次向下执行各层级构造函数中剩下的代码，直至最低层级的构造函数。同 this()一样，super()方法也应该放到构造方法的第一行。
- new 一个类的对象的时候，通过构造方法的从上至下的依次调用，就依次建立了新的根对象、父类对象和自身对象，其中，this 指向新建的对象本身，super 指向新建的直接父类对象本身。

（10）组合 VS 继承

- “is-a”关系使用继承：上面的通过在 Audi 类中增加一个 Car 属性虽然也复用了代码，但是不合逻辑不容易理解。
- “has-a”关系使用组合：计算机类、主板类。可以通过在计算机类中增加主板属性来复用主板类的代码。
- 如果仅仅从代码复用的角度考虑，组合完全可以替代继承。
- 所谓组合，就是把要组合的另一个类作为属性放到类里面。
- 是就用继承、有就用组合。

（11）final 关键字

- final 修饰变量时表示常量；
- final 修饰方法（最终方法）时表示该方法不可被子类重写。但是可以被重载；
- final 修饰类（最终类）时表示修饰的类不能有子类，不能被继承。比如 Math、String。

7、包装/封装/隐藏（encapsulation）

（1）为什么需要封装？封装的作用和含义？

- 隐藏对象内部的复杂性，只对外公开简单的接口。便于外界调用，从而提高系统的可扩展性、可维护

性。

- 我们程序设计要追求“高内聚、低耦合”（**只提供最简单的接口给别人，内部复杂的细节不给别人看，也不允许别人干预**）。高内聚就是内的内部数据操作细节自己完成，不允许外部干涉；低耦合就是仅暴露少量的方法给外部使用。

（2）使用访问控制符，实现封装（类里面就属性和方法两类代码，如何控制可访问性如何封装，由下表决定）

	同一个类	同一个包	子类	所有类
private	可以			
default 缺省	可以	可以		
Protected 保护	可以	可以	可以	
public	可以	可以	可以	可以

（3）封装要点

- 类的属性的处理：一般使用 private（除非本属性确定会让子类继承）；提供相应的 get/set 方法来访问相关属性，这些方法通常是 public，从而提供对属性的读取操作。(boolean 变量的 get 方法是 is 开头)；常量或 static 变量公开。
- 一些只用于本类的辅助性方法可以用 private，希望其它类调用的方法可以用 public。
- default：默认访问控制属性，什么都没加就是该控制符。有的书上说 friendly、package，这都是一个意思，都不能真的写出来，如果什么访问修饰符都不加，就是 default/friendly/package。
- Java 的访问控制是停留在编译层，也就是它不会在.class 文件中留下任何痕迹，只在编译的时候进行访问控制的检查。其实，通过反射的手段，可以访问任何包下任何类中的成员，例如，访问类中的私有成员也是可以的。

8、类、变量、方法、接口修饰符总结

（1）类：

访问修饰符 修饰符 class 类名称 extends 父类名称（单个） implement 接口名称（可以多个）（访问修饰符与修饰符的位置可以互换）

访问修饰符：

名称	说明	备注
public	可以被所有类使用	public 类必须定义在和类名相同的同名文件中，且一个 java 文件中只能最多有一个 public 类
缺省	可以被同一个包中的类使用	默认的访问权限，可以省略此关键字，可以定义在和 public 类同一个文件中

修饰符：

名称	说明	备注
final	使用此修饰符的类不能被继承	
abstract	如果要使用 abstract 类，之前必须要建一个继承 abstract 类的子类，子类中实现 abstract 类中的抽象方法 abstract 类不能实例化对象	类只要有一个 abstract 方法，类就必须定义为 abstract，但 abstract 类不一定非要保护 abstract 方法不可。

（2）变量：

- Java 中没有全局变量，只有局部变量、实例变量、类变量（类中的静态变量）。
- 方法中的变量不能够有访问修饰符。访问修饰符仅针对类中定义的变量。
- 声明实例变量时，如果没有赋初值，将被实例化为 null（引用类型）或 0、false（原始类型）。
- 可以通过实例变量初始化块（用{}包含的语句块）来初始化较复杂的实例变量，在类的构造器被调用时运行，运行时刻位于父类构造器之后，本类构造器之前。
- 类变量（静态变量）也可以通过类变量初始化块（用 static{}包含的语句块）来进行初始化，且该静态块在所有层级上的构造函数调用之前被调用。只可能被初始化一次。
- static{}是在类初始化时执行，不是在创建对象时执行。静态初始化块中不能访问非 static 成员变量。执行顺序：上溯到 Object 类，先执行 Object 的静态初始化块，再向下执行子类的静态初始化块，直到我们的类的静态初始化块为止。

访问修饰符：

名称	说明	备注
public	可以被任何类使用	
protected	可以被同一包中的所有类访问，可以被所有子类访问	子类不在同一个包中也可以访问
缺省	可以被同一包中的所有类访问	如果子类不在同一个包中，也不能访问
private	只能被同一类访问	

修饰符：

名称	说明	备注
static	静态变量（类变量）	可以被类的所有实例共享，并不需要创建类

		的实例就可以访问，在方法区中
final	常量，值只能分配一次，不能更改	注意不要使用 const，虽然它和 C、C++中的 const 关键字含义一样。可以同 static 一起使用，避免对类的每个实例维护一个拷贝
transient	告诉编译器，在类对象序列化的时候，此变量不需要持久保存	该变量可以通过其它变量来得到，使用它是为了性能问题
volatile	指出可能有多个线程修改此变量，要求编译器优化以保证对此变量的修改能正确被处理	

（3）方法：

访问修饰符 修饰符 返回类型 方法名称（参数列表） throws Exception 列表（访问修饰符与修饰符的位置可以互换）

- 类的构造器不能够有修饰符、返回类型和 throws 子句；
- 类的构造器方法被调用时，它首先运行静态变量初始化代码块（static{}），然后调用父类的构造器，然后调用实例变量初始化块（{}），然后运行构造器本身（会沿类层级迭代进行此步骤）；
- 如果构造器方法没有显式的调用父类的构造器，那么编译器会自动为它加上一个默认的 super()，而如果父类又没有默认的无参数构造器，则编译器报错。Super 必须是构造器方法的第一个子句。
- 同一个构造方法里面不能同时调用 super()和 this()。
- 注意理解 private 构造器方法额使用技巧。

访问修饰符：

名称	说明	备注
public	可以从所有类访问	
Protected	可以被同一包中所有类访问，可以被所有子类访问	子类没有在同一包中也可以访问
缺省	可以被同一包中的所有类访问	如果子类没有在同一包中，也不能访问
private	只能够被当前类的方法访问	

修饰符：

名称	说明	备注
static	静态方法（类方法）	提供不依赖于实例对象的服务，并不需要创建类的实例就可以访问静态方法
final	防止任何子类重写该方法，但是可以重载该方法	注意不要使用 const，虽然它和 C、C++中的 const 关键字含义一样。可以同 static 一起使用，避免对类的每个实例维护一个拷贝
abstract	抽象方法，类中已声明而没有实现的方法	不能将 static 方法、final 方法或者类的构造器方法声明为 abstract
native	用该修饰符定义的方法在类中没有实现，而大多数情况下该方法的实现是用 C、C++编写的	参见 Java Native 接口（JNI），JNI 提供了运行时加载一个 native 方法的实现，并将其与一个 java 类关联的功能
synchronized	多线程的支持	当一个此方法被调用时，没有其它线程能够调用该方法，其它的 synchronized 方法也不能调用该方法，直到该方法执行完成

（4）接口：

访问修饰符 interface 接口名称 extends 接口列表

- 接口不能够定义其声明的方法的任何实现；
- 接口中所有方法默认隐含为 public（只能）的，可以使用 static 修饰符；
- 接口中不能够定义变量，接口中的变量总是需要定义为“public static final 接口名称”，但可以不包含这些修饰符，编译器默认就是这样，显式的包含修饰符是为了程序清晰。

访问修饰符：

名称	说明	备注
public	所有类可见	
缺省	同一个包内	

（5）访问修饰符和修饰符表：

修饰符	类	成员方法	构造方法	成员变量	局部变量

abstract	可以	可以			
static		可以		可以	
public	可以	可以	可以	可以	
protected		可以	可以	可以	
缺省	可以	可以	可以	可以	
private		可以	可以	可以	
synchronized		可以			
native		可以			
transient				可以	
volatie				可以	
final	可以	可以		可以	可以

9、多态（polymorphism）

- Java 程序经历编译、运行阶段。编译用父类声明，运行时使用具体子类实例化。
- 多态性是 OOP 中的一个重要特性，主要是用来实现动态联编。就是程序的最终状态只有在执行过程中才被决定而非在编译期间就决定。这对于大型系统来说能提高系统的灵活性和扩展性。
- Java 中如何实现多态：引用变量的两种类型：编译时类型（模糊一点，一般是一个父类），由声明时的类型决定。运行时类型（运行时，具体是哪个子类就是哪个子类），由实际对应的对象类型决定。
- 多态存在的 3 个必要条件：要有继承，要有方法重写，父类引用指向子类对象。
- 多态指的是重写方法的多态。
- 方法的多态是子类重写了父类的方法，父类引用指向子类对象，通过父类.被重写方法，实际调用的是子类的重写方法（这个相当于把子类对象地址赋给父类对象引用，编译时，编译器通过声明的父类类型去查找父类代码中是否有该方法，有，则通过了编译，实际运行时，则依据实际内存地址找到了子类对象内存空间，因此，实际执行的是子类方法）。
- 上述情况下，如果想通过父类引用来调用子类扩展的方法，则编译时将不通过（因为编译时只认父类引用所声明的父类类型，而父类类型里没有该子类扩展方法），可以将该父类引用强制转型为子类类型来达到目标。
- 引用类型的强制转型，适用于将父类类型向下强制转换为子类类型。不同类型之间不能强制转型（编译不通过）。
- 子类类型的对象地址可以直接赋给父类类型的引用对象，这个称为向上转型，是实现多态的基础。
- A instanceof B：A 对象的类型是否是 B 类型，只有在 A 对象的类型和 B 类型相同，或为父子类型时，编译不报错。而在运行时，只有 A 对象类型为 B 类型的子类型或者就是 B 类型时，结果才返回 true。
- 内存分析（下述例子）：调用父类的 service()，然后调用子类的 doGet()（this 关键字指向整个包裹对象，及子对象）。

```
package oop.polymorphism.myServlet;
public class HttpServlet {
    public void service(){
        System.out.println("HttpServlet.service()");
        this.doGet();
    }
    public void doGet(){
        System.out.println("HttpServlet.doGet()");
    }
}
package oop.polymorphism.myServlet;
public class MyServlet extends HttpServlet {
    public void doGet(){
        System.out.println("MyServlet.doGet()");
    }
}
package oop.polymorphism.myServlet;
public class Test {
    public static void main(String[] args) {
        HttpServlet s = new MyServlet();
        s.service();
    }
}
```

10、抽象类

（1）为什么需要抽象类？如何定义抽象类？

- 是一种模版模式。抽象类为所有子类提供了一个通用模版，子类可以在这个模版基础上进行扩展。
- 通过抽象类，可以避免子类设计的随意性。通过抽象类，可以严格限制子类的设计，使子类之间更加通用。

（2）要点

- 有抽象方法的类只能定义成抽象类。
- 抽象类不能实例化，即不能用 new 来实例化抽象类。
- 抽象类可以包含属性、方法、构造方法（抽象类除了抽象方法，也可以定义普通方法、构造方法和普通属性）。但是构造方法不能用来 new 实例，只能用来被子类调用。
- 抽象类只能用来继承。

- 抽象方法只有方法的声明，没有方法体（即{}和里面的代码），其必须被子类实现（这也是抽象方法的设计初衷，提供规范，要求各子类实现规范）。
- 抽象类用于设计和实现的分离，但是这种分离不够彻底。
- 一个抽象类可以继承另一个抽象类。

11、接口（interface）

接口：比抽象类还要抽象得彻底的抽象类（内部只能包含常量和 public 抽象方法）。最最最抽象。可以更加规范的对实现类进行约束。

- 一般用于抽取出不容易用继承关系来联系起来的一批类型的相同功能部分（可以抽象出任何类的共同点）。
- 接口就是规范，定义的是一组规则（任何类必须遵守），体现了现实世界“如果你是…你必须能…”的思想。
- 意义也在于设计和实现的分离，接口分离得最彻底。
- 项目基本都是面向接口编程。
- 接口中常量定义时，可以不写 public static final，缺省有。
- 接口中方法定义时，可以不写 public abstract，缺省有。
- 一个类可以实现多个接口（implements），接口不能 new 一个对象，但是可用于声明引用变量类型。接口可以继承另一个接口（extends），而且支持多继承。

12、内部类（innerclasses）

一般情况，我们把类定义成独立的单元。有些情况，我们把一个类放到另一个类的内部定义，称为内部类。

（1）内部类作用：

- 内部类提供了更好的封装。只能让外部类直接访问，不允许同一个包中的其它类直接访问。
- 内部类可以直接访问外部类的私有属性，内部类被当作外部类的成员。但外部类不能访问内部类的内部属性。

（2）内部类的使用场合：

由于内部类提供了更好的封装特性，并且可以很方便的访问外部类的属性。所以，通常内部类在只为外部类提供服务的情况下优先使用。

（3）内部类的分类：

1、成员内部类（可以使用 private、protected、public 任意进行修饰。类文件：外部类$内部类.class）

1.1 非静态内部类（外部类里使用非静态内部类和平时使用其它类没什么区别）

- 非静态内部类必须寄存在一个外部类对象里。因此，如果有一个非静态内部类对象那么一定存在对应的外部类对象。非静态内部类对象单独属于外部类的某个对象。
- 非静态内部类可以使用外部类的成员 Outer.this.variableName，但是外部类不能直接访问非静态内部类成员。
- 非静态内部类不能有静态方法、静态属性、静态初始化块。
- 静态成员不能访问非静态成员：外部类的静态方法、静态代码块不能访问非静态内部类，包括不能使用非静态内部类定义变量、创建实例。
- 成员变量访问要点：内部类里方法的局部变量：变量名；内部属性：this.变量名；外部类属性：外部类名.this.变量名。
- 内部类的访问：外部类中定义内部类：new innerClass()；外部类以外的地方使用非静态内部类：Outer.Inner varname = OuterObject.new inner()；/new Outer().new Inner();

1.2 静态内部类

定义方式： static class ClassName{}

使用要点：

- 当一个静态内部类对象存在，并不一定存在对应的外部类对象。因此，静态内部类的实例方法不能直接访问外部类的实例方法。
- 静态内部类看作外部类的一个静态成员。因此，外部类的方法可以通过：静态内部类.名字 访问静态内部类的静态成员。通过 new 静态内部类()访问静态内部类的实例。
- 在外部类的外面创建静态内部类：Outer.StaticInner inner = new Outer.StaticInner();

2、匿名内部类

适合那种只需要使用一次的类。比如：键盘监听操作等。

语法：new 父类构造器(实参类表)实现接口(){//匿名内部类类体}

```
this.addWindowListener(new WindowAdapter(){
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    ]
});
```

匿名内部类使用情况：

- 只用到类的一个实例。
- 类在定义后马上用到。
- 类非常小（SUN 推荐是在 4 行代码以下）。
- 给类命名并不会导致你的代码更容易被理解。

在使用匿名内部类时，要记住以下几个原则：

- 匿名内部类不能有构造方法。
- 匿名内部类不能定义任何静态成员、方法和类。

- 匿名内部类不能是 public,protected,private,static。
- 只能创建匿名内部类的一个实例。
- 一个匿名内部类一定是在 new 的后面，用其隐含实现一个接口或实现一个类。
- 因匿名内部类为局部内部类，所以局部内部类的所有限制都对其生效。

3、局部内部类：

定义在方法内部。作用域只限于本方法。

```
public class Outer {
    public static void main(String[] args) {
        Face f = new Face();
        Face.Nose n = f.new Nose();
        n.breath();
        Face.Ear e = new Face.Ear();
        e.listen();
    }
}
class Face {
    int type;
    String shape="瓜子脸";
    static String color="红润";
    class Nose {
        void breath(){
            System.out.println(shape);
            System.out.println(Face.this.type);
            System.out.println("呼吸！");
        }
    }
    static class Ear {
        void listen(){
            System.out.println(color);
            System.out.println("我在听！");
        }
    }
}
```

13、文件操作（File 类）

（1）java.io.File 类：文件和目录路径名的抽象表示形式

（2）通过 File 对象可以访问文件的属性：

```
public boolean canRead()
public boolean canWrite()
public boolean exists()
public boolean isDirectory()
public boolean isFile()
public boolean isHidden()
public long lastModified()
public long length()
public String getName()
public String getPath()
public boolean createNewFile() throws IOException
public boolean delete()
public boolean mkdir()
public boolean mkdirs()
```

```
public class TestFile {
    public static void main(String[] args) {
        File f = new File("d:/src3/TestObject.java");
        File f2 = new File("d:/src3");
        File f3 = new File(f2,"TestThis.java");
        File f4 = new File(f2,"TestFile666.java");
        File f5 = new File("d:/src3/aa/bb/cc/ee/ddd");
        f5.mkdirs();
            //f4.createNewFile();
//      f4.delete();
        if(f.isFile()){
            System.out.println("是一个文件");
        }
        if(f2.isDirectory()){
            System.out.println("是一个目录");
        }
    }
}
```

（3）采用递归方式编写一个程序，在命令行中以树状结构展现特定的文件夹及其子文件（夹）

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

14、异常（Exception）

Java 异常是 Java 提供的用于处理程序中错误的一种机制。

int i = 1/0; → java.lang ArithmeticException: / by zero

（1）如果没有处理异常机制存在的问题：逻辑代码和错误处理代码放到一起；程序员本身需要考虑的例外情况较复杂，对程序员本身要求较高。

（2）JAVA 中采用面向对象的方式来处理异常，处理过程：

- **抛出异常**：在执行一个方法时，如果发生异常，则这个方法生成代表该异常的一个对象，停止当前执行路径，并把异常对象提交给 JRE。
- **捕获异常**：JRE 得到该异常后，寻找相应的代码来处理该异常。JRE 在方法的调用栈中查找，从生成异常的方法开始回溯，直到找到相应的异常处理代码为止。

（3）异常分类：JDK 中的异常类对应各种可能出现的异常事件，所有异常对象都是继承于 Throwable 类的一个实例。如果内置的异常类不能满足要求，还可自定义异常类。

Throwable: Error、Exception

Error:

- Error 用于指示合理的应用程序不应试图捕获的严重问题(Java运行时系统内部错误和资源耗尽错误)，这类错误我们无法控制，同时也是非常罕见的错误，编程时不处理这类错误。
- 在执行方法期间，无需在其 throws 子句中声明可能抛出但是未能捕获的 Error 的任何子类，因为这些错误可能是再也不会发生的异常条件。

Exception:

- **Checked Exception：**我们必须捕获进行处理的一类异常（运行前已经检查过。就是编译器会检查是否对这些异常做了处理）
- **RuntimeException(:Unchecked Exception)：**一类特殊的异常（编译器不会检查的一类异常），如被 0 除、数组下标超范围等，其产生比较频繁，处理麻烦，如果显式的声明或捕获将会对程序可读性和运行效率影响很大。因此由系统自动检测并将它们交给缺省的异常处理程序（用户不必对其处理），常见 RuntimeException 包括：ArithmeticException、NullPointerException 、ClassCastException 、ArrayIndexOutOfBoundsException 、NumberFormatException。

（4）try-catch-finally（处理异常）

try

- try 语句指定了一段代码，该段代码就是一次捕获并处理的范围。在执行过程中，当任意一条语句产生异常时，就会跳出该段中后面的代码。代码中可能会产生并抛出一种或几种类型的异常对象，它后面的 catch 语句要分别对这些异常做相应的处理。
- 一个 try 语句必须带有至少一个 catch 语句块或一个 finally 语句块。当异常处理的代码执行结束后，不会回到 try 语句去执行尚未执行的代码。

catch

- 每个 try 语句块可以伴随一个或多个 catch 语句，用于处理可能产生的不同类型的异常对象。
- 常用方法（继承自 Throwable 类）：toString()方法，显示异常的类名和产生异常的原因；getMessage()方法，只显示产生异常的原因，不显示类名；printStachTrace()方法，用来跟踪异常事件发生时堆栈的内容。
- 当异常类之间有继承关系，则越是顶层的类，越放到后面。

finally

- 有些语句，不管是否发生了异常，都必须要执行，那么就可以把这样的语句放到 finally 语句块中。
- 通常在 finally 中关闭程序块已打开的资源，比如：文件流、释放数据库连接等。

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class TestException {
    public static void main(String[] args) {
        FileReader reader = null;
        try {
            reader = new
FileReader("D:/workspace/dms/pom.xml");
            char temp = (char)reader.read();
            System.out.println(temp);
        } catch (FileNotFoundException e) {
            System.out.println("文件没有找到");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("读取文件错误");
        } finally {
            System.out.println("肯定执行的语句");
            try {
                reader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
public class OpenFile {
    public static void main(String[] args) {
        String string = new OpenFile().openFile();
        System.out.println(string);
    }
    String openFile(){
        try {
            System.out.println("aaa");
            FileInputStream fis = new
FileInputStream("d:/a.txt"); //checked
            int a = fis.read();
            System.out.println("bbb");
            return "step1";
```

```
        } catch (FileNotFoundException e) {
            System.out.println("catching!!!!!");
            e.printStackTrace();
            return "step2";
        } catch (IOException e) {
            e.printStackTrace();
            return "step3";
        } finally {
            System.out.println("finally!!!!!");
            //return "step4";
        }
    }
}
```

执行顺序：**执行 try,catch，给返回值赋值；执行 finally；return。**

（5）抛出异常（throws 子句）

当 Checked Exception 产生时，不一定立刻处理它，可以再把异常 throws 出去。

如果一个方法抛出多个 checked 异常，就必须在方法的首部列出所有的异常，之间逗号隔开。

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class TestReadFile {
    public static void main(String[] args) throws FileNotFoundException, IOException {
        String str;
        str = new TestReadFile().openFile();
        System.out.println(str);
    }
    String openFile() throws FileNotFoundException,IOException {
        FileReader reader = new FileReader("d:/a.txt");
        char c = (char)reader.read();
        System.out.println(c);
        return ""+c;
    }
}
```

（6）手动抛出异常（throw 子句）

- Java 异常类对象除在程序执行过程中出现异常时由系统自动生成并抛出，也可根据需要手工创建并抛出。
- 在捕获一个异常之前，必须有一段代码先生成异常对象并把它抛出。这个过程可以手工做，也可以由 jre 做，但它们调用的都是 throw 子句。
- 对于一个已存在的异常类，抛出该类异常对象过程：找到合适的异常类；创建一个该类的对象；将对象抛出。

```
File f = new File("d:/a.txt");
if(!f.exists()){
    try{
throw new FileNotFoundException("file can not found!");
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

（7）方法重写中声明异常原则

子类声明的异常范围不能超过父类声明的范围：父类没有声明异常，子类也不能；不可抛出原有方法抛出异常类的父类或上层类；抛出的异常类型的类型数目不可比原有方法抛出的还多。

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.text.ParseException;
    class A {
        public void method() throws IOException { }
    }
    class B extends A {  public void method() throws FileNotFoundException {  }
    }
    class C extends A {  public void method() {  }
    }
//  class D extends A {  public void method() throws Exception {  }  //超过父类异常的范围，会报错！
//  }
    class E extends A {  public void method() throws IOException, FileNotFoundException {  }
    }
    class F extends A {  public void method() throws IOException, ArithmeticException {  }
    }
//  class G extends A {  public void method() throws IOException, ParseException {  }
//  }
```

（8）自定义异常

在程序中，可能会遇到任何标准异常类都没有充分的描述清楚异常的情况，这时可以自己创建异常类。

从 Exception 类或者它的子类派生一个子类即可。

习惯上，定义的类应该包含 2 个构造器：一个是默认的构造器，一个是带有详细信息的构造器。

```
public class MyException extends Exception {
    public MyException(){
    }
    public MyException(String message){
```

```
        super(message);
    }
}
class TestMyException{
    void test()throws MyException{
        ///
    }
    public static void main(String[] args) {
        try {
            new TestMyException().test();
        } catch (MyException e) {
            e.printStackTrace();
        }
    }
}
```

（9）异常小结

- 要避免使用异常处理代替错误处理，这样会降低程序清晰性，并且效率低。
- 处理异常不可以代替简单测试，只在异常情况下使用异常机制。
- 不要进行小粒度的异常处理，应该将整个任务包装在一个 try 语句块中。
- 异常往往在高层处理。

15、新增知识点

（1）自动装箱与拆箱、享元模式（flyweight）：有很多小对象，它们的大部分属性相同，这是可以把它们变成一个对象，那些相同的属性为对象的内部状态，那些不同的属性可以变为方法的参数，由外部传入。例：-128~127 内的相同整数自动装箱为同一个对象。

（2）增强 for 循环（迭代变量必须在()中定义，集合部分必须实现 Iterable 接口）。

for(type 变量名：集合变量名){}

（3）可变参数（可有限替代 overload）：

- 使用场合：一个方法接收的参数个数不固定时；
- 可变参数只能出现在参数列表的最后；
- …位于变量类型和变量名之间，前后有无空格都可；
- 调用可变参数的方法时，编译器为该可变参数隐含创建一个数组，在方法体中以数组的形式访问可变参数。

（4）静态导入 import static java.lang.Math.max/Math.*

（5）枚举。枚举就是要让某个类型的变量的取值只能为若干个固定值的一个，否则，编译器就会报错。枚举可以让编译器在编译时就可以控制源程序中填写的非法值，普通变量的方式在开发阶段无法实现。

（1.1）用普通类模拟枚举的实现

```
public class WekDay {
    public final static WekDay SUN = new WekDay();
    public final static WekDay MON = new WekDay();
    private WekDay(){}
    public WekDay nextDay(){
        if (this == MON){
            return SUN;
        }else{
            return MON;
        }
    }
    public String toString(){
        return this==MON?"MON":"SUN";
    }
}
public abstract class WekDay {
    public final static WekDay SUN = new WekDay(){
        public WekDay nextDay() {
            return MON;
        }
    };
    public final static WekDay MON = new WekDay(){
        public WekDay nextDay() {
            return SUN;
        }

    };
    private WekDay(){}

    public abstract WekDay nextDay();
    public String toString(){
        return this==MON?"MON":"SUN";
    }
}
public static void main(String[] args) {
    WekDay wekDay = WekDay.SUN;
    System.out.println(wekDay.nextDay());
}
```

（1.2）枚举的基本使用

- 枚举有相当于一个类，其中也可以定义构造方法（私有）、成员变量、普通方法和抽象方法；
- 枚举元素必须位于枚举体的最开始部分，枚举元素列表的后面要有分号与其他成员分隔。
- 枚举只有一个成员时，可以作为一种单例的实现方式。

```
public class Test {
    public static void main(String[] args) {
        WekDayEnum wekDayEnum =
```

```
WekDayEnum.SUN;
            System.out.println(wekDayEnum.name());
            System.out.println(wekDayEnum.ordinal());

        System.out.println(WekDayEnum.valueOf("SUN").to
String());
        System.out.println(WekDayEnum.values().length);
        }
        public enum WekDayEnum {
            SUN,MON;
        }
}
```

（1.3）带有构造方法的枚举

```
        public enum WekDayEnum {
            SUN(1),MON;
            private
WekDayEnum(){System.out.println("sun");}
            private WekDayEnum(int
i){System.out.println("mon");}
        }
```

（1.4）带有抽象方法的枚举

```
        public enum TrafficLamp {
            RED(30){
                public TrafficLamp nextLamp() {
                    return GREEN;
                }
            },GREEN(30){
                public TrafficLamp nextLamp() {
                    return YELLOW;
                }
            },YELLOW(5){
                public TrafficLamp nextLamp() {
                    return RED;
                }
            };
            public abstract TrafficLamp nextLamp();
            private int time;
            private TrafficLamp(int time){this.time =
time;}
        }
```

16、Java 中的反射技术

（1）动态语言：

程序运行时，可以改变程序结构或变量类型。典型的语言包括：Python，ruby，javascript 等等。如下面的 js 代码：

```
Function test(){
    var s = “var a = 3; var b = 5; alert(a + b);”
    eval(s);
}
```

而 C, C++, Java 不是动态语言。不过，Java 有一定的动态性：可以利用反射机制、字节码操作获得类似动态语言的特性。

反射（reflection）机制：

指的是可以在程序运行时加载、探知、使用编译期间完全未知的类。（*反射就是把 Java 类中的各种成分映射成相应的 java 类。例如，一个 Java 类用一个 Class 类的对象来表示，一个类中的组成部分：成员变量、方法、构造方法、包等等信息也用一个个类来表示。就像汽车是一个类，汽车中的发动机，变速箱等也是一个个类。表示 java 类的 Class 类显然要提供一系列的方法，来获取其中的变量、方法、构造方法、修饰符、包等信息，这些信息就用相应类的实例对象来表示，他们是 Field、Method、Contructor、Package 等这些类。一个类中的每个成员都可以用相应的反射 API 类的一个实例对象来表示，通过调用 Class 类的方法可以得到这些实例对象。*）

程序在运行状态下，可以动态加载只有名称的类，对于任意一个已加载的类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性。

Class cls1 = Class.forName(“java.util.Date”);

加载完类之后，在堆内存中，就产生了一个 Class 类型的对象（一个类只有一个 Class 对象），这个对象就包含了完整的类的结构信息。我们可以通过这个对象看到类的结构。这个对象就像一面镜子，透过这个镜子可以探究类的结构。因此，我们将这一技术称之为“反射”。

反射机制常见的作用：动态加载类、动态获取类的信息（属性、方法、构造器）；动态构造对象；动态调用类和对象的任意方法、构造器；动态调用和处理属性；获取泛型信息；处理注解。

Class 类解析：

Java 类用于描述一类事物的共性，定义该类事物有什么属性，没有什么属性。至于这个属性的值是什么，则由这个类的实例对象来确定，不同的实例对象有不同的属性值。Java 程序中的各个 java 类，也属于同一类事物，可以用一个类型来描述，这个类就是 java.lang.Class（这个特殊的类用来表示 Java 中的类型（class/interface/enum/annotation/primitive type/void）本身）。Class 类描述了类的名字，类的访问属性，类所属于的包名，字段名称的列表、方法名称的列表，等等。

Class 类的各个实例对象就是各个类在内存中的字节码，包含了各个被加载类的结构。一个被加载的类对应一个 Class 对象。当一个 class 被加载，或当加载器 class loader 的 defineClass()被 JVM 调用，JVM 便自动产生一个 Class 对象。

Class 类是 Reflection 的根源。针对任何你想动态加载并运行的类，唯有先获得相应的 Class 对象。

（2）得到各个字节码对应的实例对象（Class 类型）

```
Class cls1 = Date.class;
Class cls1 = new Date().getClass();
Class cls1 = Class.forName("java.util.Date");//最灵活
System.out.println(cls1 == cls2); true
```

（3）八个基本类型（boolean, byte, char, short, int, long, float, double）以及 void 都是 Class 的实例对象。

```
System.out.println(int.class.isPrimitive()); true
System.out.println(int.class == Integer.class); false
System.out.println(int.class == Integer.TYPE); true
System.out.println(int[].class.isPrimitive());false
System.out.println(int[].class.isArray());true
```

（4）Constructor 类：其各个实例分别代表某个类中的一个构造方法。

```
Constructor [] constructor =
  Class.forName("java.lang.String").getConstructors();///
Constructor constructor =
  Class.forName("java.lang.String").
getConstructor(StringBuffer.class);//可以根据 String 类提供的各个构造方法来为写对应个数的形参，因为 getConstructor 方法的形参为可变参数 Class<?> …，如果不能用可变参数，可以用数组来代替。
```

使用反射来完成 new String(new StringBuffer("abc"))：

```
Constructor cons =
String.class.getConstructor(StringBuffer.class);
String str = (String)cons.newInstance(/*"abc"*/new StringBuffer("abc"));
System.out.println(str.charAt(2));
String obj =
(String)Class.forName("java.lang.String").newInstance();
```

该方法内部先得到默认的构造方法，然后用该构造方法创建实例对象。（用到了缓存机制来保存默认构造方法的实例对象）**参看 java 的 Class 类源码**

（5）Field 类：代表某个类中的一个成员变量，其某一个实例代表对应类的对应变量的定义。

```
ReflectPoint pt1 = new ReflectPoint(3, 5);
Field fieldY = pt1.getClass().getField("y");//成员变量 y 的定义
fieldY.get(pt1);//5
System.out.println(pt1.getClass().getField("x").get(pt1));//
```

私有成员 x 看不见，报错。需要换成 getDeclaredField 方法（这种方法看得见 x，但是无法访问），可以进一步加一行 fieldX.setAccessible(true);则不仅可以看到 private 的变量，还可以访问到。这种代码称为暴力反射（去抢变量）。

作业：将任意一个对象中的所有 String 类型的成员变量所对应的字符串内容中的"b"改成"a"。

```
public class ReflectPoint{
    private int x;
    public int y;
    public String str1 = "ball";
    public String str2 = "basketball";
    public String str3 = "str";
    public ReflectPoint(int x, int y) {
        super();
        this.x = x;
        this.y = y;
    }
    @Override
    public String toString(){return str1 + str2+str3;}
}
private static void changeStringValue(Object obj) throw Exception{
    Field[] fields = obj.getClass().getFields();
    for(Field field : fields) {
        //if(field.getType().equals(String.class))
        if(field.getType() == String.class){
            String oldValue = (String)field.get(obj);
            String newValue = oldValue.replace('b', 'a');
            field.set(obj, newValue);
        }
    }
}
changeStringValue(pt1);
System.out.println(pt1);
```

（6）Method 类：代表某个类中的一个成员方法。

```
得到类中某一个方法：Method charAt =
Class.forName("java.lang.String").getMethod("charAt",int.class);
调用方法：通常方式：System.out.println(str.charAt(1));
反射方式：System.out.println(charAt.invoke(str,1));
```

如果传递给 Method 对象的 invoke()方法的第一个参数为 null，说明该 Method 对象对应的是一个静态方法。

Jdk1.5 之前的 invoke 方法，没有可变参数一说，需要将一个数组作为 invoke 方法的形参，即 public Object invoke(Object obj, Object[] args)，数组中每个元素分别对

应被调用方法中的一个参数。因此，也可以这样调用 invoke 方法：charAt.invoke(str, new Object[]{1});

作业：写一个程序，这个程序能够根据用户提供的类名，去执行该类中的 main 方法。理解为什么要用反射的方式调。

问题提示：启动 Java 程序的 main 方法的参数是一个字符串数组，即 public static void main(String[] args)，通过反射方式来调用这个 main 方法时，如何为 invoke 方法传递参数？按照 jdk1.5 之后的语法，整个数组是一个参数，而按照 jdk1.5 之前的语法，数组中的每个元素对应一个参数，当吧一个字符串数组作为参数传递给 invoke 方法时，java 会按照哪种语法处理？按照向后兼容思想，肯定会把数组打散成为若干个单独的参数。所以，在给 main 方法传递参数时，不能使用代码 mainMethod.invoke(null, new String[]{"***"})，javac 只会把它当作 jdk1.5 之前的语法进行理解，因此会出现参数类型不对的问题。

解决方法：

```
class TestArguments{
    public static void main(String[] args){
        for(String arg : args){
            System.out.println(arg);
        }
    }
}
//常用方式：
//TestArguments.main(new String[]{"111"});
public class ReflectTest {
public static void main(String[] args){
Class clazz = Class.forName(arg[0]);//arg[0]为执行时传递进去的类名。
(run as → Java ReflectTest cn.day.TestArguments)
Method mainMethod = clazz.getMethod("main", String[].class);
mainMethod.invoke(null, new Object[]{new String[]{"111"}})
mainMethod.invoke(null, (Object)new String[]{"111"});
}}
```

（7）反射操作泛型：

Java 采用泛型擦除的机制来引入泛型。Java 中的泛型仅仅是给编译器 javac 使用的，确保数据的安全性和免去强制类型转换的麻烦。但是，一旦编译完成，所有的和泛型相关的类型全部擦除。

为了通过反射操作这些类型以满足实际开发的需要，Java 就新增了 ParameterizedType, GenericArrayType, TypeVariable 和 WildcardType 几种类型来代表不能被归一到 Class 类中的类型但是又和原始类型齐名的类型。

ParameterizedType: 表示一种参数化的类型，比如 Collection<String>;

GenericArrayType: 表示一种元素类型是参数化类型或者类型变量的数组类型;

TypeVariable: 是各种类型变量的公共父接口;

WildcardType: 代表一种通配符类型表达式，比如?, ? extends Number, ? super Integer.

（8）反射操作注解：

可以通过反射 API: getAnnotations, getAnnotation 获得相关的注解信息。

（9）反射机制的性能问题：

setAccessible: 启用和禁用访问安全检查的开关，值为 true 则指示反射的对象在使用时应该取消 Java 语言访问检查。值为 false 则指示反射的对象应该实施 Java 语言访问检查。并不是为 true 就能访问为 false 就不能访问。而禁止安全检查，可以提高反射的运行速度。

（10）数组的反射：

- 具有相同维数的元素类型的数组属于同一个类型，即具有相同的 Class 实例对象。
- 代表数组的 Class 实例对象的 getSuperClass()方法返回的父类为 Object 类对应的 Class。
- 基本类型的一维数组可以被当作 Object 类型使用，不能当作 Object[]类型使用；非基本类型的一维数组，既可以当作 Object 类型使用，也可以当作 Object[]类型使用。
- Arrays.asList[]方法处理 int[]和 String[]时的差异。

```
int[] a1 = new int[3];
int[] a2 = new int[4]
int[][] a3 = new int[2][3];
String[] a4 = new String[3];
System.out.println(a1.getClass() = = a2.getClass());//true
System.out.println(a1.getClass() = = a4.getClass());//false
System.out.println(a1.getClass() = = a3.getClass());//false
System.out.println(a1.getClass().getSimpleName ());//int[]
System.out.println(a1.getClass().getSuperclass().getName( ));//java.lang.Object
System.out.println(a4.getClass().getSuperclass().getName( ));//java.lang.Object
Object obj1 = a1;
Object obj2 = a4;
Object[] obj3 = a1;//错
Object[] obj4 = a3;
Object[] obj5 = a4;
```

```
int[] a1 = new int[] {1,2,3};
String[] a4 = new String[] {"a","b","c"};
System.out.println(a1);//整数数组类型@hashcode 值
System.out.println(a4);//字符串数组类型@hashcode 值
```

System.out.println(Arrays.asList(a1));//按照向后兼容原则，首先按照 1.5 之前的语法，即当成一个 Object[]数组处理，发现没法对应上，就转为 1.5 之后的语法，即当作可变参数中的一个参数进行处理。因此转化的 list 中只有一个对象，而这个对象是一个整形的数组。
System.out.println(Arrays.asList(a4));//同样原理，因为和 1.5 之前的语法对应上了，所以就按照一个字符串数组处理，因此转化的 list 中有三个对象，这三个对象都是字符串类型对象。

（11）反射的作用：

- **框架的定义**

通过框架提供针对某一问题的通用解决方案，用户基于通用框架进行定制化开发。用户写的类和框架的类之间的关系：框架的类调用用户提供的类。其与我们经常用到的工具类存在的区别：工具类有用户写的类进行调用。

- **框架需要解决的核心问题**

写框架代码中的各种类的时候，使用该框架的用户还未进行自定义类的编写，如何保证框架程序能够调用还未编写出来的用户类：因为在写框架程序时无法知道要被调用的用户类名，所以，在框架程序中无法直接 new 某个用户类的实例对象，必须借助反射方式来实现。

config.properties
className=java.util.ArrayList　(java.util.HashSet)

```
public class ReflectTest{
public static void main(String[] args) throws Exception {
    InputStream ips
= new FileInputStream("config.properties");
    Properties propes = new Properties();
    propes.load(ips);
    ips.close();
    String className = propes.getProperty("className");
    Collection collections
= (Collection) Class.forName(className).newInstance();
    ReflectPoint pt1 = new ReflectPoint(3,3);
    ReflectPoint pt2 = new ReflectPoint(4,4);
    ReflectPoint pt3 = new ReflectPoint(3,3);
    collections.add(pt1);
    collections.add(pt2);
    collections.add(pt3);
    collections.add(pt1);
    System.out.println(collections.size());
}
}

public class ReflectPoint{
    private int x;
    public int y;
    public String str1 = "ball";
    public String str2 = "basketball";
    public String str3 = "str";
    public ReflectPoint(int x, int y) {
        super();
        this.x = x;
        this.y = y;
    }
    @Override
    public int hashCode(){
        final int prime = 31;
        int result = 1;
        result = prime * result + x;
        result = prime * result + y;
        return result;
    }
    @Override
    public Boolean equals(Object obj){
        if(this = = obj) return true;
        if(obj = = null) return false;
        if(getClass() != obj.getClass()) return false;
        final ReflectPoint other = (ReflectPoint)obj;
        if(x!=other.x) return false;
        if(y!=other.y) return false;
        return true;
    }
    @Override
    public String toString(){return str1 + str2+str3;}
}
```

扩展之动态编译：

动态编译的应用场景：
可以做一个浏览器端编写 Java 代码，上传到服务器编译和运行的在线测评系统。由服务器动态加载这些类文件进行编译。具体做法：

1） 通过 JavaCompiler 动态编译：调用 JavaCompiler 的 run 方法实现，该方法需要传递如下参数：第一个参数为 Java 编译器提供参数；第二个参数得到 Java 编译器的输出信息；第三个参数接收编译器的错误信息；第四个参数为可变参数（是一个 String 数组）能传入一个或多个 Java 源文件；返回值如果为 0 表示编译成功，如果为非 0 表示编译失败。

2） 通过 Runtime 调用 javac 启动新的进程去操作。（JDK1.6 以前的做法，本质上并非动态编译！）

```
Runtime run = Runtime.getRuntime();
Process process = run.exe("javac –cp c:/java/
```

HelloWorld.java");

3） 通过反射运行编译好的类。

脚本引擎执行 javascript 代码：

使得 Java 应用程序可以通过一套固定的接口与各种脚本引擎交互，从而达到在 Java 平台上调用各种脚本语言的目的。Java 脚本 API 是连通 Java 平台和脚本语言的桥梁。可以把一些复杂异变的业务逻辑交给脚本语言处理，这将极大提高开发效率。

获得脚本引擎对象：

ScriptEngineManager sem = new ScriptEngineManager();

ScriptEngine engine = sem.getEngineByName("javascript");

Java 脚本 API 为开发者提供了如下功能：

获取脚本程序输入，通过脚本引擎运行脚本并返回运行结果，这是最核心的接口。注意是接口！

Java 可以使用各种不同的实现，从而通用的调用 javascript，groovy，python 等脚本。其中 javascript 使用的是 Nashorn（JDK1.6 之后是 Rhino，JDK1.8 之后换成了执行速度快很多的 Nashorn）。Nashorn 是一种使用 java 语言编写的 javascript 的开源实现，已被集成进 JDK。

可以通过脚本引擎的运行上下文在脚本和 Java 平台之间交换数据。通过 Java 应用程序调用脚本函数。

Java 的字节码操作：

Java 动态性的两种常见实现方式：字节码操作和反射机制。

运行时操作字节码可以让我们实现如下功能：动态生成新的类；动态改变某个类的结构（添加/删除/修改 新的属性/方法）

字节码操作比反射机制的开销小，性能高。

常见的字节码操作类库：

- **BCEL**（Byte Code Engineering Library），是 Apache Software Foundation 的 Jakarta 项目的一部分。BCEL 是 Java classworking 广泛使用的一种框架，它可以深入 JVM 汇编语言进行类操作的细节。BCEL 与 Javassist 有不同的处理字节码方法，BCEL 在实际的 JVM 指令层次上进行操作（BCEL 拥有丰富的 JVM 指令级支持）而 Javassist 所强调的是源代码级别的工作。
- **ASM** 是一个轻量级 Java 字节码操作框架，直接涉及到 JVM 底层的操作和指令。
- **CGLIB**（Code Generation Library）是一个强大的，高性能，高质量的 Code 生成类库，基于 ASM 实现。
- **Javassist** 是一个开源的分析、编辑和创建 Java 字节码的类库，性能较 ASM 差，跟 CGLIB 差不多，使用简单。

Javassist 的最外层 API 和 Java 的反射包中的 API 类似。主要由 CtClass、CtMethod、CtField 几个类组成。用以执行和 JDK 反射 API 中 java.lang.Class 、java.lang.reflect.Method、java.lang.reflect.Field 相同的操作。

JVM 运行和类加载全过程：

- 类加载过程：

JVM 把 class 文件加载到内存，并对数据进行验证、准备、解析和初始化，最终形成 JVM 可以直接使用的 Java 类型的过程。

加载：类加载器将 class 文件字节码内容（来源于硬盘、网络、数据库等等各种来源）加载到内存中，并将这些静态数据转换成方法区中的运行时数据结构，同时在堆中生成一个代表这个类的 java.lang.Class 对象（反射机制），作为方法区类数据的访问入口。

链接：将 Java 类的二进制代码合并到 JVM 的运行状态之中的过程：（1）验证：确保加载的类信息符合 JVM 规范，没有安全方面的问题；（2）准备：正式为类变量（static 变量）分配内存并设置类变量初始值的阶段，这些内容都将在方法区中进行分配；（3）解析：JVM 常量池的符号引用替换为直接引用的过程。

初始化：（1）初始化阶段是执行类构造器<clinit>()方法的过程。类构造器<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static 块）中的语句合并产生的；（2）当初始化一个类的时候，如果发现其父类还没有进行过初始化、则需要先触发其父类的初始化；（3）JVM 会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步。

类的主动引用（一定会发生类的初始化）

- new 一个类的对象；
- 调用类的静态成员（除了 final 常量）和静态方法；
- 使用 java.lang.reflect 包的方法对类进行反射调用；
- 当 JVM 启动，java Hello，则一定会初始化 Hello 类；
- 当初始化一个类，如果其父类没有被初始化，则先初始化它的父类。

类的被动引用（不会发生类的初始化）

- 当访问一个静态域时，只有真正声明这个域的类才会被初始化；（通过子类引用父类的静态变量，不会导致子类初始化）
- 通过数组定义类时，不会触发此类的初始化；
- 引用常量不会触发此类的初始化（常量在编译阶段就存入调用类的常量池中了！）

JVM 类加载器深入：

类加载器的作用：类加载器将 class 文件字节码内容（来源于硬盘、网络、数据库等等各种来源）加载到内存中，并将这些静态数据转换成方法区中的运行时数据结构，同时在堆中生成一个代表这个类的 java.lang.Class 对象（反射机制），作为方法区类数据的访问入口。

类缓存：标准的 Java SE 类加载器可以按照要求查找类，但一旦某个类被加载到类加载器中，它将维持加载（缓存）一段时间。不过，JVM 垃圾收集器可以回收这些 Class 对象。

类加载器的层次结构：

- 引导类加载器（bootstrap class loader）：用来加载 Java 的核心库（JAVA_HOME/jre/lib/rt.jar 或 sun.boot.class.path 路径下的内容），它是原生代码实现的，并不继承自 java.lang.ClassLoader；
- 扩展类加载器（extensions class loader）：用来加载 Java 的扩展库（JAVA_HOME/jre/ext/*.jar，或 java.ext.dirs 路径下的内容）。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。其由 sun.misc.Launcher$ExtClassLoader 实现。
- 应用程序类加载器（application class loader）：它根据 Java 应用的类路径（classpath）来加载。Java 应用的类都是由它来完成加载的。其由 sun.misc.Launcher$AppClassLoader 实现。
- 自定义类加载器：开发人员可以通过继承 java.lang.ClassLoader 类的方式实现自定义的类加载器。

java.lang.ClassLoader 类：

java.lang.ClassLoader 类的职责是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个 Java 类，即 java.lang.Class 的一个实例。

类加载器的代理模式：

代理模式即交给其他加载器来加载指定的类。在这里采用的是代理模式中的双亲委托机制。就是某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次回溯，直到最高的根加载器。如果父类加载器可以完成类加载任务，就成功返回。当父类加载器无法完成加载任务时，才自己尝试去加载。

这种机制可以很好的保证 Java 核心库的类型加载安全。比如：如果用户想自定义一个 java.lang.Object 类，这种机制可以防止实际加载时加载的还是核心包中的 Object 类。

自定义类加载器：

- 继承 java.lang.ClassLoader
- 首先检查请求的类型是否已经被这个类加载器加载到命名空间中，如果已经装载，直接返回。
- 委派类加载请求给父类加载器，如果父类加载器能够完成，则返回父类加载器加载的 Class 实例。
- 如果父类加载不了，调用本类加载器的 findClass 方法，试图获取对应的字节码，如果得到则调用 defineClass 方法导入类型到方法区。如果获取不到对应的字节码或其他原因失败，返回异常给 loadClass 方法，loadClass 方法转抛异常，终止加载过程。
- *注意：被两个类加载器加载的同一个类，JVM 不认为是相同的类。*

17、Java 中的注解

1、注解（Annotation）相当于一种标记，在程序中加了注解就等于为程序打上某种标记。

Annotation 的作用：

- 不是程序本身，可以对程序做出解释（这一点和注释 comment 没有什么区别）。
- 可以被其他程序（比如 javac 编译器等）读取。javac 编译器、开发工具和其它程序可以用反射来了解你的类及各种元素上有无何种标记，看你有什么标记，就去干相应的事情。（即存在对应的注解信息处理代码，这也是注解和注释的最大区别。如果没有对应的注解信息处理代码，则注解毫无意义）

Annotation 的格式：

- 注解是以“@注释名”在代码中存在的，还可以添加一些参数值，例如：@SuppressWarings(value=”unchecked”)。

Annotation 的使用位置：

- 标记（即注解）可以加在包、类、成员变量、方法、方法的参数以及局部变量上。相当于给它们添加了额外的辅助信息，我们可以通过反射机制编程实现对这些元数据的访问。

2、了解 java 原生的三个注解：@SuppressWarings、@Deprecated、@Override。

- @SuppressWarings：定义在 java.lang.SuppressWarnings 中，用来抑制编译时的警告信息。这个注解需要添加一个参数才能正确使用，这些参数值都是事先定义好的，我们选择性的使用即可。如下：deprecation:使用了过时的类或方法的警告；unchecked:执行了未检查的转换时的警告，如使用集合时未指定泛型；fallthrough:当在 switch 语句使用时发生 case 穿透；path:在类路径、源文件路径等中不存在路径的警告；serial:当在可序列化的类上缺少 serialVersionUID 定义时的警告；finally:任何 finally 子句不能完成时的警告；all:关于

以上所有情况的警告。

@SuppressWarnings(“unchecked”)

@SuppressWarnings(value={“unchecked”,”deprecation”})

- @Deprecated：定义在 java.lang.Deprecated 中，此注释可用于修饰方法、属性、类表示不鼓励程序员使用这样的对象，通常是因为它已经很危险或已经存在更好的替换选择。
- @Override：定义在 java.lang.Override 中，此注释只适用于修饰方法，表示一个方法打算重写父类中的另一个方法。

```
public class AnnotationTest{
    @SupressWarings(“deprecation”)
    public static void main(String[] args){
        System.runFinalizersOnExit(true);
    }
    @Deprecated
    public static void sayHello(){
    }
    @Override
    public String toString(){}
}
```

（因为 Object 中的 equals 方法的参数为 Object 类型，所以实际上不是重写，加上@Override 会报错）

```
    @Override
    public Boolean equals(ReflectPoint obj){
        if(this = = obj) return true;
        if(obj = = null) return false;
        if(getClass() != obj.getClass()) return false;
        final ReflectPoint other = (ReflectPoint)obj;
        if(x!=other.x) return false;
        if(y!=other.y) return false;
        return true;
    }
```

3、自定义注解：

（1）注解类：@interface 用来声明一个注解。用@interface 自定义注解时，自动继承了 java.lang.annotation.Annotation 接口。

```
public @interface A{
}
```

其中的每一个方法实际上是声明了一个配置参数：方法的名称就是参数的名称；返回值类型就是参数的类型（返回值类型只能是基本类型、Class、String、enum）；可以通过 default 来声明参数的默认值；如果只有一个参数成员，一般参数名为 value。

（2）应用了“注解类”的类：

```
@A
Class B{}
```

（3）对“应用了注解类的类”进行反射操作的类：

```
Class C{
    B.class.isAnnotionPresent(A.class);
    A a = B.class.getAnnotion(A.class);
}
```

4、元注解：

- 元注解的作用就是负责注解其他的注解，对其它的注解作解释的注解。Java 定义了 4 个标准的 meta-annotation 类型，它们被用来提供对其它 annotation 类型作说明。@Target;@Retention;@Documented;@Inherited。这些类型和它们所支持的类在 java.lang.annotation 包中可以找到。
- @Retention 元注解：包括 RetentionPolicy.SOURCE（java 源文件存在该注解）、RetentionPolicy.CLASS（class 文件存在该注解）、RetentionPolicy.RUNTIME（加载到内存中的字节码存在该注解，为 RUNTIME 时，则可以使用反射机制读取）/**默认只保持在 CLASS 阶段**

思考题：考虑@Override、@SuppressWarnings 和@Deprecated 注解的 Retention 属性值分别为什么？

- @Target 元注解的使用（用于描述注解的使用范围，即被描述的注解可以用在什么地方）：Target 的默认值为任何元素，即标识自定义注解可以加到任何元素上。其值可以取 ElementType.METHOD、FIELD、CONSTRUCTOR、LOCAL_VARIABLE、PACKAGE、PARAMETER、TYPE（类、接口、枚举、Annotation 类型）。

5、为注解增加基本属性：

可以在增加注解的基础上进一步增加属性来更细粒度的对目标对象进行标注。

（1）定义基本类型的属性和应用属性：

在注解类中增加 String color();

在应用“注解类”类上修改@MyAnnotation(color=”red”)

（2）用反射方式获得注解对应的实例对象，再通过该对象调用属性对应的方法：

```
MyAnnotation a =
(MyAnnotation)AnnotationTest.class.getAnnotation(MyAnnotation.class);
System.out.println(a.color());
```

可以认为上面的@MyAnnotation 是 MyAnnotation 类的一个实例对象。

（3）为属性制定缺省值：

String color() default “blue”;

（4）Value 属性（特殊属性，用的时候可以省略 value=）：

String value() default “abc”;

（5）数组类型的属性：

int[] arrayAttr() default{1,2,3};

@MyAnnotation(arrayAttr={2,3,4})

如果数组属性中只有一个元素，这时候属性值部分可以省略数组大括号。

（6）枚举类型的属性

EnumTest.TrafficLamp lamp();

@MyAnnotation(lamp=EnumTest.TrafficLamp.GREEN)

（7）注解元素必须要有值。我们定义注解元素时，经常使用空字符串、0 作为默认值。也经常使用负数表示不存在的含义。

6、注解练习：

使用注解实现一个 ORM，即使用注解完成类和表结构的映射关系。

18、JAVA 多线程

（1）线程：代码一行行向下执行的线路和流程，多线程就是分出另外一条执行线路，同时有多条线路并行运行。

（2）创建线程的两种传统方式

- 在 Thread 子类覆盖的 run 方法中编写运行代码

  涉及一个以往知识点：能否在 run 方法声明中抛出 InterruptedException 异常，以便省略 run 方法内部对 Thread.sleep()语句的 try…catch 处理？
- 在传递给 Thread 对象的 Runnable 对象的 run 方法中编写代码

总结：查看 Thread 类的 run()方法的源代码，可以看到其实这两种方式都是在调用 Thread 对象的 run 方法，如果 Thread 类的 run 方法没有被覆盖，并且为该 Thread 对象设置了一个 Runnable 对象，该 run 方法会调用 Runnable 对象的 run 方法。

问题：如果在 Thread 子类覆盖的 run 方法中编写了运行代码，也为 Thread 子类对象传递了一个 Runnable 对象，那么，线程运行时的执行代码是子类的 run 方法的代码？还是 Runnable 对象的 run 方法的代码？（弄清楚匿名内部类对象的构造方法如何调用父类的非默认构造方法？）

```
public class TraditionalThread {
    public static void main(String[] args) {
        Thread thread = new Thread(){
            @Override
            public void run() {
                while (true) {
                    try {
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("1:
"+Thread.currentThread().getName());
                    System.out.println("2:
"+this.getName());
                }
            }
        };
        thread.start();

        Thread thread2 = new Thread(new Runnable(){

            @Override
            public void run() {
                while (true) {
                    try {
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("3:
"+Thread.currentThread().getName());
                    //System.out.println("2:
"+this.getName());
                }
            }

        });
        thread2.start();

        new Thread(
            new Runnable() {
                public void run() {
                    while (true) {
                        try {
                            Thread.sleep(500);
                        } catch
(InterruptedException e) {
                            e.printStackTrace();
                        }

    System.out.println("runnable :
"+Thread.currentThread().getName());
                    }
                }
            }
        ){
```

```
            public void run() {
                while (true) {
                    try {
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("thread :
"+Thread.currentThread().getName());
                }
            }
        }.start();
    }
}
```

（3）定时器的应用

Timer 类

TimerTask 类

问题：如何让一个事件 2 秒后发生，然后 4 秒后再发生，然后又 2 秒后发生，如此往复？

```
public class TraditionalTimerTest {
    private static int count = 0;
    public static void main(String[] args) {
        /*10秒后炸，然后每隔3秒炸
         * new Timer().schedule(new TimerTask() {
            @Override
            public void run() {
                System.out.println("Bombing!");
            }
        }, 10000, 3000);*/

        /*炸两次
        new Timer().schedule(new TimerTask() {
            @Override
            public void run() {
                System.out.println("Bombing!");
                new Timer().schedule(new TimerTask()
{
                    @Override
                    public void run() {

    System.out.println("Bombing!");

                    }
                }, 2000);
            }
        }, 2000);*/

        /*3秒后炸，然后每隔2秒炸
         * class MyTimerTask extends TimerTask {
            @Override
            public void run() {
                System.out.println("Bombing!");
                new Timer().schedule(new
MyTimerTask(), 2000);
            }
        }

        new Timer().schedule(new MyTimerTask(),
3000);*/

        //3秒后炸，然后隔2秒炸一次，然后3秒炸一次，依次循环
        class MyTimerTask extends TimerTask {

            @Override
            public void run() {
                count = (count+1)%2;
                System.out.println("Bombing!");
                new Timer().schedule(new
MyTimerTask(), 2000+2000*count);
            }
        }

        new Timer().schedule(new MyTimerTask(),
3000);

        while (true) {
            System.out.println(new
Date().getSeconds());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        }
    }
}
```

（4）线程的互斥问题

- 线程安全问题（代码的原子性：有个线程来执行我的时候，别的线程就不能来执行我）（多个线程在操作共享的数据；操作共享数据的线程代码有多条。当一个线程在执行操作共享数据的多条代码过程

中，其它线程参与了运算，就会导致线程安全问题的产生。）

- 使用 synchronized 代码块及其原理（一段代码或两段代码被两个线程执行时要互斥，则需用 synchronized 代码块包围起来）
- 使用 synchronized 方法
- 分析静态方法所使用的同步监视器对象是什么？

```
public class TraditionalThreadSynchronization {
    public static void main(String[] args) {
        new TraditionalThreadSynchronization().init();
    }
    private void init(){
        final Outputer outputer = new Outputer();
        new Thread(new Runnable() {
            @Override
            public void run() {
                while(true){
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated
catch block
                        e.printStackTrace();
                    }

    outputer.output1("11111111111111111111");
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                while(true){
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated
catch block
                        e.printStackTrace();
                    }

    outputer.output3("22222222222222222222");
                }
            }
        }).start();
    }

    /*class Outputer{
        public void output(String name){
            int len = name.length();
            synchronized (this)
            {
                for (int i = 0; i < len; i++) {
                    System.out.print(name.charAt(i));
                }
                System.out.println();
            }
        }
        public synchronized void output2(String name){
            int len = name.length();
                for (int i = 0; i < len; i++) {
                    System.out.print(name.charAt(i));
                }
                System.out.println();
        }
    }*/
    //多个synchronized会死锁
    static class Outputer{
        public void output1(String name){
            int len = name.length();
            synchronized (Outputer.class)//this
            {
                for (int i = 0; i < len; i++) {
                    System.out.print(name.charAt(i));
                }
                System.out.println();
            }
        }
        public synchronized void output2(String
name){
            int len = name.length();

                for (int i = 0; i < len; i++) {
                    System.out.print(name.charAt(i));
                }
                System.out.println();

        }
        public static synchronized void output3(String
name){
            int len = name.length();
                for (int i = 0; i < len; i++) {
                    System.out.print(name.charAt(i));
                }
```

```
                System.out.println();
            }
        }
}
```

（5）Lock 线程锁和读写锁技术

Lock 比传统线程模型中的 synchronized 方式更加面向对象，与生活中的锁类似，锁本身也应该是一个对象。两个线程执行的代码片段要实现同步互斥的效果，它们必须用同一个 Lock 对象。锁是上在代表要操作的资源的类的内部方法中，而不是线程代码中。

读写锁：分为读锁和写锁，多个读锁不互斥，读锁和写锁互斥，写锁和写锁互斥，这是由 JVM 自己控制的，我们只要上好相应的锁即可。如果代码只是读数据，可以很多人同时读，但不能同时写，那就上读锁；如果你的代码修改数据，只能有一个人在写，且不能同时读取，那就上写锁。总之，读的时候上读锁，写的时候上写锁。

```
class CachedData {
    Object data;
    volatile boolean cacheValid;
    ReentrantReadWriteLock rwl = new
ReentrantReadWriteLock();

    void processCachedData() {
      rwl.readLock().lock();
      if (!cacheValid) {
          // Must release read lock before acquiring write
lock
          rwl.readLock().unlock();
          rwl.writeLock().lock();
          // Recheck state because another thread might
have acquired
          //     write lock and changed state before we did.
          if (!cacheValid) {
             data = ...
             cacheValid = true;
          }
          // Downgrade by acquiring read lock before
releasing write lock
          rwl.readLock().lock();
          rwl.writeLock().unlock(); // Unlock write, still
hold read
      }

      use(data);
      rwl.readLock().unlock();
    }
 }
```

```
public class ReadWriteLockTest {
    public static void main(String[] args) {
        final Queue3 q3 = new Queue3();
        for (int i = 0; i < 3; i++) {
            new Thread(){
                public void run() {
                    while (true) {
                        q3.get();
                    }
                }
            }.start();

            new Thread(){
                public void run() {
                    while(true){
                        q3.put(new
Random().nextInt(10000));
                    }
                }
            }.start();
        }
    }

}
class Queue3{
    private Object data = null;
    ReadWriteLock rwl = new
ReentrantReadWriteLock();
    public void get(){
        rwl.readLock().lock();
        try {

    System.out.println(Thread.currentThread().getName()
+" be ready to read data!");
            Thread.sleep((long)(Math.random()*1000));

    System.out.println(Thread.currentThread().getName()
+" have read data: "+data);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            rwl.readLock().unlock();
        }
    }

    public void put(Object data){
        rwl.writeLock().lock();
```

```
            try {

        System.out.println(Thread.currentThread().getName()
+" be ready to read data!");
                Thread.sleep((long)(Math.random()*1000));
                this.data = data;

        System.out.println(Thread.currentThread().getName()
+" have read data: "+data);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                rwl.writeLock().unlock();
            }
        }
}
```

```
public class LockThreadSynchronization {

    public static void main(String[] args) {
        new LockThreadSynchronization().init();
    }

    private void init(){
        final Outputer outputer = new Outputer();
        new Thread(new Runnable() {
            @Override
            public void run() {
                while(true){
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated
catch block
                        e.printStackTrace();
                    }

        outputer.output1("11111111111111111111");
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                while(true){
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated
catch block
                        e.printStackTrace();
                    }

        outputer.output3("22222222222222222222");
                }
            }
        }).start();
    }

    /*class Outputer{
        public void output(String name){
            int len = name.length();
            synchronized (this)
            {
                for (int i = 0; i < len; i++) {
                    System.out.print(name.charAt(i));
                }
                System.out.println();
            }
        }
        public synchronized void output2(String name){
            int len = name.length();
                for (int i = 0; i < len; i++) {
                    System.out.print(name.charAt(i));
                }
                System.out.println();
        }
    }*/
    //多个synchronized会死锁
    static class Outputer{
        Lock lock = new ReentrantLock();
        public void output1(String name){
            int len = name.length();
            lock.lock();
            try{
                for (int i = 0; i < len; i++) {
                    System.out.print(name.charAt(i));
                }
                System.out.println();
            }finally{
                lock.unlock();
            }
        }
        public synchronized void output2(String
name){
```

```
            int len = name.length();

                for (int i = 0; i < len; i++) {
                    System.out.print(name.charAt(i));
                }
                System.out.println();

        }
        public static synchronized void output3(String name){
            int len = name.length();
                for (int i = 0; i < len; i++) {
                    System.out.print(name.charAt(i));
                }
                System.out.println();
        }
    }
}
```

（6）线程的同步与通信问题

- wait 与 notify 实现线程间的通信

问题：子线程循环 10 次，接着主线程循环 100 次，接着又回到子线程循环 10 次，接着再回到主线程又循环 100 次，如此循环 50 次，请写出程序。

设计提示：要用到共同数据（包括同步锁）或共同算法的若干个方法应该归在同一个类身上，这种设计体现了高类聚和程序的健壮性。

存在虚假唤醒问题：while()代替 if()

```
public class TraditionalThreadCommunications {

    public static void main(String[] args) {
        final Business business = new Business();
        new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 0; i < 50; i++) {
                    /*synchronized (TraditionalThreadCommunications.class) {
                        for (int j = 0; j < 10; j++) {
                            System.out.println("sub thread: sequence of " + j + ", loop of " + i);
                        }
                    }*/
                    business.sub(i);
                }
            }
        }).start();

        for (int i = 0; i < 50; i++) {
            /*synchronized (TraditionalThreadCommunications.class) {
                for (int j = 0; j < 100; j++) {
                    System.out.println("main thread: sequence of " + j + ", loop of " + i);
                }
            }*/
            business.main(i);
        }
    }
}
class Business{
    private boolean shouldSub = true;
    public synchronized void sub(int i){
        //if(!shouldSub){
        while(!shouldSub){
            try {
                this.wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        for (int j = 0; j < 10; j++) {
            System.out.println("sub thread: sequence of " + j + ", loop of " + i);
        }
        shouldSub = false;
        this.notify();
    }
    public synchronized void main(int i){
        //if(shouldSub){
        while(shouldSub){
            try {
                this.wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        for (int j = 0; j < 100; j++) {
            System.out.println("main thread: sequence of " + j + ", loop of " + i);
        }
        shouldSub = true;
```

```
            this.notify();
        }
    }
```

（7）Condition

- Condition 的功能类似在传统线程技术中的 Object.wait 和 Object.notify 的功能。在等待 Condition 时，允许发生“虚假唤醒”，这通常作为对基础平台语义的让步。对于大多数程序，这带来的影响很小，因为 Conidtion 总是在一个循环中被等待，并测试正在被等待的状态声明。某个实现可以随意移除可能的虚假唤醒，但建议总是假定这些虚假唤醒可能发生，因此总是在一个循环中等待。
- 一个锁内部可以有多个 Condition，即有多路等待和通知，可以参考 jdk1.5 提供的 Lock 和 Condition 实现的可阻塞队列的应用案例。在传统的线程机制中一个监视器对象只能有一路等待和通知，要想实现多路等待和通知，必须嵌套使用多个同步监视器对象。

```
class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull  = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException
{
      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() throws InterruptedException {
      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();
      }
    }
  }
```

```
public class ThreeConditionThreadCommunications {

    public static void main(String[] args) {
        final Business business = new Business();
        new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 0; i < 50; i++) {
                    business.subA(i);
                }
            }
        }).start();

        new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 0; i < 50; i++) {
                    business.subB(i);
                }
            }
        }).start();

        new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 0; i < 50; i++) {
                    business.subC(i);
                }
            }
        }).start();

    }

    static class Business{
        Lock lock = new ReentrantLock();
```

```
        Condition conditionA = lock.newCondition();
        Condition conditionB = lock.newCondition();
        Condition conditionC = lock.newCondition();
        private int shouldSub = 1;

        public void subA(int i){
            lock.lock();
            try{
                //if(!shouldSub){
                while(shouldSub!=1){
                    try {
                        //this.wait();
                        conditionA.await();
                    } catch (Exception e) {
                        // TODO Auto-generated
catch block
                        e.printStackTrace();
                    }
                }
                for (int j = 0; j < 10; j++) {
                    System.out.println("A sub thread:
sequence of " + j + ", loop of " + i);
                }
                shouldSub = 2;
                //this.notify();
                conditionB.signal();
            } finally {
                lock.unlock();
            }
        }

        public void subB(int i){
            lock.lock();
            try{
                //if(!shouldSub){
                while(shouldSub!=2){
                    try {
                        //this.wait();
                        conditionB.await();
                    } catch (Exception e) {
                        // TODO Auto-generated
catch block
                        e.printStackTrace();
                    }
                }
                for (int j = 0; j < 10; j++) {
                    System.out.println("B sub thread:
sequence of " + j + ", loop of " + i);
                }
                shouldSub = 3;
                //this.notify();
                conditionC.signal();
            } finally {
                lock.unlock();
            }
        }

        public void subC(int i){
            lock.lock();
            try{
                //if(!shouldSub){
                while(shouldSub!=3){
                    try {
                        //this.wait();
                        conditionC.await();
                    } catch (Exception e) {
                        // TODO Auto-generated
catch block
                        e.printStackTrace();
                    }
                }
                for (int j = 0; j < 10; j++) {
                    System.out.println("C sub thread:
sequence of " + j + ", loop of " + i);
                }
                shouldSub = 1;
                //this.notify();
                conditionA.signal();
            } finally {
                lock.unlock();
            }
        }
    }
}
```

（8）线程范围内共享变量的概念和作用（ThreadLocal）

- ThreadLocal 的作用与目的：用于实现线程内部的数据共享，即对于相同的程序代码，多个模块在同一个线程中运行时要共享一份数据，而在另外线程中运行时又共享另外一份数据。
- 每个线程调用全局 ThreadLocal 对象的 set 方法，就相当于往其内部的 map 中增加一条记录，key 分别是各自的线程，value 是各自的 set 方法传递进去的值。在线程结束时可以调用 ThreadLocal.clear()方法，这样会更快释放内存，不调用也可，因为线程结束

后也可以自动释放相关的 ThreadLocal 变量。

- ThreadLocal 的应用场景：如 Struct2 的 ActionContext，同一段代码被不同的线程调用运行时，该代码操作的数据是每个线程各自的状态和数据，对于不同的线程来说，getContext 方法拿到的对象都不相同，对同一个线程来说，不管调用 getContext 方法多少次和在哪个模块中 getContext 方法，拿到的都是同一个。
- 实现对 ThreadLocal 变量的封装，让外界不要直接操作 ThreadLocal 变量。对于基本类型的数据的封装，这种应用很少见；对于对象类型的数据封装，较常见，即让某个类针对不同线程分别创建一个独立的实例对象。

总结：一个 ThreadLocal 代表一个变量，故其中只能放一个数据，有两个变量都要线程范围内共享，则要定义两个 ThreadLocal 对象。可以先定义一个对象来装多个要线程内共享的变量，然后在 ThreadLocal 中存储这个对象。

```
public class ThreadLocalTest {
    private static ThreadLocal<Integer> x = new ThreadLocal<Integer>();
    public static void main(String[] args) {
        for (int i = 0; i < 2; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    int data = new Random().nextInt();

    System.out.println(Thread.currentThread().getName() + " has put data: " + data);
                    x.set(data);
                    new A().get();
                    new B().get();
                }
            }).start();
        }
    }
    static class A{
        public void get(){
            int data = x.get();
            System.out.println("A from " + Thread.currentThread().getName() + " has put data: " + data);
        }
    }
    static class B{
        public void get(){
            int data = x.get();
            System.out.println("B from " + Thread.currentThread().getName() + " has put data: " + data);
        }
    }
}
```

```
public class ThreadLocalTest1 {
    private static ThreadLocal<Integer> x = new ThreadLocal<Integer>();
    public static void main(String[] args) {
        for (int i = 0; i < 2; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    int data = new Random().nextInt();

    System.out.println(Thread.currentThread().getName() + " has put data: " + data);
                    x.set(data);

    MyThreadScopeData1.getInstance().setName("name "+ data);

    MyThreadScopeData1.getInstance().setAge(data);
                    new A().get();
                    new B().get();
                }
            }).start();
        }
    }
    static class A{
        public void get(){
            int data = x.get();
            MyThreadScopeData1 myDate = MyThreadScopeData1.getInstance();
            System.out.println("A from " + Thread.currentThread().getName() + " has put data: " + data);
            System.out.println("A from " + Thread.currentThread().getName() + " getMyData: " + myDate.getName() + "," + myDate.getAge());
        }
    }
    static class B{
```

```
        public void get(){
            int data = x.get();
            MyThreadScopeData1 myDate =
MyThreadScopeData1.getInstance();
            System.out.println("B from " +
Thread.currentThread().getName() + " has put data: " +
data);
            System.out.println("B from " +
Thread.currentThread().getName() + " getMyData: " +
myDate.getName() + "," + myDate.getAge());
        }
    }
}
class MyThreadScopeData1{
    private MyThreadScopeData1(){}
    public static /*synchronized*/ MyThreadScopeData1
getInstance(){
        MyThreadScopeData1 instance = map.get();
        if(instance == null){
            instance = new MyThreadScopeData1();
            map.set(instance);
        }
        return instance;
    }
    //private static MyThreadScopeData1 instance = null;
    private static ThreadLocal<MyThreadScopeData1>
map = new ThreadLocal<MyThreadScopeData1>();
    private String name;
    private int age;
    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;
    }
}
```

（9）多个线程访问共享对象和数据的方式

问题：设计 4 个线程，其中两个线程每次对 j 增加 1，另外两个线程对 j 每次减少 1。写出程序。

A．如果每个线程执行的代码相同，可以使用同一个 Runnable 对象，这个 Runnable 对象中有那个共享数据；

B．如果每个线程执行的代码不同，这时候需要用不同的 Runnable 对象，有如下两种方式来实现这些 Runnable 对象之间的数据共享：

- 将共享数据封装在另外一个对象中，然后将这个对象逐一传递给各个 Runnable 对象。每个线程对共享数据的操作方法也分配到那个对象身上完成，这样容易实现针对该数据进行的各个操作的互斥与通信。
- 将这些 Runnable 对象作为某一个类中的内部类，共享数据作为这个外部类中的成员变量，每个线程对共享数据的操作方法也分配给外部类，以便实现对共享数据进行的各个操作的互斥与通信，作为内部类的各个 Runnable 对象调用外部类的这些方法。
- 上面两种方式的组合：将共享数据封装在另外一个对象中，每个线程对共享数据的操作方法也分配到那个对象身上去完成，对象作为这个外部类中的成员变量或方法中的局部变量，每个线程的 Runnable 对象作为外部类中的成员内部类或局部内部类。
- 总之，要同步互斥的几段代码最好是分别放在几个独立的方法中，这些方法再放到同一个类中，这样比较容易实现它们之间的同步互斥和通信。

```
public class MultiThreadShareDate {
    private int j;
    public static void main(String[] args) {
        MultiThreadShareDate multiThreadShareDate =
new MultiThreadShareDate();
        Inc inc = multiThreadShareDate.new Inc();
        Dec dec = multiThreadShareDate.new Dec();
        for (int i = 0; i < 2; i++) {
            Thread thread = new Thread(inc);
            thread.start();
            thread = new Thread(dec);
            thread.start();
        }
    }
    private synchronized void inc(){
        j++;

    System.out.println(Thread.currentThread().getName()
+ ".inc:" + j);
    }

    private synchronized void dec(){
        j--;

    System.out.println(Thread.currentThread().getName()
+ ".dec:" + j);
```

```
    }
    class Inc implements Runnable{
        public void run(){
            for (int i = 0; i < 100; i++) {
                inc();
            }
        }
    }
    class Dec implements Runnable{
        public void run(){
            for (int i = 0; i < 100; i++) {
                dec();
            }
        }
    }
}
```

19、JAVA GUI

（1）AWT（Abstract Window Toolkit）：包括 JAVA 中各种用于构建 GUI（Graphics User Interface）的类和接口（java.awt.*），如 Component（所有可以显示出来的图形元素都叫 Component），Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent 等等。

（2）Container 类：为 Component 的子类，其实例化的对象用于装载其它的 Component 对象。包括 Window 和 Panel 两种常用类。Window 对象表示顶级窗口，Panel 对象用于装载其它的 Component 对象，不能独立存在和显示。Window 对象可以作为一个应用程序的窗口独立显示出来，Panel 也可以容纳其他的 Component 对象，但其不能作为应用程序的窗口独立显示出来，必须被添加到其他 Containner 中才能显示出来。

（3）Window 类：两种常用类 Frame 和 Dialog。由 Frame 或其子类创建的对象为一个窗体，由 Dialog 或其子类创建的对象为一个对话框。

```
public class TestFrame {
    public static void main( String args[]) {
        JFrame f = new JFrame("My First Test");
        f.setLocation(300, 300);
        f.setSize( 170,100);
        f.getContentPane().setBackground( Color.blue);
        f.setResizable(true);
        f.setVisible( true);
    }
}
public class TestFrameWithPanel {
    public static void main(String args[]) {
        Frame f = new Frame("MyTest Frame");
        Panel pan = new Panel();
        f.setSize(200,200);
        f.setBackground(Color.blue);
        f.setLayout(null); // 取消默认布局管理器
        pan.setSize(100,100);
        pan.setBackground(Color.green);
        f.add(pan);
        f.setVisible(true);
    }
}
public class TenButtons {
    public static void main(String args[]) {
        Frame f = new Frame("Java Frame");
        f.setLayout(new GridLayout(2, 1));
        f.setLocation(300, 400);
        f.setSize(300, 200);
        f.setBackground(new Color(204, 204, 255));
        Panel p1 = new Panel(new BorderLayout());
        Panel p2 = new Panel(new BorderLayout());
        Panel p11 = new Panel(new GridLayout(2, 1));
        Panel p21 = new Panel(new GridLayout(2, 2));
        p1.add(new Button("BUTTON"),
BorderLayout.WEST);
        p1.add(new Button("BUTTON"),
BorderLayout.EAST);
        p11.add(new Button("BUTTON"));
        p11.add(new Button("BUTTON"));
        p1.add(p11, BorderLayout.CENTER);
        p2.add(new Button("BUTTON"),
BorderLayout.WEST);
        p2.add(new Button("BUTTON"),
BorderLayout.EAST);
        for (int i = 1; i <= 4; i++) {
            p21.add(new Button("BUTTON"));
        }
        p2.add(p21, BorderLayout.CENTER);
        f.add(p1);
        f.add(p2);
        f.setVisible(true);
    }
}
public class TestMultiFrame {
    public static void main(String args[]) {
        MyFrame222 f1 =
            new
MyFrame222(100,100,200,200,Color.BLUE);
```

```
        MyFrame222 f2 =
            new
MyFrame222(300,100,200,200,Color.YELLOW);
        MyFrame222 f3 =
            new
MyFrame222(100,300,200,200,Color.GREEN);
        MyFrame222 f4 =
            new
MyFrame222(300,300,200,200,Color.MAGENTA);
    }
}
class MyFrame222 extends Frame{
    static int id = 0;
    MyFrame222(int x,int y,int w,int h,Color color){
        super("MyFrame " + (++id));
        setBackground(color);
        setLayout(null);
        setBounds(x,y,w,h);
        setVisible(true);
    }
}
public class TestMultiPanel {
    public static void main(String args[]) {
        new
MyFrame2("MyFrameWithPanel",300,300,400,300);
    }
}
class MyFrame2 extends Frame{
    private Panel p1,p2,p3,p4;
    MyFrame2(String s,int x,int y,int w,int h){
        super(s);
        setLayout(null);
        p1 = new Panel(null); p2 = new Panel(null);
        p3 = new Panel(null); p4 = new Panel(null);
        p1.setBounds(0,0,w/2,h/2);
        p2.setBounds(0,h/2,w/2,h/2);
        p3.setBounds(w/2,0,w/2,h/2);
        p4.setBounds(w/2,h/2,w/2,h/2);
        p1.setBackground(Color.BLUE);
        p2.setBackground(Color.GREEN);
        p3.setBackground(Color.YELLOW);
        p4.setBackground(Color.PINK);
        add(p1);add(p2);add(p3);add(p4);
        setBounds(x,y,w,h);
        setVisible(true);
    }
}
```

（4）布局管理器：LayoutManager，用于管理 Component 在 Container 中的布局，不必自己通过 setBounds 等方法来设置位置和大小。每个 Container 都有一个布局管理器对象，当容器需要对某个组件进行定位或判断其大小尺寸时，就会自动调用其对应的布局管理器。常用的布局管理器包括：FlowLayout，BorderLayout，GridLayout，其中，Frame 默认使用 BorderLayout，Panel 默认使用 FlowLayout。

```
public class TestBorderLayout {
    public static void main(String args[]) {
        Frame f = new Frame("Border Layout");
        Button bn = new Button("BN");
        Button bs = new Button("BS");
        Button bw = new Button("BW");
        Button be = new Button("BE");
        Button bc = new Button("BC");
        f.add(bn, "North");
        f.add(bs, "South");
        f.add(bw, "West");
        f.add(be, "East");
        f.add(bc, "Center");
        // 也可使用下述语句
        /*
        f.add(bn, BorderLayout.NORTH);
        f.add(bs, BorderLayout.SOUTH);
        f.add(bw, BorderLayout.WEST);
        f.add(be, BorderLayout.EAST);
        f.add(bc, BorderLayout.CENTER);
        */
        f.setSize(200,200);
        f.setVisible(true);
    }
}
public class TestFlowLayout {
    public static void main(String args[]) {
        Frame f = new Frame("Flow Layout");
        Button button1 = new Button("Ok");
        Button button2 = new Button("Open");
        Button button3 = new Button("Close");
        f.setLayout(new
FlowLayout(FlowLayout.LEFT));
        f.add(button1);
        f.add(button2);
        f.add(button3);
        f.setSize(100,100);
        f.setVisible(true);
    }
```

```
}
```

（5）事件监听模式和监听器：事件源对象→事件对象→监听对象。

```
public class TestActionEvent {
    public static void main(String args[]) {
            Frame f = new Frame("Test");
            Button b = new Button("Press Me!");
            Monitor bh = new Monitor();
            b.addActionListener(bh);
            f.add(b,BorderLayout.CENTER);
            f.pack();
            f.setVisible(true);
    }
}

class Monitor implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("a button has been pressed");
    }
}
public class TestActionEvent2 {
    public static void main(String args[]) {
            Frame f = new Frame("Test");
            Button b1 = new Button("Start");
            Button b2 = new Button("Stop");
            Monitor2 bh = new Monitor2();
            b1.addActionListener(bh);
            b2.addActionListener(bh);
            b2.setActionCommand("game over");
            f.add(b1,"North");
            f.add(b2,"Center");
            f.pack();
            f.setVisible(true);
    }
}
class Monitor2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("a button has been pressed," +
        "the relative info is:\n " +
e.getActionCommand());
    }
}
public class TestKey {
    public static void main(String[] args) {
        new KeyFrame().launchFrame();
    }
}
class KeyFrame extends Frame {
    public void launchFrame() {
        setSize(200, 200);
        setLocation(300,300);
        addKeyListener(new MyKeyMonitor());
        setVisible(true);
    }
    class MyKeyMonitor extends KeyAdapter {
        public void keyPressed(KeyEvent e) {
            int keyCode = e.getKeyCode();
            if(keyCode == KeyEvent.VK_UP) {
                System.out.println("UP");
            }
        }
    }
}
public class TestWindowClose {
  public static void main(String args[]) {
    new MyFrame55("MyFrame");
  }
}
class MyFrame55 extends Frame {
  MyFrame55(String s) {
    super(s);
    setLayout(null);
    setBounds(300, 300, 400, 300);
    this.setBackground(new Color(204, 204, 255));
    setVisible(true);
    //this.addWindowListener(new
MyWindowMonitor());

    this.addWindowListener(
    new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        setVisible(false);
        System.exit(-1);
      }
    });
  }
  /*
  class MyWindowMonitor extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        setVisible(false);
        System.exit(0);
    }
  }
  */
```

```
}
```
（6）Graphics 类和 Paint 方法：每个 Component 有一个 paint(Graphics g)方法用于实现绘图，每次重画该 Component 时都会自动调用 paint 方法。

```
public class TestPaint {
    public static void main(String[] args) {
        new PaintFrame().launchFrame();
    }
}
class PaintFrame extends Frame {
    public void launchFrame() {
        setBounds(200,200,640,480);
        setVisible(true);
    }
    public void paint(Graphics g) {
        Color c = g.getColor();
        g.setColor(Color.red);
        g.fillOval(50, 50, 30, 30);
        g.setColor(Color.green);
        g.fillRect(80,80,40,40);
        g.setColor(c);
    }
}
```
（7）适配器：实现了某一个监听器接口的类，如鼠标、键盘、窗口事件适配器等等。

```
public class MyMouseAdapter{
  public static void main(String args[]) {
    new MyFrame("drawing...");
  }
}
class MyFrame extends Frame {
    private static final long serialVersionUID = 1L;
    ArrayList<Point> points = null;
    MyFrame(String s) {
        super(s);
        points = new ArrayList<Point>();
        setLayout(null);
        setBounds(300,300,400,300);
        this.setBackground(new Color(204,204,255));
        setVisible(true);
        this.addMouseListener(new Monitor11());
    }
    public void paint(Graphics g) {
        Iterator<Point> i = points.iterator();
        while(i.hasNext()){
          Point p = (Point)i.next();
          g.setColor(Color.BLUE);
          g.fillOval(p.x,p.y,10,10);
        }
    }
    public void addPoint(Point p){
        points.add(p);
    }
}
class Monitor11 extends MouseAdapter {
  public void mousePressed(MouseEvent e) {
    MyFrame f = (MyFrame)e.getSource();
    f.addPoint(new Point(e.getX(),e.getY()));
    f.repaint();
  }
}
```

20、JAVA 网络编程

（1）基本概念：计算机网络定义、计算机网络主要功能、IP地址、端口号、资源定位（URL(Uniform Resource Locator:协议、存放资源的主机域名、端口号和资源文件名)、URI(Uniform resource identifier)）、通信协议的分层、传输层通信协议（TCP、UDP）。

（2）相关类：InetAddress、InetSocketAddress、URL、TCP类（ServerSocket、Socket）、UDP类（DatagramSocket、DatagramPacket）。

（3）InetAddress：封装计算机的ip地址和域名，没有端口信息。InetSocketAddress：包含端口，用于socket通信。

```
public class InetDemo01 {
    public static void main(String[] args) throws UnknownHostException{
        //使用getLocalHost方法创建InetAddress对象
        InetAddress addr = InetAddress.getLocalHost();

        System.out.println(addr.getHostAddress());  //返回：192.168.1.100
        System.out.println(addr.getHostName());  //输出计算机名
        //根据域名得到InetAddress对象

        //addr = InetAddress.getByName("www.163.com");

        //System.out.println(addr.getHostAddress());  //返回 163服务器的ip:61.135.253.15
        //System.out.println(addr.getHostName());  //输出：www.163.com
        //根据ip得到InetAddress对象
```

```
        //addr =
InetAddress.getByName("61.135.253.15");

        //System.out.println(addr.getHostAddress());  //
返回 163服务器的ip:61.135.253.15
        //System.out.println(addr.getHostName());  //输
出ip而不是域名。如果这个IP地 址不存在或DNS服务器
不允许进行IP地址和域名的映射，getHostName方法就直
接返回这个IP地址。
    }
}
public class InetSockeDemo01 {
    public static void main(String[] args) throws
UnknownHostException {
        InetSocketAddress  address = new
InetSocketAddress("127.0.0.1",9999);
        address = new
InetSocketAddress(InetAddress.getByName("127.0.0.1"),9
999);
        System.out.println(address.getHostName());
        System.out.println(address.getPort());
        InetAddress addr =address.getAddress();
        System.out.println(addr.getHostAddress());  //
返回：地址
        System.out.println(addr.getHostName());  //输
出计算机名
    }
}
```

（4）URL：指向互联网“资源”的指针，资源可以是简单的文件或目录，也可以是更为复杂的对象的引用，例如对数据库或搜索引擎的查询。

```
public class URLDemo01 {
    public static void main(String[] args) throws
MalformedURLException {
        //绝对路径构建
        URL url = new
URL("http://www.baidu.com:80/index.html?uname=whut"
);
        System.out.println("协议:"+url.getProtocol());
        System.out.println("域名:"+url.getHost());
        System.out.println("端口:"+url.getPort());
        System.out.println("资源:"+url.getFile());
        System.out.println("相对路径:"+url.getPath());
        System.out.println("锚点:"+url.getRef()); //锚点
        System.out.println("参数:"+url.getQuery());//?参
数 :存在锚点  返回null ,不存在，返回正确
        url = new URL("http://www.baidu.com:80/a/");
        url = new URL(url,"b.txt"); //相对路径
        System.out.println(url.toString());
    }
}
public class URLDemo02 {
    public static void main(String[] args) throws
IOException {
        URL url = new URL("http://localhost:8080/"); //
主页 默认资源
        //获取资源 网络流
        InputStream is =url.openStream();
        byte[] flush = new byte[1024];
        int len =0;
        while(-1!=(len=is.read(flush))){
            System.out.println(new String(flush,0,len));
        }
        is.close();
        /*BufferedReader  br =
                new BufferedReader(new
InputStreamReader(url.openStream(),"utf-8"));
        BufferedWriter bw = new BufferedWriter(new
OutputStreamWriter(new
FileOutputStream("whut.html"),"utf-8"));
        String msg =null;
        while((msg=br.readLine())!=null){
            //System.out.println(msg);
            bw.append(msg);
            bw.newLine();
        }
        bw.flush();
        bw.close();
        br.close();*/
    }
}
```

（5）UDP：以数据为中心，无连接、不可靠、效率相对较高。DatagramSocket、DatagramPacket

客户端：

1、创建客户端（DatagramSocket类） + 端口

2、准备数据 double --> 字节数组 字节数组输出流

3、打包（DatagramPacket + 发送的地点 及端口）

4、发送

5、释放资源

服务器端：

1、创建服务端 （DatagramSocket类） + 端口

2、准备接受容器(字节数组) 封装为DatagramPacket

3、封装成包

4、接受数据
5、分析数据 字节数组 --> double
6、释放

```
public class MyServer {
    public static void main(String[] args) throws
IOException {
        //1、创建服务端 +端口
        DatagramSocket server = new
DatagramSocket(8888);
        //2、准备接受容器
        byte[] container = new byte[1024];
        //3、封装成 包 DatagramPacket(byte[] buf, int
length)
        DatagramPacket packet =new
DatagramPacket(container, container.length) ;
        //4、接受数据
        server.receive(packet);
        //5、分析数据
        byte[] data =packet.getData();
        int len =packet.getLength();
        System.out.println(new String(data,0,len));
        //6、释放
        server.close();
    }
}
public class MyClient {
    public static void main(String[] args) throws
IOException {
        //1、创建客户端 +端口
        DatagramSocket client = new
DatagramSocket(6666);
        //2、准备数据
        String msg ="udp编程";
        byte[] data =msg.getBytes();
        //3、打包（发送的地点 及端口）
DatagramPacket(byte[] buf, int length, InetAddress address,
int port)
        DatagramPacket packet = new
DatagramPacket(data,data.length,new
InetSocketAddress("localhost",8888));
        //4、发送
        client.send(packet);
        //5、释放
        client.close();
    }
}
```

```
public class Server {
    /**
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws
IOException {
        //1、创建服务端 +端口
        DatagramSocket server = new
DatagramSocket(8888);
        //2、准备接受容器
        byte[] container = new byte[1024];
        //3、封装成 包 DatagramPacket(byte[] buf, int
length)
        DatagramPacket packet =new
DatagramPacket(container, container.length) ;
        //4、接受数据
        server.receive(packet);
        //5、分析数据
        double data =convert(packet.getData());
        System.out.println(data);
        //6、释放
        server.close();

    }
    /**
     * 字节数组 +Data 输入流
     * @param data
     * @return
     * @throws IOException
     */
    public static double convert(byte[] data) throws
IOException{
        DataInputStream dis =new
DataInputStream(new ByteArrayInputStream(data));
        double num =dis.readDouble();
        dis.close();
        return num;
    }
}
public class Client {
    public static void main(String[] args) throws
IOException {
        //1、创建客户端 +端口
        DatagramSocket client = new
DatagramSocket(6666);
        //2、准备数据
        double num =89.12;
```

```
        byte[] data =convert(num);
        //3、打包（发送的地点 及端口）
DatagramPacket(byte[] buf, int length, InetAddress address, int port)
        DatagramPacket packet = new DatagramPacket(data,data.length,new InetSocketAddress("localhost",8888));
        //4、发送
        client.send(packet);
        //5、释放
        client.close();

    }
    /**
     * 字节数组 数据源  +Data 输出流
     * @param num
     * @return
     * @throws IOException
     */
    public static byte[] convert(double num) throws IOException{
        byte[] data =null;
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        DataOutputStream dos =new DataOutputStream(bos);
        dos.writeDouble(num);
        dos.flush();
        //获取数据
        data = bos.toByteArray();
        dos.close();
        return data;
    }
}
```

（6）TCP：面向连接、建立稳定连接的点对点的通信，实时、快速、可靠性高、占用系统资源多、效率低。Socket(代表连接、发送TCP消息)、ServerSocket(创建服务器+端)

服务器端：

1、创建服务器 指定端口 ServerSocket(int port)

2、接收客户端连接

3、发送数据 + 接收数据

客户端：

1、创建客户端 必须指定服务器+端口 此时就在连接 Socket(String host, int port)

2、接收数据 +发送数据

```
public class Server {
    public static void main(String[] args) throws IOException {
        //1、创建服务器 指定端口 ServerSocket(int port)
        ServerSocket server = new ServerSocket(8885);
        //2、接收客户端连接 阻塞式
        Socket socket =server.accept();
        System.out.println("一个客户端建立连接");
        //3、发送数据
        String msg ="欢迎使用";
        //输出流
        /*
        BufferedWriter bw = new BufferedWriter(
                new OutputStreamWriter(
                socket.getOutputStream()));

        bw.write(msg);
        bw.newLine();
        bw.flush();
        */
        DataOutputStream dos = new DataOutputStream(socket.getOutputStream());
        dos.writeUTF(msg);
        dos.flush();
    }
}
public class Client {
    public static void main(String[] args) throws UnknownHostException, IOException {
        //1、创建客户端 必须指定服务器+端口 此时就在连接
        Socket client = new Socket("localhost",8885);
        //2、接收数据
        /*
        BufferedReader br = new BufferedReader(new InputStreamReader(client.getInputStream()));
        String echo =br.readLine(); //阻塞式方法
        System.out.println(echo);
        */
        DataInputStream dis = new DataInputStream(client.getInputStream());
        String echo = dis.readUTF();
        System.out.println(echo);
    }
}
public class MultiServer {
    public static void main(String[] args) throws
```

```
IOException {
        //1、创建服务器  指定端口   ServerSocket(int port)
        ServerSocket server = new ServerSocket(8888);
        //2、接收客户端连接   阻塞式
        while(true){ //死循环  一个accept()一个客户端
            Socket socket =server.accept();
            System.out.println("一个客户端建立连接");
            //3、发送数据
            String msg ="欢迎使用";
            //输出流
            DataOutputStream dos = new DataOutputStream(socket.getOutputStream());
            dos.writeUTF(msg);
            dos.flush();
            while(true){
            }
        }
    }
}
```

21、JAVA IO

1、Java 流式输入/输出（所谓输入/输出，这都是站在程序内存的角度来说的）：在 Java 程序中，对于数据的输入/输出操作以流（stream）方式进行；Java JDK 提供了各种流的类，以获取不同种类的数据。

2、java.io 包中定义了多个流类型（类或抽象类）来实现输入/输出功能；可以从不同的角度对其进行分类：

（1）按数据流的方向不同可以分为输入流和输出流；

（2）按处理数据单位不同可以分为字节流和字符流（一个字符 2 个字节，UTF-16）；

（3）按照功能不同可以分为节点流和处理流。

例：**InputStream、OutputStream、Reader、Writer**（Java 所有流类型都位于 java.io 中，都分别继承上述前两种字节流抽象类和后两种字符流抽象类）

3、节点流和处理流：节点流为可以从一个特定的数据源（节点）读写数据（如：文件、内存）。处理流是“连接”在已存在的流（节点流或处理流）之上，通过对数据的处理为程序提供更为强大的读写功能。

4、InputStream：继承自 InputStream 的流都是用于向程序中输入数据，且数据的单位为字节（8bit），其包括的节点流有：FileInputStream、PipedInputStream、ByteArrayInputStream、StringBufferInputStream；包括的处理流有：FilterInputStream、SequenceInputStream、ObjectInputStream。查看 JDK API 中 InputStream 的 read、close 等方法。

```
import java.io.*;
public class TestFileInputStream {
   public static void main(String[] args) {
      int b = 0;
      FileInputStream in = null;
      try {
         in = new FileInputStream("d:\\workspace\\JAVATest\\src\\cn\\edu\\whut\\io\\TestFileInputStream.java");
      } catch (FileNotFoundException e) {
         System.out.println("找不到指定文件");
         System.exit(-1);
      }

      try {
         long num = 0;
         while((b=in.read())!=-1){
            System.out.print((char)b);
            num++;
         }
         in.close();
         System.out.println();
         System.out.println("共读取了 "+num+" 个字节");
      } catch (IOException e1) {
         System.out.println("文件读取错误");
         System.exit(-1);
      }
   }
}
```

5、OutputStream：继承自 OutputStream 的流是用于从程序中输出数据，且数据的单位为字节。其包括的节点流为 FileOutputStream、PipedOutputStream、ByteArrayOutputStream；包括的处理流有：FilterOutputStream、ObjectOutputStream。查看 JDK API 中的 write、close、flush 等方法。

```
import java.io.*;
public class TestFileOutputStream {
   public static void main(String[] args) {
      int b = 0;
      FileInputStream in = null;
      FileOutputStream out = null;
      try {
         in = new FileInputStream("d:\\workspace\\JAVATest\\src\\cn\\edu\\whut\\io\\TestFileOutputStream.java");
```

```
            out = new
FileOutputStream("d:\\workspace\\JAVATest\\src\\cn\\edu\
\whut\\io\\HW.java");
            while((b=in.read())!=-1){
               out.write(b);
            }
            in.close();
            out.close();
         } catch (FileNotFoundException e2) {
            System.out.println("找不到指定文件");
System.exit(-1);
         } catch (IOException e1) {
            System.out.println("文件复制错误");
System.exit(-1);
         }
         System.out.println("文件已复制");
   }
}
```

6、Reader：继承自 Reader 的流都是用于向程序中输入数据，且数据的单位为字符（16bit），其包括的节点流有：CharArrayReader、PipedReader、StringReader；包括的处理流有：BufferedReader、InputStreamReader、FilterReader。查看 JDK API 中的 read、close 等方法。

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

7、Writer：继承自 Writer 的流都是用于从程序中输出数据，且数据的单位为字符（16bit）。其包括的节点流有：CharArrayWriter、PipedWriter、StringWriter；包括的处理流有：BufferedWriter、OutputStreamWriter、FilterWriter。查看 JDK API 中的 write、close、flush 等方法。

```
import java.io.*;
public class TestFileWriter {
   public static void main(String[] args) {
      FileWriter fw = null;
      try {
         fw = new
FileWriter("d:\\workspace\\JAVATest\\src\\cn\\edu\\whut\\
io\\unicode.dat");
         for(int c=0;c<=50000;c++){
            fw.write(c);
         }
         fw.close();
      } catch (IOException e1) {
            e1.printStackTrace();
         System.out.println("文件写入错误");
         System.exit(-1);
      }
   }
}
import java.io.*;
public class TestFileWriter2 {
      public static void main(String[] args) throws
Exception {
            FileReader fr = new
FileReader("d:/java/io/TestFileWriter2.java");
            FileWriter fw = new
FileWriter("d:/java/io/TestFileWriter2.bak");
            int b;
            while((b = fr.read()) != -1) {
                  fw.write(b);
            }
            fr.close();
            fw.close();
      }
}
```

8、缓冲流：缓冲流要“套接”在相应的节点流之上，对读写的数据提供缓冲的功能，提高了读写的效率，减少了读硬盘的次数。

缓冲输入流支持其父类的 mark 和 reset 方法；BufferedReader 提供了 readLine 方法用于读取一行字符串；BufferedWriter 提供了 newLine 方法用于写入一个行分隔符；对于输出的缓冲流，写出的数据会先在内存中缓存，使用 flush 方法会使内存中的数据立刻写出。

```
import java.io.*;
public class TestBufferStream1 {
    public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream(

    "d:\\workspace\\JAVATest\\src\\cn\\edu\\whut\\io\\TestFileInputStream.java");
            BufferedInputStream bis = new BufferedInputStream(fis);
            int c = 0;
            System.out.println(bis.read());
            System.out.println(bis.read());
            bis.mark(100);
            for (int i = 0; i <= 10 && (c = bis.read()) != -1; i++) {
                System.out.print((char) c + " ");
            }
            System.out.println();
            bis.reset();
            for (int i = 0; i <= 10 && (c = bis.read()) != -1; i++) {
                System.out.print((char) c + " ");
            }
            bis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
import java.io.*;
public class TestBufferStream2 {
    public static void main(String[] args) {
        try {
            BufferedWriter bw = new BufferedWriter(
                    new FileWriter(

    "d:\\workspace\\JAVATest\\src\\cn\\edu\\whut\\io\\dat2.txt"));
            BufferedReader br = new BufferedReader(new FileReader(

    "d:\\workspace\\JAVATest\\src\\cn\\edu\\whut\\io\\TestFileInputStream.java"));
            String s = null;
            for (int i = 1; i <= 100; i++) {
                s = String.valueOf(Math.random());
                bw.write(s);
                bw.newLine();
            }
            bw.flush();
            while ((s = br.readLine()) != null) {
                System.out.println(s);
            }
            bw.close();
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

9、转换流：InputStreamReader 和 OutputStreamWriter 用于字节数据到字符数据之间的转换。InputStreamReader 需要和 InputStream 套接，OutputStreamWriter 需要和 OutputStream 套接。转换流在构造的时候可以指定其编码集。

```
import java.io.*;
public class TestTransForm1 {
  public static void main(String[] args) {
    try {
      OutputStreamWriter osw = new OutputStreamWriter(
              new FileOutputStream("d:\\workspace\\JAVATest\\src\\cn\\edu\\whut\\io\\char.txt"));
      osw.write("mircosoftibmsunapplehp");
      System.out.println(osw.getEncoding());
      osw.close();
      osw = new OutputStreamWriter(
                                      new FileOutputStream("d:\\workspace\\JAVATest\\src\\cn\\edu\\whut\\io\\char.txt", true),
                                      "ISO8859_1");
// latin-1
      osw.write("mircosoftibmsunapplehp");
      System.out.println(osw.getEncoding());
      osw.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
import java.io.*;
public class TestTransForm2 {
```

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

10、数据流：DataInputStream 和 DataOutputStream 分别继承自 InputStream 和 OutputStream，它属于处理流，需要分别套接在 InputStream 和 OutputStream 类型的节点流上。其提供了可以存取与机器无关的 Java 原始类型数据的方法。

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

      System.out.println(dis.readDouble());

      System.out.println(dis.readBoolean());
      dos.close();   dis.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

11、Print 流：PrintWriter 和 PrintStream 都属于输出流，分别针对字符和字节。PrintWriter 和 PrintStream 提供了重载的 Print 和 Println 方法用于多种数据类型的输出。PrintWriter 和 PrintStream 的输出操作不会抛出异常，用户通过检测错误状态获取错误信息。PrintWriter 和 PrintStream 有自动 flush 功能。

```
import java.io.*;
public class TestPrintStream1 {
  public static void main(String[] args) {
    PrintStream ps = null;
    try {
      FileOutputStream fos =
                  new
FileOutputStream("d:\\workspace\\JAVATest\\src\\cn\\edu\
\whut\\io\\log.dat");
      ps = new PrintStream(fos);
    } catch (IOException e) {
      e.printStackTrace();
    }
    if(ps != null){
      System.setOut(ps);
    }
    int ln = 0;
    for(char c = 0; c <= 60000; c++){
      System.out.print(c+ " ");
      if(ln++ >=100){ System.out.println(); ln = 0;}
    }
  }
}
import java.io.*;
public class TestPrintStream2 {
  public static void main(String[] args) {
    String filename = args[0];
    if(filename!=null){list(filename,System.out);}
  }
  public static void list(String f,PrintStream fs){
    try {
      BufferedReader br =
                  new BufferedReader(new
FileReader(f));
      String s = null;
      while((s=br.readLine())!=null){
        fs.println(s);
      }
```

```
            br.close();
        } catch (IOException e) {
            fs.println("无法读取文件");
        }
    }
}
import java.util.*;
import java.io.*;
public class TestPrintStream3 {
    public static void main(String[] args) {
        String s = null;
        BufferedReader br = new BufferedReader(
                                    new
InputStreamReader(System.in));
        try {
            FileWriter fw = new FileWriter

("d:\\workspace\\JAVATest\\src\\cn\\edu\\whut\\io\\logfile.
log", true); //Log4J
            PrintWriter log = new PrintWriter(fw);
            while ((s = br.readLine())!=null) {
                if(s.equalsIgnoreCase("exit")) break;
                System.out.println(s.toUpperCase());
                log.println("-----");
                log.println(s.toUpperCase());
                log.flush();
            }
            log.println("==="+new Date()+"===");
            log.flush();
            log.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

12、Object 流：直接将 Object 写入或读出。serializeable 接口、externalizable 接口、transient 关键字。

```
import java.io.*;
public class TestObjectIO {
    public static void main(String args[]) throws
Exception {
            T t = new T();
            t.k = 8;
            FileOutputStream fos = new
FileOutputStream("d:\\workspace\\JAVATest\\src\\cn\\edu\
\whut\\io\\testobjectio.dat");
            ObjectOutputStream oos = new
ObjectOutputStream(fos);
            oos.writeObject(t);
            oos.flush();
            oos.close();
            FileInputStream fis = new
FileInputStream("d:\\workspace\\JAVATest\\src\\cn\\edu\\
whut\\io\\testobjectio.dat");
            ObjectInputStream ois = new
ObjectInputStream(fis);
            T tReaded = (T)ois.readObject();
            System.out.println(tReaded.i + " " + tReaded.j +
" " + tReaded.d + " " + tReaded.k);
        }
}
class T implements Serializable
{
    int i = 10;
    int j = 9;
    double d = 2.3;
    transient int k = 15;
}
```

22、JAVA　JDBC

1、JDBC（Java DataBase Connectivity），java 数据库连接，JAVA 提供的一套操作数据库、执行 SQL 语句的 API 标准，可以为多种关系数据库提供统一、标准的访问。具体实现由不同的数据库厂商来实现，为我们屏避了具体的实现细节。

2、SQL 语言：

Select: Select * from T where…;

Insert：Insert into T values(…);

Create：Create table T (…);

Delete：Delete from T where…;

Update：Update T set t1 = … and t2 = … where t3 = …;

Drop：Drop table T。

3、JDBC 编程步骤：

（1）Load the Driver

Class.forName() | Class.forName().newInstance() | new DriverName()

实例化时自动向 DriverManager 注册，不需显式调用 DriverManager.registerDriver 方法

（2）Connect to the DataBase

DriverManager.getConnection()

（3）Execute the SQL

Connection.CreateStatement()

Statement.executeQuery()

Statement.executeUpdate()

（4）Retrieve the result data
循环取得结果 while(rs.next())
（5）Show the result data
将数据库中的各种类型转换为 Java 中的类型（getXXX 方法）
（6）Close
close the resultset/close the statement/close the connection

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {

    public static void main(String[] args) {

        Connection conn = null;
        Statement stmtStatement = null;
        ResultSet rSet = null;
        try {
            //new oracle.jdbc.driver.OracleDriver();

    Class.forName("oracle.jdbc.driver.OracleDriver");
            conn = DriverManager.getConnection(

    "jdbc:oracle:thin:@127.0.0.1:1521:orcl", "oa", "oa");



            stmtStatement = conn.createStatement();
            rSet = stmtStatement.executeQuery("select
* from T_S_TYPE");
            while (rSet.next()) {

    System.out.println(rSet.getString("TYPECODE"));

    System.out.println(rSet.getString("TYPENAME"));
                ;
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if(rSet != null) rSet.close();
                if(stmtStatement != null)
stmtStatement.close();
                if(conn != null) conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestDML {

    public static void main(String[] args) {

        Connection conn = null;
        Statement stmtStatement = null;
        try {

    Class.forName("oracle.jdbc.driver.OracleDriver");
            conn = DriverManager.getConnection(

    "jdbc:oracle:thin:@127.0.0.1:1521:orcl", "oa", "oa");
            stmtStatement = conn.createStatement();
            String sqlString    = "insert into T_S_ROLE
values ('xxx', 'uuu', 'yyy')";
            stmtStatement.executeUpdate(sqlString);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if(stmtStatement != null)
stmtStatement.close();
                if(conn != null) conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
```

```
}

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestDML2 {

	public static void main(String[] args) {

		if(args.length != 3){
			System.out.println("Parameter Error !
Please Input Again!!!");
			System.exit(-1);
		}

		Connection conn = null;
		Statement stmtStatement = null;
		try {

	Class.forName("oracle.jdbc.driver.OracleDriver");
			conn = DriverManager.getConnection(

	"jdbc:oracle:thin:@127.0.0.1:1521:orcl", "oa", "oa");
			stmtStatement = conn.createStatement();
			String sqlString    = "insert into T_S_ROLE
values ('"+ args[0] +"', '"+ args[1] +"', '"+ args[2] +"')";
			stmtStatement.executeUpdate(sqlString);
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		} catch (SQLException e) {
			e.printStackTrace();
		} finally {
			try {
				if(stmtStatement != null)
stmtStatement.close();
				if(conn != null) conn.close();
			} catch (SQLException e) {
				e.printStackTrace();
			}
		}
	}
}

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.PreparedStatement;

public class TestPrepStmt {

	public static void main(String[] args) {

		if(args.length != 3){
			System.out.println("Parameter Error !
Please Input Again!!!");
			System.exit(-1);
		}

		Connection conn = null;
		PreparedStatement preparedStatement = null;
		try {

	Class.forName("oracle.jdbc.driver.OracleDriver");
			conn = DriverManager.getConnection(

	"jdbc:oracle:thin:@127.0.0.1:1521:orcl", "oa", "oa");
			preparedStatement =
conn.prepareStatement("insert into T_S_ROLE values
(?, ?, ?)");
			preparedStatement.setString(1, args[0]);
			preparedStatement.setString(2, args[1]);
			preparedStatement.setString(3, args[2]);
			preparedStatement.executeUpdate();
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		} catch (SQLException e) {
			e.printStackTrace();
		} finally {
			try {
				if(preparedStatement != null)
preparedStatement.close();
				if(conn != null) conn.close();
			} catch (SQLException e) {
				e.printStackTrace();
			}
		}
	}
}

import java.sql.Connection;
import java.sql.DriverManager;
```

```
import java.sql.SQLException;
import java.sql.Statement;

public class TestTransaction {

    public static void main(String[] args) {

        Connection connection = null;
        Statement statement = null;

        try {

    Class.forName("oracle.jdbc.driver.OracleDriver");
            connection =
DriverManager.getConnection(

    "jdbc:oracle:thin:@127.0.0.1:1521:orcl", "oa", "oa");
            connection.setAutoCommit(false);
            statement = connection.createStatement();
            statement.addBatch("insert into T_S_ROLE
values ('1', '2', '3')");
            statement.addBatch("insert into T_S_ROLE
values ('4', '5', '6')");
            statement.addBatch("insert into T_S_ROLE
values ('7', '8', '9')");
            statement.executeBatch();
            connection.commit();
            connection.setAutoCommit(true);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
            try {
                if(connection != null){
                    connection.rollback();
                    connection.setAutoCommit(true);
                }
            } catch(SQLException eq) {
                eq.printStackTrace();
            }
        } finally {
            try {
                if(statement != null) statement.close();
                if(connection != null)
connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

23、设计模式

创建型模式：

- 单例模式、工厂模式、抽象工厂模式、建造者模式、原型模式

结构型模式：

- 适配器模式、桥接模式、装饰模式、组合模式、外观模式、享元模式、代理模式

行为型模式：

- 模版方法模式、命令模式、迭代器模式、观察者模式、中介者模式、备忘录模式、解释器模式、状态模式、策略模式、职责链模式、访问者模式

1）单例模式

核心作用：保证一个类只有一个实例，并且提供一个访问该实例的全局访问点。

常见应用场景：

- Windows 的 TaskManager 任务管理器是一个很典型的单例模式；
- 数据库连接池的设计一般也采用单例模式，因为数据库连接是一种数据库资源；
- 在 Spring 中，每个 Bean 默认是一个单例；
- 在 Servlet 编程中，每个 Servlet 是一个单例；
- 在 Spring MVC 框架/Struts 框架中，控制器对象是单例。

单例模式的优点：

- 由于单例模式只生成一个实例，减少了系统性能开销，当一个对象的产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后永久驻留内存的方式来解决。
- 单例模式可以在系统设置全局的访问点，优化共享资源访问，例如可以设计一个单例类，负责所有数据表的映射处理。

常见的五种单例模式实现方式：

- 饿汉式（线程安全，调用效率高。但不能延时加载）；
- 懒汉式（线程安全，调用效率不高。但可以延时加载）；
- 双重检测锁式；
- 静态内部类式（线程安全，调用效率高。可以延时加载）；
- 枚举单例（线程安全，调用效率高，不能延时加载）。