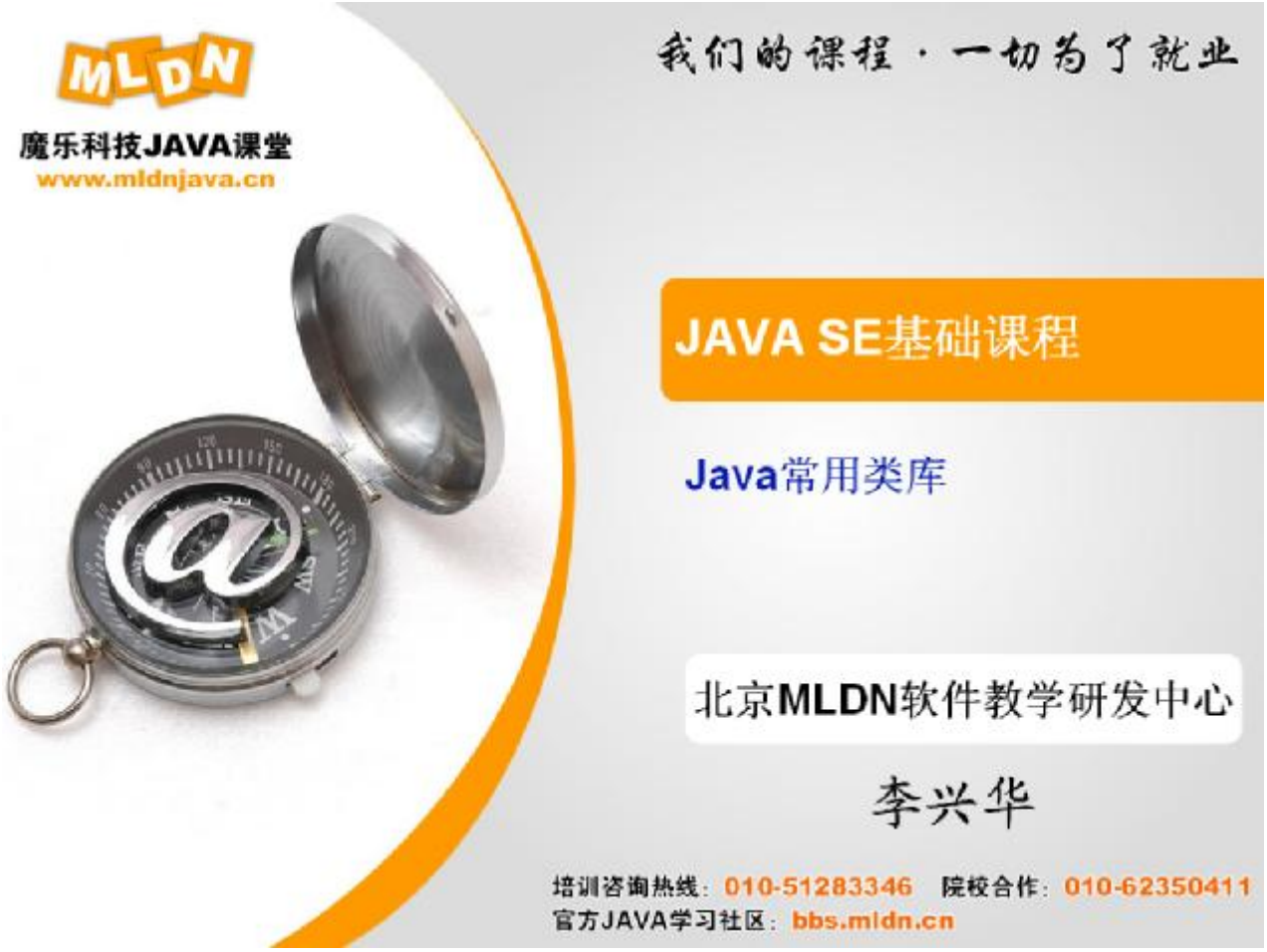
1、课程名称：Java 常用类库



2、知识点

2.1、上次课程的主要知识点

1、 Eclipse 的 JDT、Junit、CVS

2、 三种 Annotation：@Override、@Deprecated、@SuppresssWarnings。

2.2、本次预计讲解的知识点

1、 掌握 StringBuffer 的使用

2、 掌握正则表达式、大数操作、日期格式化、Random 类的使用

3、 理解 System、Runtime、Process、Math、Calendar 类的使用

4、 理解反射机制在程序中的应用

3、具体内容

类库（API，应用程序接口），主要是由系统为了完成某些特定的操作而使用的系统提供的开发包，这些开发包的中的各个类依然符合与之前讲解的面向对象中的各个基本概念，但是对于类库的学习一定要学会查询 DOC 文档。

3.1、StringBuffer（重点）

在程序中可以使用 String 表示一个字符串的操作，但是 String 本身有如下特点：

- 两种声明的方式，而且比较的时候靠 equals()比较内容
- **一个字符串的内容声明之后则不可改变**

实际上 String 最要命的一个问题就是内容无法改变，但是这种功能在实际的开发中又不可能避免掉，此时就可以依靠 StringBuffer 类来完成这种功能。**当字符串的内容需要被改变的时候就使用 StringBuffer**。

3.1.1、StringBuffer 简介

StringBuffer 是 java.lang 提供的一个开发包，类的定义如下：

```
public final class StringBuffer
extends Object
implements Serializable, CharSequence
```

此类实现了一个 CharSequence 接口，这个接口中定义了 charAt()、length()等方法，而且 String 也是这个接口的子类。

在 String 中如果要想完成两个字符串的连接，依靠“+”，但是在 StringBuffer 类中要想完成连接则需要使用的是 append()方法，此方法被重载过很多次，而且每个方法返回类型都是 StringBuffer：public StringBuffer append(char c)。

根据这种返回类型的特性，所以对于 append()方法，可以采用代码链的形式，例如：StringBuffer 对象.append().append()。

范例：进行字符串的连接

```
package org.lxh.api.stringbufferdemo;
public class StringBufferDemo01 {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer();
        buf.append("hello"); // 增加内容
        buf.append("world").append("!!!");
        System.out.println(buf);
    }
}
```

但是一定要注意的是，StringBuffer 虽然与 String 都属于 CharSequence 的子类，但是这两个类的对象是不能够直接转换的，如果现在需要将一个 StringBuffer 的类型变成 String 的话，则必须依靠 toString()方法完成。

```
package org.lxh.api.stringbufferdemo;
public class StringBufferDemo02 {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer();
        buf.append("hello"); // 增加内容
        buf.append(" world").append("!!!");
```

```
        String str = buf.toString() ;  // StringBuffer --> String
        System.out.println(str);
    }
}
```

由于 StringBuffer 与 String 相比内容是允许改变的，所以以后在进行引用传递的时候，肯定可以将方法中对 StringBuffer 操作的结果返回。

```
package org.lxh.api.stringbufferdemo;
public class StringBufferDemo03 {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer();
        buf.append("hello"); // 增加内容
        buf.append(" world").append("!!!");
        fun(buf);
        System.out.println(buf);
    }
    public static void fun(StringBuffer temp) {
        temp.append("\n").append(1).append(" + ").append(2).append(" = ")
                .append(1 + 2);
    }
}
```

从程序中的最终结果来看，StringBuffer 的内容是完全允许改变的。

3.1.2、StringBuffer 的应用

根据 StringBuffer 的特点实际上就可以发现，对于需要经常改变内容的字符串肯定要使用 StringBuffer，而相反，如果某些内容不需要随时改变的话，那么使用 String 就够了。

范例：以下情况就必须使用 StringBuffer

```
package org.lxh.api.stringbufferdemo;
public class StringBufferDemo04 {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer();
        for (int x = 0; x < 1000; x++) {
            buf.append(x);
        }
        System.out.println(buf);
    }
}
```

一般在此种情况下往往都会使用 StringBuffer。

3.1.3、StringBuffer 的方法

StringBuffer 本身也属于一个类，所以里面也提供了很多的操作方法，但是这些方法大部分都与 String 类似。

但是 StringBuffer 也有许多自己的定义的方法，下面依次来看这些方法。

1、 字符串反转

- 方法：public **StringBuffer** reverse()

```
package org.lxh.api.stringbufferdemo;
public class ReverseDemo {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer();
        buf.append("hello world!!!");
        System.out.println(buf.reverse());
    }
}
```

2、 字符串替换

- 在 String 类中依靠 replaceAll()方法完成替换
- 方法：public StringBuffer replace(int start,int end,String str)
  - |- 需要指定替换的开始和结束位置

```
package org.lxh.api.stringbufferdemo;
public class ReplaceDemo {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer();
        buf.append("hello world!!!");
        System.out.println(buf.replace(0, buf.length(), "XXX"));
    }
}
```

3、 插入内容

在使用 append()方法的时候实际上所有的内容都是采用顺序的方式依次连接到后面的位置，也可以使用 insert()方法完成在指定位置上的增加：public StringBuffer insert(int offset,char c)，此方法被重载过很多次，可以插入各种数据类型。

```
package org.lxh.api.stringbufferdemo;
public class InsertDemo {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer();
        buf.append("world!!!").insert(0, "hello");
        System.out.println(buf);
    }
}
```

3.2、Runtime（理解）

Runtime 表示的是运行时的状态对象，即：每当有一个 JVM 进程产生的时候都会自动生成一个 Runtime 类的对象。但是在此类的定义中发现并没有构造方法的声明，很明显构造方法被私有化了，那么一旦构造方法被私有化，则内部一定会存在一个方法，可以取得本类的实例化对象：public static Runtime getRuntime()。

以后查询 DOC 文档的时候只要没有发现构造方法，则构造方法一定被私有化了，属于单例设计，则此时一定可以从类的内部找到一个 static 方法，取得本类的实例。

```
Runtime run = Runtime.getRuntime() ;
```

Runtime 类可以取得一些系统的信息或者是建立一个新的进程。

3.2.1、取得系统信息

在 Runtime 类中定义了如下的三个方法，这些方法可以取得系统的内存使用情况：

- 总共可以使用的内存：public long totalMemory()
- 取得最大可用的内存：public long maxMemory()
- 取得当前空余的内存：public long freeMemory()

范例：观察内存的使用情况

```
package org.lxh.api.runtimedemo;
public class RuntimeDemo01 {
    public static void main(String[] args) {
        Runtime run = Runtime.getRuntime();
        System.out.println("** 1、maxMemory：" + run.maxMemory());
        System.out.println("** 1、totalMemory：" + run.totalMemory());
        System.out.println("** 1、freeMemory：" + run.freeMemory());
        String str = "";
        for (int x = 0; x < 10000; x++) { // 将产生大量的垃圾
            str += x;
        }
        System.out.println("========= 产生垃圾之后 ============");
        System.out.println("** 2、maxMemory：" + run.maxMemory());
        System.out.println("** 2、totalMemory：" + run.totalMemory());
        System.out.println("** 2、freeMemory：" + run.freeMemory());
    }
}
```

由于本程序中产生大量的垃圾，所以当操作完成之后，freeMemory()的空间明显减少。

在 Java 中对于垃圾收集实际上存在两种形式：

- 手工回收：public void gc()
- 自动回收：由系统完成

范例：再次观察垃圾收集之后的内存情况

```
package org.lxh.api.runtimedemo;
public class RuntimeDemo01 {
    public static void main(String[] args) {
        Runtime run = Runtime.getRuntime();
        System.out.println("** 1、maxMemory：" + run.maxMemory());
        System.out.println("** 1、totalMemory：" + run.totalMemory());
        System.out.println("** 1、freeMemory：" + run.freeMemory());
        String str = "";
        for (int x = 0; x < 10000; x++) { // 将产生大量的垃圾
            str += x;
        }
        System.out.println("========= 产生垃圾之后 ============");
        System.out.println("** 2、maxMemory：" + run.maxMemory());
        System.out.println("** 2、totalMemory：" + run.totalMemory());
```

```
        System.out.println("** 2、freeMemory：" + run.freeMemory());
        run.gc() ;  // 垃圾收集
        System.out.println("========= 垃圾收集之后 ============");
        System.out.println("** 2、maxMemory：" + run.maxMemory());
        System.out.println("** 2、totalMemory：" + run.totalMemory());
        System.out.println("** 2、freeMemory：" + run.freeMemory());
    }
}
```

3.2.2、运行本机程序（了解）

Runtime 类最大的好处在于，可以直接运行本机的可执行程序，例如：运行：notepad.exe。

使用的方法：public Process exec(String command) throws IOException

此方法返回一个 Process 类的对象实例，那么此类可以用于进程的控制。

- 销毁进程：public void destroy()

范例：观察程序的运行

```
package org.lxh.api.runtimedemo;
public class RuntimeDemo02 {
    public static void main(String[] args) throws Exception {
        Runtime run = Runtime.getRuntime();
        Process pro = run.exec("notepad.exe"); // 运行程序
        Thread.sleep(2000) ;
        pro.destroy() ;
    }
}
```

3.3、System 类（理解）

System.out.println()代码一直在使用着，根据之前学习到的知识，可以发现，System 肯定是一个类，而 out 是一个静态属性，而 println()是 out 对象所提供的一个方法。

3.3.1、取得计算的时间

System 类本身可以取得一个系统的当前时间，只是返回的时候按照 long 返回。

如果要想求出一个操作所花费的时间，采用的公式：结束时间- 开始时间。

取得当前时间的方法：public static long currentTimeMillis()

```
package org.lxh.api.systemdemo;
public class SystemDemo01 {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        String str = "";
        for (int x = 0; x < 10000; x++) { // 将产生大量的垃圾
```

```
            str += x;
        }
        long end = System.currentTimeMillis();
        System.out.println("花费的时间：" + (end - start));
    }
}
```

3.3.2、垃圾回收与对象生命周期（重点）

在 System 类中定义了如下的方法：public static void gc()。

当调用此方法的时候实际上就等同于调用了 Runtime 类中提供的 gc()方法，两个是一样的。

当一个对象被回收之前实际上也是可以进行一些收尾工作，这些工作是依靠覆写 Object 类中的以下方法：

- 方法：protected void finalize() throws Throwable

当一个对象被回收之前都会默认调用此方法，但是此方法上抛出的异常是 Throwable。Throwable 表示 Error 和 Exception，证明此方法有可能产生错误或者是异常，但是此方法的好处即便是产生了问题也不会影响程序的执行。

范例：观察对象的回收

```
package org.lxh.api.systemdemo;
class Person {
    public Person() {
        System.out.println("我出生了。。。");
    }
    @Override
    protected void finalize() throws Throwable {
        System.out.println("我完了，我被回收了。。。");
    }
}
public class ObjectGCDemo {
    public static void main(String[] args) {
        Person per = new Person();
        per = null;
        System.gc(); // 手工回收
    }
}
```

一般一个对象都要经历以下的生命周期：

- 加载 à 初始化 à 使用 à 回收 à 卸载

3.4、Math 类（理解）

Math 类本身表示的是数学的操作类，里面提供了各种各样的数学操作方法，例如：sin()等等。

方法：public static long round(double a)，四舍五入

```
package org.lxh.api.mathdemo;
public class MathDemo {
    public static void main(String[] args) {
```

```
        System.out.println(Math.round(90.356));
        System.out.println(Math.round(90.556));
    }
}
```

这个四舍五入是将小数点之后的内容全部进行了省略，那么如果要想进行准确的四舍五入的话，最早的做法是通过 BigDecimal 类完成的，这个类也称为大数操作类。

3.5、大数操作类（重点）

现在如果假设有两个数字非常的大，那么如果进行相乘的话，则肯定数字的会更大，此时，传统的数据类型已经无法装下了。

```
package org.lxh.api.bignumberdemo;
public class BigNumberDemo {
    public static void main(String[] args) {
        double d1 = Double.MAX_VALUE;
        double d2 = Double.MAX_VALUE;
        System.out.println(d1 * d2);
    }
}
```

这个时候数字已经太大了，所以根本就无法计算了，但是从现实世界中这种数字不可能避免。所以对于大数字的操作最早时候是通过字符串完成。所以在 Java 中为了解决这种大数的操作，提供了两个类：BigInteger、BigDecimal。

3.5.1、大整数操作类：BigInteger

BigInteger 本身是一个类，里面提供了一些基本的计算方法，方法如下：

- 构造：public BigInteger(String val)
- 四则运算：
  - |- 加法：public BigInteger add(BigInteger val)
  - |- 减法：public BigInteger subtract(BigInteger val)
  - |- 乘法：public BigInteger multiply(BigInteger val)
  - |- 除法：public BigInteger divide(BigInteger val)、public BigInteger[] divideAndRemainder(BigInteger val)

范例：观察大数字的操作

```
package org.lxh.api.bignumberdemo;
import java.math.BigInteger;
public class BigIntegerDemo {
    public static void main(String[] args) {
        BigInteger bi1 = new BigInteger(Long.MAX_VALUE + "");
        BigInteger bi2 = new BigInteger(Long.MAX_VALUE + "");
        System.out.println("加法：" + bi1.add(bi2));
        System.out.println("减法：" + bi1.subtract(bi2));
        System.out.println("乘法：" + bi1.multiply(bi2));
        System.out.println("除法：" + bi1.multiply(new BigInteger("333")));
        BigInteger res[] = bi1.divideAndRemainder(new BigInteger("333")) ;
```

```
        System.out.println("整数部分：" + res[0]) ;
        System.out.println("小数部分：" + res[1]) ;
    }
}
```

3.5.2、大小数操作类：BigDecimal

对于 BigDecimal 本身和 BigInteger 是非常类似的，包括各个计算方法也都是非常类似的。

范例：观察四则运算

```
package org.lxh.api.bignumberdemo;
import java.math.BigDecimal;
public class BigDecimalDemo {
    public static void main(String[] args) {
        BigDecimal bd1 = new BigDecimal(Double.MAX_VALUE);
        BigDecimal bd2 = new BigDecimal(Double.MAX_VALUE);
        System.out.println("加法：" + bd1.add(bd2));
        System.out.println("减法：" + bd1.subtract(bd2));
        System.out.println("乘法：" + bd1.multiply(bd2));
        System.out.println("除法：" + bd1.divide(bd2));
    }
}
```

但是，在 BigDecimal 类中存在着一种除法操作，这种除法操作可以指定小数的保留位数：

- 方法：public BigDecimal divide(BigDecimal divisor,int scale,int roundingMode)

范例：观察四舍五入操作

```
package org.lxh.api.bignumberdemo;
import java.math.BigDecimal;
class MyMathRound {
    public static double round(double num, int scale) {
        BigDecimal bd = new BigDecimal(num);
        return bd.divide(new BigDecimal(1), scale, BigDecimal.ROUND_HALF_UP)
                .doubleValue();
    }
    public static BigDecimal round(String num, int scale) {
        BigDecimal bd = new BigDecimal(num);
        return bd.divide(new BigDecimal(1), scale, BigDecimal.ROUND_HALF_UP);
    }
}
public class RoundDemo {
    public static void main(String[] args) {
        System.out.println("四舍五入：" + MyMathRound.round(98.56137, 2));
        System.out.println("四舍五入：" + MyMathRound.round(98.56637, 2));
        System.out.println("四舍五入：" + MyMathRound.round("98.56637", 2));
    }
```

```
}
```

如果编写的是一些财务程序的话，则肯定会有四舍五入的要求。

3.6、随机数（理解）

在 java.util.Random 类中主要的功能是用于产生随机数的。

范例：产生 10 个数字，都不大于 100

```
package org.lxh.api.randomdemo;
import java.util.Random;
public class RandomDemo {
    public static void main(String[] args) {
        Random rand = new Random();
        for (int x = 0; x < 10; x++) {
            System.out.println(rand.nextInt(100));
        }
    }
}
```

3.7、日期操作（重点）

日期肯定是一种比较常见的类型，在 Java 的日期操作中也存在着许多的类。

3.7.1、Date 类

java.util.Date 类是一个专门取得日期的操作类，本身的使用非常的简单，直接实例化对象输出即可。

```
package org.lxh.api.bignumberdemo;
import java.util.Date;
public class DateDemo {
    public static void main(String[] args) {
        System.out.println(new Date());
    }
}
```

此时的输出：

```
Thu Dec 24 10:59:44 CST 2009
```

确实是取得了一个日期，但是这个日期的格式并不符合于中国的习惯。

3.7.2、Calendar 类（理解）

Calendar 类是采用手工的方式取得日期，可以通过此类精确到毫秒，此类的定义如下：

```
public abstract class Calendar extends Object
implements Serializable, Cloneable, Comparable<Calendar>
```

这个类本身是一个抽象类，抽象类要想实例化肯定使用子类：GregorianCalendar

```
package org.lxh.api.datedemo;
import java.util.Calendar;
import java.util.GregorianCalendar;
public class CalendarDemo {
    public static void main(String[] args) {
        Calendar calendar = new GregorianCalendar();
        System.out.println("YEAR: " + calendar.get(Calendar.YEAR));
        System.out.println("MONTH: " + (calendar.get(Calendar.MONTH) + 1));
        System.out.println("DATE: " + calendar.get(Calendar.DATE));
        System.out
                .println("HOUR_OF_DAY: " + calendar.get(Calendar.HOUR_OF_DAY));
        System.out.println("MINUTE: " + calendar.get(Calendar.MINUTE));
        System.out.println("SECOND: " + calendar.get(Calendar.SECOND));
        System.out
                .println("MILLISECOND: " + calendar.get(Calendar.MILLISECOND));
    }
}
```

但是现在有一个问题，如果每次取时间都按照这种方式取的话，那么得累死，那么能不能设计一个类，可以通过此类直接取得时间呢，例如：现在给出了如下的一个接口：

```
package org.lxh.api.datedemo;
public interface DateTime {
    /**
     * 取得日期
     * @return 日期的字符串，例如：2009-12-22
     */
    public String getDate() ;
    /**
     * 取得日期时间
     * @return 日期时间的字符串，例如：2009-12-22 11:06:23.345
     */
    public String getDateTime() ;
    /**
     * 取得时间戳
     * @return 返回时间戳的字符串，例如：20091222110623345
     */
    public String getTimeStamp() ;
}
```

前面需要补 0 的问题需要注意，例如：01。

```
package org.lxh.api.datedemo;
import java.util.Calendar;
import java.util.GregorianCalendar;
public class DateTimeImpl implements DateTime {
    private Calendar calendar;
```

```
    public DateTimeImpl() {
        this.calendar = new GregorianCalendar();
    }
    @Override
    public String getDate() {
        StringBuffer buf = new StringBuffer();
        buf.append(calendar.get(Calendar.YEAR)).append("-");
        buf.append(this.addZero((calendar.get(Calendar.MONTH) + 1), 2)).append("-");
        buf.append(this.addZero(calendar.get(Calendar.DATE), 2));
        return buf.toString();
    }
    @Override
    public String getDateTime() {
        StringBuffer buf = new StringBuffer();
        buf.append(calendar.get(Calendar.YEAR)).append("-");
        buf.append(this.addZero((calendar.get(Calendar.MONTH) + 1), 2)).append("-");
        buf.append(this.addZero(calendar.get(Calendar.DATE), 2)).append(" ");
        buf.append(this.addZero(calendar.get(Calendar.HOUR_OF_DAY), 2)).append(":") ;
        buf.append(this.addZero(calendar.get(Calendar.MINUTE), 2)).append(":") ;
        buf.append(this.addZero(calendar.get(Calendar.SECOND), 2)).append(".") ;
        buf.append(this.addZero(calendar.get(Calendar.MILLISECOND), 3)) ;
        return buf.toString();
    }
    @Override
    public String getTimeStamp() {
        StringBuffer buf = new StringBuffer();
        buf.append(calendar.get(Calendar.YEAR));
        buf.append(this.addZero((calendar.get(Calendar.MONTH) + 1), 2));
        buf.append(this.addZero(calendar.get(Calendar.DATE), 2));
        buf.append(this.addZero(calendar.get(Calendar.HOUR_OF_DAY), 2));
        buf.append(this.addZero(calendar.get(Calendar.MINUTE), 2)) ;
        buf.append(this.addZero(calendar.get(Calendar.SECOND), 2)) ;
        buf.append(this.addZero(calendar.get(Calendar.MILLISECOND), 3)) ;
        return buf.toString();
    }
    private String addZero(int num, int len) {
        StringBuffer buf = new StringBuffer();
        buf.append(num);
        while (buf.length() < len) {
            buf.insert(0, 0);
        }
        return buf.toString();
    }
}
```

3.7.3、SimpleDateFormat 类（绝对重点）

Date 本身提供的数据已经属于一个完整的日期时间了，但是其显示的效果并不完整，所以如果现在需要将显示的日期进行格式化的话，就可以通过 java.text.SimpleDateFormat 类完成功能。

但是如果要进行格式化的话，有一点必须注意的是，需要指定一个格式化日期的模板：

- 年（yyyy）、月（MM）、日（dd）、时（HH）、分（mm）、秒（ss）、毫秒（SSS）

范例：格式化日期

```
package org.lxh.api.datedemo;
import java.text.SimpleDateFormat;
import java.util.Date;
public class SimpleDateFormatDemo01 {
    public static void main(String[] args) {
        // 准备好了一个要格式化的模板
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
        Date date = new Date();
        String str = sdf.format(date);
        System.out.println("格式化后的日期：" + str);
    }
}
```

通过 SimpleDateFormat 类可以将一个 Date 型的数据变为 String 型的数据。

范例：将 String 变回 Date 型

```
package org.lxh.api.datedemo;
import java.text.SimpleDateFormat;
import java.util.Date;
public class SimpleDateFormatDemo02 {
    public static void main(String[] args) throws Exception {
        String str = "2009-12-24 11:51:57.500" ;
        // 准备好了一个要格式化的模板
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
        Date date = sdf.parse(str) ;    // 将String --> Date
        System.out.println("格式化后的日期：" + date);
    }
}
```

依靠 SimpleDateFormat 可以完成 String 和 Date 型数据的转换。

实际上 SimpleDateFormat 类还可以完成日期格式的转换，例如：有两种不同类型的日期格式，可以通过此类转换：

```
package org.lxh.api.datedemo;
import java.text.SimpleDateFormat;
import java.util.Date;
public class SimpleDateFormatDemo03 {
    public static void main(String[] args) throws Exception {
        String oldDate = "2009-12-24 11:51:57.500";
        String newDate = null;
        // 准备好了一个要格式化的模板
```

```
        SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
        SimpleDateFormat sdf2 = new SimpleDateFormat(
                "yyyy年MM月dd日HH时mm分ss秒SSS毫秒");
        Date date = sdf1.parse(oldDate); // 从原日期字符串中取出日期数字
        newDate = sdf2.format(date); // 新的日期格式
        System.out.println("格式化后的日期：" + newDate);
    }
}
```

所以，上面的 DateTime 接口的实现类通过 SimpleDateFormat 类完成应该是最方便的。

```
package org.lxh.api.datedemo;
import java.text.SimpleDateFormat;
public class DateTimeImpl implements DateTime {
    private SimpleDateFormat sdf;
    @Override
    public String getDate() {
        String str = null;
        this.sdf = new SimpleDateFormat("yyyy-MM-dd");
        str = this.sdf.format(new java.util.Date());
        return str;
    }
    @Override
    public String getDateTime() {
        String str = null;
        this.sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
        str = this.sdf.format(new java.util.Date());
        return str;
    }
    @Override
    public String getTimeStamp() {
        String str = null;
        this.sdf = new SimpleDateFormat("yyyyMMddHHmmssSSS");
        str = this.sdf.format(new java.util.Date());
        return str;
    }
}
```

3.8、Arrays 类（理解）

Arrays 类是一个数组的操作类，之前曾经使用过此类完成过排序的操作，但是此类还有填充及输出的功能。

```
package org.lxh.api.arraysdemo;
import java.util.Arrays;
public class ArraysDemo {
    public static void main(String[] args) {
```

```
        int temp[] = { 50, 45, 8, 9, 1, 23, 4, 5, 23, 21, 324 };
        Arrays.sort(temp);
        System.out.println(Arrays.toString(temp));
        Arrays.fill(temp, 1);
        System.out.println(Arrays.toString(temp));
    }
}
```

但是在这个类的 sort()方法中有一个以下的排序操作：public static void sort(Object[] a)

可以对一个对象数组进行排序，那么下面观察一下：

```
package org.lxh.api.arraysdemo;
import java.util.Arrays;
class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "姓名：" + this.name + "，年龄：" + this.age;
    }
}
public class SortObjectArray {
    public static void main(String[] args) {
        Person per[] = { new Person("张三", 20), new Person("李四", 19),
                new Person("王五", 23) };
        Arrays.sort(per);
        for (int x = 0; x < per.length; x++) {
            System.out.println(per[x]);
        }
    }
}
```

但是，此时执行的时候出现了以下的错误提示：

```
Exception in thread "main" java.lang.ClassCastException: org.lxh.api.arraysdemo.Person
cannot be cast to java.lang.Comparable
```

此时提示的是一个类转换异常：Person 类的对象不能向 Comparable 转换。

3.9、比较器（绝对重点）

当需要对一组对象进行排序的时候，一定要指定比较规则，这种比较规则主要是将某一个类中的几个属性进行比较。

对于这种比较器的实现，在 Java 中有两种接口完成： **Comparable：使用的最广泛的一种**

Comparator：属于挽救的比较器

3.9.1、Comparable（重点）

Comparable 接口主要是用于执行比较器操作的接口，定义如下：

```
public interface Comparable<T>{
    public int compareTo(T o) ;
}
```

Comparable 接口中只存在了一个 compareTo()的方法，此方法的主要功能是编写比较规则的，此方法返回一个 int 型的数据，此值会返回三种结果：

- 0：两个对象相等
- 1：大于
- -1：小于

如果要使用 Arrays.sort()进行排序的话，则肯定要在对象所在的类中进行编写比较器的操作。

```
package org.lxh.api.arraysdemo;
import java.util.Arrays;
class Person implements Comparable<Person> {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "姓名：" + this.name + "，年龄：" + this.age;
    }
    @Override
    public int compareTo(Person o) {
        if (this.age > o.age) {
            return 1;
        } else if (this.age < o.age) {
            return -1;
        } else {
            return 0;
        }
    }
}
public class SortObjectArray {
    public static void main(String[] args) {
        Person per[] = { new Person("张三", 20), new Person("李四", 19),
                new Person("王五", 23) };
        Arrays.sort(per);
        for (int x = 0; x < per.length; x++) {
            System.out.println(per[x]);
```

```
        }
    }
}
```

如果对象数组要排序，则对象所在的类必须实现 Comparable 接口。

3.9.2、Comparable 排序原理（理解）

Comparable 排序的原理实际上就属于 Binary Tree 排序的操作形式。

下面就模拟一个排序的操作：

```
package org.lxh.api.arraysdemo;
class BinaryTree{
    class Node {
        private Comparable data;
        private Node left; // 保存左子树
        private Node right; // 保存右子树
        public Node(Comparable data) {
            this.data = data;
        }
        public void addNode(Node newNode) {
            if (this.data.compareTo(newNode.data) >= 0) { // 放在左子树
                if (this.left == null) {
                    this.left = newNode;
                } else {
                    this.left.addNode(newNode);
                }
            }
            if (this.data.compareTo(newNode.data) < 0) { // 放在右子树
                if (this.right == null) {
                    this.right = newNode;
                } else {
                    this.right.addNode(newNode);
                }
            }
        }
        public void printNode() { // 左 - 根 - 右
            if (this.left != null) {
                this.left.printNode();
            }
            System.out.println(this.data);
            if (this.right != null) {
                this.right.printNode();
            }
        }
```

```
    }
    private Node root; // 根节点
    public void add(Comparable data) {
        Node newNode = new Node(data);
        if (this.root == null) {
            this.root = newNode;
        } else {
            this.root.addNode(newNode);
        }
    }
    public void print() {
        if (this.root != null) {
            this.root.printNode();
        }
    }
}
public class BinaryTreeDemo {
    public static void main(String[] args) {
        BinaryTree bt = new BinaryTree();
        bt.add("B");
        bt.add("B");
        bt.add("A");
        bt.add("C");
        bt.print();
    }
}
```

3.9.3、Comparator

在 Arrays 类中可以使用此排序：public static <T> void sort(T[] a,Comparator<? super T> c)

此排序接收一个 Comparator 的接口实例。

```
package org.lxh.api.comdemo;
public class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = 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;
    }
    @Override
    public String toString() {
        return "姓名：" + this.name + "，年龄：" + this.age;
    }
}
```

但是，此类中由于没有使用 Comparable 接口，所以肯定是无法通过 Arrays.sort()进行排序的。

所以，可以通过 Comparator 挽救一个排序的规则。

```
package org.lxh.api.comdemo;
import java.util.Comparator;
public class PersonComparator implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        if (o1.getAge() > o2.getAge()) {
            return -1;
        } else if (o1.getAge() < o2.getAge()) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

所以，之后就可以继续使用 Arrays 类提供的 sort()方法进行排序。

```
package org.lxh.api.comdemo;
import java.util.Arrays;
public class Demo {
    public static void main(String[] args) {
        Person per[] = { new Person("张三", 20), new Person("李四", 19),
                new Person("王五", 23) };
        Arrays.sort(per, new PersonComparator());
        for (int x = 0; x < per.length; x++) {
            System.out.println(per[x]);
        }
    }
}
```

两种比较接口的区别：Comparable 属于定义类的时候使用，而 Comparator 属于挽救的比较器，所在的包不同。

Comparable 只有一个方法，而 Comparator 有两个方法。

3.10、正则表达式（绝对重点）

正则表达式是一个重要的组成部分，主要的功能可以用于复杂的验证、替换、拆分等操作，最早的时候正则表达式是在 PHP 下应用较为广泛的一种语句，但是后来在 Java 中如果要使用正则的话，则需要单独安装 **APAHCE** 的一个正则的组件包。

在 JDK 1.4 之后 Java 为了方便用户开发在 JDK 中已经加入了正则的支持，主要是在 java.util.regex 包中。

3.10.1、正则的作用

判断一个字符串是否由数字组成。

范例：如果现在不使用正则则采用的方式

- 将一个字符串变成一个字符数组，之后分别判断数组中的每个内容是否是数字。

```
package org.lxh.api.regexdemo;
public class CheckNumberDemo01 {
    public static void main(String[] args) {
        String str = "1234567";
        boolean flag = true; // 假设是正确的
        char c[] = str.toCharArray(); // 将字符串变为字符数组
        for (int x = 0; x < c.length; x++) {
            if (c[x] > '9' || c[x] < '0') { // 不是数字
                flag = false;
            }
        }
        if (flag) {
            System.out.println("是由数字组成！");
        } else {
            System.out.println("不是由数字组成！");
        }
    }
}
```

此时功能确实完成了，但是这样编写太复杂了，而且此处只是完成了一个简单的验证。

范例：现在使用正则

```
package org.lxh.api.regexdemo;
public class CheckNumberDemo02 {
    public static void main(String[] args) {
        String str = "1234s567";
        if (str.matches("\\d+")) {
            System.out.println("是由数字组成！");
        } else {
            System.out.println("不是由数字组成！");
        }
    }
}
```

两种方式很明显，第二种方式的操作更加简单，第二种就是采用了正则，其中的“\\d+”就是一个正则表达式的符号。

3.10.2、正则表达式的符号

构成正则的核心部分就是这些标记，这些标记都在 Pattern 类的 DOC 文档中有所说明。

1、 字符表示：
- \\：表示一个\
- 字母：表示一个具体的字符，例如：A
- \t：表示的是“\t”的转义字符
- \n：表示的是“\m”的转义字符

2、 字符集：表示一串完整的字符
- [abc]：表示内容可能是 a、b、c 字母的任意一个
- [^abc]：表示内容可能不是 a、b、c 字母的任意一个
- [a-zA-Z]：全部的英文字母
- [0-9]：全部的数字

3、 特殊表达式
- .：表示任意的一个字符
- \d：与[0-9]的含义一样；\D：与[^0-9]含义一样
- \w：与[a-zA-Z0-9_]的含义一样；\W：取反

4、 表达式的出现次数：默认的情况以上的所有表达式都只能表示一个字符，如果要表示多个，则要加上次数
- X?：出现 0 次或 1 次
- X+：出现 1 次或多次
- X*：出现 0 次、1 次或多次
- X{n}：出现正好 n 次
- X{n,m}：出现 n~m 次
- X{n,}：出现 n 次以上

如果要想操作以上的表达式，需要 Pattern 类 Matcher 两个类的支持。

3.10.3、Pattern 类

Pattern 类的主要功能是用于进行正则的匹配的规则验证，在此类中没有构造方法，所以构造方法被私有话了，只能使用：public static Pattern compile(String regex)

Pattern 类的主要功能就是拆分：public String[] split(CharSequence input)

范例：验证拆分

```
package org.lxh.api.regexdemo;
import java.util.regex.Pattern;
public class PatternDemo01 {
    public static void main(String[] args) {
        String str = "a1b22c333d4444e55555f";
        Pattern pat = Pattern.compile("\\d+");
        String s[] = pat.split(str); // 拆分
        for (int x = 0; x < s.length; x++) {
            System.out.println(s[x]);
```

```
        }
    }
}
```

3.10.4、Matcher 类

Matcher 类的主要功能是完成字符串的验证及替换的，但是如果要想取得此类的实例，必须依靠以下方法：

- 方法：public Matcher matcher(CharSequence input)

Matcher 的方法：

- 验证：public boolean matches()
- 替换：public String replaceAll(String replacement)

范例：例如美国的车牌号码的组成：12-335-222

```
package org.lxh.api.regexdemo;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class MatcherDemo01 {
    public static void main(String[] args) {
        String str = "11-234-45";
        Pattern pat = Pattern.compile("\\d{2}-\\d{3}-\\d{2}");
        Matcher mat = pat.matcher(str);
        if (mat.matches()) {
            System.out.println("验证通过！");
        }
    }
}
```

范例：一个用户的用户名应该是 6~15 位的字母、数字、_

```
package org.lxh.api.regexdemo;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class MatcherDemo02 {
    public static void main(String[] args) {
        String str = "helloworld";
        Pattern pat = Pattern.compile("\\w{6,15}");
        Matcher mat = pat.matcher(str);
        if (mat.matches()) {
            System.out.println("验证通过！");
        }
    }
}
```

范例：完成替换

```
package org.lxh.api.regexdemo;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
public class MatcherDemo03 {
    public static void main(String[] args) {
        String str = "a1b22c333d4444e55555f";
        Pattern pat = Pattern.compile("\\d+");
        Matcher mat = pat.matcher(str);
        System.out.println(mat.replaceAll("X")) ;
    }
}
```

但是在一般的使用中，验证使用的是最多的，尤其是针对于客户端提交的数据验证更是多。

3.10.5、String 类对正则的支持（重点）

因为在进行数据传送的时候字符串类型的数据出现较多，所以在 JDK 1.4 之后对 String 类进行了改进了，可以直接使用正则进行验证，有以下几个方法支持正则：

- 验证：public boolean matches(String regex)
- 拆分：public String[] split(String regex)
- 替换：public String replaceAll(String regex,String replacement)

范例：完成验证，验证 email

```
package org.lxh.api.regexdemo;
public class StringMatch {
    public static void main(String[] args) {
        String email = "aa@aa.com";
        if (email.matches("\\w+@\\w+\\.\\w+")) {
            System.out.println("验证通过！");
        }
    }
}
```

范例：拆分

```
package org.lxh.api.regexdemo;
public class StringSplit {
    public static void main(String[] args) {
        String ip = "192.168.1.2";
        String s[] = ip.split("\\.");
        for (int x = 0; x < s.length; x++) {
            System.out.println(s[x]);
        }
    }
}
```

以后在开发中使用最多的肯定是直接利用 String 操作正则，而 Pattern 类和 Matcher 类基本上是不使用的。

3.11、反射机制（理解）

反射是 Java 的最大特征，而且所有的开源项目都是依靠了反射的操作原理，但是反射并不会在具体的开发中出现。

3.11.1、认识反射

现在如果要产生一个对象，肯定需要类。也就是说现在要先有类再有对象。

那么如果倒过来呢？很想知道从一个对象掌握其对象所在的包.类的话，则就需要使用反射。

```
package org.lxh.api.reflectdemo.classdemo;
class Person{}
public class ClassDemo {
    public static void main(String[] args) {
        Person per = new Person() ;
        System.out.println(per.getClass().getName());
    }
}
```

此时得到的是一个对象所在的“包.类”名称，那么 getClass()方法实际上就是作为反射的取得，此方法是 Object 类定义的：public final **Class**<?> getClass()

Class 就是反射的源头。

3.11.2、Class

Class 实际上表示的就是一个完整的类，但是 Class 类如果要想进行实例化的话，可以采用如下三种方式：

- 利用 Object 类中的 getClass()方法
- 利用“类.class”的形式实例化
- **静态方法：public static Class<?> forName(String className) throws ClassNotFoundException**

```
package org.lxh.api.reflectdemo.classdemo;
class Person{}
public class ClassDemo {
    public static void main(String[] args) {
        Class<?> cls = Person.class ;
        System.out.println(cls.getName());
    }
}
```

但是在开发中使用最多的实例化方法不是以上的，而是通过 forName()方法完成，只需要在方法中传递一个完整的“包.类”名称即可。

```
package org.lxh.api.reflectdemo.classdemo;
class Person {
}
public class ClassDemo {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class
                .forName("org.lxh.api.reflectdemo.classdemo.Person");
        System.out.println(cls.getName());
    }
}
```

3.11.3、通过反射实例化对象（重点）

正常情况下一个对象肯定是通过构造方法进行实例化，但是一旦引入了反射的概念之后，就可以通过反射的方式进行对象的实例化了，使用方法：public T newInstance() throws InstantiationException,IllegalAccessException

```
package org.lxh.api.reflectdemo.classdemo;
class Person {
    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;
    }
}
public class ClassDemo {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class
                .forName("org.lxh.api.reflectdemo.classdemo.Person");
        Person per = (Person) cls.newInstance();   // 实例化对象
        per.setName("张三") ;
        per.setAge(30) ;
        System.out.println("姓名：" + per.getName());
        System.out.println("年龄：" + per.getAge());
    }
}
```

本代码实际上是有局限性的，只能用在类中存在无参构造的时候。

3.11.4、定义有参构造

在之前强调过，可以通过构造方法为对象进行实例化的操作。

```
    public Person(String name,int age){
        this.setName(name) ;
        this.setAge(age) ;
    }
```

增加了一个有两个参数的构造方法，但是此时，再次运行程序会出现问题：

```
Exception in thread "main" java.lang.InstantiationException:
```

现在的错误提示表示的是对象的实例化错误，因为现在的类中不存在无参构造了。一旦不存在之后，之前的代码再操作的时候肯定会出现找不到无参构造而造成实例化异常。

如果现在一个类中没有无参构造，而只存在有参构造的话，那么就必须明确的找到类中的有参构造，同时设置相关的内容进行对象的实例化，此时就需要使用 java.lang.reflect.Constructor 类完成。

一个类中本身会存在多个构造方法，如果要找到构造方法：

|- public Constructor<?>[] getConstructors() throws SecurityException

取得构造之后，可以通过构造的 newInstance()方法实例化：

|- public T newInstance(Object... initargs) throws InstantiationException,
IllegalAccessException,IllegalArgumentException,InvocationTargetException

```
package org.lxh.api.reflectdemo.classdemo;
import java.lang.reflect.Constructor;
class Person {
    private String name ;
    private int age ;
    public Person(String name,int age){
        this.setName(name) ;
        this.setAge(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;
    }
}
public class ClassDemo {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class
                .forName("org.lxh.api.reflectdemo.classdemo.Person");
        Constructor<?> cons[] = cls.getConstructors() ;
        Person per = (Person) cons[0].newInstance("张三",30);  // 实例化对象
        System.out.println("姓名：" + per.getName());
        System.out.println("年龄：" + per.getAge());
    }
}
```

如果此时类中没有无参构造，则只能通过此种形式进行对象的实例化，所以，之前在开发类的时候已经明确要求过，一个类至少存在一个无参构造方法。

3.11.5、在工厂中应用反射

工厂设计模式，通过一个工厂类取得一个接口的对象，但是最早的工厂模式中可以发现一个问题，如果现在要扩充子类，则肯定需要修改工厂，那么这个时候就可以通过反射来解决此类问题。

```
package org.lxh.api.reflectdemo.classdemo;
interface Fruit{
    public void eat() ;
}
class Apple implements Fruit{
    public void eat(){
        System.out.println("** 吃苹果！") ;
    }
}
class Orange implements Fruit{
    public void eat(){
        System.out.println("** 吃橘子！") ;
    }
}
class Factory {
    public static Fruit getInstance(String className) {
        Fruit f = null;
        try {
            f = (Fruit) Class.forName(className).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return f;
    }
}
public class FactoryDemo {
    public static void main(String[] args) {
        Fruit f = Factory.getInstance("org.lxh.api.reflectdemo.classdemo.Apple") ;
        f.eat() ;
    }
}
```

在工厂设计模式之中，应用反射的最大好处时，即使在增加了新的子类，工厂也可以不用做任何的修改。

3.11.6、取得方法

实际上通过反射可以取得一个类的完整组成部分，例如：一个类的父类，或者是实现的接口。为了方便操作，建立一个完整的类，并且通过此类进行反射操作。

```
package org.lxh.api.reflectdemo.classdemo;
interface Info{
```

```
    public String FLAG = "HELLO" ;
    public void fun() ;
    public String say(String name,int age) ;
}
public class Student implements Info {
    private String school ;
    public String getSchool() {
        return school;
    }
    public void setSchool(String school) {
        this.school = school;
    }
    @Override
    public void fun() {
        System.out.println("Hello World!!!");
    }
    @Override
    public String say(String name, int age) {
        return "姓名：" + name + "，年龄：" + age ;
    }
}
```

这个时候可以通过反射取得本类中全部的方法：

- 方法：public Method[] getMethods() throws SecurityException

```
package org.lxh.api.reflectdemo.classdemo;
import java.lang.reflect.Method;
public class MethodDemo {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class
                .forName("org.lxh.api.reflectdemo.classdemo.Student");
        Method met[] = cls.getMethods(); // 取得全部方法
        for (int x = 0; x < met.length; x++) {
            System.out.println(met[x]);
        }
    }
}
```

此时的信息输出，是通过 toString()的方法得来的。

```
package org.lxh.api.reflectdemo.classdemo;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
public class MethodDemo {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class
                .forName("org.lxh.api.reflectdemo.classdemo.Student");
        Method met[] = cls.getMethods(); // 取得全部方法
```

```
        for (int x = 0; x < met.length; x++) {
            System.out.print(Modifier.toString(met[x].getModifiers()) + " ") ;
            System.out.print(met[x].getReturnType().getName() + " ") ;
            System.out.print(met[x].getName() + "(");  // 取得方法名称
            Class<?> params[] = met[x].getParameterTypes();
            for (int y = 0; y < params.length; y++) {
                System.out.print(params[y].getName() + " arg" + y);
                if (y < params.length - 1) {
                    System.out.print(",");
                }
            }
            System.out.print(") ");
            Class<?> exp[] = met[x].getExceptionTypes();
            if (exp.length > 0) {
                System.out.print("throws ");
                for (int y = 0; y < exp.length; y++) {
                    System.out.print(exp[y].getName());
                    if (y < exp.length - 1) {
                        System.out.print(",");
                    }
                }
            }
            System.out.println() ;
        }
    }
}
```

Method 中还有一个最大的特点，是可以通过反射调用方法。

范例：调用无参无返回值的方法

```
package org.lxh.api.reflectdemo.classdemo;
import java.lang.reflect.Method;
public class InvokeDemo {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class
                .forName("org.lxh.api.reflectdemo.classdemo.Student");
        Object obj = cls.newInstance() ;
        Method funMet = cls.getMethod("fun") ;
        funMet.invoke(obj) ;    // 反射调用方法
    }
}
```

范例：调用有参有返回值的方法

```
package org.lxh.api.reflectdemo.classdemo;
import java.lang.reflect.Method;
public class InvokeDemo {
    public static void main(String[] args) throws Exception {
```

```
        Class<?> cls = Class
                .forName("org.lxh.api.reflectdemo.classdemo.Student");
        Object obj = cls.newInstance() ;
        Method funMet = cls.getMethod("say",String.class,int.class) ;
        String ret = (String)funMet.invoke(obj,"张三",20) ;// 反射调用方法
        System.out.println(ret);
    }
}
```

3.11.7、取得属性

在 Java 的反射机制之中，可以发现，通过反射可以完整的恢复一个类的结构，对于属性也一样，可以通过以下方法取得全部的属性： 本类属性：public Field[] getDeclaredFields() throws SecurityException

继承属性：public Field[] getFields() throws SecurityException

范例：取得属性

```
package org.lxh.api.reflectdemo.classdemo;
import java.lang.reflect.Field;
public class FieldDemo {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class
                .forName("org.lxh.api.reflectdemo.classdemo.Student");
        {
            Field f[] = cls.getFields();
            for (int x = 0; x < f.length; x++) {
                System.out.println(f[x]);
            }
        }
        {
            Field f[] = cls.getDeclaredFields() ;
            for (int x = 0; x < f.length; x++) {
                System.out.println(f[x]);
            }
        }
    }
}
```

使用 Field 还有一个好处，就是可以直接调用本类中的属性。

```
package org.lxh.api.reflectdemo.classdemo;
import java.lang.reflect.Field;
public class InvokeFieldDemo {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class
                .forName("org.lxh.api.reflectdemo.classdemo.Student");
        Field school = cls.getDeclaredField("school");
```

```
        Object obj = cls.newInstance();
        school.setAccessible(true); // 取消封装
        school.set(obj, "清华大学");
        System.out.println(school.get(obj));
    }
}
```

3.12、对象克隆（理解）

克隆就是复制，将一个对象直接复制。

不是说只要是一个类就必须进行克隆，如果一个类的对象要想完成对象的克隆操作的话，则必须实现一个接口：Cloneable。此接口没有任何的方法。

如果要想克隆，则必须依靠 Object 类中的 clone 方法：protected Object clone() throws CloneNotSupportedException

但是此方法属于受保护的访问权限，所以如果一个非本包的非子类，要访问肯定不能访问。

```
package org.lxh.api.clonedemo;
class Person implements Cloneable{
    private String name ;
    public Person(String name){
        this.name = name ;
    }
    public void setName(String name){
        this.name = name ;
    }
    public String getName(){
        return this.name ;
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
public class CloneDemo {
    public static void main(String[] args) throws Exception {
        Person per = new Person("张三") ;
        Person cp = (Person) per.clone() ; // 克隆
        System.out.println(per + ", " + per.getName());
        System.out.println(cp + ", " + cp.getName());
    }
}
```

在本程序中唯一重要的就是，Cloneable 是一个标识接口， 用于表示一种能力。

4、总结

1、 StringBuffer
2、 对象生命周期，及处理的方法
3、 Date、SimpleDateFormat（字符串和 Date 型的转换）
4、 BigInteger
5、 比较器
6、 正则
7、 通过反射实例化对象

5、预习任务

File、OutputStream、InputStream、Writer、Reader、PrintStream、ByteArrayOutputStream、ByteArrayInputStream、对象序列化。

6、作业

1、 对象比较

现在要求完成以下的一个操作：

- 现在假设输入的信息是（姓名：年龄：成绩）：张三:20:89|李四:20:90|王五:19:90|赵六:21:100。
- 要求按照成绩由高到低排序显示，如果成绩相同，则按照年龄由低到高排序。