



内容提要

第5章 常用Java类库

Java API 简介

Object类

语言包

实用包(java.util)

文本包(java.text)



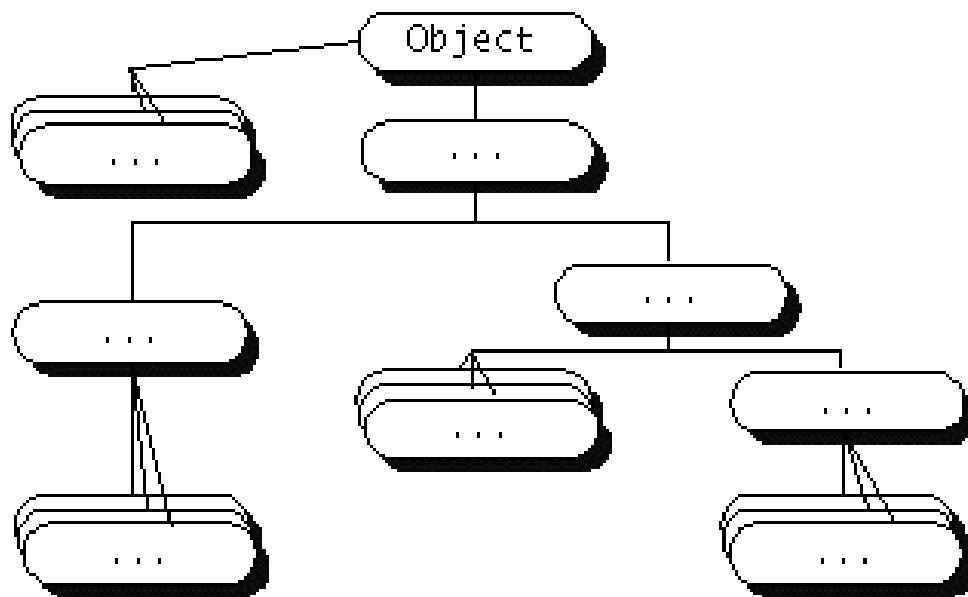
Java API 简介

- Java提供了用于语言开发的类库，称为Java基础类库(JFC, Java Foundational Class)，也称应用程序编程接口(API, Application Programming Interface)，分别放在不同的包中
- Java提供的包主要有
[*java.lang*](#), [*java.util*](#), [*java.text*](#), [*java.io*](#)
[*java.applet*](#), [*java.awt*](#), [*java.awt.datatransfer*](#)
[*java.awt.event*](#), [*java.awt.image*](#), [*javax.swing*](#)
[*java.net*](#), [*java.sql*](#), [*java.beans*](#), [*java.rmi*](#), [*java.security*](#)
，等



Object类

- 所在包：java.lang
- Java程序中所有类的直接或间接父类，类库中所有类的父类，处在类层次最高点
- 包含了所有Java类的公共属性，其构造方法是Object()





Object类：主要方法

- Object类定义了所有对象必须具有的状态和行为，较主要的方法如下
 - public final Class `getClass()`
获取当前对象所属的类信息，返回Class对象
 - public String `toString()`
返回当前对象本身的有关信息，按字符串对象返回
 - public boolean `equals(Object obj)`
比较两个对象是否是同一对象，是则返回true
 - protected Object `clone()`
生成当前对象的一个拷贝，并返回这个复制对象
 - public int `hashCode()` : 返回该对象的哈希代码值
 - protected void `finalize()` throws Throwable
定义回收当前对象时所需完成的资源释放工作
- 你的类不可以覆盖终结方法，即有final修饰的方法



Object类： 主要方法

● 相等和同一的概念

- 两个对象具有相同的类型，及相同的属性值，则称二者**相等**(equal)
- 如果两个引用变量指向的是同一个对象，则称这两个变量(对象)**同一**(identical)
- 两个变量同一，则其所引用的对象肯定相等
- 两个对象相等，但并不一定同一
- 比较运算符“==”判断的是这两个对象是否同一



Object类：主要方法

● equals 方法

- 由于Object是类层次结构中的树根节点，因此所有其他类都继承了equals()方法
- Object类中的 equals() 方法的定义如下，可见，也是判断两个对象是否同一

```
public boolean equals(Object x) {  
    return this == x;  
}
```



Object类：主要方法

- 判断两个对象是否同一

//一般对象的==操作是判断同一

```
Circle cir1=new Circle(10), cir2=new Circle(10);
```

```
____.println(cir1==cir2); // false, 引用的不是同一个对象
```

```
____.println(cir1.equals(cir2)); // false
```

```
cir2=cir1; // 引用同一个对象
```

```
____.println(cir==cir2); // true
```

```
____.println(cir.equals(cir2)); // true
```



Object类：主要方法

● equals方法的重写

- 要判断两个对象各个属性域的值是否相同，则不能使用从Object类继承来的equals方法，而需要在类声明中对equals方法进行重写
- String类中已经重写了Object类的equals方法，可以判别两个字符串是否内容相同

String (Java Platform SE 8) ☒

equals

```
public boolean equals(Object anObject)
```

Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

Overrides:

equals in class Object

Parameters:

anObject - The object to compare this String against

Returns:

true if the given object represents a String equivalent to this string, false otherwise



Object类： 主要方法

● 判断两个对象是否同一

//常量池内字符串和基本类型数据的==操作是判断相等

String **a**="abc", **b**="abc", **c**=new String("abc");

____.println(a.equals(b)); // true, a与b内容相等

____.println(a==b); // true, a与b引用了常量池内的同一串字符

____.println(a.equals(c)); // true, a与c内容也相等

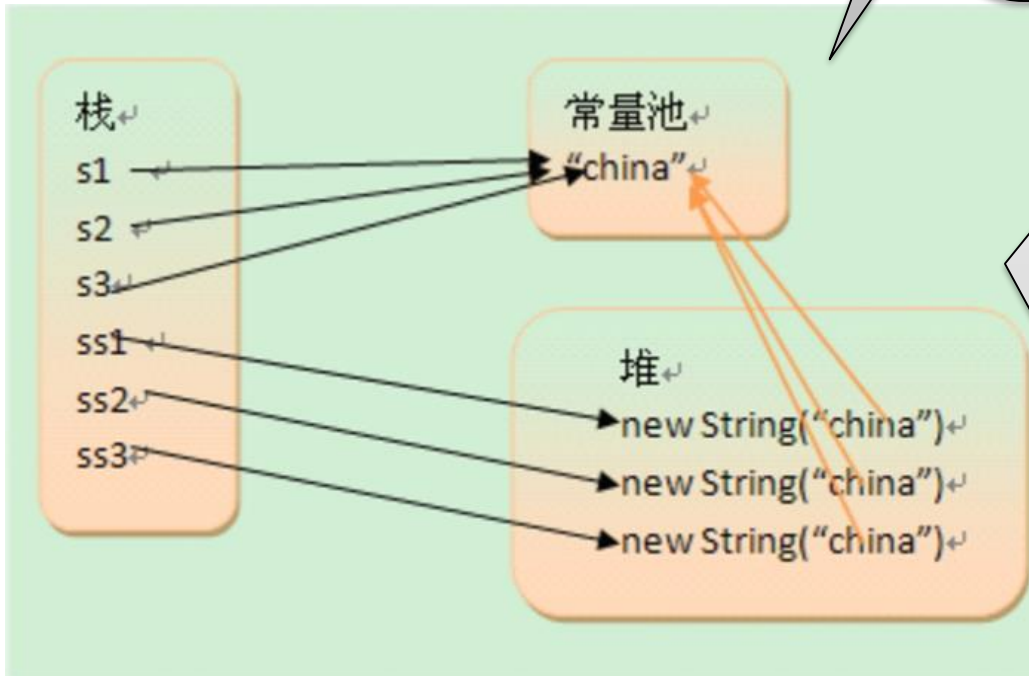
____.println(a==c); // false, a与c分别引用了不同的对象

判断两个字符串的内容是否相同，请使用equals方法。
运算符==用于判断是否同一。

```
String s1 = "china";  
String s2 = "china";  
String s3 = "china";
```

```
String ss1 = new String("china");  
String ss2 = new String("china");  
String ss3 = new String("china");
```

常量池指的是在编译期被确定，并被保存在已编译的.class文件中的一些数据。包含代码中所定义的各种基本类型（如int）和对象型（如String及数组）的常量值(final)。



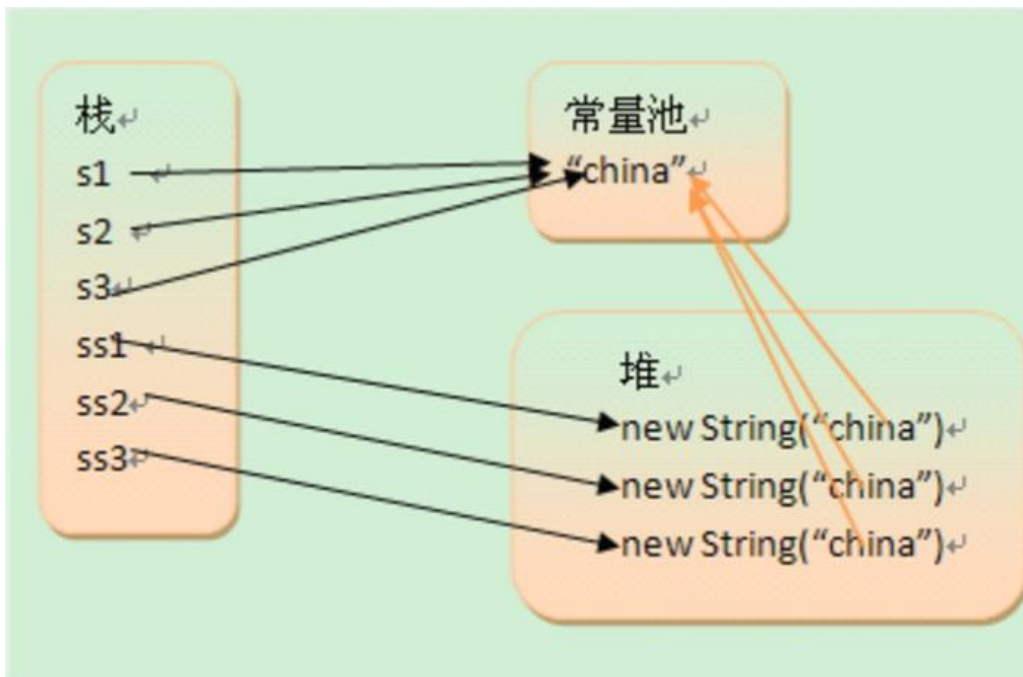
字符串是一个特殊包装类，其引用是存放在栈里的，而对象内容必须根据创建方式不同确定(常量池和堆)。

- 编译期就已经创建好，存放在字符串常量池中，
- 运行时创建（使用new）的，则存放在堆中。

```
String s1 = "china";  
String s2 = "china";  
String s3 = "china";
```

```
String ss1 = new String("china");  
String ss2 = new String("china");  
String ss3 = new String("china");
```

s1, s2, s3 引用同一个字符串字面量（位于常量池）对象，因而其引用的对象的标识相同



名称	值
> s1	"china" (标识=31)
> s2	"china" (标识=31)
> s3	"china" (标识=31)
> ss1	"china" (标识=32)
> ss2	"china" (标识=33)
> ss3	"china" (标识=34)

ss1, ss2, ss3 引用相等的字符串对象，但是它们位于堆上的不同位置，因而具有不同的标识



Object类：主要方法

- equals方法的重写

```
public boolean equals(Object x) {
```

```
//首先，判断两个对象的类型是否相同
```

```
if (this.getClass() != x.getClass())  
    return false;
```

```
//然后，强制类型转换（塑型）
```

```
Circle b = (Circle) x; // x是根类Object的引用
```

```
//最后，判断两个对象各项内容是否全部相同
```

```
return this.getRadius()==b.getRadius();
```

```
}
```

注意：一般
对象，==还
是判断同一！



Object类：主要方法

● clone方法

- 根据已存在的对象构造一个新的对象
- 在根类中被定义为protected，所以需要覆盖为public
- 实现Cloneable 接口，赋予一个对象被克隆的能力

public interface Cloneable

- Cloneable 是一个空接口
- 一个类实现了Cloneable 接口，以指示Object.clone()方法可以合法地对该类实例进行按字段复制
- 在没有实现Cloneable接口的实例上调用根类的clone方法，则会抛出CloneNotSupportedException异常
- 注意，此接口不包含clone方法。若某个类覆盖了根类中的Clone方法而没有实现Cloneable接口，则其对象不具有克隆能力



Object类： 主要方法

- 覆盖clone方法，赋予一个对象被克隆的能力
public class Circle implements Shape2D, Cloneable{
.....
public Object Clone() //覆盖clone方法
throws CloneNotSupportedException{
return this.clone();
}
}

// 测试代码

```
Circle cir1=new Circle(20),cir2=new Circle(50);  
System.out.println(cir1.equals(cir2)); // false  
try { cir1=(Circle)(cir2.Clone()); }  
catch (CloneNotSupportedException e) {e.printStackTrace();}  
System.out.println(cir1.equals(cir2)); // true
```



Object类： 主要方法

● finalize 方法

- 在对象被垃圾回收器回收之前，系统自动调用对象的 finalize 方法
- 在根类Object 中被定义为protected，所以需要覆盖为 public
- 在对象回收之前需要释放数据库连接、网络连接、IO流等非系统资源时，可以考虑覆盖finalize方法
- 如果要覆盖finalize方法，覆盖方法的最后必须调用 super.finalize



Object类： 主要方法

● getClass 方法

- final方法，返回一个Class对象，用来代表对象隶属的类
- 通过Class对象，可以查询Class对象的各种信息，比如：
 - ◆ 它的名字(getName()方法)
 - ◆ 它的基类(getSuperclass()方法)
 - ◆ 它所实现接口的名字(getInterfaces()方法，返回数组)等。

```
void PrintClassInfo(Object obj) {  
    System.out.println("The Object's class is " +  
        obj.getClass().getName());  
    System.out.println("The Object's super class is " +  
        obj.getClass().getSuperclass().getName());  
}
```




Object类： 主要方法

- notify、notifyAll、wait方法
 - final方法，不能覆盖
 - 这三个方法主要用在多线程程序中，用于线程唤醒或等待



语言包(java.lang)

- 语言包java.lang提供了Java语言最基础的类，包括
 - Object类
 - 数据类型包装类(the Data Type Wrapper)
 - 字符串类(String、StringBuffer)
 - 数学类(Math)
 - 系统和运行时类(System、Runtime)
 - 类操作类(Class, ClassLoader)



语言包：数据类型包装类

- Java的每一个基本数据类型(primitive data type)都有一个数据包装类，且都为final类型，不能派生

数据类型包装类	基本数据类型
Boolean	boolean
Byte	byte
Character	char
Short	short
Integer	int
Long	long
Float	float
Double	double



语言包：数据类型包装类

- 生成数据类型包括类对象的方法
 - 从基本数据类型的变量或常量生成包装类对象
`double x = 1.2;`
`Double a = new Double(x);`
`Double b = new Double(-5.25);`
 - 从字符串生成包装类对象
`Double c = new Double("-2.34");`
`Integer i = new Integer("1234");`
 - 已知字符串，可使用valueOf方法（静态）将其转换成包装类对象：
`Integer.valueOf("125");` //静态方法，返回Integer对象
`Double.valueOf("5.15");` //静态方法，返回Double对象
 - 自动装箱
`Integer i = 3; Double d = -5.25`



语言包：数据类型包装类

● 得到基本数据类型数据的方法

- 每一个包裹类都提供相应的方法将包裹类对象转换回基本数据类型的数据

`anIntegerObject.intValue()` // 返回 `int` 类

`aCharacterObject.charValue()` // 返回 `char` 类型的数据

- `Integer`、`Float`、`Double`、`Long`、`Byte` 及 `Short` 类提供了特殊的方法能够将字符串类型的对象直接转换成对应的 `int`、`float`、`double`、`long`、`byte` 或 `short` 类型的数据

`Integer.parseInt("234")` // 静态方法，返回 `int` 类型的数据

`Float.parseFloat("234.78")` // 静态方法，返回 `float` 类型的数据

- 自动拆箱

`Integer a = new Integer(3);`

`int i = a;`



语言包：常量字符串String类

- String字符串的值和长度都不可变（**immutable**），称为**常量字符串**。
- String类内部结构

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for the string */
    private int hash; // Default to 0

    /** use serialVersionUID from JDK 1.0.2 for interoperability */
    private static final long serialVersionUID = -6849794470754667710L;
```

属性value为不可变数组，存储String字符串中的所有字符。定义多少，字符数组的长度就是多少，不存在字符数组扩容一说。



语言包：常量字符串String类

- 生成String类对象的方法
 - 可以这样生成一个常量字符串
`String aString;`
`aString = "This is a string"`
 - 调用构造方法生成字符串对象
`new String();`
`new String(String value);`
`new String(char[] value);`
`new String(char[] value, int offset, int count);`
`new String(StringBuffer buffer);`



语言包：常量字符串String类

● String类常用方法（1）

名称	解释
int length ()	返回字符串中字符的个数
char charAt (int index)	返回序号 index 处的字符
int indexOf (String s)	在接收者字符串中进行查找，如果包含子字符串 s ，则返回匹配的第一个字符的位置序号，否则返回-1
String substring (int begin, int end)	返回接收者对象中序号从 begin 开始到 end-1 的子字符串
String[] split (String regex) String[] split (String regex, int limit)	以指定字符为分隔符，分解字符串
String concat (String s)	返回接收者字符串与参数字符串 s 进行连接后的字符串



语言包：常量字符串String类

● String类常用方法（2）

名称	解释
String replace (char oldChar, char newChar);	将接收者字符串的oldChar替换为newChar
int compareTo (String s);	将接收者对象与参数对象进行比较
boolean equals (String s);	接收者对象与参数对象的值进行比较
String trim ();	将接收者字符串两端的空字符串都去掉
String toLowerCase ()	将接收者字符串中的字符都转为小写
String toUpperCase ()	将接收者字符串中的字符都转为大写
static String format (String format, Object... args)	使用指定的格式字符串和参数返回一个格式化字符串



语言包：常量字符串String类

- 注意：substring、replace、concat、trim、toLowerCase、toUpperCase、split等方法返回的是一个新String对象或对象数组。

例如：

```
public String substring(int beginIndex) {  
    if (beginIndex < 0) {  
        throw new StringIndexOutOfBoundsException(beginIndex);  
    }  
    int subLen = value.length - beginIndex;  
    if (subLen < 0) {  
        throw new StringIndexOutOfBoundsException(subLen);  
    }  
    return (beginIndex == 0) ? this : new String(value, beginIndex, subLen);  
}  
.....
```



语言包：常量字符串String类

- 使用正则表达式拆分和替换字符串

```
String s="AA,BB CC:DD";
```

```
String[] words=s.split("[, :]"); //使用正则表达式拆分字符串
```

```
for(String word:words)
```

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

```
String t=s.replaceAll("[,:]", " "); //使用正则表达式将标点替换成空格
```

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

```
AA
```

```
BB
```

```
CC
```

```
DD
```

```
AA BB CC DD
```



语言包：常量字符串String类

- 使用正则表达式包`java.util.regex`匹配字符串
 - **Pattern对象**：表示正则表达式编译对象，由Pattern类的静态方法`compile()`产生
 - 静态的布尔方法`matches()`可直接用于字符串形式的模式去匹配目标字符串。
 - **Matcher对象**：匹配器，由Pattern的`matcher()`方法产生
 - 布尔方法`matches()`用于从开头进行匹配，成功则返回`true`。
 - 布尔方法`find()`可用于循环搜索匹配的子串，成功则返回`true`，再次调用则继续搜索，类似于Python RE模块的`search`方法。
 - `group()`方法可以得到匹配的项；如果分组，则通过`group(1)`, `group(2)`, ..., `group(n)`得到各分组匹配结果。



语言包：常量字符串String类

- 使用正则表达式包java.util.regex匹配字符串

//匹配6位邮政编码

```
Pattern p=Pattern.compile("[1-9]\\d{5}$"); //正则表达式的编译对象
```

```
Matcher m=p.matcher("266580"); //生成匹配器
```

```
System.out.println(m.matches()); //用匹配器匹配
```

```
m=p.matcher("2665801"); //匹配器
```

```
System.out.println(m.matches()); //匹配
```

//或者

```
System.out.println(Pattern.matches("[1-9]\\d{5}$","266580")); //匹配
```

```
true  
false  
true
```



语言包：常量字符串String类

- 使用正则表达式包java.util.regex匹配字符串

// 匹配以th打头的单词，不区分大小写

```
words=new String[]{"the","Thank","you","That","tree",  
"tHis","Th_","Th-"};
```

```
Vector<String> v=new Vector<String>();//向量，用于存放匹配的单词
```

```
p=Pattern.compile("^th\\w*", Pattern.CASE_INSENSITIVE);
```

```
for(String word:words){
```

```
    m=p.matcher(word);
```

```
    if(m.matches())
```

```
        v.add(word);
```

```
}
```

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

[the, Thank, That, tHis, Th_]



语言包：常量字符串String类

- 使用正则表达式包java.util.regex匹配字符串

//使用**find**方法循环搜索所有匹配的子串,类似于Python RE模块中的**search**方法

```
p=Pattern.compile("\\w*ain\\b");//
```

```
m=p.matcher("No pain no gain");
```

```
System.out.println(m.matches()); // false, 开头匹配不上就失败
```

```
while(m.find()){ // 循环匹配搜索（搜索子串，下一次调用则继续搜索）
```

```
    System.out.println(m.group());
```

```
}
```

```
false  
pain  
gain
```



语言包：常量字符串String类

- 使用正则表达式包java.util.regex匹配字符串

// 匹配分组，提取格式化子串

p=Pattern.compile(".*E-mail:(.*?), Tel:(\\d*)."); //正则表达式编译对象

m=p.matcher("联系方式： E-mail:ruanzl@upc.edu.cn, Tel:18500001ABC");

if(m.matches()){ // 匹配

System.out.println(m.group()); // 返回由以前匹配操作所匹配的输入子序列。

System.out.println(m.group(0)); // 同上

System.out.println(m.group(1)); // 输出第1个分组

System.out.println(m.group(2)); // 输出第2个分组

}

联系方式： E-mail:ruanzl@upc.edu.cn, Tel:18500001ABC

联系方式： E-mail:ruanzl@upc.edu.cn, Tel:18500001ABC

ruanzl@upc.edu.cn

18500001



语言包：可变字符串StringBuffer类

第5章 常用Java类库

- 可变 (**mutable**) 字符串类StringBuffer，其对象是可以修改的字符串
 - 字符的个数称为对象的长度(length)
 - 分配的存储空间称为对象的容量(capacity)
- 只要字符串缓冲区所包含的字符序列的长度没有超出此容量，就无需分配新的内部缓冲区数组。如果内部缓冲区溢出（不够用时），则**此容量自动增大**。



语言包：可变字符串StringBuffer类

● StringBuffer类内部结构

```
public final class StringBuffer extends AbstractStringBuilder
    implements java.io.Serializable, CharSequence
{
    private transient char[] toStringCache;
    static final long serialVersionUID = 3388685877147921107L;

    public StringBuffer() { super(16); }

    public StringBuffer(int capacity) { super(capacity); }

    public StringBuffer(String str) {
        super(str.length() + 16);
        append(str);
    }

    public StringBuffer(CharSequence seq) {
        this(seq.length() + 16);
        append(seq);
    }
}
```

为什么要用transient关键字？
在持久化对象时，对于一些特殊的数据成员（如用户的密码，银行卡号等），我们不想用序列化机制来保存它。为了在一个特定对象的一个成员变量上关闭序列化，可以在这个成员变量前加上关键字transient。

.....



语言包：可变字符串StringBuffer类

● StringBuffer类内部结构

```
abstract class AbstractStringBuilder implements Appendable, CharSequence  
{
```

```
    /**  
     * The value is used for character storage.  
     */  
    char[] value;  
    /**  
     * The count is the number of characters used.  
     */  
    int count;
```

```
    AbstractStringBuilder() {  
    }
```

```
    AbstractStringBuilder(int capacity) {  
        value = new char[capacity];  
    }
```

```
    .....
```

- StringBuffer的count和value 是包级别的成员变量，因此它只能在StringBuffer所属的包（java.lang）内被访问，因此也就成了内部变量
- 例如，aBuffer.count 是不可见



语言包：可变字符串StringBuffer类

第5章 常用Java类库

```
public final class StringBuffer extends AbstractStringBuilder .....
{
    .....
    @Override
    public synchronized StringBuffer append(String str) {
        toStringCache = null;
        super.append(str);
        return this;
    }
    .....
}
```

```
abstract class AbstractStringBuilder implements Appendable, CharSequence
{
    .....
    public AbstractStringBuilder append(String str) {
        if (str == null)
            return appendNull();
        int len = str.length();
        ensureCapacityInternal(count + len);
        str.getChars(0, len, value, count); // 拷贝str中的字符到value数组内
        count += len;
        return this;
    }
    .....
}
```



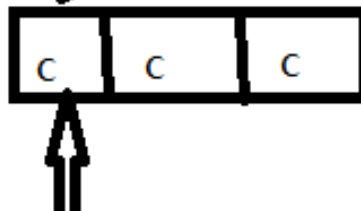
语言包：可变字符串StringBuffer类

```
StringBuffer sb = new StringBuffer();  
String s = null; //没有分配空间  
String s2 = "ccc";  
sb.append("aaa");  
sb.append(s);  
sb.append(s2);  
sb.append("bbb");  
System.out.println(sb); //aaanullcccbbb
```

StringBuffer类，可变字符数组，初始容量16



String类型，不可变字符串





语言包：可变字符串StringBuffer类

- 与String类的对象相比，执行效率要低一些
- 该类的方法不能被用于String类的对象
- 生成StringBuffer类的对象
 - `new StringBuffer();`
生成容量为16的空字符串对象
 - `new StringBuffer(int size);`
生成容量为size的空字符串对象
 - `new StringBuffer(String aString);`
生成aString的一个备份，容量为其长度 +16



语言包：可变字符串StringBuffer类

● StringBuffer类常用方法（1）

名称	解释
int length ()	返回字符串对象的长度
int capacity ()	返回字符串对象的容量
void ensureCapacity (int size)	设置字符串对象的容量
void setLength (int len)	设置字符串对象的长度。如果len的值小于当前字符串的长度，则尾部被截掉
char charAt (int index)	返回index处的字符
void setCharAt (int index, char c)	将index处的字符设置为c



语言包：可变字符串StringBuffer类

● StringBuffer类常用方法（2）

名称	解释
void getChars(int start, int end, char [] charArray, int newStart)	将接收者对象中从 start 位置到 end-1 位置的字符拷贝到字符数组 charArray 中，从位置 newStart 开始存放
StringBuffer reverse()	返回将接收者字符串逆转后的字符串
StringBuffer insert(int index, Object ob)	将 ob 插入到 index 位置
StringBuffer append(Object ob)	将 ob 连接到接收者字符串的末尾， ob 可以是 char/int/double 等基本数据类型、 String 类型/ StringBuffer 类型
String toString()	返回一个包含StringBuffer中字符序列的 String 对象。



语言包：可变字符串StringBuffer类

- StringBuffer类示例：已知一个字符串，返回将字符串中的非字母字符都删除后的字符串

```
public class StringEditor {  
    public static String removeNonLetters(String original) {  
        StringBuffer aBuffer = new StringBuffer(original.length());  
        char ch;  
        for (int i=0; i<original.length(); i++) {  
            ch = original.charAt(i);  
            if (Character.isLetter(ch))  
                aBuffer.append(ch);  
        }  
        return new String(aBuffer);  
    }  
}
```

也可以不指定长度


String original =
"Hello123, My Name is
Mark, 234I think you
are my classmate?!!"



HelloMyNameisMarkIt
hinkyouaremyclassmate

if(ch>=65 && ch<=122) // 'a'~'Z'

第5章 常用Java类库



(x)= 变量 断点

名称	值
> original	"Hello123, My Name is Mark,234I think..."
▼ aBuffer	StringBuffer (标识=22)
▲ count	4
■ toStringCache	null
▼ ▲ value	(标识=26)
▲ [0]	H
▲ [1]	e
▲ [2]	l
▲ [3]	l
▲ [4]	
▲ [5]	
▲ [6]	
▲ [7]	
▲ [8]	
▲ [9]	
▲ [10]	
▲ [11]	
▲ [12]	
▲ [13]	
▲ [14]	
▲ [15]	
ch	l
i	3

[H, e, l, l, □, □, □, □, □, □, □, □, □, □, □, □]

- 容量不够用时，内部的value数组也会重新分配内存，并复制原有内容，但是value数组的标识改变了，因此它是一个新数组对象。
- 容量自动增长：16 -> 34

> original	"Hello123, My Name is Mark,234I think..."
▼ aBuffer	StringBuffer (标识=22)
▲ count	17
■ toStringCache	null
▼ ▲ value	(标识=28)
▲ [0]	H
▲ [1]	e
▲ [2]	l
▲ [3]	l
▲ [4]	o
▲ [5]	M
▲ [6]	y
▲ [7]	N
▲ [8]	a
▲ [9]	m
▲ [10]	e
▲ [11]	i
▲ [12]	s
▲ [13]	M
▲ [14]	a
▲ [15]	r
▲ [16]	k
▲ [17]	
▲ [18]	
▲ [19]	
▲ [20]	
▲ [21]	
▲ [22]	
▲ [23]	
▲ [24]	
▲ [25]	
▲ [26]	
▲ [27]	
▲ [28]	
▲ [29]	
▲ [30]	
▲ [31]	
▲ [32]	
▲ [33]	
ch	k
i	24



语言包：数学Math类

- 数学类Math提供一组常量和数学函数，例如
 - E和PI常数
 - 求绝对值的abs方法
 - 开根号sqrt方法、幂函数pow(x, n)方法
 - 计算三角函数的sin/cos方法、反三角函数asin/acos方法、双曲函数sinh/cosh方法
 - 求最小/大值的min/max方法
 - 指数函数exp方法
 - 自然对数/以10为底的对数函数log/log10
 - 四舍五入round方法、向上/下取整ceil/floor方法
 - 求随机数(0~1之间的double)的random方法等
- 其中所有的属性和方法都是静态的(static)
- 是终结类(final)，不能从中派生其他的新类



语言包：System类和Runtime类

- 系统类**System**（成员都是静态的）
 - 访问系统资源
 - arraycopy()** 复制一个数组
 - exit()** 结束当前运行的程序
 - currentTimeMillis()** 获得系统当前日期和时间等
 - 访问标准输入输出流
 - System.in** 标准输入，表示键盘
 - System.out** 标准输出，表示显示器
- 运行时类**Runtime**
 - 可直接访问运行时资源
 - totalMemory()** 返回JVM内存总量（字节）
 - freeMemory()** 返回JVM内存的剩余空间（字节）



语言包：类操作类和ClassLoader类

● Class 类

- 提供运行时信息，如名字、类型以及父类
- **Object**类中继承的getClass方法返回当前对象所在的类，返回类型是**Class**
- 它的getName方法返回一个类的名称，返回值是**String**
- 它的getSuperclass方法可以获得当前对象的父类
- 它的getInterfaces方法可以获得当前对象的实现的所有接口，返回**Class**类型的数组。

● ClassLoader 类

- 提供把类装入运行时环境的方法



实用包(java.util)

- 实用包(java.util)实现各种不同实用功能
 - **StringTokenizer**类：允许以某种分隔标准将字符串分隔成单独的子字符串
 - **Scanner**类：简单文本扫描器，使用某种分隔符将输入文本分隔，可以分析基本数据类型和字符串
 - 集合类/接口
 - **Collection**（无序集合）、**Set**（不重复集合）、**List**（有序不重复集合）、**Queue**等接口
 - **Vector**、**ArrayList**、**Stack**、**LinkedList**、**HashSet**、**TreeSet**等类
 - **Hashtable**（散列表）
 - **Enumeration**（枚举）、**Iterator**(迭代)接口
 - **Collections**类、**Array**类
 - 日期类：描述日期和时间
 - **Date**、**Calendar**、**GregorianCalendar**



实用包：StringTokenizer类

- **StringTokenizer** 类允许以某种分隔标准将字符串分隔成单独的子字符串，如可以将单词从语句中分离出来
- 术语分隔符 (delimiter) 是指用于分隔单词 (也称为标记, tokens) 的字符
- 常用方法
 - **int countTokens()** 返回单词的个数
 - **String nextToken()/nextElement()** 返回下一个单词
 - **boolean hasMoreTokens()/hasMoreElement()** 是否还有单词

"数学::英语::语文::化学"

↑
nextToken()

↑
nextToken()



实用包：StringTokenizer类

- 生成StringTokenizer类的对象的方法
 - **new StringTokenizer(String aString);**
指定了将被处理的字符串，默认的分隔符为空白（单个或多个空格、tab符）
 - **new StringTokenizer(String aString, String delimiters);**
除了指定将被处理的字符串，还指定了分隔符字符串，如分隔符字符串可以为“,:;|_()”
 - **new StringTokenizer(String aString, String delimiters, boolean returnDelimiters);**
第三个参数如果为true，则分隔符本身也作为标记返回



实用包：StringTokenizer类

- 示例：使用StringTokenizer类分割字符串

```
String str = "数学::英语::语文::化学";
```

```
StringTokenizer st=new StringTokenizer(str,"::");//以::作为分隔符
```

```
int m=st.countTokens();
```

```
System.out.println("课程数为："+m);
```

```
while(st.hasMoreTokens())
```

```
    System.out.println(st.nextToken());
```

//以空白作为字符串分隔符

```
String sentence="Yesterday once more!";
```

```
st=new StringTokenizer(sentence);
```

```
m=st.countTokens();
```

```
System.out.println("单词数为："+m);
```

```
while(st.hasMoreTokens())
```

```
    System.out.println(st.nextToken());
```

课程数为：4

数学

英语

语文

化学

单词数为：3

Yesterday

once

more!



实用包：Scanner类

- **Scanner** 类一个可以使用正则表达式来解析基本类型和字符串的简单文本扫描器。
- **Scanner** 使用分隔符模式将其输入分解为标记，默认情况下该分隔符模式与空白匹配。然后可以使用不同的 **next** 方法将得到的标记转换为不同类型的值。
- 常用方法
 - **String next ()** 读取下一个标记（字符串）
 - **boolean hasNext()** 是否有下一个标记（字符串）
 - **xxx Nextxxx()** 读取xxx类型（Int/Double等）的数据
 - **boolean hasNextxxx()** 下一个是否是xxx类型的数据
 - **void close()** 关闭扫描器
 - **Scanner useDelimiter(String pattern)** 设置分隔符



实用包：Scanner类

- 生成Scanner类的对象的方法
 - `new Scanner(File source);`
创建从指定文件扫描数据的扫描器
 - `new Scanner(String source);`
创建从指定字符串扫描数据的扫描器
 - `new Scanner(InputStream source);`
创建从指定输入流扫描数据的扫描器



实用包：Scanner类

- 示例：使用Scanner类分解键盘输入数据或字符串数据

//扫描器从字符串扫描数据（以空格作为字符串分隔符）

```
String sentence="Yesterday once more!";
```

```
Scanner scan=new Scanner(sentence);
```

```
while(scan.hasNext())
```

```
    System.out.println(scan.next());
```

```
scan.close(); // 关闭扫描器
```

Yesterday
once
more!



实用包：Scanner类

第5章 常用Java类库

```
String name;  int age;
scan = new Scanner(System.in); // 从键盘输入扫描数据,输入q结束
System.out.println("请输入姓名和年龄(以空白分隔), 输入q结束: ");
while (scan.hasNext()) {
    name = scan.next();
    if(name.toUpperCase().equals("Q"))
        break;
    age = scan.nextInt();
    System.out.println(name + "\t" + age);
}
scan.close(); // 关闭扫描器
```

请输入姓名和年龄(以空白分隔), 输入q结束:
Tom 22 Kate 54 q
Tom 22
Kate 54



实用包：Vector<E> 类

- **Vector** 类似于数组，但与数组相比在使用上有两个优点：
 - 无需声明上限，随着元素的增加，向量的容量（即内部数据数组的大小）会根据需要自动增加或减少。
 - 提供额外的方法来增加、删除元素，比数组操作高效
- 内部属性

<code>protected Object[] elementData</code>	存储向量组件的数组缓冲区。 vector 的容量就是此数据缓冲区的长度，该长度至少要足以包含向量的所有元素。 Vector 中的最后一个元素后的任何数组元素都为 null 。
<code>protected int capacityIncrement</code>	向量的大小大于其容量时，容量自动增加的量。如果容量的增量小于等于零，则每次需要增大容量时，向量的容量将 增大一倍 。
<code>protected int elementCount</code>	Vector 对象中的有效组件数。从 elementData[0] 到 elementData[elementCount-1] 的组件均为实际项。



实用包：Vector<E> 类

- Vector 类似于数组，但与数组相比在使用上有两个优点：
 - 无需声明上限，随着元素的增加，向量的容量（即内部数据数组的大小）会自动增加
 - 提供额外的方法来增加、删除元素，比数组操作高效
- 向量中的元素必须是引用类型，基本类型数据保存到向量前会自动封箱（即自动转换成相应的包装类对象）
- 构造方法
 - **Vector<E>()** 构造一个元素类型为E、初始容量为10的空向量，其容量增量为零。
 - **Vector<E>(int initialCapacity)** 构造一个具有指定初始容量的向量，其容量增量为零。
 - **Vector<E>(int initialCapacity, int capacityIncrement)** 构造一个具有指定初始容量和容量增量的向量。



实用包：Vector<E> 类

● 常用方法

— 向向量添加元素

- **addElement(E obj)** 将对象obj添加到向量末尾
- **boolean add(E e)** 同上。对于向量，add方法总是返回true
- **insertElementAt(E obj, int index)** 将对象obj插入向量的index处

— 删除或修改向量中元素

- **removeElementAt(int index)** 移除向量index处的元素
- **boolean removeElement(E obj)** 移除向量中第一个与对象obj相同的元素，不存在则返回false
- **removeAllElements()** 移除向量中所有元素
- **setElementAt(E obj, int index)** 将向量index处的元素设置为obj
- **E set(int index, E element)** 用指定的元素替换此向量中指定位置处的元素，返回被替换的那个元素



实用包：Vector<E> 类

— 访问向量中元素

- **E elementAt(int index)** 返回指定位置index处的元素。
- **E get(int index)** 同上
- **E firstElement()** 返回向量的第一个元素
- **E lastElement()** 返回向量的最后一个元素
- **boolean isEmpty()** 判断向量中是否有元素
- **boolean contains(Object o)** 判断向量是否包含对象o
- **Enumeration elements()** 返回包含向量中所有元素的枚举对象

Vector类不支持下标[], 而是用以下方法实现其元素对象的读取与修改:

- **E elementAt(int index)**
- **setElementAt(E obj, int index)**

也可以来自**List**接口的**get(int index) / set(int index, E obj)** 方法获取、修改元素, 使用来自**Collection**接口的**add(E obj)**方法增加元素。



实用包：Vector<E> 类

- 查找向量中元素
 - **int indexOf(Object obj)** 返回向量中第一次出现指定对象obj的索引。如果对象不存在，则返回 -1
 - **int indexOf(Object obj, int index)** 从index处开始正向搜索，返回向量中第一次出现指定对象obj的索引。如果对象不存在，则返回 -1
- 向量大小的操作
 - **int size()** 返回向量中元素个数
 - **int capacity ()** 返回向量容量大小
 - **setSize(int newSize)** 设置向量的大小。如果新大小大于当前大小，则会在向量的末尾添加相应数量的 null 值元素。如果新大小小于当前大小，则丢弃索引 newSize 处及其之后的所有元素。
 - **trimToSize()** 把向量容量调整到正好等于向量元素个数，以压缩向量的存储空间



实用包：Vector<E> 类

- 示例：创建三个元素类型不同的向量，对向量添加元素，并进行元素访问、插入、修改、删除等操作。

```
Vector<String> v1=new Vector<String>(); // 存储String对象
v1.addElement("北京"); v1.addElement("上海"); v1.add("山东");
System.out.println("v1=" + v1);
___.println("v1:元素个数=" + v1.size() + ", 容量=" + v1.capacity());
v1.insertElementAt("青岛", 1);
___.println("v1=" + v1);
v1.removeElementAt(0);
___.println("v1=" + v1);
```

```
v1=[北京, 上海, 山东]
v1:元素个数=3, 容量=10
v1=[北京, 青岛, 上海, 山东]
v1=[青岛, 上海, 山东]
```

```
___.println("是否包含上海: "+v1.contains("上海"));
___.println(v1.get(0)); //获取元素
v1.set(0, "Beijing"); //修改元素
v1.add("天津"); //添加元素
___.println("v1=" + v1);
```

```
是否包含上海: true
```

```
青岛
```

```
v1=[Beijing, 上海, 山东, 天津]
```



实用包：Vector<E> 类

```
Vector<Integer> v2=new Vector<Integer>(); //存储Integer对象  
for(int i=0;i<15;i++)
```

```
    v2.addElement((int)(Math.random()*9)); //注意，这里int类型  
    变量将自动封箱成Integer对象
```

```
System.out.println("v2="+v2);
```

```
____.println("v2:元素个数="+v2.size() + "， 容量="+v2.capacity());
```

```
____.println("v2的第4个元素为： " + v2.elementAt(3));
```

```
v2.setElementAt(9999, 3); //修改第4个元素为9999
```

```
____.println("v2="+v2);
```

```
____.println("查找第一个元素8的位置： "+v2.indexOf(8));
```

```
v2=[6, 8, 7, 3, 8, 7, 5, 3, 6, 7, 8, 8, 8, 1, 8]
```

```
v2:元素个数=15， 容量=20
```

```
v2的第4个元素为： 3
```

```
v2=[6, 8, 7, 9999, 8, 7, 5, 3, 6, 7, 8, 8, 8, 1, 8]
```

```
查找第一个元素8的位置： 1
```



实用包：Vector<E> 类

```
Vector<Circle> v3=new Vector<Circle>(); // 存储Circle对象
v3.addElement(new Circle(1));
v3.addElement(new Circle(3));
System.out.println("v3="+v3);
___.println("v3:元素个数="+v3.size()+"， 容量="+v3.capacity());
Circle cir=null;
___.println("圆序号\t半径\t面积");
for(int i=0;i<v3.size();i++){
    cir=v3.elementAt(i);
    ___.println(i+"\t"+cir.getRadius()+"\t"+cir.area());
}
```

v3=[Circle@106d69c, Circle@52e922]

v3:元素个数=2， 容量=10

圆序号	半径	面积
-----	----	----

0	1.0	3.141592653589793
---	-----	-------------------

1	3.0	28.274333882308138
---	-----	--------------------



实用包：Hashtable<K,V> 类

- 哈希表（散列表）Hashtable将键（Key）映射到相应的值(Value)，即哈希表中存储是键-值对
- 哈希表中的键必须是唯一的，但一个键可以对应多个值
- 为了成功地在哈希表中存储和获取对象，用作键的对象必须重写 hashCode 方法和 equals 方法（这两个方法在String类中已经重写）
- 容量（capacity）和加载因子（load factor）
 - 容量 是哈希表中桶的数量
 - 加载因子 是对哈希表在其容量自动增加之前可以达到多满的一个尺度



实用包：Hashtable<K,V> 类

- 构造方法（默认的初始容量 =11 和加载因子 =0.75）
 - **Hashtable()** 用默认参数构造一个新的空哈希表
 - **Hashtable(int initialCapacity)** 用指定初始容量构造一个新的空哈希表
 - **Hashtable(int initialCapacity, float loadFactor)** 用指定初始容量和指定加载因子构造一个新的空哈希表
- 常用方法
 - **V put(V key, V value)** 将指定的键-值对存放到哈希表中，键存在则覆盖它。返回此哈希表中指定键的以前的值；如果不存在该值，则返回 **null**
 - **V get(Object key)** 返回指定键所映射到的值，如果此映射不包含此键的映射，则返回 **null**



实用包：Hashtable<K,V> 类

- 常用方法
 - **int size()** 返回哈希表中的键的数量
 - **boolean isEmpty()** 测试哈希表是否为空
 - **clear()** 清空哈希表
 - **boolean containsKey(Object key)** 测试指定对象是否为此哈希表中的键。
 - **Enumeration<V> elements()** 返回值的枚举。对返回的对象使用 **Enumeration** 方法，以便按顺序获取这些元素。
 - **Enumeration<K> keys()** 返回键的枚举
 - **Collection<K> values()** 返回值的 **Collection**



实用包：Hashtable<K,V> 类

- 示例：使用哈希表存储学生姓名与成绩（键-值对）

```
Hashtable<String, Integer> ht  
= new Hashtable<String, Integer> ();
```

```
ht.put("张三", 95); //存入键对应的值
```

```
ht.put("李四", 86); ht.put("王五", 95);
```

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

```
____.println("人数="+ht.size());
```

```
____.println("李四="+ht.get("李四"));
```

```
ht.put("李四", 99); //修改键对应的值
```

```
____.println(ht);
```

```
ht.remove("李四"); //移除键值对
```

```
____.println(ht);
```

{王五=95, 张三=95, 李四=86}

人数=3

李四=86

{王五=95, 张三=95, 李四=99}

{王五=95, 张三=95}



实用包：Enumeration<E>接口

- 实现Enumeration接口的对象生成一系列元素，通过nextElement()方法依次读取下一个元素。
- 枚举接口只有两个方法
 - **boolean hasMoreElements()** 测试此枚举是否包含更多的元素
 - **E nextElement()** 返回此枚举的下一个元素
- Enumeration接口及其方法通常与Vector、Hashtable一起连用，用来枚举Vector中的元素和Hashtable中的键或值，例如：

```
for (Enumeration<E> e = v.elements(); e.hasMoreElements(); )  
    System.out.println(e.nextElement());
```
- Vector和Hashtable等类提供了**Enumeration<E> elements()/keys()**方法获取其枚举接口，StringTokenizer类实现了枚举接口



实用包：Enumeration<E>接口

- 示例：使用枚举接口遍历向量

```
Vector<String> v1=new Vector<String>();  
v1.addElement("北京"); v1.addElement("上海"); v1.addElement("山东");  
Enumeration<String> e = v1.elements();  
while(e.hasMoreElements()) ____println(e.nextElement());
```

- 示例：使用枚举接口遍历哈希表的键或值

```
Hashtable<String, Integer> ht = new Hashtable<String, Integer> ();  
ht.put("张三", 95); ht.put("李四", 86); ht.put("王五", 95);  
Enumeration<String> eK=ht.keys(); //返回哈希表中的键的枚举接口  
while(eK.hasMoreElements())  
____.println(eK.nextElement());  
  
Enumeration<Integer> eV=ht.elements(); //返回哈希表中的值的枚举接口  
while(eV.hasMoreElements())  
____.println(eV.nextElement());
```



实用包：Iterator<E>接口

- 对 **collection** 进行迭代的迭代器
- 迭代接口取代了 **Java Collections Framework** 中的 **Enumeration**。迭代器与枚举有两点不同：
 - 在迭代期间可以从其所指向的 **collection** 移除元素
 - 方法名称得到了改进
- 迭代接口只有三个方法
 - **boolean hasNext()** 判断是否仍有元素可以迭代
 - **E next()** 返回迭代的下一个元素
 - **remove()** 从迭代器指向的 **collection** 中移除迭代器返回的最后一个元素。每次调用 **next** 只能调用一次此方法。
- 实现了 **collection <E>** 接口的 **Vector**、**LinkedList**，**HashSet**、**TreeSet**等类提供了**Iterator<E> iterator()**方法获取其迭代器，**Scanner**类实现了迭代接口



实用包： Iterator<E> 接口

- 示例：使用迭代接口遍历向量

```
Vector<String> v1=new Vector<String>();  
v1.addElement("北京"); v1.addElement("上海"); v1.addElement("山东");  
for(Iterator<String> e = v1.iterator(); e.hasNext(); )  
    _____.println(e.next());
```

```
Vector<Circle> v3=new Vector<Circle>();  
v3.addElement(new Circle(1)); v3.addElement(new Circle(3));  
Circle cir=null;  
for(Iterator<Circle> e=v3.iterator();e.hasNext();){  
    cir=e.next();  
    _____.println(cir.getRadius()+"\t"+cir.area());  
}
```



实用包： Iterator<E> 接口

- 小结：遍历向量Vector对象的四种方式
 - ① 利用元素索引通过其**elementAt(index)**方法遍历

```
for( int i=0; i<v1.size();i++)  
    _____.println(v1.elementAt(i));
```
 - ② 用增强的for循环遍历

```
for( String str : v1 )  
    _____.println(str);
```
 - ③ 通过**Enumeration**接口遍历

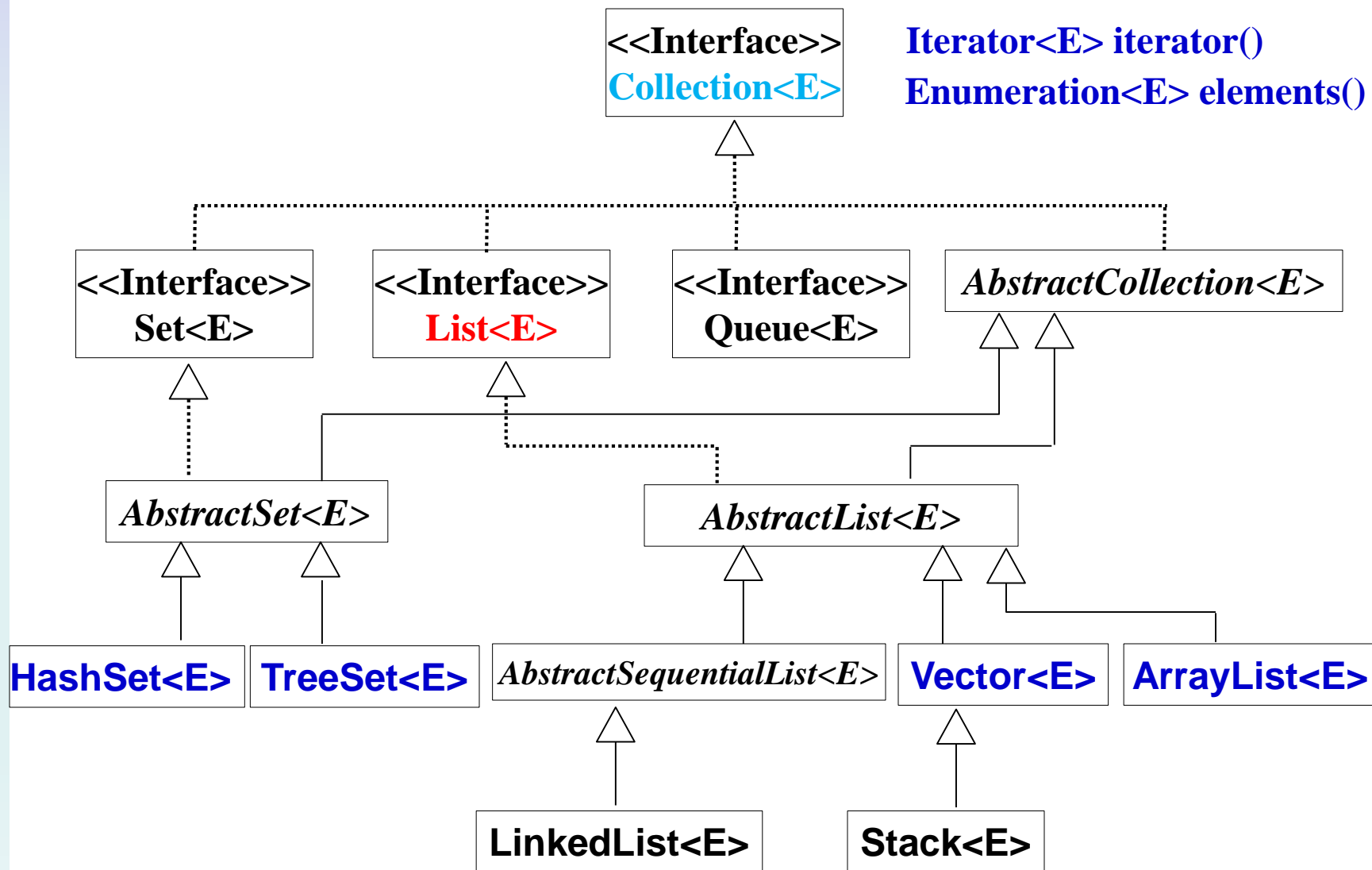
```
for(Enumeration<String> e = v1. elements();  
     e. hasMoreElements(); )  
    _____.println(e.nextElement());
```
 - ④ 通过**Iterator**接口遍历

```
for(Iterator<String> e = v1. iterator(); e.hasNext(); )  
    _____.println(e.next());
```



实用包：部分集合/接口的层次关系

第5章 常用Java类库





实用包：Collection接口

- 接口特点：元素是Object
- 常用方法：
 - `add(Object o)`: 将对象o添加到集合中
 - `addAll(Collection c)`: 将集合c中所有的元素添加到集合中
 - `clear()`: 清空集合
 - `contains(Object o)`: 判断集合中是否存在元素o
 - `remove(Object o)`: 从集合中删除元素o
 - `size()`: 返回集合的长度
 - `toArray()`: 将集合转为数组



实用包：ArrayList类

- 列表ArrayList类表示大小可变数组，用法类似于向量。

ArrayList<Integer> list=new ArrayList<Integer>(); //构造一个初始容量为 10 的空列表。

System.out.println(list.size()); //列表中元素个数

list.add(5); //在末尾添加元素

list.add(0, 8); //在指定位置0添加元素

list.add(3);

System.out.println(list.size()); //列表中元素个数

System.out.println(list); //输出列表

list.remove(1); //移除第2个元素

System.out.println(list); //输出列表

System.out.println("获取第1个元素: "+list.get(0));

System.out.println("修改第1个元素: "+list.set(0,99));

System.out.println(list); //输出列表



实用包：ArrayList类

// 用Iterator接口遍历列表元素, 注意不支持Enumeration接口

```
System.out.println("---用Iterator接口遍历列表元素---");
```

```
Iterator<Integer> it=list.iterator();
```

```
while(it.hasNext()) System.out.println(it.next());
```

```
System.out.println("---用普通for循环遍历列表元素---");
```

```
for(int i=0;i<list.size();i++)
```

```
    System.out.println(list.get(i));
```

```
System.out.println("---用增强for循环遍历列表元素---");
```

```
for(int x:list) System.out.println(x);
```

//返回数组形式

```
Integer[] a=new Integer[list.size()]; //a必须是int的包裹类型
```

```
list.toArray(a);
```

```
System.out.println(a.length);
```

```
for(int x:a) System.out.println(x);
```



实用包：HashSet类与TreeSet类

- **HashSet**类表示**无序集合**，元素不允许重复，可以包含一个null元素，但是不能保证元素的排列顺序，顺序有可能发生变化。
 - 当向**HashSet**集合中存入一个元素时，**HashSet**会调用该对象的**hashCode()**方法来得到该对象的**hashCode**值，然后根据**hashCode**值来决定该对象在**HashSet**中存储位置。
- **TreeSet**类表示**有序集合**，可以确保集合元素处于排序状态，但是不允许放入null值。
 - 支持自然排序（默认）和定制排序。自然排序是根据集合元素的大小，以升序排列；要定制排序，则应提供实现**Comparator**接口的比较器，其中实现 **int compare(T o1,T o2)**方法。
 - 使用自然排序时，插入该**set**的所有元素都必须实现 **Comparable**接口，否则将抛出**ClassCastException**异常。**Integer, String**等类都已实现该接口。
 - 方法**descendingSet()**返回降序排列的**NavigableSet**集合。



实用包：HashSet类与TreeSet类

- 例：利用HashSet去掉一个Vector中重复的元素

```
Vector<Integer> v=new Vector<Integer>();  
v.add(5); v.add(9); v.add(5); v.add(3); v.add(34);
```

//用向量初始化hashSet

```
HashSet<Integer> set=new HashSet<Integer>(v);  
System.out.println("v= "+v);  
System.out.println("set= "+set);
```

//用hashSet初始化向量

```
Vector<Integer> v1=new Vector<Integer>(set);  
System.out.println("v1= "+v1);
```

```
v= [5, 9, 5, 3, 34]  
set= [34, 3, 5, 9]  
v1= [34, 3, 5, 9]  
set2= [3, 5, 9, 34]
```

//用向量初始化有序集合TreeSet

```
TreeSet<Integer> set2=new TreeSet<Integer>(v);  
System.out.println("set2= "+set2);
```



Comparable接口和Comparator接口

- **java.lang.Comparable** 接口 用于定义对元素进行 **自然排序** 规则的方法 `int compareTo(T o)`，在元素所属类中实现
 - 实现此接口的对象列表（和数组）可以通过 **Collections.sort**（和 **Arrays.sort**）进行自动排序。实现此接口的对象可以用作有序映射中的键或有序集合中的元素，无需指定比较器。
 - 用法形如：**Collections.sort(list);** // 自然排序，不指定比较器
- **java.util.Comparator** 接口 用于定义强行对某个对象 **collection** 进行整体排序的比较器（实现 `int compare(T o1, T o2)` 方法），用于 **定制排序**
 - 先单独定义一个实现了 **Comparator** 接口的比较器类，然后可以将其比较器对象传递给 **sort** 方法（如 **Collections.sort** 或 **Arrays.sort**），从而允许在排序顺序上实现精确控制。还可以使用 **Comparator** 来控制某些数据结构（如有序 **set** 或有序映射）的顺序，或者为那些没有自然顺序的对象 **collection** 提供排序。
 - 用法形如：**Collections.sort(list, comparator);** // 指定比较器



Comparable接口和Comparator接口

第5章 常用Java类库

- 例：为Circle类定义自然排序规则——实现Comparable接口

```
public class Circle implements Comparable<Circle>{  
    private float r;  
    public Circle(float r) {this.r = r;}  
    public void setR(float r) {this.r = r;}  
    public float getR(){return r;}  
    @Override  
    public String toString(){return "Circle [r=" + r + "];"}
```

// 实现Comparable接口的抽象方法：用于自然排序

@Override

```
public int compareTo(Circle o) { // 规则：半径较大的对象就较大  
    if(this.r > o.r)  
        return 1;  
    else if(this.r == o.r)  
        return 0;  
    else  
        return -1;  
}
```

return Double.compare(this.r, o.r);



Comparable接口和Comparator接口

第5章 常用Java类库

- 例：为Circle类定义自然排序规则——实现Comparable接口

```
public static void main(String[] args) {  
    ArrayList<Circle> list=new ArrayList<Circle>();  
    list.add(new Circle(5)); list.add(new Circle(3));  
    list.add(new Circle(1)); list.add(new Circle(9));  
    System.out.println(list);  
  
    System.out.println("ArrayList排序后: ");  
    Collections.sort(list); //按自然顺序对list排序  
    System.out.println(list);  
}
```

```
[Circle [r=5.0], Circle [r=3.0], Circle [r=1.0], Circle [r=9.0]]  
ArrayList排序后:  
[Circle [r=1.0], Circle [r=3.0], Circle [r=5.0], Circle [r=9.0]]
```




Comparable接口和Comparator接口

第5章 常用Java类库

- 例：为Circle类定义自然排序规则——实现Comparable接口

```
public static void main(String[] args) {  
    ArrayList<Circle> list=new ArrayList<Circle>();  
    list.add(new Circle(5)); list.add(new Circle(3));  
    list.add(new Circle(1)); list.add(new Circle(9));  
    System.out.println(list);
```

// TreeSet中元素会自动按自然顺序排列

```
TreeSet<Circle2> set=new TreeSet<Circle2>(list);  
System.out.println("TreeSet中元素: ");  
System.out.println(set);
```

```
}
```

```
}
```

```
[Circle [r=5.0], Circle [r=3.0], Circle [r=1.0], Circle [r=9.0]]  
TreeSet中元素:  
[Circle [r=1.0], Circle [r=3.0], Circle [r=5.0], Circle [r=9.0]]
```




Comparable接口和Comparator接口

- 例：为Circle类的对象定义比较器—实现Comparator接口

```
class ComparatorForCircle implements Comparator<Circle> {  
    // 实现Comparator接口的抽象方法  
    // 比较规则：半径较大的对象就较大  
    @Override  
    public int compare(Circle o1, Circle o2) {  
        double r1=o1.getR();  
        double r2=o2.getR();  
        if (r1 > r2)  
            return 1;  
        else if (r1 == r2)  
            return 0;  
        else  
            return -1;  
    }  
}
```

} return Double.compare(r1, r2);



Comparable接口和Comparator接口

- 例：为Circle类的对象定义比较器——实现Comparator接口

```
public class Circle{
```

```
.....
```

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

```
    ArrayList<Circle> list=new ArrayList<Circle>();
```

```
    list.add(new Circle(5)); list.add(new Circle(3));
```

```
    list.add(new Circle(1)); list.add(new Circle(9));
```

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

```
//创建比较器
```

```
    ComparatorForCircle comparator=new ComparatorForCircle();
```

```
    System.out.println("ArrayList排序后: ");
```

```
    Collections.sort(list,comparator); //根据指定比较器，对list排序
```

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

```
}
```

```
] [Circle [r=5.0], Circle [r=3.0], Circle [r=1.0], Circle [r=9.0]]
```

```
ArrayList排序后:
```

```
[Circle [r=1.0], Circle [r=3.0], Circle [r=5.0], Circle [r=9.0]]
```



Comparable接口和Comparator接口

- 例：为Circle类的对象定义比较器—实现Comparator接口

```
public static void main(String[] args) {  
    ArrayList<Circle> list=new ArrayList<Circle>();  
    list.add(new Circle(5)); list.add(new Circle(3));  
    list.add(new Circle(1)); list.add(new Circle(9));  
    System.out.println(list);
```

//定义比较器

```
ComparatorForCircle comparator=new ComparatorForCircle();
```

// 创建TreeSet，并指定比较器

```
TreeSet<Circle> set=new TreeSet<Circle>(comparator);
```

```
set.addAll(list); // 添加元素
```

```
System.out.println("TreeSet中元素：");
```

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

```
[Circle [r=5.0], Circle [r=3.0], Circle [r=1.0], Circle [r=9.0]]  
TreeSet中元素：  
[Circle [r=1.0], Circle [r=3.0], Circle [r=5.0], Circle [r=9.0]]
```



比较器的其他实现方式

- 比较简单属性，可使用函数式接口(**FunctionalInterface**) **Comparator**提供的**comparing**方法或**comparingXxx**方法(例如**comparingDouble**)得到比较器

```
// comparingXxx(keyExtractor):  
// 参数keyExtractor是一个函数，用于指定从比较的对象获取属性的方法，  
// 一般对应某个getter  
// 返回一个比较器实例
```

```
Collections.sort(list, Comparator.comparingDouble(Circle::getR));  
// ::表示引用方法
```

```
// 或者  
Collections.sort(list, Comparator.comparing(Circle::getR));
```



比较器的其他实现方式

● 使用Lambda表达式简化匿名比较器

// 方式2: 创建匿名类的对象

```
Collections.sort(v, new Comparator<Circle>(){  
    @Override  
    public int compare(Circle o1, Circle o2) {  
        double r1=o1.getR(), r2=o2.getR();  
        if(r1>r2)  
            return 1;  
        else if(r1==r2)  
            return 0;  
        else  
            return -1;  
    }  
});
```

使用Lambda表达式, 可以简化定义那些实现了只有一个抽象方法的接口的匿名类



```
Collections.sort(v, (Circle o1, Circle o2) ->  
    { return o1.getR()>o2.getR()? 1 : (o1.getR()==o2.getR()? 0 : -1); } );
```



比较器的其他实现方式

● 使用Lambda表达式简化匿名比较器

```
Collections.sort(v, (Circle o1, Circle o2) -> { return o1.getR()>o2.getR()? 1 : (o1.getR()==o2.getR()? 0 : -1); });
```

Lambda表达式可以进一步简化：省略return和花括号

↓

```
Collections.sort(v, (Circle o1, Circle o2) -> o1.getR()>o2.getR()? 1 : (o1.getR()==o2.getR()? 0 : -1) );
```

再进一步简化：省略形参类型

↓

```
Collections.sort(v, (o1, o2) -> o1.getR()>o2.getR()? 1 : (o1.getR()==o2.getR()? 0 : -1) );
```

↕

```
Collections.sort(v, (o1, o2) -> Double.compare(o1.getR(), o2.getR() ) );
```



实用包：Stack类

- Stack 类表示后进先出（LIFO）的对象堆栈。
- 通过5个操作对类 Vector 进行了扩展，允许将向量视为堆栈。
 - 提供了通常的 push 和 pop 操作
 - 取堆栈顶点的 peek 方法
 - 测试堆栈是否为空的 empty 方法
 - 在堆栈中查找项并确定到堆栈顶距离的 search 方法
- 首次创建堆栈时，它不包含项。构造方法：

```
Stack<E> stack=new Stack<E>();
```

栈顶---> | 7 | 5 | 2 | 8 | 栈底



实用包：Stack类

- 示例：使用 **Stack** 对象存储字符串。

Stack<String> stack=**new Stack<String>()**; //创建空栈

System.**out.println(stack.isEmpty());** //测试堆栈是否为空

stack.add("山东"); //将指定元素添加到此向量的末尾。

stack.add("江苏");

stack.add(1, "湖北"); //在此向量的指定位置插入指定的元素

stack.addElement("四川"); //功能与 **add(E)** 方法完全相同

stack.push("辽宁"); //元素压栈,作用与 **add(E)** 方法完全相同

System.**out.println(stack.isEmpty());**

System.**out.println(stack);**



实用包：Stack类

- 示例：使用 **Stack** 对象存储字符串。
//移除堆栈顶部的对象，并作为此函数的值返回该对象。
`System.out.println(stack.pop());`
`System.out.println(stack);`
//查看堆栈顶部的对象，但不从堆栈中移除它。
`System.out.println(stack.peek());`
`System.out.println(stack);`
//移除此向量中指定位置的元素，并返回该元素。
`System.out.println(stack.remove(0));`
`System.out.println(stack);`
//从此向量中移除全部组件，功能与 **clear()** 方法完全相同。
`stack.removeAllElements(); //stack.clear();`
`System.out.println(stack.isEmpty());`



实用包：Deque<E>接口

- **public interface Deque<E> extends Queue<E>**
- 一个线性 collection，支持在两端插入和移除元素。名称 deque 是“double ended queue（双端队列）”的缩写，通常读为“deck”。
- 此接口定义在双端队列两端访问元素的方法。提供插入、移除和检查元素的方法。
 - 插入 **addFirst(e)** **addLast(e)**
 - 移除 **removeFirst()** **removeLast()**
 - 获取 **getFirst()** **getLast()** **peekFirst()** **peekLast()**



实用包：Deque<E>接口

- 此接口扩展了 **Queue** 接口。在将双端队列用作队列时，将得到 **FIFO**（先进先出）行为。将元素添加到双端队列的末尾，从双端队列的开头移除元素。从 **Queue** 接口继承的方法完全等效于 **Deque** 方法，如下表所示：

Queue 方法	等效 Deque 方法
<u>add(e)</u>	<u>addLast(e)</u>
<u>remove()</u>	<u>removeFirst()</u>
<u>element()</u>	<u>getFirst()</u>



实用包：Deque<E>接口

- 双端队列也可用作 **LIFO**（后进先出）堆栈。应优先使用此接口而不是遗留 **Stack** 类。在将双端队列用作堆栈时，元素被推入双端队列的开头并从双端队列开头弹出。堆栈方法完全等效于 **Deque** 方法，如下表所示：

堆栈Stack方法	等效 Deque 方法
<u>push(e)</u>	<u>addFirst(e)</u>
<u>pop()</u>	<u>removeFirst()</u>
<u>peek()</u>	<u>peekFirst()</u>

(队列首) 栈顶---> | 7 | 5 | 2 | 8 | 栈底 (队列尾)



实用包： Deque<E>接口

- 示例：双端队列也可用作 **LIFO**（后进先出）栈。

//创建空双端队列 **ArrayDeque** (作为栈)

// **ArrayDeque**: **Deque** 接口的大小可变数组的实现，没有容量限制，可根据需要增加以支持使用。

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

```
____.println(stack.isEmpty()); //测试堆栈是否为空
```

//将指定元素插入此双端队列的开头(压栈)

```
stack.push(8); stack.push(2); //等效于 addFirst(E)
```

```
stack.addFirst(5); stack.addFirst(7);
```

```
____.println(stack);
```



实用包： Deque<E>接口

- 示例：双端队列也可用作 **LIFO**（后进先出）栈。

//获取并移除此双端队列第一个元素(弹栈)

`stack.pop();` // 等效于 `removeFirst()`

`stack.remove();` //也等效于 `removeFirst()`

`____.println(stack);`

//获取双端队列的第一个元素

`____.println(stack.peek());` //等效于 `peekFirst()`

`System.out.println(stack);`

`stack.clear();` //移除此 **collection** 中的所有元素

`____.println(stack.isEmpty());`



实用包：Collections 类

- **Collections** 是 **Collection** 的一个工具类，提供静态方法，帮助 **collection** 完成一些功能，例如反转、排序(升序、降序、随机排列)、获得最大/小元素等。
- 常用方法

static <T> boolean	addAll (Collection<? super T> c,T... elements) 将所有指定元素添加到指定 collection 中。
static int	frequency (Collection<?> c, Object o) 返回指定 collection 中等于指定对象的元素数。
static <T extends Object & Comparable<? super T>> T	max (Collection<? extends T> coll) 根据元素的自然顺序，返回给定 collection 的最大元素。
static void	reverse (List<?> list) 反转指定列表中元素的顺序
static void	shuffle (List<?> list) 使用默认随机源对指定列表进行置换。
static <T extends Comparable<? super T>> void	sort (List<T> list) 根据元素的自然顺序 对指定列表按升序进行排序。
static <T> void	sort (List<T> list, Comparator<? super T> c) 根据指定比较器产生的顺序对指定列表进行排序。
static void	swap (List<?> list,int i,int j) 在指定列表的指定位置处交换元素。



实用包：Collections类

- 例：用**Collections**类对向量进行求最大元素和排序等操作

```
Vector<Integer> v=new Vector<Integer>();  
v.add(5);v.add(9);v.add(5);v.add(3);v.add(34);
```

```
System.out.println(Collections.max(v)); // 34
```

```
Collections.reverse(v); // 反转排列向量元素  
System.out.println(v); // [34, 3, 5, 9, 5]
```

```
Collections.sort(v); //升序排列向量元素  
System.out.println(v); // [3, 5, 5, 9, 34]
```

```
// 如何降序排列：可以先升序排列，然后反转  
Collections.reverse(v); // 反转排列向量元素  
System.out.println(v); // [34, 9, 5, 5, 3]
```




实用包：Arrays 类

- **Arrays** 类包含用来操作 **普通数组**（比如排序和搜索）的各种静态方法。
- 例子：

```
int[ ] a={5,9,3,6};
```

```
Arrays.sort(a);//对指定的 int 型数组按数字升序进行排序
```

```
for(int x:a)
```

```
    System.out.print(x+" ");
```

```
int index=Arrays.binarySearch(a, 6);//二分法搜索元素
```

```
System.out.println("\n"+index); //元素不存在，则index<0
```

```
List<Integer> list=Arrays.asList(2,0,4,8); //返回一个列表
```

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

```
3 5 6 9
```

```
2
```

```
[2, 0, 4, 8]
```



实用包：Date类

- 构造方法
 - **Date()**: 获得系统当前日期和时间值。
 - **Date(long date)**: 以date创建日期对象，date表示从GMT（格林威治）时间1970-1-1 00:00:00开始至某时刻的毫秒数
- 常用方法
 - **getTime()**
返回一个长整型表示时间，单位为毫秒（millisecond）
 - **after(Date d)**
返回接收者表示的日期是否在给定的日期之后
 - **before(Date d)**
返回接收者表示的日期是否在给定的日期之前
- 说明：**getYear()**等其他很多方法已过时，由Calendar类的get方法取代



实用包：Date类

- 示例

```
Date date1=new Date(); // 用系统当前时间构造Date对象
long time1=date1.getTime();
____.println(date1.toString());
____.println(time1);
Date date2=new Date(time1+5);
____.println(date1.after(date2));
____.println(date1.before(date2));
```

运行结果：

```
Tue May 31 23:46:20 CST 2016
1464709580256
false
true
```



实用包：Calendar类

- 一个抽象的基础类，支持将Date对象转换成一系列单个的日期整型数据集，如YEAR、MONTH、DAY、HOUR等常量
- 它派生的GregorianCalendar类实现标准的Gregorian日历（格里高利历）
- 由于Calendar是抽象类，不能用new方法生成Calendar的实例对象，可以使用getInstance()方法创建一个GregorianCalendar类的对象



实用包：Calendar类

- Calendar类中声明了许多的静态常量，例如
 - 表示上、下午：
Calendar.AM , PM
 - 表示周天~周六：
Calendar.SUNDAY, MONDAY ...
 - 表示一月~十二月：
Calendar.JANUARY, FEBRUARY...
 - 指示get/set方法获取/设置年，月，日，时，分，秒，毫秒：
 - **Calendar.HOUR, MINUTE, SECOND, MILLISECOND**
 - **Calendar.YEAR, MONTH, DAY**



实用包：Calendar类

- Calendar类中的方法
 - **isLeapYear(int year)** 返回给定的年份是否是闰年
 - **after/befor(Object ob)** 比较日期
 - **get(int field)** 取得特定Calendar对象的信息
 - **aCalendar.get(Calendar.YEAR);**
 - **aCalendar.get(Calendar.MONTH);**
 - **aCalendar.get(Calendar.DAY_OF_WEEK);**
 - **aCalendar.get(Calendar.MINUTE);**
 - ...
 - **set(int field, int value)** 给日期域设定特定的值
 - **aCalendar.set(Calendar.MONTH, Calendar.JANUARY);**
 - **aCalendar.set(1999, Calendar.AUGUST, 15);**
 - ...



实用包：GregorianCalendar类

- 抽象Calendar类的一个具体实现，用于查询及操作日期
- 构造方法
 - `new GregorianCalendar()` // 当前日期
 - `new GregorianCalendar(1999, 11, 31)` // 特定日期
 - `new GregorianCalendar(1999, 11, 31, 8, 11, 55)` // 日期和时间
- `getTime()`方法：返回Date对象，显示日历
 - `____.println(new GregorianCalendar().getTime());`
 - `____.println(new GregorianCalendar(1999, 11, 31).getTime());`
- `void add(int field, int amount)`方法：根据日历规则，将指定的（有符号的）时间量添加到给定的日历字段中
 - `time.add(Calendar.MONTH, -4);` //从time表示的日期减去4个月



实用包：GregorianCalendar类

● 示例

Calendar time=new GregorianCalendar(); //在具有默认语言环境的默认时区内使用当前时间构造GregorianCalendar对象

```
____.println(time.getTime());
```

```
____.println(time.get(Calendar.YEAR));
```

```
____.println(" 星期"+time.get(Calendar.DAY_OF_WEEK));
```

```
time.add(Calendar.MONTH, -4); //从当前日期减去4个月
```

```
____.println(time.getTime());
```

```
time.add(Calendar.DAY_OF_MONTH, 5); //从当前日期加5天
```

```
____.println(time.getTime());
```

运行结果：

Wed Jun 01 00:18:38 CST 2016

2016

星期4

Mon Feb 01 00:18:38 CST 2016

Sat Feb 06 00:18:38 CST 2016



实用包：GregorianCalendar类

● 示例(续)

`time.set(1998, 7, 25);` //改变年月日。Month 值是基于 0 的。例如, 0 表示 January

`____.println(time.getTime());`

`time.set(Calendar.MONTH, Calendar.SEPTEMBER);` //改变日期到当年的九月份

`____.println(time.getTime());`

`time.set(Calendar.DAY_OF_WEEK, Calendar.WEDNESDAY);` //改变日期到本周三

`____.println(time.getTime());`

运行结果:

Fri Aug 25 00:18:38 CST 1998

Fri Sep 25 00:18:38 CST 1998

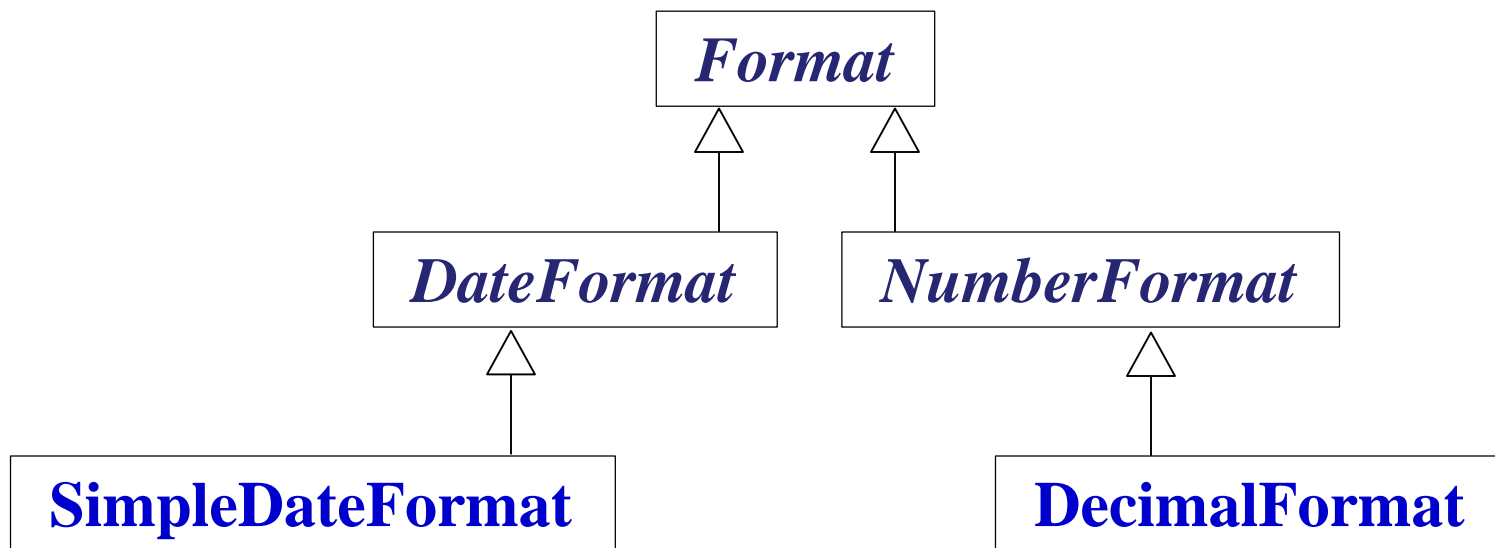
Wed Sep 23 00:18:38 CST 1998

可以用java.text包中的SimpleDateFormat类对日期进行格式化



文本包(java.text)

- 提供各种文本或日期格式
- 包含有
 - **Format** 类
 - **DateFormat** 类, **SimpleDateFormat** 类
 - **NumberFormat** 类, **DecimalFormat** 类





文本包：SimpleDateFormat类

- 使用已定义的格式对日期对象进行格式化
- 构造方法：以一指定格式的字符串作为参数

SimpleDateFormat(String pattern);

— pattern用于指定格式，常见

- "yyyy-MM-dd" // 1998-07-25
- "yyyy.MM.dd" // 1998.07.25
- "HH:mm:ss" // 24小时制时间, 15:08:02
- "HH:mm" // 24小时制时间, 15:08
- "yyyy-MM-dd, HH:mm:ss" // 日期+24小时制时间
- "yyyy-MM-dd, hh:mm:ss a" // 日期+12小时制时间+上/下午

- **format(Date d)**方法：将此种格式应用于给定的日期
aSimpleDateFormat.format(aDate);
- **applyPattern(String pattern)**方法：指定/改变格式

模式字母	日期或时间元素	表示	示例
G	Era 标志符	<u>Text</u>	AD
y	年	<u>Year</u>	1996; 96
M	年中的月份	<u>Month</u>	July; Jul; 07
w	年中的周数	<u>Number</u>	27
W	月份中的周数	<u>Number</u>	2
D	年中的天数	<u>Number</u>	189
d	月份中的天数	<u>Number</u>	10
F	月份中的星期	<u>Number</u>	2
E	星期中的天数	<u>Text</u>	Tuesday; Tue
a	Am/pm 标记	<u>Text</u>	PM
H	一天中的小时数 (0-23)	<u>Number</u>	0
k	一天中的小时数 (1-24)	<u>Number</u>	24
K	am/pm 中的小时数 (0-11)	<u>Number</u>	0
h	am/pm 中的小时数 (1-12)	<u>Number</u>	12
m	小时中的分钟数	<u>Number</u>	30
s	分钟中的秒数	<u>Number</u>	55
S	毫秒数	<u>Number</u>	978
z	时区	<u>General time zone</u>	Pacific Standard Time; PST; GMT-08:00
Z	时区	<u>RFC 822 time zone</u>	-0800

注意字母区
分大小写



文本包：SimpleDateFormat类

● 示例

```
Date date=new Date();
```

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

```
____.println(sdf.format(date)); // 格式化时间
```

2016-06-01

```
sdf.applyPattern("HH:mm:ss"); // 改变格式
```

```
____.println(sdf.format(date));
```

20:17:37

```
sdf.applyPattern("hh:mm:ss a");
```

```
____.println(sdf.format(date));
```

08:17:37 下午

```
sdf.applyPattern("yyyy.MM.dd, hh:mm a");
```

```
____.println(sdf.format(date));
```

2016.06.01,08:17 下午



文本包：DecimalFormat类

- 使用已定义的格式对十进制数进行格式化
- 构造方法：以一指定格式的字符串作为参数

DecimalFormat (String pattern);

— pattern用于指定格式，常见

- "000" // 123->123, 12->012
- "0.00" // 1.234->1.23, 1.2->1.20, 1->1.00
- "0.0#" // 1.234->1.23, 1.2->1.2, 1->1.0
- "#0.0%" // 0.1234->12.3%, 0.01->1.0%
- "#,##00.0#" // 12345.6->12,345.6 1.23456->01.23 12->12.0
- "###,###,###" // 1234567->1,234,567
- "0.00E0" // 1234567->1.23E6

- **format(Object ob)**方法：将此种格式应用于给定的十进制数
aDecimalFormat.format(aDecimal);
- **applyPattern(String pattern)**方法：指定/改变格式



文本包：DecimalFormat类

● 示例

`DecimalFormat df=new DecimalFormat("000");` //格式：至少3位整数（不够则左边补0）

```
System.out.println("12->" + df.format(12));  
____.println("1234->" + df.format(1234));  
____.println("12.34->" + df.format(12.34));
```

12->012
1234->1234
12.34->012

`df.applyPattern("0.00");` //格式：至少1位整数（不够则左边补0）和两位小数（不够则右边补0）

```
____.println("1.234->" + df.format(1.234));  
____.println("12.3->" + df.format(12.3));  
____.println("-12.3->" + df.format(-12.3));
```

1.234->1.23
12.3->12.30
-12.3->-12.30

`df.applyPattern("0.0#");` //格式：至少1位整数（不够则左边补0），和两位小数（至少1位，不够则右边补0）

```
____.println("1.234->" + df.format(1.234));  
____.println("12->" + df.format(12));
```

1.234->1.23
12->12.0



文本包：DecimalFormat类

● 示例（续）

df.applyPattern("#,##,###"); //格式：千分位用逗号分隔

System.out.println("123->" + df.format(123));

123->123

__.println("1234->" + df.format(1234));

1234->1,234

__.println("1234567->" + df.format(1234567));

1234567->1,234,567

df.applyPattern("#,##,##0.0"); //格式：千分位用逗号分隔

__.println("123.456->" + df.format(123.456));

123.456->123.5

__.println("1234->" + df.format(1234));

1234->1,234.0

__.println("1234567.89->" + df.format(1234567.89));

1234567.89->1,234,567.9

df.applyPattern("0.00E0"); //格式：科学计数法

__.println("123.456->" + df.format(123.456));

123.456->1.23E2

__.println("-1234->" + df.format(-1234));

-1234->-1.23E3

__.println("1234567.89->" + df.format(1234567.89));

1234567.89->1.23E6

__.println("0.01234->" + df.format(0.01234));

0.01234->1.23E-2



Java中如何使用JSON

- Java中并没有内置JSON的解析，因此使用JSON需要借助第三方类库。
- 下面是几个常用的JSON解析类库：
 - Gson: 谷歌开发的JSON库，功能十分全面。
 - FastJson: 阿里巴巴开发的JSON库，性能十分优秀。
 - Jackson: 社区十分活跃且更新速度很快。
- FastJson库中，JSON对象与字符串的相互转化

方法	作用
JSON.parseObject()	从字符串解析JSON对象
JSON.parseArray()	从字符串解析JSON数组
JSON.toJSONString(obj/array)	将JSON对象或JSON数组转化为字符串

下载FastJson <https://github.com/alibaba/fastjson>



Java中如何使用JSON

- 示例：将数据装入JSON对象

// 将数据装入JSON对象

```
JSONObject json = new JSONObject();
```

```
json.put("string", "China");
```

```
json.put("int", 2);
```

```
json.put("boolean", true);
```

// array

```
List<Integer> integers = Arrays.asList(1, 2, 3);
```

```
json.put("list", integers);
```

// null

```
json.put("null", null);
```

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

```
{"boolean":true,"string":"China","list":[1,2,3],"int":2}
```



Java中如何使用JSON

- 示例：从JSON对象提取数据

```
JSONObject json = JSONObject.parseObject(  
    "{\"boolean\":true,\"string\":\"China\",\"list\":[1,2,3],\"int\":2}");
```

```
String s = json.getString("string");
```

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

```
int i = json.getIntValue("int");
```

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

```
boolean b = json.getBooleanValue("boolean");
```

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

```
// list
```

```
List<Integer> list = JSONObject.parseArray(  
    json.getJSONArray("list").toJSONString(), Integer.class);
```

```
list.forEach(System.out::println);
```

```
// null
```

```
System.out.println(json.getString("null"));
```

```
China  
2  
true  
1  
2  
3  
null
```



习 题

1. 用StringTokenizer类或者Scanner类统计英文句子中的单词个数，并分别输出每个单词。
2. 建立两个类及并进行相应测试：
 - ① 建立一个学生，数据成员包括学号、姓名、性别，成绩等，方法成员包括构造方法、set/get方法、toString()方法等。
 - ② 建立一个管理学生对象的类StudentManager。在其中定义向量（或列表）成员，用于存储学生对象；提供若干方法，分别用于向量（或列表）进行学生对象的插入、移除、修改、排序（按成绩升序）、浏览、查找（按姓名）、统计人数等操作。



习 题

提示：

■ StudentManager类的关键方法设计举例：

public void add(Student stu)

public Student remove(int i) ---i超出索引，则返回null

public Student remove(Student stu) ---删除第一个匹配的元素

public Student get(int i) --- i超出索引，则返回null

public void set(int i, Student stu)

public void sort()

public void display()

public Student search(String name) ---未找到则返回null

public int size()

■ 排序功能可以通过Collections类实现，此时需要用到Comparable接口或Comparator接口

■ 可以采用枚举或迭代接口实现对向量（或列表）中的学生对象的浏览输出或查找。

■ 在StudentManager类添加main方法进行各种操作测试。



谢谢大家!