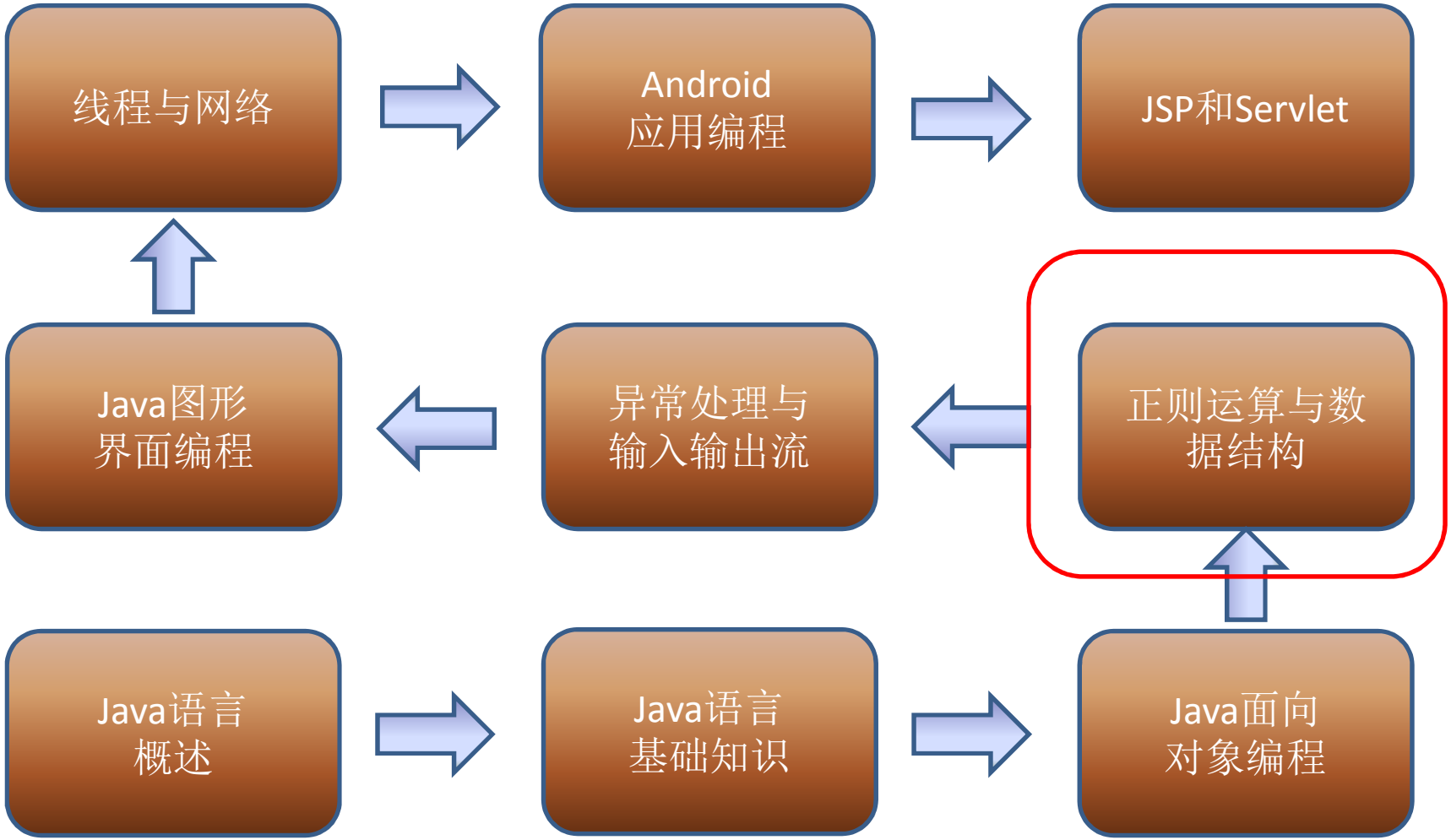
第四讲 正则运算与数据结构

课程内容安排


线程与网络
Android
应用编程
JSP和Servlet
Java图形
界面编程
异常处理与
输入输出流
正则运算与数
据结构
Java语言
概述
Java语言
基础知识
Java面向
对象编程

课前思考

- 如何统计一个文档中的特定字符串出现的次数？例如一个网页中“中国好声音”出现的次数？
- [A-Za-z]+[0-9] 表示什么字符串？
- 0\d\d-\d\d\d\d\d\d\d\d是否表示了中国电话号码的模式？ 0\d{2}-\d{8} 呢？
- Java中有没有哈希表？
- 如何对数组排序？
- 如何将对象进行排序？

正则表达式

例子：统计字符串出现的次数

统计下面字符串中“DCOM”出现的次数，并把”DCOM”替换成“Java”。

“解决 DCOM 的问题主要是解决程序配置和部署的问题。由于 DCOM 涉及到在多台计算机上运行的程序，所以潜在的问题比在单机上使用COM时要大。其他可能需要解决的问题包括程序和网络协议之间的安全机制。因为在默认情况下 COM 安全是打开的，所以只要试图访问COM对象的COM程序或客户程序启动COM对象，就会开始进行安全检查。”

正则表达式定义

- 正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。
- 给定一个正则表达式和另一个字符串，我们可以达到如下的目的：
  - 1. 给定的字符串是否符合正则表达式的过滤逻辑（称作“匹配”）；
  - 2. 可以通过正则表达式，从字符串中获取我们想要的特定部分。

正则表达式的特点

- 1. 灵活性、逻辑性和功能性非常的强；
- 2. 可以迅速地用极简单的方式达到字符串的复杂控制。
- 3. 对于刚接触的人来说，比较晦涩难懂。

由于正则表达式主要应用对象是文本，因此它在各种文本编辑器场合都有应用，小到著名编辑器EditPlus，大到Microsoft Word、Visual Studio等大型编辑器，都可以使用正则表达式来处理文本内容。

```
import java.util.regex.*;
public class StringState {
   public static void main(String args[]){
      String s="解决 DCOM 的问题主要是解决程序配置和部署的问题。由于
     DCOM 涉及到在多台计算机上运行的程序，所以潜在的问题比在单机上使
     用COM时要大。其他可能需要解决的问题包括程序和网络协议之间的安全
     机制。因为在默认情况下 COM 安全是打开的，所以只要试图访问COM对
     象的COM程序或客户程序启动COM对象，就会开始进行安全检查。";
       Pattern p=Pattern.compile("DCOM");
       Matcher m=p.matcher(s);
       int count=0;
       System.out.println(s);
        while(m.find())     {count++;System.out.println(m.group());}
       System.out.println("The number is: "+count);
       System.out.println("after replace DCOM with Java");
       s=m.replaceAll("Java");
       System.out.println(s);
   }
}
```

Regular Expressions

- Regular expressions are an extremely useful tool for manipulating text, heavily used
  - in the automatic generation of Web pages (网页自动生成)
  - in the specification of programming languages, (编程语言规格)
  - in text search. (文本搜索)
- generalized to patterns that can be applied to text (or strings) for string matching.
- A pattern can either match the text (or part of the text), or fail to match
  - If matching, you can easily find out which part.
  - For complex regular expression, you can find out which parts of the regular expression match which parts of the text
  - With this information, you can readily extract parts of the text, or do substitutions in the text

Java Regular Expressions

- since jdk 1.4, Java has a regular expression package: java.util.regex
  - java.util.regex.Pattern
  - java.util.regex.Matcher

A first example

- The regular expression "[a-z]+"

  will match a sequence of one or more lowercase letters.
  - [a-z] means any character from a through z, inclusive
  - + means “one or more”

Some simple patterns

abc
- exactly this sequence of three letters

[abc]
- any *one* of the letters a, b, or c

[^abc]
- any character *except* one of the letters a, b, or c

[ab^c]
- a, b, ^ or c.
- (immediately within [, ^ mean "not," but anywhere else mean the character ^)

[a-z]
- any *one* character from a through z, inclusive

[a-zA-Z0-9]
- any *one* letter or digit

Sequences and alternatives

- If one pattern is followed by another, the two patterns must match consecutively
  - Ex: [A-Za-z]+ [0-9] will match one or more letters immediately followed by one digit

- The vertical bar, |, is used to separate alternatives
  - Ex: the pattern abc|xyz will match either abc or xyz

元字符

.	一个点表示任意字符，除了换行符
\d	一个数字: [0-9]
\D	一个非数字: [^0-9]
\s	任意的空白符，包括空格，制表符(Tab)，换行符，中文全角空格等: [\t\n\x0B\f\r]
\S	非空白符: [^\s]
\w	用于单词的字符，例如字母或数字或下划线或汉字等: [a-zA-Z_0-9]
\W	不是用于单词的字符，与\w相反: [^\w]

边界匹配的元字符

- 下列模式用于匹配特定的位置:
  - ^ 字符串的开始
  - $ 字符串的结束
  - \b 单词的开头或者结尾
  - \B 不是单词的开头或者结尾
  - \A the beginning of the input (can be multiple lines)
  - \Z the end of the input except for the final terminator, if any
  - \z the end of the input
  - \G the end of the previous match

表示数量的限定符

*	重复零次或者更多次
+	重复一次或者更多次
？	重复零次或者一次
{n}	重复n次
{n,}	重复n次或者更多次
{n,m}	重复n到m次

正则表达式若干例子

- 验证QQ号是否为5位到12位数字？

  ^\d{5,12}$

- 刚好是10个字符的单词？

  \b\w{10}\b

- 匹配字符串中的第一个单词？

  ^\w+

分组

- 如何重复多个字符？
- 用小括号来指定**子表达式**(也叫做**分组**)，然后指定这个子表达式的重复次数
- 例如(\d{1,3}\.){3}\d{1,3}是一个简单的IP地址匹配表达式，比如 166.111.8.28

Pattern match in Java

- First, you must *compile* the pattern

```
import java.util.regex.*;
Pattern p = Pattern.compile("[a-z]+");
```

- Next, create a *matcher* for a target text by sending a message to your pattern

```
Matcher m = p.matcher("The game is over");
```

- Notes:
  - Neither Pattern nor Matcher has a public constructor;
    - use static Pattern.compile(String regExpr) for creating pattern instances
    - using Pattern.matcher(String text) for creating instances of matchers.
  - The matcher contains information about *both* the pattern *and* the target text.

Pattern match in Java (continued)

After getting a matcher m,

- use m.match() to check if there is a match.
  - returns true if the pattern matches the entire text string, and false otherwise.
- use m.lookingAt() to check if the pattern matches a prefix of the target text.
- m.find() returns
  - true iff the pattern matches any part of the text string,
  - If called again, m.find() will start searching from where the last match was found
  - m.find() will return true for as many matches as there are in the string; after that, it will return false
  - When m.find() returns false, matcher m will be *reset* to the beginning of the text string (and may be used again).

Finding what was matched

- *After a successful match,*
  - m.start() will return the index of the first character matched
  - m.end() will return the index of the last character matched, *plus one*
- If no match was attempted, or if the match was unsuccessful,
  - m.start() and m.end() will throw an IllegalStateException (a RuntimeException).
- Example:
  - "The game is over".substring(m.start(), m.end()) will return exactly the matched substring.

A complete example

```
import java.util.regex.*;

public class RegexTest {
    public static void main(String args[]) {
        String pattern = "[a-z]+";
        String text = “The game is over";
        Pattern p = Pattern.compile(pattern);
        Matcher m = p.matcher(text);
        while (m.find()) {
            System.out.print(text.substring(m.start(),
m.end()) + "*");
        }
    }
}
```

Output: he*game*is*over*

Additional methods

If m is a matcher, then

- m.replaceFirst(*newText*)
  - returns a new String where the first substring matched by the pattern has been replaced by *newText*
- m.replaceAll(*newText*)
  - returns a new String where every substring matched by the pattern has been replaced by *newText*
- m.find(*startIndex*)
  - looks for the next pattern match, starting at the specified index
- m.reset() resets this matcher
- m.reset(*newText*) resets this matcher and gives it new text to examine.

Spaces

- One important thing to remember about spaces (blanks) in regular expressions:
  - *Spaces are significant!*
  - I.e., A space is an ordinary char and stands for itself, a *space*
  - So It's a *bad idea* to put spaces in a regular expression just to make it look better.
- Ex:
  - Pattern.compile("a b+").matcher("abb"). matches()
  - return false.

实例应用

- .验证电话号码：（"^(\d{3,4}-)\d{7,8}$"）正确格式：xxx/xxxx-xxxxxxx/xxxxxxxx;
- 验证手机号码："^1[3|4|5|8][0-9]\d{8}$"
- 匹配帐号是否合法(字母开头，允许5-16字节，允许字母数字下划线)：^[a-zA-Z][a-zA-Z0-9_]{4,15}$
- 只能输入m~n位的数字："^\d{m,n}$“
- 只能输入由26个大写英文字母组成的字符串："^[A-Z]+$"

Summary

- Regular expressions are *not* easy to use at first
  - It's a bunch of punctuation, not words
  - it takes practice to learn to put them together correctly.
- Regular expressions form a sublanguage
  - It has a different syntax than Java.
  - It requires new thought patterns
  - can't *use* regular expressions directly in java; you have to create Patterns and Matchers first.
- Regular expressions is powerful and convenient to use for string manipulation
  - It is worth learning !!

集合框架 Collection

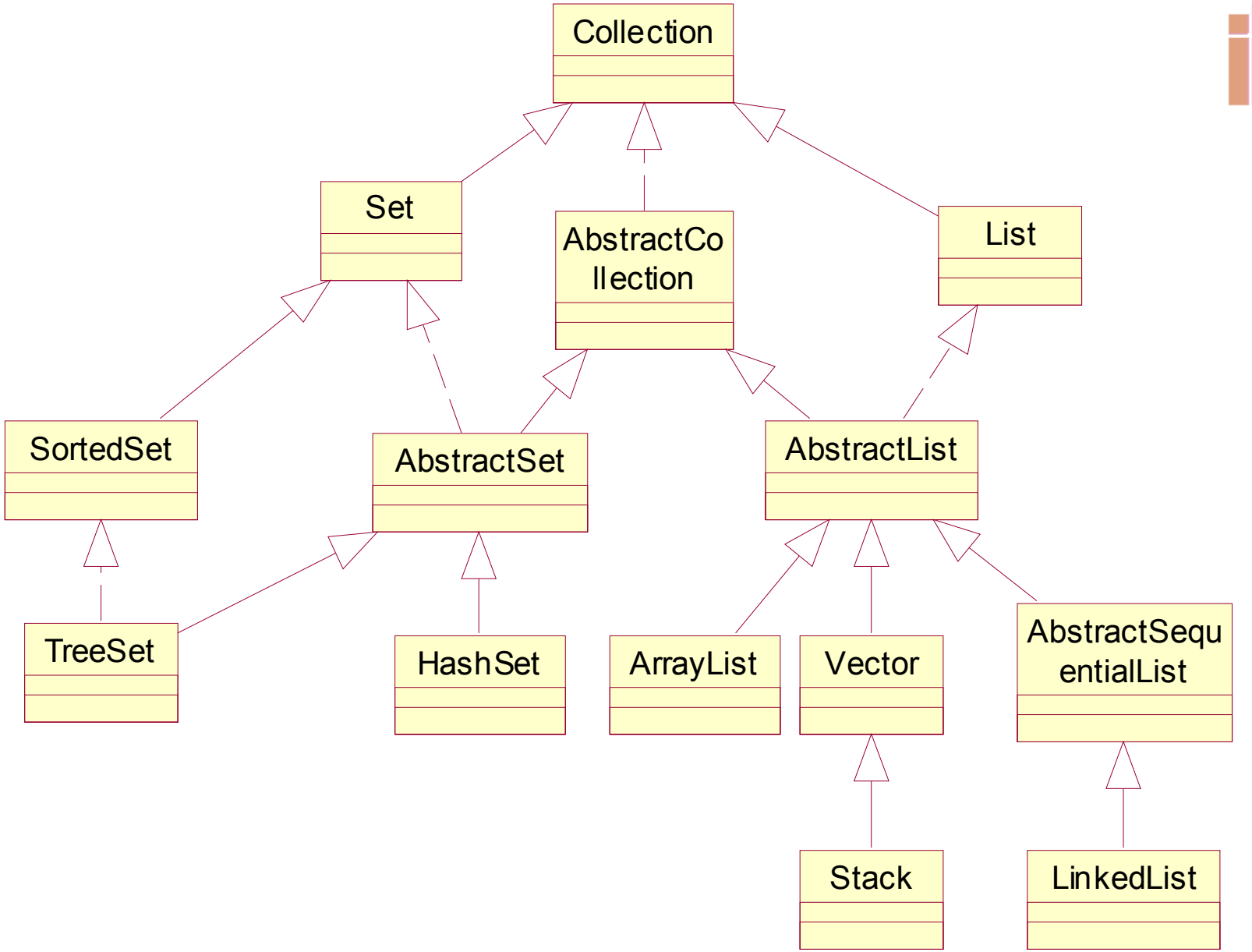
集合框架Collection

一个集合是代表一组对象的对象，集合中的对象成为它的元素。集合主要用来处理各种类型的对象的聚集，每一个对象都具有一定的数据类型。

集合包含三个重要的接口：

Collection、Set、List

- Collection：对象的无序聚集，允许重复（即可以存在相同的对象）
- Set：对象的无序聚集，不允许重复（相同的对象只能在集合中出现一次）
- List：有序的对象聚集，允许重复。



(实线箭头代表继承、虚线箭头代表实现）

ArrayList

- ArrayList相当于一个长度可变的数组，但是其中的每个元素都是对象。
- ArrayList中存放的元素是按照一定顺序的，元素的顺序按照添加的顺序排列，而且允许重复元素的存在。

```
import java.util.*;
public class ArrayListDemo {
public static void main(String[] args) {
        List l = new ArrayList();
        l.add("Hello");
        l.add("World");
        l.add(new Character('我'));
        l.add(new Integer(23));
        l.add("Hello");
        String[] as = {"W","o","r","l","d"};
        l.add(as);
        l.add(new Integer(23));
        System.out.println(l);
    }
}
```

运行结果

[Hello,world,我,23,Hello,[Ljava.lang.String;@20c10f,23]

HashSet

- 元素没有顺序，显示结果与我们加入元素的顺序没有必然的联系。
- HashSet不允许重复的元素。
- HashSet基于哈希表，当其中存放的元素数量较大的时候，其访问速度会比线性列表快。

```
import java.util.*;
public class HashSetDemo {
    public static void main(String[] args) {
        Set s = new HashSet();
        s.add("Hello");
        s.add("World");
        s.add(new Character('我'));
        s.add(new Integer(23));
        s.add("Hello");
        String[] as = {"W","o","r","l","d"};
        s.add(as);
        s.add(null);
        s.add(new Integer(23));
        s.add(null);
        System.out.println(s);
    }
}
```

运行结果

[World,我,[Ljava.lang.String;@47e553,23,null,Hello]

如何访问集合中的元素

- 通过使用接口Iterator来访问集合中的元素，该接口定义了如下方法：

  hasNext( );

  next( );

  remove( );

- 对于Set来说，通过Iterator获取的元素顺序不是一定的。

Iterator


```
import java.util.*;
public class SetIteratorDemo {
    public static void main(String[] args) {
        Set s = new HashSet();
        s.add("Hello");
        s.add("World");
        s.add(new Character('我'));
        s.add(new Integer(23));
        s.add(new Double(23.12));
        s.add(null);
        Iterator i = s.iterator();
        System.out.print("[ ");
```

```
        boolean first = true;
        while(i.hasNext()) {
                System.out.print(i.next()+"  ");
                if ( first ) {
                        i.remove();
                        first = false;
                }
        }
        System.out.println("]");
        i = s.iterator();
        System.out.print("[ ");
        while(i.hasNext())
                System.out.print(i.next()+"  ");
        System.out.println(")");
    }
}
```

运行结果

[World 我 23.12 23 null Hello]

[我 23.12 23 null Hello]

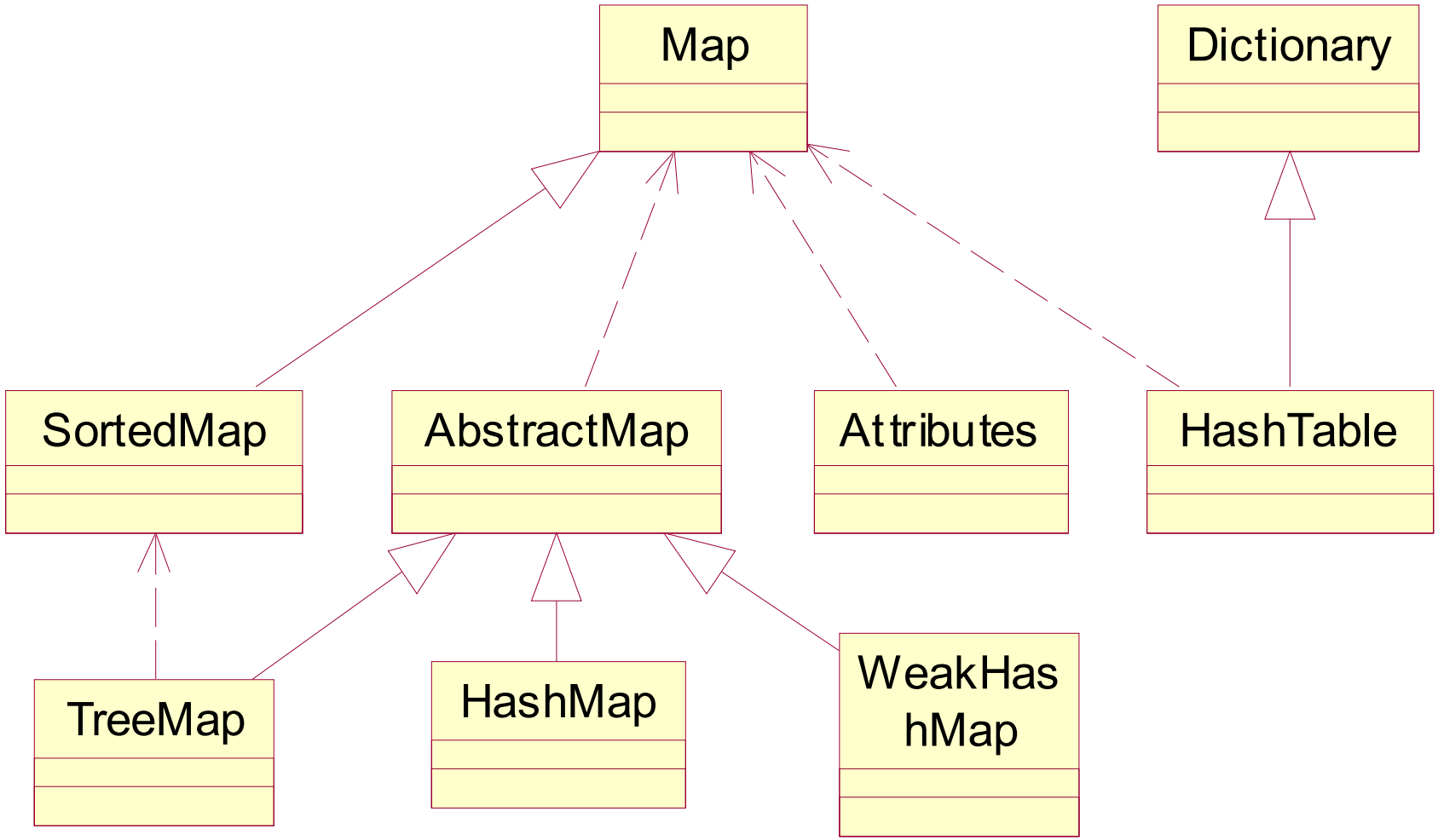
映射（Map）

- 映射实际上是一系列的“键—值”，也就是一个“键（Key）—值（Value）”对的集合，可以通过一个键找到相应的值。其中“键”和“值”可以是任意类型的对象（Object）。
- Map接口的方法：

put（）：加入一个“键—值”对，如果加入的键已经存在，则新插入的值将取代旧的值。

get（）：从映射中取出指定键所对应的值。

- keySet( )：返回一个Set，其中包含了映射中的所有键(key）。
- values( )：返回一个Collection，其中包含了本映射中所有的值（value）。


Map
Dictionary
SortedMap
AbstractMap
Attributes
HashTable
TreeMap
HashMap
WeakHas
hMap

HashMap


```
import java.util.*;
import java.io.*;
public class HashMapDemo {
   public static void main(String[] args) {
      HashMap hm = new HashMap();
      hm.put("1",new Integer(1));
      hm.put(new File("test.txt"),"2");
      hm.put(new Byte((byte)3),System.out);
      hm.put(null,"nothing");
```


```
        Set s = hm.keySet();
        Iterator i = s.iterator();
        while ( i.hasNext() ) {
                Object k = i.next();
                Object v = hm.get(k);
                System.out.print("  "+k+"="+v);
                if ( v instanceof PrintStream )
                        ((PrintStream)v).print("(It's Me!)");
        }
    }
}
```

运行结果

1=1 3=java.io.PrintStream@310d42(It’s Me!)
null=nothing test.txt=2

注意几点：

- 输出结果的顺序与加入键值对的顺序没有必然联系。
- 先获取Key的一个Set，然后用反复器访问Set取出Key，再用Key去得到Value，这是访问Map的所有元素的一种方法。
- 键允许为null，值也允许为null

HashTable


```
import java.util.*;
import java.io.*;
public class HashTableDemo {
    public static void main(String[] args) {
        Hashtable ht = new Hashtable();
        Hashtable tempHt = new Hashtable();
        tempHt.put("1","one");
        tempHt.put("two",new Long(2));
        ht.put("1",new Integer(1));
```


```
        ht.put(new File("test.txt"),"2");
        ht.put(new Byte((byte)3),"three");
        ht.put("four",new Double(4.0));
        ht.put(tempHt,"hashtable");
        Enumeration e = ht.keys();
        while ( e.hasMoreElements() ) {
                Object k = e.nextElement();
                Object v = ht.get(k);
                System.out.print(k+"="+v+"  ");
        }
    }
}
```

运行结果

1=1 four=4.0 3=three {two=2,1=one}=hashtable
test.txt=2

TreeMap

- 输出结果按照键（key）从小到大排列，因此要求key实现Comparable接口。

TreeMap


```
import java.util.*;
import java.io.*;
public class TreeMapDemo {
   public static void main(String[] args) {
        TreeMap tm = new TreeMap();
        Hashtable tempHt = new Hashtable();
        tempHt.put("2",new Long(2));
        tempHt.put("1","one");
        tm.put("6.Hashtable",tempHt);
```


```
        tm.put("3.Integer",new Integer(1));
        tm.put("5.Double",new Double(4.0));
        tm.put("4.File",new File("test.txt"));
        Set s = tm.keySet();
        Iterator i = s.iterator();
        while ( i.hasNext() ) {
                Object k = i.next();
                Object v = tm.get(k);
                System.out.print(k+"="+v+"  ");
        }
    }
}
```

运行结果

3.Integer=1 4.File=test.txt 5.Double=4.0
6.Hashtable={2=2,1=one}

排序问题

- 对简单类型的数组的排序：Arrays.sort(byte[] a);
- 对象数组排序： Arrays.sort(Object[] a);

  或者Arrays.sort(Object[] a,Comparator c);
- 对于List排序，可以使用Collections的sort( )方法来排序。

ArraySort


```
import java.util.*;
public class ArraySortDemo {
    public static void main(String[] args) {
        int[] iArray = {13,1,-1,78,0,-44,98,101, -789,15,4};
        System.out.println("Original:");
      showArray(iArray);
      System.out.println("Sorted:");
      Arrays.sort(iArray);
      showArray(iArray);
}
```


```
private static void showArray(int[] iArray) {
   System.out.print("[ ");
   int i;
   for ( i=0 ; i<iArray.length ; i++ ) {
        System.out.print(iArray[i]+" ,");
   }
   System.out.println(" ]");
}
}
```

运行结果

Original:

[13,1,-1,78,0,-44,98,101,-789,15,4]

Sorted:

[-789,-44,-1,0,1,4,13,15,78,98,101]

对象数组比较


```
public class MyObject implements Comparable {
    private String name;
    public MyObject() {
        this("Object");
    }
    public MyObject(String s) {
        name = s;
    }
```


```
    public int compareTo(Object o) {
        String moName = ((MyObject)o).getName();
        if ( this.name.length() > moName.length() )
                return 1;
        else if ( this.name.length() < moName.length() )
                return -1;
        else    return 0;
    }
    public String getName() {
        return name;
    }
    public String toString() {
        return getName();
    }
}
```

ListSortDemo.java


```
import java.util.*;
public class ListSortDemo{
public static void main(String args[]){
        List l=new ArrayList();
        l.add(new MyObject("Congratulation"));
        l.add(new MyObject("where"));
        l.add(new MyObject("I"));
        l.add(new MyObject("Student"));
```


```
        l.add(new MyObject("Form"));
        l.add(new MyObject("Is"));
        l.add(new MyObject());
        l.add(new MyObject("JavaBean"));
      System.out.println("Original :");
      showArray(l);
      Collections.sort(l);
      System.out.println("Sorted :");
      showArray(l);
}
```


```
    private static void showArray(List l){
        ListIterator li=l.listIterator( );
        System.out.print("[");
        while(li.hasNext( )){
                System.out.print(li.next()+",  " );
        }
        System.out.println("\b]");
    }
}
```

运行结果

Original:

[Congratulation，Where，I，Student，Form，Is，Object，JavaBean]

Sorted：

[I，Is，Form，Where，Object，Student，JavaBean，Congratulation]

Java 5中的新增特性

泛型Generics

- 泛型（Generic type 或者 generics）是对 Java 语言的类型系统的一种扩展，以支持创建可以按类型进行参数化的类。
- 该代码不使用泛型：

```
List li = new ArrayList();
li.put(new Integer(3));
Integer i = (Integer) li.get(0);
```

该代码使用泛型：

```
List<Integer> li = new ArrayList<Integer>();
li.put(new Integer(3));
Integer i = li.get(0);
```

例子1

```
Vector v = new Vector();
v.add(new Integer(4));
OtherClass.expurgate(v);
...
static void expurgate(Collection c) {
        for (Iterator it = c.iterator();
                it.hasNext();)
        /* ClassCastException */
        if (((String)it.next()).length() == 4)
        it.remove();
}
```

- 问题: Collection元素类型
  - 编译器无法帮助验证类型
  - 赋值必须进行强制类型转换
  - 有可能产生运行时的错(**ClassCastException**)
- 解决办法:
  - 告诉编译器元素类型
  - 让编译器来做类型的匹配和转换
  - 保证运行成功

使用泛型修饰对象

- 创建与特定类型关联的泛型对象实例

```
Vector<String> x = new Vector<String>();
x.add(new Integer(5)); // 编译器错误
```

- 类型变量定义在<>之间
- 不同类型变量之间用逗号分隔

泛型的兼容性问题

- 与现成代码兼容

```
/* Old class */
public Vector getVector() { return new Vector(); }

/* New class */
public Vector<String> s = oldCode.getVector();
/* 产生编译器警告, 注意不是错误 */
```

例子2

- 定义二元关系对：

```
public class IntPair {
        int first; int second;
        public NumberPair(int f, int s) {
                first = f; second = s;
        }
}
public class StringPair {
        String first; String second;
        public StringPair(String f, String s) {
                first = f; second = s;
        }
}
```

减少繁琐

- 泛型修饰的类

```
public class Pair<F, S> {
        F first; S second;
        public Pair(F f, S s) {
                first = f; second = s;
        }
}
Pair<String, String> stp = new Pair<String,String>(“Xu", “Bin");
Pair<Shoe, Shoe> shp =
                new Pair<Shoe,Shoe>(LeftShoe, RightShoe);
```

例子3

- 如何打印任何Collection的内容?
- 一种解决方案

```
void printCollection(Collection<Object> c) {
        for (Object o : c)
        System.out.println(o);
}
```

错误!

- 如果传入一个Collection<String>编译器将报错
- Collection<Object>不是所有Collection的父类

泛型通配符

```
void printCollection(Collection<?> c) {
        for (Object o : c)
        System.out.println(o);
}
```

- ? 是通配符
- Collection<?>可以匹配任意类型的Collection

泛型通配符

- 用作参数的类型不能像普通对象那样继承
- 通配符可以规定类型上限

```
public void drawAll(List<? extends Shape>s) {
        …
}
List<Circle> c = getCircles();
drawAll(c);
List<Triangle> t = getTriangles();
drawAll(t);
```

例子4

- 如何将任意类型的数组元素复制到相应集合中

```
static void aToC(Object[] a, Collection<?> c) {
        for (Object o : a)
        c.add(o);
}
```

泛型方法

- 问题：如何对方法使用泛型？

```
static void aToC(Object[] a, Collection<?> c) {
for (Object o : a)
c.add(o); /* 编译器错误 */
}
```

- ? 表示未知类型，不是任何一个确定的类型

泛型方法

- 正确的解决方法
  - 用类型参数来定义泛型方法
  - 在方法调用发生时编译器匹配具体类型

```
static <T> void aToC(T[] a, Collection<T> c) {
        for (T o : a)
        c.add(o); /* 不会产生编译错误 */
}
String[] sa = new String[100];
Collection<Object> co = new ArrayList<Object>();
Collection<String> cs = new ArrayList<String>();
aToC(sa, cs); /* T 匹配为类型 String */
aToC(sa, co); /* T 匹配为类型 Object */
```

基本类型的自动装箱

- 问题:
  - 基础类型与其包装类之间的相互转换
  - 比如当需要把基础类型加入集合的时候
- 解决方案：让编译器帮忙！

```
Byte byteObj = 22; // 包装转换
int i = byteObj // 解包转换
ArrayList al = new ArrayList();
al.add(22); // 包装转换
```

For循环增强(foreach)

- 问题
  - 集合遍历容易出错
  - 通常情况下, iterator只被用来得到一个元素
- 解决方法: 让编译器帮忙!
- 新的for循环句法:

  for (variable : collection)

对集合与数组起作用

Loop循环增强举例

- 早先的代码

```
void cancelAll(Collection c) {
        for (Iterator i = c.iterator(); i.hasNext(); ){
        TimerTask task = (TimerTask)i.next();
        task.cancel();
        }
}
```

- 如今的代码

```
void cancelAll(Collection<TimerTask> c) {
        for (TimerTask task : c)
        task.cancel();
}
```

可变参数

- 问题
  - 含有可变参数个数的方法
  - 可以通过数组来传递，但是不方便
- 解决方法: 让编译器帮你！
- 新的语法:

public static String format (String fmt, Object... args);

静态导入

- 问题:
  - 外部静态引用必须要引用类名
- 解决方法: 新的导入语法
- **import static TypeName.Identifier;**
- **import static Typename.*;**
- 静态方法和枚举类型同样适用
- 比如**Math.sin(x)**变成 **sin(x)**

小结

- 正则表达式：元符号、边界匹配元字符、表示数量的限定符
- 集合框架：Collection、Set、List
- 泛型

谢谢！