

# JAVA程序设计

潘微科

感谢：教材《Java大学实用教程》的作者和其他老师提供PowerPoint讲义等资料！  
说明：本课程所使用的所有讲义，都是在以上资料上修改的。

# Outline

- 7.1 Date类
- 7.2 Calendar类
- 7.3 Math类与BigInteger类
- 7.4 数字格式化
- 7.5 LinkedList<E>泛型类
- 7.6 HashSet<E>泛型类
- 7.7 HashMap<K,V>泛型类
- 7.8 TreeSet<E>泛型类
- 7.9 TreeMap<K,V>泛型类
- 7.10 Stack<E>泛型类

# 7.1 Date类

- 1.Date对象
  - Date类在**java.util**包中。
  - 使用Date类的无参数构造方法创建的对象可以获取**本地**当前时间。
  - 用Date的构造方法**Date(long time)**创建的Date对象表示相对1970年1月1日0点（Greenwich Mean Time, GMT, 格林威治标准时间）的时间。例如，参数time取值**60\*60\*1000毫秒**表示**Thu Jan 01 01:00:00 GMT 1970**。
  - 可以用System类的**静态**方法**public long currentTimeMillis()**获取系统当前时间，这个时间是从1970年1月1日0点（GMT）到目前时刻所走过的**毫秒数**。

## 7.1 Date类

- 2.格式化时间
  - Date对象表示时间的默认顺序：星期 月 日 小时 分 秒 年
    - Sat Apr 28 21:59:38 CST 2001
  - 我们可能希望按照某种习惯来输出时间
    - 年 月 星期 日
    - 年 月 星期 日 小时 分 秒
  - 可以使用DateFormat的子类SimpleDateFormat来实现日期的格式化

## 7.1 Date类

- SimpleDateFormat构造方法: public **SimpleDateFormat**(String pattern)
  - 用参数pattern指定的格式创建一个对象sdf

```
String pattern = "yyyy-MM-dd";  
SimpleDateFormat sdf = new SimpleDateFormat(pattern);
```

- 用public String **format**(Date date)方法格式化时间对象

```
Date currentTime = new Date();  
String currentTime2 = sdf.format(currentTime);
```

# 7.1 Date类

- 常用**时间元字符**

- y, **yy**: 2位数字年份, 如14
- yyyy: 4位数字年份, 如2014
- M, **MM**: 2位数字月份, 如08
- MMM: 汉字月份, 如八月
- d, **dd**: 2位数字日期, 如09, 22
- **a**: 上午或下午
- H, **HH**: 2位数字小时 (00-23)
- h, **hh**: 2位数字小时 (am/pm, 01-12)
- m, **mm**: 2位数字分
- s, **ss**: 2位数字秒
- E, **EE**: 星期

注: 关于pattern中的普通字符 (非**时间元字符**), 如果是ASCII字符集中的字符, 必须用""转义符

" 'Time' yyyy-MM-dd"

## 7.1 Date类

```
import java.util.Date;
import java.text.SimpleDateFormat;
```

```
public class Example7_1
{
    public static void main(String args[])
    {
        Date currentTime = new Date();
        System.out.println("Current time: " + currentTime);

        SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd ");
        System.out.println("Current time: " + sdf1.format(currentTime));

        SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss(a)(EE)");
        System.out.println("Current time: " + sdf2.format(currentTime));

        long time = -1000L;
        Date date = new Date(time);
        System.out.println(time + "ms: " + sdf2.format(date));

        time = 1000L;
        date = new Date(time);
        System.out.println(time + "ms: " + sdf2.format(date));
    }
}
```

```
Current time: Wed Oct 22 14:31:19 CST 2014
Current time: 2014-10-22
Current time: 2014-10-22 14:31:19(下午)(星期三)
-1000ms: 1970-01-01 07:59:59(上午)(星期四)
1000ms: 1970-01-01 08:00:01(上午)(星期四)
```

# Outline

- 7.1 Date类
- 7.2 Calendar类
- 7.3 Math类与BigInteger类
- 7.4 数字格式化
- 7.5 LinkedList<E>泛型类
- 7.6 HashSet<E>泛型类
- 7.7 HashMap<K,V>泛型类
- 7.8 TreeSet<E>泛型类
- 7.9 TreeMap<K,V>泛型类
- 7.10 Stack<E>泛型类



## 7.2 Calendar类

- Calendar类在java.util包中。
- 使用Calendar类的static方法getInstance()可以初始化一个日历对象

```
Calendar calendar = Calendar.getInstance();
```

- 然后，calendar对象可以调用方法：
  - public final void **set**(int year, int month, int date)
  - public final void **set**(int year, int month, int date, int hour, int minute)
  - public final void **set**(int year, int month, int date, int hour, int minute, int second)

将日历**翻到**任何一个时间，当参数year取负数时表示公元前。

## 7.2 Calendar类

- calendar对象调用方法public int get(int field)可以获取有关年份、月份、小时、星期等信息，参数field的有效值由Calendar的静态常量指定，如：

```
calendar.get(Calendar.MONTH);
```

- 返回一个整数，0表示一月，1表示二月，等。

## 7.2 Calendar类

默认情况下，一周中第1天是星期日

```
import java.util.*;
public class Example7_2
{
    public static void main(String args[])
    {
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(new Date());

        String day_of_week = String.valueOf(calendar.get(Calendar.DAY_OF_WEEK)-1);
        System.out.println(day_of_week);

        calendar.set(1931,8,18);
        long timeOne = calendar.getTimeInMillis();

        calendar.set(1945,7,15);
        long timeTwo = calendar.getTimeInMillis();

        long days = (timeTwo-timeOne)/(1000*60*60*24);
        System.out.println("1945年8月15日和1931年9月18日相隔: " + days + "天");
    }
}
```

星期三

3  
1945年8月15日和1931年9月18日相隔: 5080天

注：我常用类似的时间差来计算一段代码的运行时间。

## 7.2 Calendar类

```
import java.util.*;
public class Example7_3
{
    public static void main(String args[])
    {
        Calendar calendar = Calendar.getInstance();
        calendar.set(1931,8,1);
        int day_of_week = calendar.get(Calendar.DAY_OF_WEEK)-1;
        String a[] = new String[day_of_week+30];
        for(int i=0; i<day_of_week; i++)
            a[i] = "";

        for(int i=day_of_week,n=1; i<day_of_week+30; i++)
        {
            a[i] = String.valueOf(n);
            n++;
        }
        for(int i=0;i<a.length;i++)
        {
            if(i%7==0&&i!=0)
                System.out.printf("\n");
            System.out.printf("%5s",a[i]);
        }
    }
}
```

| 日  | 一  | 二  | 三  | 四  | 五  | 六  |
|----|----|----|----|----|----|----|
|    |    | 1  | 2  | 3  | 4  | 5  |
| 6  | 7  | 8  | 9  | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| 27 | 28 | 29 | 30 |    |    |    |

# Outline

- 7.1 Date类
- 7.2 Calendar类
- 7.3 Math类与BigInteger类
- 7.4 数字格式化
- 7.5 LinkedList<E>泛型类
- 7.6 HashSet<E>泛型类
- 7.7 HashMap<K,V>泛型类
- 7.8 TreeSet<E>泛型类
- 7.9 TreeMap<K,V>泛型类
- 7.10 Stack<E>泛型类

## 7.3 Math类与BigInteger类

- 1.Math类
- 在编写程序时，可能需要计算一个数的平方根、绝对值、获取一个随机数等。java.lang包中的类包含许多用来进行科学计算的**静态方法**（static methods，又称类方法），这些方法可以直接通过**类名**调用。
- 另外，Math类还有两个**静态常量**，E和PI，它们的值分别是：
  - 2.7182828284590452354
  - 3.14159265358979323846

## 7.3 Math类与BigInteger类

- Math类常用方法
  - public static long **abs**(double a): 返回a的绝对值
  - public static double **max**(double a, double b): 返回a, b的最大值
  - public static double **min**(double a, double b): 返回a, b的最小值
  - public static double **random**(): 产生一个0到1之间的随机数, 范围是**[0,1)**
  - public static double **pow**(double a, double b): 返回a的b次幂
  - public static double **sqrt**(double a): 返回a的平方根
  - public static double **log**(double a): 返回a的对数
  - public static double **sin**(double a): 返回正弦值
  - public static double **asin**(double a): 返回反正弦值

## 7.3 Math类与BigInteger类

- 2.BigInteger类
- 程序有时需要处理大整数，java.math包中的BigInteger类提供任意精度的整数运算。可以使用如下构造方法创建一个十进制的BigInteger对象
  - `public BigInteger(String val)`
- 参数val中如果含有非数字字符就会发生NumberFormatException异常



## 7.3 Math类与BigInteger类

- BigInteger类的常用方法
  - public BigInteger **add**(BigInteger val): 返回当前大整数对象与参数指定的大整数对象的**和**
  - public BigInteger **subtract**(BigInteger val): 返回当前大整数对象与参数指定的大整数对象的**差**
  - public BigInteger **multiply**(BigInteger val): 返回当前大整数对象与参数指定的大整数对象的**积**
  - public BigInteger **divide**(BigInteger val): 返回当前大整数对象与参数指定的大整数对象的**商**
  - public BigInteger **remainder**(BigInteger val): 返回当前大整数对象与参数指定的大整数对象的**余**

## 7.3 Math类与BigInteger类

- public int **compareTo**(BigInteger val): 返回当前大整数对象与参数指定的大整数的比较结果，返回值是1、-1或0，分别表示当前大整数对象大于、小于或等于参数指定的大整数
- public BigInteger **abs**(): 返回当前大整数对象的**绝对值**
- public BigInteger **pow**(int exponent): 返回当前大整数对象的exponent**次幂**
- public String **toString**(): 返回当前大整数对象十进制的字符串表示
- public String **toString**(int p): 返回当前大整数对象**p进制**的字符串表示

## 7.3 Math类与BigInteger类

```
import java.math.*;
public class Example7_6
{
    public static void main(String args[])
    {
        BigInteger n1 = new BigInteger("987654321987654321987654321");
        BigInteger n2 = new BigInteger("123456789123456789123456789");
        System.out.println("add: " + n1.add(n2));
        System.out.println("subtract: " + n1.subtract(n2));
        System.out.println("multiply: " + n1.multiply(n2));
        System.out.println("divide: " + n1.divide(n2));
        BigInteger m = new BigInteger("77889988");
        BigInteger COUNT = new BigInteger("0");
        BigInteger ONE = new BigInteger("1");
        BigInteger TWO = new BigInteger("2");
        for( BigInteger i=TWO; i.compareTo(m)<0; i=i.add(ONE) )
        {
            if((n1.remainder(i).compareTo(BigInteger.ZERO))==0)
            {
                COUNT = COUNT.add(ONE);
            }
        }
        System.out.println(COUNT);
    }
}
```

← 要熟悉面向对象的编程方式

```
add: 11111111111111111111111111110
subtract: 864197532864197532864197532
multiply: 121932631356500531591068431581771069347203169112635269
divide: 8
30
```

# Outline

- 7.1 Date类
- 7.2 Calendar类
- 7.3 Math类与BigInteger类
- 7.4 数字格式化
- 7.5 LinkedList<E>泛型类
- 7.6 HashSet<E>泛型类
- 7.7 HashMap<K,V>泛型类
- 7.8 TreeSet<E>泛型类
- 7.9 TreeMap<K,V>泛型类
- 7.10 Stack<E>泛型类

## 7.4 数字格式化

- 有时我们可能需要对输出的数字结果进行必要的格式化，例如，对于3.14356789，我们希望保留小数位为3位、整数部分至少要显示3位，即将3.14356789格式化为003.144。
- 可以使用**java.text**包中的NumberFormat类，该类调用如下静态方法来实例化一个NumberFormat对象
  - public static final NumberFormat getInstance()

```
NumberFormat f = NumberFormat.getInstance();
```

## 7.4 数字格式化

- NumberFormat常用方法：
  - public void setMaximumFractionDigits(int newValue)
  - public void setMinimumFractionDigits(int newValue)
  - public void setMaximumIntegerDigits(int newValue)
  - public void setMinimumIntegerDigits(int newValue)
- **对象f**可调用public final String **format**(double number)方法来格式化数字number。

## 7.4 数字格式化

- 【例子】

```
import java.text.NumberFormat;
public class Example_Format
{
    public static void main(String args[])
    {
        double a = Math.sqrt(10);
        System.out.println("Before: " + a);

        NumberFormat f = NumberFormat.getInstance();
        f.setMaximumFractionDigits(5);
        f.setMinimumIntegerDigits(3);
        String s = f.format(a);
        System.out.println("After: " + s);
    }
}
```

```
Before: 3.1622776601683795
After: 003.16228
```

## 7.4 数字格式化

- 【例子】  
小数点后最多保留n位

```
class MyNumberFormat
{
    public String format(double a, int n)
    {
        String str = String.valueOf(a);
        int index = str.indexOf(".");

        String temp = str.substring(index+1);
        int fractionLeng = temp.length();
        n = Math.min(fractionLeng, n);
        str = str.substring(0, index+n+1);

        return str;
    }
}
```

↓  
小数点占1位

```
public class Example_Format2
{
    public static void main(String args[])
    {
        double a = Math.sqrt(10);
        System.out.println("Before: " + a);
        MyNumberFormat myFormat=new MyNumberFormat();
        System.out.println("After: " + myFormat.format(a,5));
    }
}
```

```
Before: 3.1622776601683795
After: 3.16227           24
```



# Outline

- 7.1 Date类
- 7.2 Calendar类
- 7.3 Math类与BigInteger类
- 7.4 数字格式化
- 7.5 LinkedList<E>泛型类
- 7.6 HashSet<E>泛型类
- 7.7 HashMap<K,V>泛型类
- 7.8 TreeSet<E>泛型类
- 7.9 TreeMap<K,V>泛型类
- 7.10 Stack<E>泛型类

## 7.5 LinkedList<E>泛型类

- 使用LinkedList<E>泛型类可以创建**链表**结构的数据对象。
- 链表是由若干个节点组成的一种数据结构，每个节点含有一个数据和下一个节点的引用（**单**链表），或含有一个数据并含有上一个节点的引用和下一个节点的引用（**双**链表），节点的索引从0开始。
- 链表适合**动态地改变存储的数据**，如，增加、删除节点等操作。

## 7.5 LinkedList<E>泛型类

- 1. LinkedList<E>对象
- java.util包中的LinkedList<E>泛型类创建的对象以链表结构存储数据，习惯上称LinkedList类创建的对象为**链表对象**。例如，

```
LinkedList<String> mylist = new LinkedList<String>();
```

- 创建一个空**双链表**。然后mylist可以使用**add(String obj)**方法向链表依次**增加**节点，节点中的数据是参数obj指定对象的**引用**。

```
mylist.add("How");  
mylist.add("Are");  
mylist.add("You");  
mylist.add("Java");
```

- 这时，双链表mylist就有了有4个节点，节点是自动连接在一起的，不需要我们去做连接，也就是说，**不需要我们去操作安排节点中所存放的下一个或上一个节点的引用**。

## 7.5 LinkedList<E>泛型类

- 2.常用方法
- 以下是LinkedList<E>泛型类的一些常用方法：
  - `public boolean add(E element)`: 向链表末尾**添加**一个新的节点，该节点中的数据是参数`element`指定的对象。
  - `public void add(int index, E element)`: 向链表的指定位置**添加**一个新的节点，该节点中的数据是参数`element`指定的对象。
  - `public void addFirst(E element)`: 向链表的头**添加**新节点，该节点中的数据是参数`element`指定的对象。

## 7.5 LinkedList<E>泛型类

- `public E removeFirst()`: **删除**第一个节点，并返回这个节点中的对象。
- `public E removeLast()`: **删除**最后一个节点，并返回这个节点中的对象。
- `public E get(int index)`: **得到**链表中指定位置处节点中的对象。
- `public E getFirst()`: **得到**链表中第一个节点中的对象。
- `public E getLast()`: **得到**链表中最后一个节点中的对象。

## 7.5 LinkedList<E>泛型类

- `public int indexOf(E element)`: **返回**含有数据`element`的节点在链表中**首次出现**的位置，如果链表中无此节点则返回-1。
- `public int lastIndexOf(E element)`: **返回**含有数据`element`的节点在链表中最后出现的位置，如果链表中无此节点则返回-1。
- `public E set(int index, E element)`: 将当前链表`index`位置节点中的对象替换为参数`element`指定的对象，并返回被**替换**的对象。

## 7.5 LinkedList<E>泛型类

- `public int size()`: 返回链表的**长度**，即节点的个数。
- `public boolean contains(Object element)`: 判断链表节点中是否有节点**包含**对象`element`。
- `public Object clone()`: 得到当前链表的一个**克隆**链表，该克隆链表中节点数据的改变不会影响到当前链表中节点的数据，反之亦然。

## 7.5 LinkedList<E>泛型类

- 【例子7】

```
import java.util.*;
class Student
{
    String name;
    int score;
    Student(String name, int score)
    {
        this.name = name;
        this.score = score;
    }
}
```

```
public class Example7_7
{
    public static void main(String args[])
    {
        LinkedList<Student> mylist = new LinkedList<Student>();
        Student stu1 = new Student("S1",78);
        Student stu2 = new Student("S2",98);
        mylist.add(stu1);
        mylist.add(stu2);

        int number = mylist.size();
        for(int i=0; i<number; i++)
        {
            Student temp = mylist.get(i);
            System.out.printf("%s:%d\n",temp.name,temp.score);
        }
    }
}
```

```
S1:78
S2:98
```



## 7.5 LinkedList<E>泛型类

- 【例子8】

```
import java.util.*;
class Student
{
    String name;
    int score;
    Student(String name, int score)
    {
        this.name = name;
        this.score = score;
    }
}
```

```
public class Example7_7
{
    public static void main(String args[])
    {
        LinkedList<Student> mylist = new LinkedList<Student>();
        Student stu1 = new Student("S1",78);
        Student stu2 = new Student("S2",98);
        mylist.add(stu1);
        mylist.add(stu2);

        Iterator<Student> iter = mylist.iterator();
        while(iter.hasNext())
        {
            Student temp = iter.next();
            System.out.printf("%s:%d\n",temp.name,temp.score);
        }
    }
}
```

```
S1:78
S2:98
```

## 7.5 LinkedList<E>泛型类

- 4.LinkedList<E>泛型类实现的接口
- LinkedList<E>泛型类实现了泛型接口List<E>，而List<E>接口是Collection<E>接口的子接口。
- LinkedList<E>类中的绝大部分方法都是接口方法的实现。
- 编程时，可以使用**接口回调技术**，即把LinkedList<E>对象的引用赋值给Collection<E>接口变量或List<E>接口变量，那么接口就可以调用类实现的接口方法。

## 7.5 LinkedList<E>泛型类

- 5.JDK1.5之前<sup>之前</sup>的LinkedList类
- JDK1.5之前没有泛型的LinkedList类，可以用普通的LinkedList创建一个链表对象，例如：

```
LinkedList mylist = new LinkedList();
```

- 创建了一个空双链表。然后mylist链表可以使用add(Object obj)方法向这个链表依次添加节点。由于任何类都是Object类的子类，因此可以把<sup>任何一个对象</sup>作为链表节点中的对象。

## 7.5 LinkedList<E>泛型类

- 需要注意的是当使用`get()`获取一个节点中的对象，要用**类型转换运算符**转换回原来的类型。
- Java泛型的主要目的是可以建立具有**类型安全**的集合框架（*Java Collections Framework*），如链表、散列表等数据结构，最重要的一个优点就是：在使用这些泛型类建立的数据结构时，**不必进行强制类型转换**，即**不要求进行运行时类型检查（在编译阶段已经完成检查）**。
- JDK1.5是支持泛型的编译器，它将运行时类型检查**提前到编译时**执行，使代码更安全。如果你使用**旧版本**的LinkedList类，1.5编译器会给出警告信息，但程序仍能正确运行。

## 7.5 LinkedList<E>泛型类

- 【例子9】

旧版本的LinkedList

```
import java.util.*;

public class Example7_9
{
    public static void main(String args[])
    {
        LinkedList mylist = new LinkedList();
        mylist.add("A");
        mylist.add(1);
        String str = (String) mylist.get(0); // 必须强制转换取出的数据,否则报错
        System.out.println(str);

        int num = (int) mylist.get(1); // 必须强制转换取出的数据,否则报错
        System.out.println(num);
    }
}
```

A  
1

# Outline

- 7.1 Date类
- 7.2 Calendar类
- 7.3 Math类与BigInteger类
- 7.4 数字格式化
- 7.5 LinkedList<E>泛型类
- 7.6 HashSet<E>泛型类
- 7.7 HashMap<K,V>泛型类
- 7.8 TreeSet<E>泛型类
- 7.9 TreeMap<K,V>泛型类
- 7.10 Stack<E>泛型类

## 7.6 HashSet<E>泛型类

- HashSet<E>泛型类在数据组织上类似数学上的集合，可以进行"交"、"并"、"差"等运算。
- 1.HashSet<E>对象
- HashSet<E>泛型类创建的对象称作集合，例如

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

- 对象set是一个可以存储String类型数据的集合，可以调用add(String s)方法将String类型的数据添加到集合中，添加到集合中的数据称做集合的元素。

## 7.6 HashSet<E>泛型类

- 集合不允许有相同的元素，也就是说，如果b已经是集合中的元素，那么再执行set.add(b)操作是无效的。
- 集合对象的初始容量（capacity）是16个字节，装载因子（load factor）是0.75，也就是说，如果集合添加的元素超过总容量的75%时，集合的容量将增加一倍。



## 7.6 HashSet<E>泛型类

- 2. 常用方法

- `public boolean add(E o)`: 向集合**添加**参数指定的元素。
- `public void clear()`: **清空**集合，使集合不含有任何元素。
- `public boolean contains(Object o)`: 判断集合**是否包含**参数指定的数据。
- `public boolean isEmpty()`: 判断集合**是否为空**。
- `public boolean remove(Object o)`: **删除**参数指定的元素。
- `public int size()`: 返回集合中元素的**个数**。
- `Object[] toArray()`: 将集合元素存放到**数组**中，并返回这个数组。
- `boolean containsAll(HashSet set)`: 判断当前集合**是否包含**参数指定的集合。
- `public Object clone()`: 得到当前集合的一个**克隆**对象，该对象中元素的改变不会影响到当前集合中的元素，反之亦然。

## 7.6 HashSet<E>泛型类

- 3.集合的交、并与差
- 集合对象调用boolean **addAll**(HashSet set)方法可以和参数指定的集合求并运算，使得**当前集合**成为两个集合的**并**。
- 集合对象调用boolean **retainAll**(HashSet set)方法可以和参数指定的集合求交运算，使得**当前集合**成为两个集合的**交**。
- 集合对象调用boolean **removeAll**(HashSet set)方法可以和参数指定的集合求差运算，使得**当前集合**成为两个集合的**差**。
- 参数指定的集合和当前集合必须是**同种类型的集合**，否则上述方法返回false。

## 7.6 HashSet<E>泛型类

- 4.HashSet<E>泛型类实现的接口
- HashSet<E>泛型类实现了泛型接口Set<E>，而 Set<E>接口是 Collection<E>接口的子接口。
- HashSet<E>类中的绝大部分方法都是接口方法的实现。
- 编程时，可以使用**接口回调技术**，即把HashSet<E>对象的引用赋值给 Collection<E>接口变量或Set<E>接口变量，那么接口就可以调用类实现的接口方法。

## 7.6 HashSet<E>泛型类

- 【例子10】

```
import java.util.*;
class Student
{
    String name;
    int score;
    Student(String name, int score)
    {
        this.name = name;
        this.score = score;
    }
}
```

```
public class Example7_10
{
    public static void main(String args[])
    {
        Student stu1 = new Student("S1",78);
        Student stu2 = new Student("S2",98);
        HashSet<Student> set = new HashSet<Student>();
        HashSet<Student> subset = new HashSet<Student>();
        set.add(stu1);
        set.add(stu2);
        subset.add(stu1);
        System.out.println("set contains subset:" + set.containsAll(subset));
        Object s[] = set.toArray();
        for(int i=0; i<s.length;i++)
        {
            System.out.printf("%s:%d\n",((Student)s[i]).name, ((Student)s[i]).score);
        }
    }
}
```

```
set contains subset:true
S2:98
S1:78
```

## 7.6 HashSet<E>泛型类

- 【例子11】

```
public class Example7_11
{
    public static void main(String args[])
    {
        Student stu1 = new Student("S1",78);
        Student stu2 = new Student("S2",98);
        HashSet<Student> set = new HashSet<Student>();
        HashSet<Student> subset = new HashSet<Student>();
        set.add(stu1);
        set.add(stu2);
        subset.add(stu1);

        HashSet<Student> tempSet = (HashSet<Student>)set.clone();
        tempSet.removeAll(subset);
        Iterator<Student> iter = tempSet.iterator();
        while(iter.hasNext())
        {
            Student temp = iter.next();
            System.out.printf("%s:%d\n",temp.name,temp.score);
        }
    }
}
```

```
import java.util.*;
class Student
{
    String name;
    int score;
    Student(String name, int score)
    {
        this.name = name;
        this.score = score;
    }
}
```

# Outline

- 7.1 Date类
- 7.2 Calendar类
- 7.3 Math类与BigInteger类
- 7.4 数字格式化
- 7.5 LinkedList<E>泛型类
- 7.6 HashSet<E>泛型类
- 7.7 HashMap<K,V>泛型类
- 7.8 TreeSet<E>泛型类
- 7.9 TreeMap<K,V>泛型类
- 7.10 Stack<E>泛型类

## 7.7 HashMap<K,V>泛型类

- HashMap<K,V>也是一个很实用的类，HashMap<K,V>对象采用**散列表**这种数据结构存储数据，习惯上称HashMap<K,V>对象为**散列映射对象**。
- 散列映射用于存储键/值数据对，允许把任何数量的键/值数据对存储在一起。
- **键（Key）不可以发生逻辑冲突**，即不要对两个数据项使用相同的键，如果出现两个数据项使用**相同的键**，那么，先前散列映射中的**键/值对将被替换**。

## 7.7 HashMap<K,V>泛型类

- 散列映射在它需要更多的存储空间时会**自动增大容量**。
  - 例如，如果散列映射的**装载因子**是0.75，那么当散列映射的容量被使用了75%时，它就把容量增加到原始容量的 2 倍。
  - 对于**数组**和**链表**这两种数据结构，如果要**查找**它们存储的某个特定的元素却不知道它的位置，就需要从头开始访问元素直到找到匹配的为止；如果数据结构中包含很多的元素，就会浪费时间。
  - 这时最好使用**散列映射**来存储要查找的数据，使用散列映射可以**减少检索的开销**。



## 7.7 HashMap<K,V>泛型类

- 1.HashMap<K,V>对象
- HashMap<K,V>泛型类创建的对象称作散列映射，例如：

```
HashMap<String, Student> hashtable = new HashMap<String, Student>();
```

- 那么，hashtable就可以存储"键/值"对数据，其中的键必须是一个String对象，键对应的值必须是Student对象。hashtable可以调用public V put(K key, V value)将键/值对数据存放到散列映射中，该方法同时返回键所对应的值。

## 7.7 HashMap<K,V>泛型类

- 2.常用方法
  - `public void clear()`: **清空**散列映射。
  - `public Object clone()`: 返回当前散列映射的一个**克隆**。
  - `public boolean containsKey(Object key)`: 如果散列映射有键/值对**使用了**参数指定的键, 方法返回`true`, 否则返回`false`。
  - `public boolean containsValue(Object value)`: 如果散列映射有键/值对的值是参数指定的**值**, 方法返回`true`, 否则返回`false`。
  - `public V get(Object key)`: 返回散列映射中使用`key`做键的键/值对中的值。
  - `public boolean isEmpty()`: 如果散列映射不含任何键/值对, 方法返回`true`, 否则返回`false`。
  - `public V remove(Object key)`: **删除**散列映射中键为参数指定的键/值对, 并返回键对应的值。
  - `public int size()`: 返回散列映射的**大小**, 即键/值对的数目。

## 7.7 HashMap<K,V>泛型类

- 3.遍历散列映射
- 如果想获得散列映射中所有键/值对中的值，首先使用

```
public Collection<V> values()
```

- 该方法返回一个实现Collection<V>接口的类创建的对象引用，并要求将该对象的引用返回到Collection<V>接口变量中。
- values()方法返回的对象中存储了散列映射中所有"键/值"对中的"值"，这样接口变量就可以调用类实现的方法，比如获取Iterator对象，然后输出所有的值。
- 见例子12（下一页）


## 7.7 HashMap<K,V>泛型类

- 【例子12】

```
import java.util.*;

public class Example7_12
{
    public static void main(String args[])
    {
        HashMap<String, Integer> map = new HashMap<String, Integer>();
        map.put("a", 1);
        map.put("b", 2);

        Collection<Integer> collection = map.values();
        Iterator<Integer> iter = collection.iterator();
        while(iter.hasNext())
        {
            Integer temp = iter.next();
            System.out.println(temp.toString());
        }
    }
}
```



## 7.7 HashMap<K,V>泛型类

- 4.HashMap<K,V>泛型类实现的接口
- HashMap<K,V>泛型类实现了泛型接口Map<K,V>，HashMap<K,V>类中的绝大部分方法都是Map<K,V>接口方法的实现。
- 编程时，可以使用**接口回调技术**，即把HashMap<K,V>对象的引用赋值给Map<K,V>接口变量，那么接口就可以调用类实现的接口方法。

# Outline

- 7.1 Date类
- 7.2 Calendar类
- 7.3 Math类与BigInteger类
- 7.4 数字格式化
- 7.5 LinkedList<E>泛型类
- 7.6 HashSet<E>泛型类
- 7.7 HashMap<K,V>泛型类
- 7.8 TreeSet<E>泛型类
- 7.9 TreeMap<K, V>泛型类
- 7.10 Stack<E>泛型类

## 7.8 TreeSet<E>泛型类

- TreeSet<E>类是实现Set接口的类，它的大部分方法都是接口方法的实现。TreeSet<E>泛型类创建的对象称作**树集**，例如

```
TreeSet<Student> tree = new TreeSet<Student>();
```

- 那么，tree就是一个可以存储Student类型数据的集合，tree可以调用add()方法将Student类型的数据添加到树集中，存放到树集中的对象**按对象的字符串**表示**升序排列**。

## 7.8 TreeSet<E>泛型类

- TreeSet<E>类的常用方法
  - `public boolean add(E o)`: 向树集**添加**对象，添加成功返回`true`，否则返回`false`。
  - `public void clear()`: **清空**树集中的所有对象。
  - `public void contains(Object o)`: 如果**包含**对象`o`，方法返回`true`，否则返回`false`。
  - `public E first()`: 返回树集中的**第一个对象**（**最小**的对象）。
  - `public E last()`: 返回**最后一个对象**（**最大**的对象）。
  - `public isEmpty()`: 判断是否是**空树集**，如果树集不含对象返回`true`。
  - `public boolean remove(Object o)`: **删除**树集中的对象`o`。
  - `public int size()`: 返回树集中对象的**数目**。



## 7.8 TreeSet<E>泛型类

- 对象调用toString()方法就可以获得自己的字符串表示。
- 但很多对象不适合按照字符串排列大小。
- 我们在创建树集时可自己规定树集中的对象按着什么样的"大小"顺序排列。

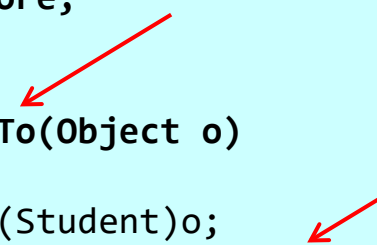
## 7.8 TreeSet<E>泛型类

- 【例子13】

```
public class Example7_13
{
    public static void main(String args[])
    {
        TreeSet<Student> mytree = new TreeSet<Student>();
        Student stu1 = new Student("S1",78);
        Student stu2 = new Student("S2",98);
        mytree.add(stu1);
        mytree.add(stu2);
        Iterator<Student> iter = mytree.iterator();
        while(iter.hasNext())
        {
            Student temp = iter.next();
            System.out.printf("%s:%d\n",temp.name,temp.score);
        }
    }
}
```

```
import java.util.*;
class Student implements Comparable
{
    String name;
    int score;
    Student(String name, int score)
    {
        this.name = name;
        this.score = score;
    }

    public int compareTo(Object o)
    {
        Student stu = (Student)o;
        return (this.score - stu.score);
    }
}
```



```
S1:78
S2:98
```

## 7.8 TreeSet<E>泛型类

- 注：树集中不容许出现大小相等的两个节点，例如，在上述例子中如果再添加语句：

```
Student stu3 = new Student("S3",98);  
mytree.add(stu3);
```

- 是无效的。**如果允许成绩相同**，可把上述例子中Student类中的compareTo方法更改为：

```
public int compareTo(Object o)  
{  
    Student stu = (Student)o;  
    if(this.score==stu.score)  
        return 1;  
    else  
        return (this.score - stu.score);  
}
```

## 7.8 TreeSet<E>泛型类

- Comparator接口
- Comparator是java.util包中的一个接口，compare(Object o1,Object o2)是接口中的方法。

## 7.8 TreeSet<E>泛型类

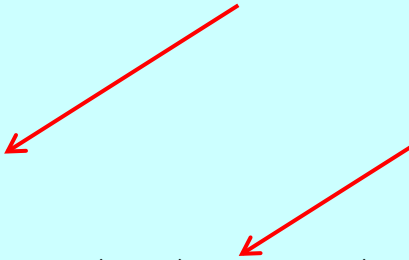
- 【例子1/2】

```
import java.util.*;

class Student
{
    String name;
    int score;
    Student(String name, int score)
    {
        this.name = name;
        this.score = score;
    }
}
```

```
class StudentComparator implements Comparator
{
    String name;
    int score;

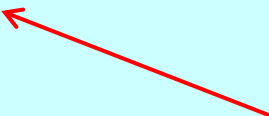
    public int compare(Object o1, Object o2)
    {
        return ( ((Student)o1).score - ((Student)o2).score );
    }
}
```



## 7.8 TreeSet<E>泛型类

- 【例子2/2】

```
public class ExampleComparator
{
    public static void main(String args[])
    {
        Student [] students = new Student[]{new Student("S1",78), new
Student("S2",98)};
        Arrays.sort(students, new StudentComparator());
        for(int i=0; i<students.length; i++)
        {
            Student temp = students[i];
            System.out.printf("%s:%d\n",temp.name,temp.score);
        }
    }
}
```

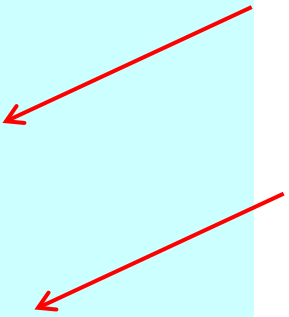


```
S1:78
S2:98
```

## 7.8 TreeSet<E>泛型类

- 【例子1/2】`import java.util.*;`

```
class Student implements Comparable
{
    String name;
    int score;
    Student(String name, int score)
    {
        this.name = name;
        this.score = score;
    }
    public int compareTo(Object o)
    {
        Student stu = (Student)o;
        return (this.score - stu.score);
    }
}
```



## 7.8 TreeSet<E>泛型类

- 【例子2/2】

```
public class ExampleComparable
{
    public static void main(String args[])
    {
        Student [] students = new Student[]{new
Student("S1",78), new Student("S2",98)};
        Arrays.sort(students);
        for(int i=0; i<students.length; i++)
        {
            Student temp = students[i];
            System.out.printf("%s:%d\n",temp.name,temp.score);
        }
    }
}
```

```
S1:78
S2:98
```



## 7.8 TreeSet<E>泛型类

- Comparator与Comparable接口的区别
- 一个类实现了Comparable接口，表明这个类的对象之间是**可以相互比较的**（i.e., it is comparable），这个类对象组成的集合就可以直接使用sort方法排序。
- Comparator可以看成一种算法的实现，**将算法和数据分离**，Comparator也可以在下面两种环境下使用：
  - 类的设计师没有考虑到比较问题而没有实现Comparable，可以通过Comparator来实现排序而不必改变对象本身
  - 可以使用多种排序标准，比如升序、降序等

# Outline

- 7.1 Date类
- 7.2 Calendar类
- 7.3 Math类与BigInteger类
- 7.4 数字格式化
- 7.5 LinkedList<E>泛型类
- 7.6 HashSet<E>泛型类
- 7.7 HashMap<K,V>泛型类
- 7.8 TreeSet<E>泛型类
- 7.9 TreeMap<K,V>泛型类
- 7.10 Stack<E>泛型类

## 7.9 TreeMap<K, V>泛型类

- TreeMap类实现了Map接口。TreeMap提供了按排序顺序存储"关键字/值"对的有效手段。
- 应该注意的是，不像散列映射（HashMap），树映射（TreeMap）保证它的元素按照**关键字升序排列**。下面是TreeMap构造方法：

```
TreeMap<K,V>()  
TreeMap<K,V>(Comparator<K> comp)
```

- 第一种形式构造的树映射，**按关键字的大小**顺序来排序树映射中的"键/值"对，键的大小顺序是按其**字符串表示的字典顺序**。
- 第二种形式构造的树映射，键的大小顺序**按comp接口规定的大小顺序**来排序树映射中的"键/值"对。

## 7.9 TreeMap<K, V>泛型类

- TreeMap类的常用方法与HashMap<K,V>类相似。

## 7.9 TreeMap<K, V>泛型类

- 【例子1/2】

```
class MyKey implements Comparable
{
    int number=0;
    MyKey(int number)
    {
        this.number=number;
    }
    public int compareTo(Object o)
    {
        MyKey mykey = (MyKey)o;
        if(this.number == mykey.number)
            return 1;
        else
            return (this.number - mykey.number);
    }
}
```

```
import java.util.*;
class Student
{
    String name = null;
    int height, weight;
    Student(int w, int h, String name)
    {
        weight=w;
        height=h;
        this.name=name;
    }
}
```

## 7.9 TreeMap<K, V>泛型类

- 【例子2/2】

```
public class Example7_14
{
    public static void main(String args[])
    {
        Student s1 = new Student(65,177,"Zhang"), s2 = new
Student(85,168,"Li");
        TreeMap<MyKey,Student> treemap = new TreeMap<MyKey,Student>();
        treemap.put(new MyKey(s1.weight),s1);
        treemap.put(new MyKey(s2.weight),s2);
        Collection<Student> collection = treemap.values();
        Iterator<Student> iter = collection.iterator();
        while(iter.hasNext())
        {
            Student te=iter.next();
            System.out.printf("%s,%d(kg)\n",te.name,te.weight);
        }
    }
}
```

```
Zhang,65(kg)
Li,85(kg)
```

# Outline

- 7.1 Date类
- 7.2 Calendar类
- 7.3 Math类与BigInteger类
- 7.4 数字格式化
- 7.5 LinkedList<E>泛型类
- 7.6 HashSet<E>泛型类
- 7.7 HashMap<K,V>泛型类
- 7.8 TreeSet<E>泛型类
- 7.9 TreeMap<K,V>泛型类
- 7.10 Stack<E>泛型类

## 7.10 Stack<E>泛型类

- 栈是一种“**后进先出**”的数据结构，只能在一端进行输入或输出数据的操作。栈把第一个放入该栈的数据放在最底下，而把后续放入的数据放在已有数据的顶上。
- 向栈中输入数据的操作称为“压栈”，从栈中输出数据的操作称为“弹栈”。由于栈总是在顶端进行数据的输入输出操作，所以弹栈总是输出（删除）最后压入堆栈中的数据，这就是“**后进先出**”的来历。



## 7.10 Stack<E>泛型类

- 使用java.util包中的Stack类创建一个堆栈对象
- 常用方法：
  - public E push(E item): 压栈
  - public E pop(): 弹栈
  - public boolean empty(): 判断栈是否还有数据
  - public E peek(): 获取栈顶端的数据，**但不删除该数据**
  - public int search(Object data): 获取数据在栈中的位置，最顶端的位置是 1 ，向下依次增加，如果栈不含此数据，则返回-1

## 7.10 Stack<E>泛型类

- 栈是很灵活的数据结构，使用栈可以节省内存的开销。
- 比如，**递归**是一种很消耗内存的算法，我们可以借助栈消除大部分递归，达到和递归算法同样的目的。
- 斐波那契整数序列（**Fibonacci sequence**）是我们熟悉的一个递归序列，它的第n项是前两项的和，第一项和第二项都是 1 。

## 7.10 Stack<E>泛型类

- 【例子15】

```
import java.util.*;
public class Example7_15
{
    public static void main(String args[])
    {
        Stack<Integer> stack=new Stack<Integer>();
        stack.push(new Integer(1));
        stack.push(new Integer(1));
        int k=1;
        while(k<=5)
        {
            Integer F1 = stack.pop();
            int f1 = F1.intValue();
            Integer F2 = stack.pop();
            int f2 = F2.intValue();
            Integer temp = new Integer(f1+f2);
            System.out.println(temp.toString());
            stack.push(temp);
            stack.push(F2);

            k++;
        }
    }
}
```

2  
3  
5  
8  
13

# Outline

- 7.1 Date类
- 7.2 Calendar类
- 7.3 Math类与BigInteger类
- 7.4 数字格式化
- 7.5 LinkedList<E>泛型类
- 7.6 HashSet<E>泛型类
- 7.7 HashMap<K,V>泛型类
- 7.8 TreeSet<E>泛型类
- 7.9 TreeMap<K,V>泛型类
- 7.10 Stack<E>泛型类



Data Structures

# 简要小节

- Queue
  - Stack
- List
  - ArrayList
  - LinkedList
- Set
  - HashSet
  - LinkedHashSet 根据insertion order
  - TreeSet 排序的Set
- Map
  - HashMap
  - LinkedHashMap 根据insertion order
  - TreeMap 排序的Map