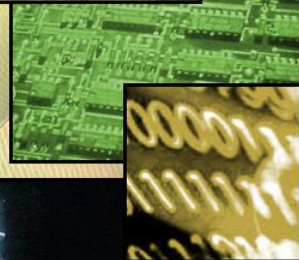
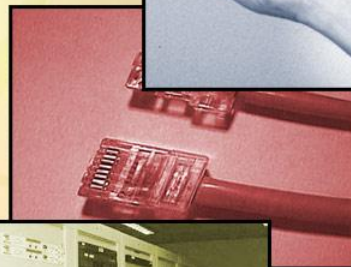


# JAVA程序设计



# 第7章 对象的容纳





# 本章目录

## 7.1 数组

7.1.1 数组创建、初始化和使用

7.1.2 多维数组

7.1.3 数组和数组引用

7.1.4 数组工具类 **Arrays**

7.1.5 对象比较

## 7.2 枚举

7.2.1 枚举的定义

7.2.2 枚举的使用

## 7.3 容器

7.3.1 List

7.3.2 泛型

7.3.3 Set

7.3.4 Map

7.3.5 **Collections** 工具类

7.3.6 容器的选择





# 7.1 数组

- ◆ Java数组的概念
- ◆ 数组创建、初始化和使用
- ◆ 多维数组
- ◆ `java.util.Arrays`工具类
- ◆ 对象比较用法



## 7.1 数组 (Arrays)

◆ **数组**是一种数据结构，用来存储相同类型(对象，基本类型)值的集合。

◆ 数组元素的访问：

**数组名[索引值]**

➤ 如： `a[10]`， `b[0]`， `c[c.length-1]`



## 7.1 数组 (Arrays)

◆ Java数组的索引（下标）从0开始编号。

➤ `a[0]`代表数组a中第一个元素

◆ 数组中除了数组元素之外，还存在唯一一个可被访问的属性`length`，记录了数组中元素的个数。

➤ `a[a.length-1]`代表数组中最后一个元素



# 1.数组的创建、初始化和使用

## (1) 数组的声明

### ◆ 声明数组变量有两种形式:

类型[ ] 数组名; //int a[];

类型 数组名[ ]; //int[] a;

➤ “[]”表示声明的变量是一个数组类型。

➤ “类型”可以是基本类型也可以是引用类型。

➤ 例: int a[ ]; String[ ] b;



## (2) 数组初始化

◆数组在声明之后还不可以使用，在使用数组之前必须为数组分配足够的系统资源，称为“初始化”。

### ◆静态初始化

➤`int a[]={21, 34, 7, 8, 10}; //a.length=5`

➤`int[] a = new int[] {1, 2, 3, 4};`





## ◆ 动态初始化

➤ `int[] a = new int[10];`     `//a.length=10`

### • 动态初始化可以使用变量作为参数:

➤ `int a[] = new int[len];` `//len是一个变量`

### • 这时数组元素值被自动赋为一个默认值:

- 数值型数据 (`int` , `double`...)     : 0
- 字符型数据 (`char`)     : `null`
- 布尔类型数据 (`boolean`)     : `false`
- 对象数据 (`String`, `Object`...)     : `null`



## ◆对象数组的初始化

- 数组内的元素若是对象，可使用以下语法：

```
类型[] 数组名=new 类型[] {  
    new构造方法(), ..., new构造方法() }
```

- 例

```
Apple a[] = new Apple[] { new Apple(),  
    new Apple() } ;
```

- 对于大型数组，一般配合循环语句来赋值



## 2. 多维数组

◆  $n$ 维数组中存放着多个  $n-1$  维数组的引用。

◆ 两种常用的声明方式：

```
int[][] a;    int a[][];
```

◆ 初始化：

```
int a[][] = { {1, 2, 3} , {4, 5, 6} };
```

```
int[][] a = new int[][] { {1, 2, 3} ,  
{4, 5, 6} };
```

◆ 动态初始化：

```
int a[][] = new int[2][3];
```



## 2. 多维数组

◆ 数组元素引用的数组长度不等的多维数组称为**不规则数组**:

```
➤ int[][] a = new int[2][];  
a[0]=new int[2]; a[1]=new int[3];
```

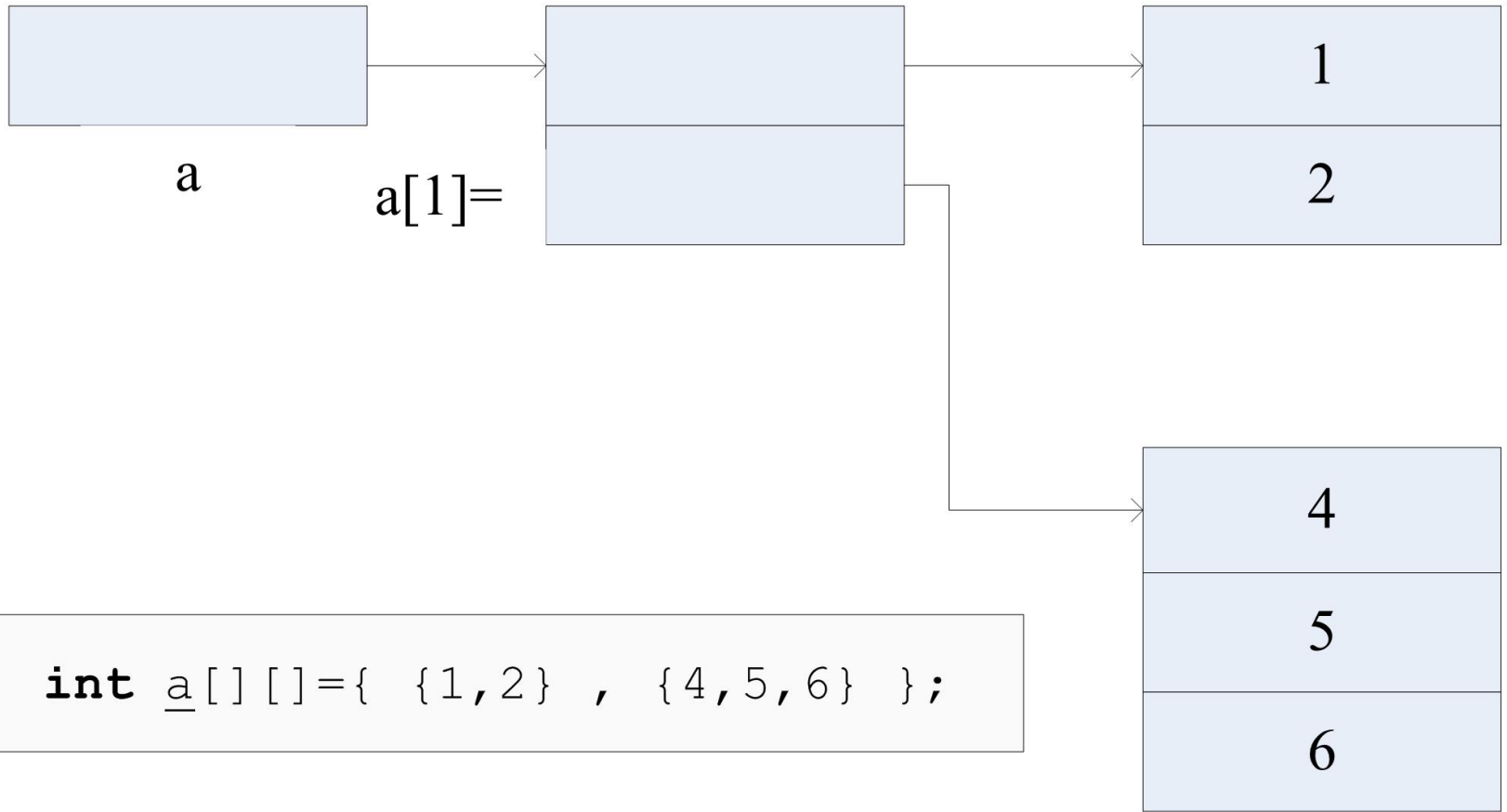
◆ 多维数组的**length**属性:

- a.length 指第一维数组的长度
- a[0].length 数组中第1个元素引用的数组的长度

◆ 多维数组的索引也是从**0**开始的



## 2. 多维数组







### 3. 数组与数组的引用

◆ Java数组是特殊的对象，数组变量存放一个数组对象的引用

◆ 可以将数组变量作为方法参数达到改变数组元素值的效果

```
public static void  
changeArrayValue(int[] para) {  
    para[0]=2;  
}
```



### 3. 数组与数组的引用

```
public class ArrayPara {  
    public static void changeArrayValue(int[] para) {  
        para[0] = 2;  
    }  
    public static void changeArrayRef(int[] para) {  
        int temp[] = {2,3,4,5};  
        para = temp;  
    }  
    public static void main(String[] args) {  
        int a[] = new int[] {99,100};  
        changeArrayValue(a);  
        System.out.println(a[0]);  
        a = new int[] {99,100};  
        changeArrayRef(a);  
        System.out.println(a[0]);  
    }  
}
```

2

99



## 4. 数组工具类Arrays

◆ Arrays类有一套static方法，提供了操作数组的实用功能

◆ Java.util.Arrays

方 法	描 述
copyOf()	将一个数组中的值拷贝到新的数组中
sort()	将一个数组中的值进行排序，默认是升序排列
binarySearch()	在已排好序的数组中查找特定值
equals()	判断两个数组是否相等
asList()	将数组重构为集合



# 示例：数组排序

```
int a[] = { 28, 14, 117, 66, 1, 70 };
```

```
Arrays.sort(a);
```

```
for (int i = 0; i < a.length; i++)
```

```
    System.out.print(a[i]+" , ");
```

```
1 , 14 , 28 , 66 , 70 , 117
```



# 示例：数组排序

```
public class Employee{

    int id;
    String name;

    public Employee(int id){
        this.id = id;
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Employee[] es = new Employee[]{new Employee(12), new Employee(2)};
        Arrays.sort(es);
    }
}
```

```
Exception in thread "main" java.lang.ClassCastException: Employee cannot be cast to java.lang.Comparable
    at java.util.ComparableTimSort.countRunAndMakeAscending(Unknown Source)
    at java.util.ComparableTimSort.sort(Unknown Source)
    at java.util.ComparableTimSort.sort(Unknown Source)
    at java.util.Arrays.sort(Unknown Source)
    at Employee.main(Employee.java:18)
```





## 5. 对象比较接口

◆当数组为对象类型时，为数组排序必须能判断数组中两个对象之间的大小。

◆java.lang.Comparable接口

➤int compareTo(Object o)

◆java.util.Comparator接口

➤int compare(Object o1, Object o2)



# 数组排序 comparable接口

```
1 import java.util.Arrays;
2 public class Employee implements Comparable{
3
4     int id;
5     String name;
6     public int compareTo(Object o){
7         Employee e = (Employee)o;
8         return this.id - e.id;
9     }
10    public Employee(int id){
11        this.id = id;
12    }
13    public static void main(String[] args) {
14        // TODO Auto-generated method stub
15        Employee[] es = new Employee[]{new Employee(12), new Employee(2)};
16        System.out.println(es[0].id);
17        System.out.println(es[1].id);
18        Arrays.sort(es);
19        System.out.println(es[0].id);
20        System.out.println(es[1].id);
21    }
22 }
```

<terminated> Employee [Java Application]  
12  
2  
2  
12|



## 5. 对象比较接口

◆当数组为对象类型时，为数组排序必须能判断数组中两个对象之间的大小。

◆java.lang.Comparable接口

➤int compareTo(Object o)

◆java.util.Comparator接口

➤int compare(Object o1, Object o2)



# 数组排序 comparator接口

```
import java.util.Arrays;
import java.util.*;
public class Employee{

    int id;
    String name;

    public Employee(int id){
        this.id = id;
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Employee[] es = new Employee[]{new Employee(12),new Employee(2),new Employee(8)};
        Arrays.sort(es, new DescComparator());
        System.out.println(es[0].id);
        System.out.println(es[1].id);
        System.out.println(es[2].id);
    }
}

class DescComparator implements Comparator{
    public int compare(Object o1, Object o2) {
        return ((Employee)o2).id - ((Employee)o1).id;
    }
}
```

<terminated

12

8

2





# 数组排序

A **Comparator**与**Comparable**两个接口都能使对象比较大小，但是**Comparator**拥有更优良的模块性，**Comparator**可以看成一种算法的实现，将算法和数据分离。

- 1、类的设计者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。
- 2、通过构造不同的比较器，可以实现多种排序标准，比如升序、降序等。

```
sort(float[] a) : void - Arrays
sort(int[] a) : void - Arrays
sort(long[] a) : void - Arrays
sort(Object[] a) : void - Arrays
sort(short[] a) : void - Arrays
sort(T[] a, Comparator<? super T> c) : void - Arrays
sort(byte[] a, int fromIndex, int toIndex) : void - Arrays
sort(char[] a, int fromIndex, int toIndex) : void - Arrays
```

Press 'Alt+/' to show Template Proposals

Sorts the specified array into ascending numerical order.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers  $O(n \log(n))$  performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Press 'Tab' from proposal table or click for focus





## 4. 数组工具类Arrays

◆ Arrays类有一套static方法，提供了操作数组的实用功能

◆ Java.util.Arrays

方 法	描 述
copyOf()	将一个数组中的值拷贝到新的数组中
sort()	将一个数组中的值进行排序，默认是升序排列
binarySearch()	在已排好序的数组中查找特定值
equals()	判断两个数组是否相等
asList()	将数组重构为集合



# 数组拷贝

```
1 import java.util.Arrays;
2 public class ArrayCopy {
3     public static void main(String[] args) {
4         // TODO Auto-generated method stub
5         int[] a = new int[]{1,2,3};
6         int[] b = Arrays.copyOf(a, a.length);
7         for(int i =0;i<b.length;i++)
8             System.out.println(b[i]);
9         int[] c = Arrays.copyOf(a, a.length*2);
10        for(int i =0;i<c.length;i++)
11            System.out.println(c[i]);
12    }
13 }
```

Problems | @ Jav  
<terminated> Arra  
1  
2  
3  
1  
2  
3  
0  
0  
0



# 数组拷贝

```
Arrays.c.copyOf(a, a.length*2);  
=0;i<c.  
n.out.pr
```

- copyOf(int[] arg0, int arg1) : int[] - Arrays
- copyOfRange(int[] arg0, int arg1, int arg2) : int[] - Arrays
- class : Class<java.util.Arrays>
- copyOf(boolean[] arg0, int arg1) : boolean[] - Arrays
- copyOf(byte[] arg0, int arg1) : byte[] - Arrays
- copyOf(char[] arg0, int arg1) : char[] - Arrays
- copyOf(double[] arg0, int arg1) : double[] - Arrays
- copyOf(float[] arg0, int arg1) : float[] - Arrays
- copyOf(long[] arg0, int arg1) : long[] - Arrays
- copyOf(short[] arg0, int arg1) : short[] - Arrays
- copyOf(T[] arg0, int arg1) : T[] - Arrays

Press 'Alt+/' to show Template Proposals



# 数组拷贝

<b>obj1</b>
<b>obj2</b>
<b>objn</b>





# 数组拷贝

- 判断如果超出当前数组存储空间了
- 申请存储空间更大的数组
- 将原来数据拷贝过去





# 数组拷贝

```
1 import java.util.Arrays;
2 public class ArrayCopy {
3     public static void main(String[] args) {
4         // TODO Auto-generated method stub
5         int[] a = new int[]{1,2,3};
6         int[] b = Arrays.copyOf(a, a.length);
7         for(int i =0;i<b.length;i++)
8             System.out.println(b[i]);
9         int[] c = Arrays.copyOf(a, a.length*2);
10        for(int i =0;i<c.length;i++)
11            System.out.println(c[i]);
12    }
13 }
```

Problems | @ Jav  
<terminated> Arra  
1  
2  
3  
1  
2  
3  
0  
0  
0



# 数组的优缺点

- ◆数组可以快速地随机访问数组中的元素。数组可以保存基本类型和对象。
- ◆数组的容量固定。当数组空间不足的时候需要做数据迁移，效率很低。
- ◆经常不知正在编写的程序需要使用多少个对象。



# (1) ArrayList (线性表)

- ◆ `Java. utils. ArrayList`
- ◆ 实现了线性表数据结构
- ◆ 内部维护了一个Object类型数组
- ◆ 容量可以动态变化，也被称为动态数组。
- ◆ 快速的随机存取。中间插入和删除元素时，速度较慢。



# (1) ArrayList (线性表)

方 法	描 述
<code>add()</code>	重载的 <code>add()</code> 方法，可在指定位置处插入元素
<code>remove()</code>	重载的 <code>remove()</code> 方法，可在指定位置处删除元素
<code>get(int)</code>	获取指定位置的元素
<code>set()</code>	重设指定位置的元素的值
<code>indexOf()</code>	获取指定元素的位置
<code>lastIndexOf()</code>	获取指定元素的最后一次出现的位置



# (1) ArrayList (线性表)

```
1 import java.util.ArrayList;
2 public class TestArrayList {
3     public static void main(String[] args) {
4         ArrayList a = new ArrayList();
5         a.add("cat");
6         a.add("dog");
7         a.add("cow");
8         a.remove(0);
9         for(int i = 0; i < a.size(); i++)
10             System.out.println(a.get(i) + ",");
11     }
12 }
```

<terminated:

dog,  
cow,



# (1) ArrayList (线性表)

## ◆构造方法

`ArrayList()`

构造一个空的线性表，容量为10

`ArrayList(Collection)`

根据已有集合构造线性表

`ArrayList(int)`

构造指定初始容量大小的线性表

示例

```
ArrayList a = new ArrayList();
```

```
ArrayList b = new ArrayList(20);
```

```
ArrayList c = new ArrayList(b);
```

◆新建一个`ArrayList`同时就创建并初始化了一个`Object`数组，容量指此数组的容量

◆正确的估计容量是提高`ArrayList`使用效率的重要途径



## (2) LinkedList (链表)

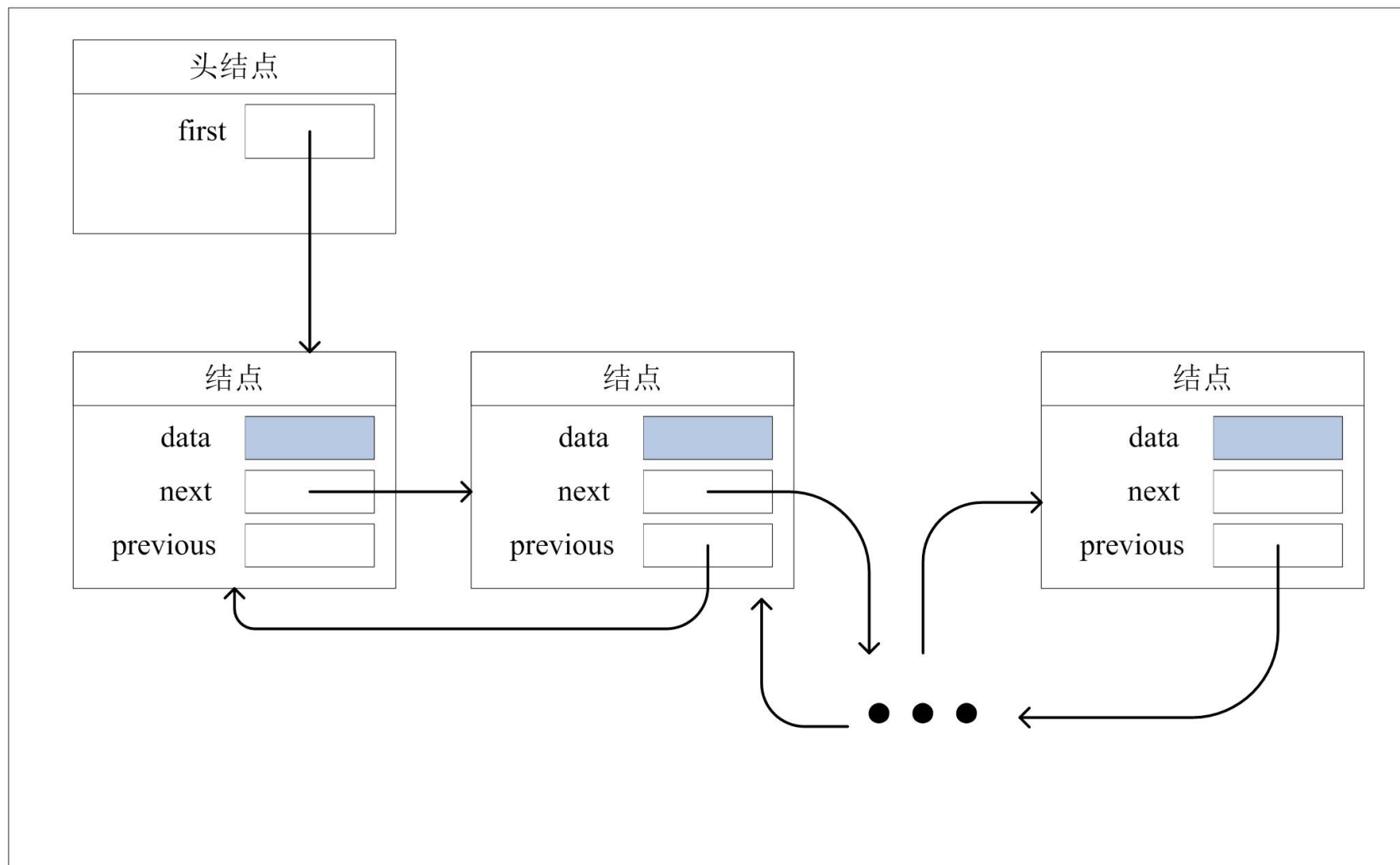
- ◆ 实现了“**链表**” 数据结构
- ◆ 较快的顺序存取速度。快速的中间插入删除速度。随机存取速度较低。
- ◆ 内部维护了一个带头结点的双向链表
- ◆ 定义了一些关于链表操作的特有方法





## (2) LinkedList (链表)

### ◆带头结点的双向链表





## (2) LinkedList (链表)

### ◆特有方法

---

<code>addFirst()</code>	向链表头插入元素
<code>addLast()</code>	向链表尾插入元素
<code>getFirst()</code>	获取链表头元素
<code>getLast()</code>	获取链表尾元素
<code>removeFirst()</code>	移除链表头元素
<code>removeLast()</code>	移除链表尾元素

---

◆使用LinkedList可以很简单的构造类似“栈”(`stack`)、“队列”(`queue`)这样的数据结构



# 示例：使用LinkedList构造栈

```
import java.util.LinkedList;

class StackL { //构造“栈”
    private LinkedList list = new LinkedList();

    public void push(Object o) {           //进栈
        list.addFirst(o);
    }
    public Object top() {                  //查看栈顶元素
        return list.getFirst();
    }
    public Object pop() {                  //出栈
        return list.removeFirst();
    }
}
```



数组

Arrays.sort()

Comparable和Comparator

Arrays.copy()

动态数组ArrayList、LinkedList

Collection

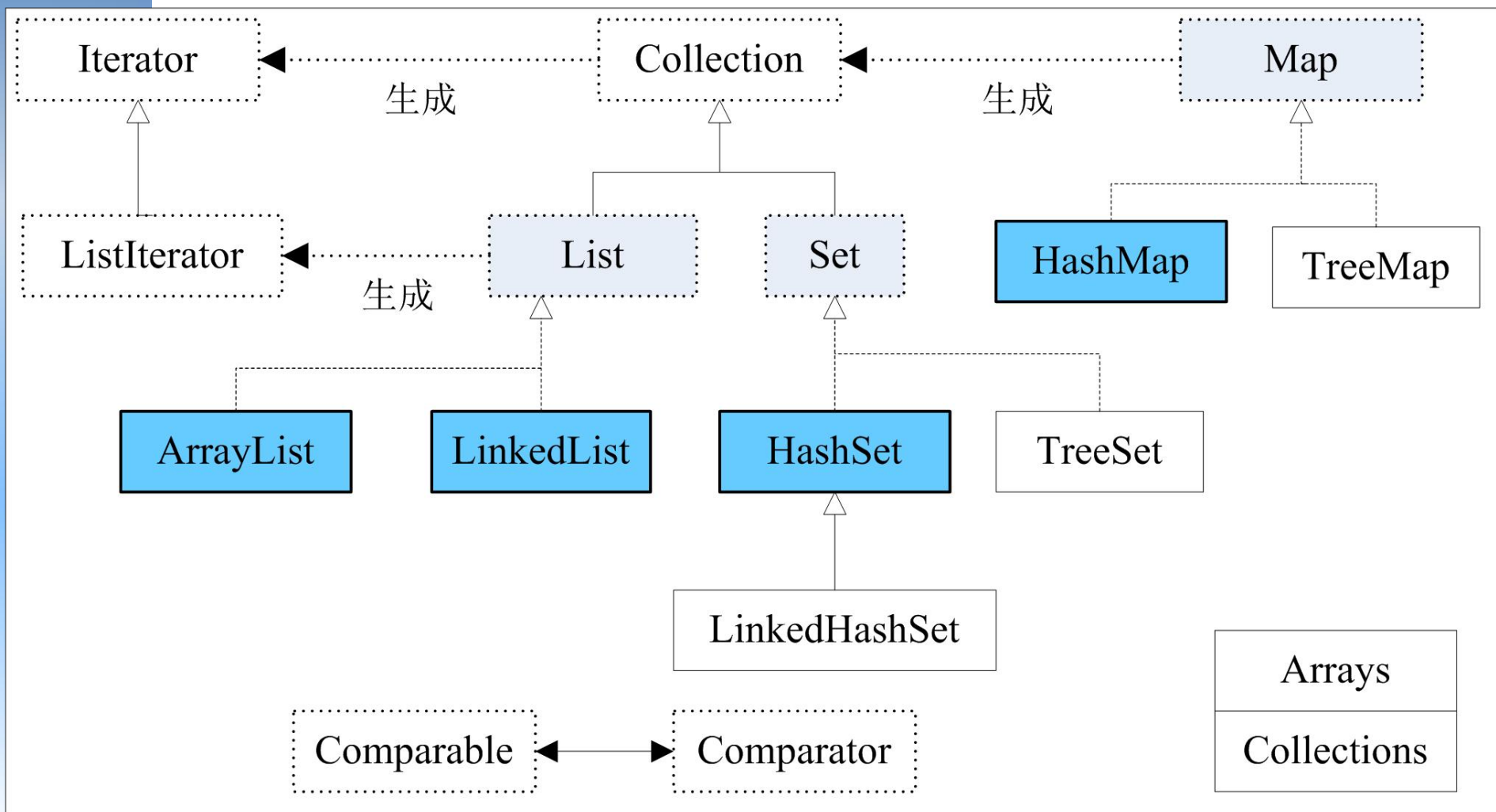


## 7.3 容器

- ◆ 列表List
- ◆ 线性表ArrayList与链表LinkedList
- ◆ 泛型
- ◆ 集合Set
- ◆ 迭代器Iterator
- ◆ 映射Map
- ◆ HashSet与HashMap
- ◆ Collections工具类



# 容器类关系







## 7.3 容器

### ◆ 动态的容量

### ◆ Collection（类集）：一组单独元素集合

➤ **List**：保存的对象有顺序，按照元素的索引位置检索对象。允许重复元素。

➤ **Set**：不允许保存重复的元素。元素之间没有顺序。靠元素值检索对象。

### ◆ Map（映射）：一组键值对

➤ 元素包括“键”对象和“值”对象。键必须是唯一的，值可以重复。



# Collection接口

方 法	描 述
<code>add()</code>	向类集中添加一个元素
<code>addAll()</code>	将另一个类集的所有元素添加到本类集
<code>clear()</code>	清空本类集
<code>contains()</code>	判断类集中是否存在某元素
<code>containsAll()</code>	判断一个类集是否为本类集的子集
<code>equals()</code>	判断两个类集是否相等
<code>isEmpty()</code>	判断类集是否为空
<code>remove()</code>	从类集中移除一个元素
<code>size()</code>	查看类集中元素的数量
<code>toArray()</code>	将本类集转化为一个同类型数组并返回数组引用
<code>iterator()</code>	生成一个本类集的迭代器



# 1. List（列表）

◆ List是Collection的子接口，是有序的Collection

◆ 顺序是List 和其它集合最大的区别，索引位置是其重要属性

---

方 法

描 述

---

add()

重载的add()方法，可在指定位置处插入元素

remove()

重载的remove()方法，可在指定位置处删除元素

get(int)

获取指定位置的元素

set()

重设指定位置的元素的值

indexOf()

获取指定元素的位置

lastIndexOf()

获取指定元素的最后一次出现的位置



# 1. List（列表）

◆ List常用实现类有ArrayList和LinkedList，两种类型存在性能差异

```
List a = new ArrayList();  
//List a = new LinkedList();  
a.add( "cat" );  
a.add( "dog" );  
a.add( "cat" );  
a.remove(0);  
for(int i=0; i<a.size(); i++) {      // a.size()=2  
    System.out.print(a.get(i)+",");  
}
```

dog,cat,



# 使用接口定义

◆上例中使用接口List定义类集的方法有一个好处：**改变类型时其它代码不用变化**

◆ArrayList与LinkedList常用的大部分方法都是接口List定义的

◆当使用实现类特有方法时不能使用接口定义，而应该使用具体类定义

➤`ArrayList a = new ArrayList();`



## (2) LinkedList (链表)

### ◆特有方法

---

<code>addFirst()</code>	向链表头插入元素
<code>addLast()</code>	向链表尾插入元素
<code>getFirst()</code>	获取链表头元素
<code>getLast()</code>	获取链表尾元素
<code>removeFirst()</code>	移除链表头元素
<code>removeLast()</code>	移除链表尾元素

---



## 2. 泛型

```
List a = new ArrayList();  
a.add( "haha" );           //插入一个String类  
a.add( new Cat() );        //插入一个Cat类  
a.add( new Dog() );        //插入一个Dog类  
String s = (String)a.get(0); //取出第一个元素  
Cat c = (Cat)a.get(0);
```

- ◆插入集合的元素没有类型检查
- ◆从集合取出的元素需要强制转换





## 2. 泛型

```
1 import java.util.*;
2 class Cat{}
3 class Dog{}
4 public class noGenerics {
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7         List a = new ArrayList();
8         a.add("haha");
9         a.add(new Cat());
10        a.add(new Dog());
11        String s = (String) a.get(0);
12        Cat c = (Cat) a.get(0);
13    }
14 }
```

Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to Cat  
at noGenerics.main(noGenerics.java:12)



## 2. 泛型

```
1 import java.util.*;
2 class Cat{}
3 class Dog{}
4 public class WithGenerics {
5     public static void main(String[] args) {
6         List<String> a = new ArrayList<String>();
7         a.add("haha");
8         a.add(new Cat());
9         a.add(new String());
10    }
11 }
12
```



## 2. 泛型

### (1) 泛型语法

- 在定义泛型类或声明泛型类的变量时，使用尖括号“`<>`”来指定形式类型参数

`class 类名<类型参数列表> {类体}`

例：Class A `<X1, X2>` {X1 a; X2 b;}

- 在应用泛型类时，必须用具体类型填入类型参数，即泛型类的具体化

`类名<具体类型列表> 变量名=new 类名<具体类型列表>`  
(构造函数的参数列表)；

例：A`<String, Integer>` m = new  
A`<String, Integer>` ();



## 2. 泛型

### (2) 泛型的使用

◆ 类可以使用泛型，接口可以使用泛型，方法也可以使用泛型

```
public <T> T testGen (boolean b, T first, T second)
    {return b ? first : second;}
```

```
String s = testGen(true, "a", "b");           //正确
```

```
Integer i=testGen(false,new Integer(1),new Integer(2));
                                           //正确
```

```
String s = testGen(true, "pi", new Float(3.14));
                                           //错误
```



## (3) 容器的泛型

### ◆接口声明

---

Collection    interface Collection<E> extends Iterable<E>

Map            interface Map<K, V>

List           Interface List<E> extends Collection<E>

Set            interface Set<E> extends Collection<E>

---

### ◆接口中的方法也是泛型的

➤ boolean    add( E e);



# 示例：定义泛型的类

```
import java.util.LinkedList;

class StackL<T> { //构造“栈”
    private LinkedList<T> list = new LinkedList();

    public void push(T o) { //进栈
        list.addFirst(o);
    }

    public T top() { //查看栈顶元素
        return list.getFirst();
    }

    public T pop() { //出栈
        return list.removeFirst();
    }
}
```



## 2. 泛型

- (1) 泛型是Java SE 1.5的新特性。
- (2) 泛型的本质是参数化类型，也就是说数据类型被认为成参数。
- (3) 使用尖括号“<>”来指定形式类型参数。
- (4) 在创建类、接口、方法时分别称为泛型类、泛型接口、泛型方法。
- (5) 在使用泛型时指定具体的类型参数的类型。





## 2. 泛型

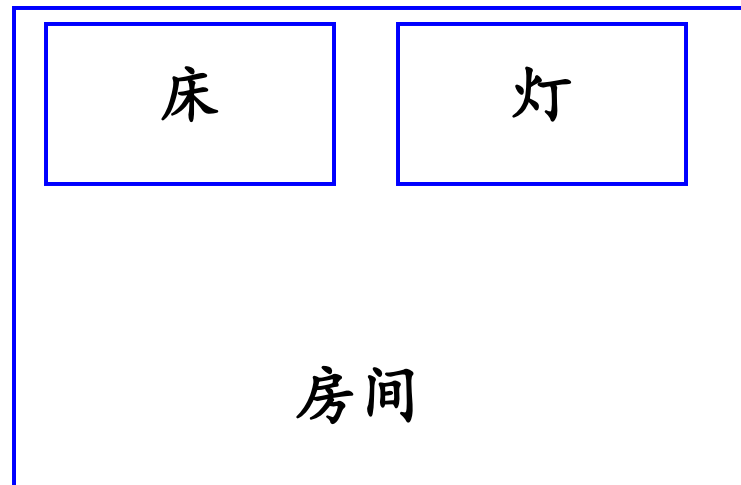
```
public class Dog <T> {  
    T weight;  
    String b = "wang";  
    public Dog(T weight) {  
        this.weight = weight;  
    }  
}  
  
Dog <double> d = new Dog(23.2);
```



## 2. 泛型



```
public class Cat{  
    int weight;  
    String b = "miao";  
    public Cat(int weight) {  
        this.weight = weight;  
    }  
}  
Cat cat = new Cat(20);
```



```
public class Dog <T> {  
    T weight;  
    String b = "wang";  
    public Dog(T weight) {  
        this.weight = weight;  
    }  
}  
Dog <double> d = new Dog(23.2);
```



## 2. 泛型

```
public class Dog <T> {  
    T weight;  
    String b = "wang";  
    public Dog(T weight) {  
        this.weight = weight;  
    }  
    public <M> M speak(M m) {  
        return m;  
    }  
}  
  
Dog <double> d = new Dog(23.2);  
String s = d.speak("hello");
```



## 2. 泛型

### ◆ 接口声明

---

Collection    interface Collection<E> extends Iterable<E>

Map            interface Map<K, V>

List           Interface List<E> extends Collection<E>

Set            interface Set<E> extends Collection<E>

---

```
interface Collection <E> {  
    boolean add(E e);  
}
```



数组

# 整体思路

Arrays.sort()

Comparable和Comparator

Arrays.copy()

动态数组ArrayList、LinkedList

Collection

List

泛型

Set 和 Map

方 法

描 述

add()

向类集中添加一个元素

addAll()

将另一个类集的所有元素添加到本类集

clear()

清空本类集

contains()

判断类集中是否存在某元素

containsAll()

判断一个类集是否为本类集的子集

equals()

判断两个类集是否相等

isEmpty()

判断类集是否为空

remove()

从类集中移除一个元素

size()

查看类集中元素的数量

toArray()

讲本类集转化为一个同类型数组并返回数组引用

iterator()

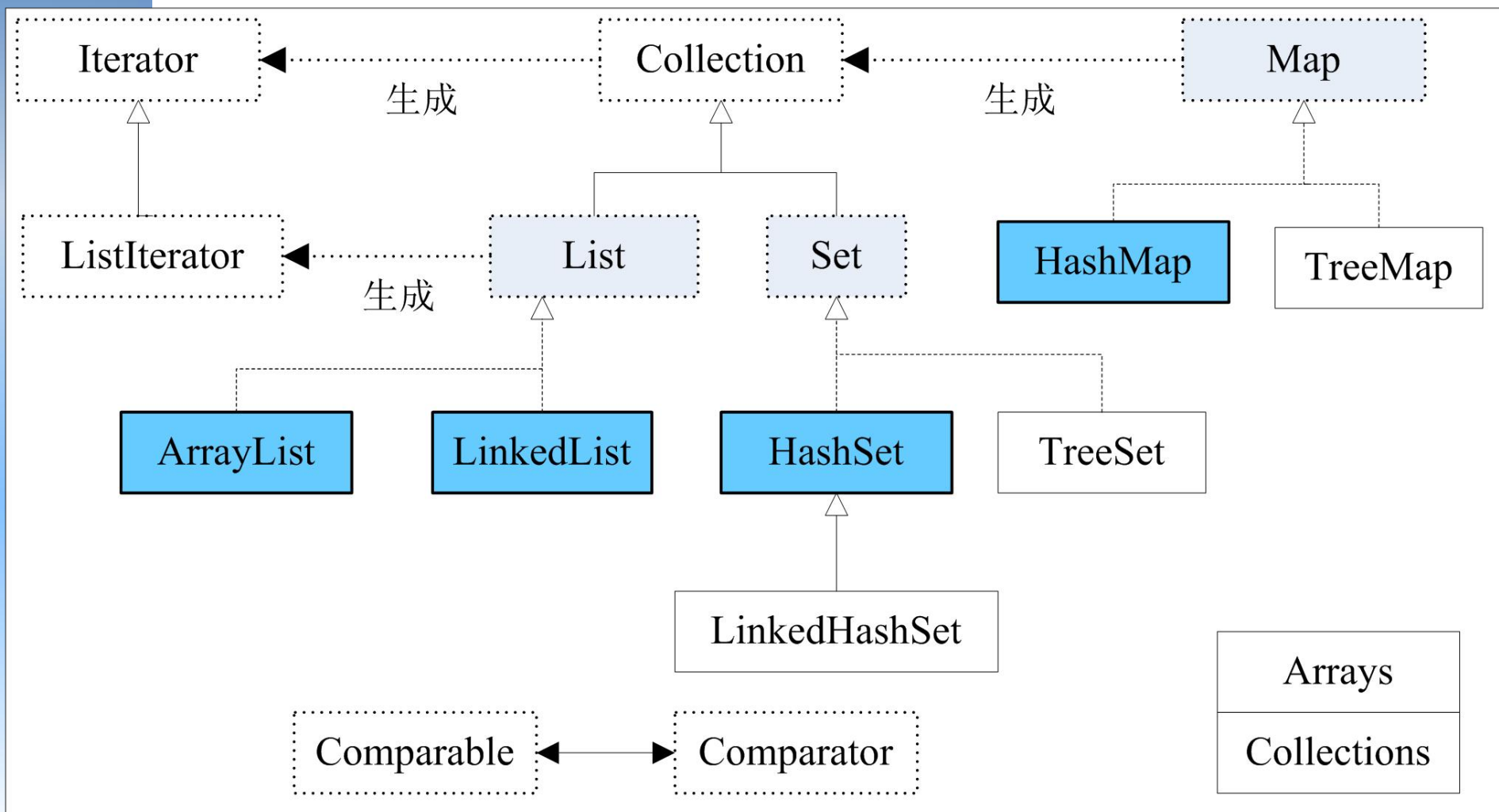
生成一个本类集的迭代器

removeLast()

移除链表尾元素



# 容器类关系





数组

Arrays.sort()

Comparable和Comparator

Arrays.copy()

动态数组ArrayList、LinkedList

Collection

List

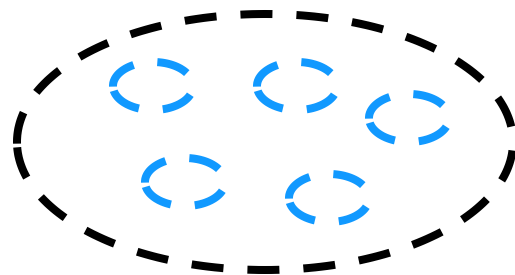
泛型

Set 和 Map



## 3. Set（集合）

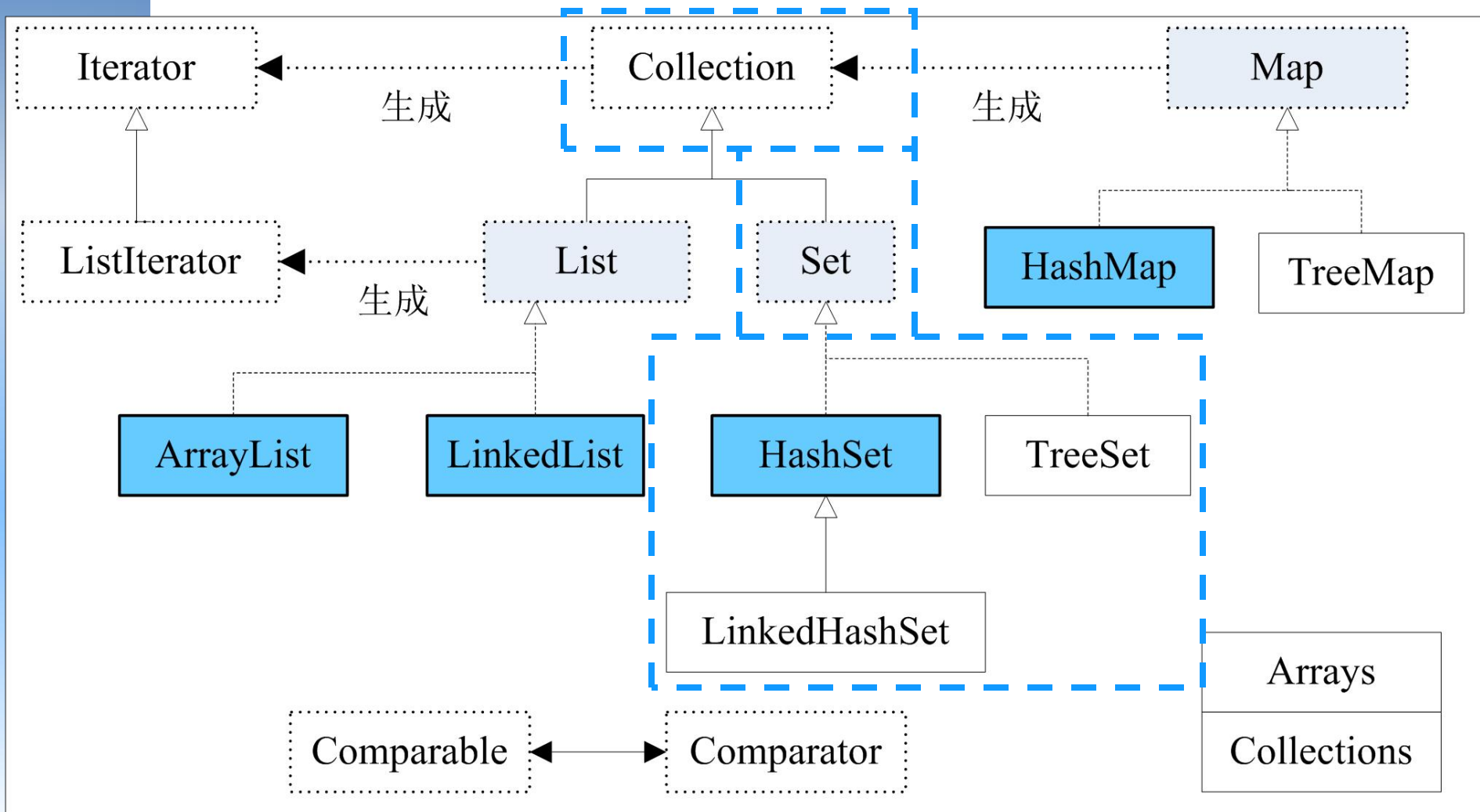
- ◆实现数学上**集合**（Set）数据结构
- ◆最大的特点就是不允许保存重复的元素
- ◆元素之间不要求有顺序，元素无索引
- ◆继承自Collection 接口。没有额外的方法
- ◆常用的实现类：**HashSet** 和 **TreeSet** 以及**LinkedHashSet**。







# 容器类关系





# 3. Set（集合）

方 法	描 述
<code>add()</code>	向类集中添加一个元素
<code>addAll()</code>	将另一个类集的所有元素添加到本类集
<code>clear()</code>	清空本类集
<code>contains()</code>	判断类集中是否存在某元素
<code>containsAll()</code>	判断一个类集是否为本类集的子集
<code>equals()</code>	判断两个类集是否相等
<code>isEmpty()</code>	判断类集是否为空
<code>remove()</code>	从类集中移除一个元素
<code>size()</code>	查看类集中元素的数量
<code>toArray()</code>	讲本类集转化为一个同类型数组并返回数组引用
<code>iterator()</code>	生成一个本类集的迭代器



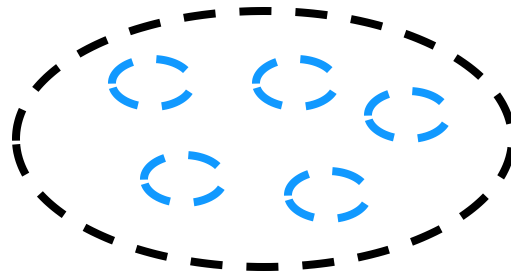
# 示例：Set（集合）

```
Set<String> set = new HashSet<String>();  
set.add("cat");  
set.add("dog");  
set.add("cat");+
```

```
set.add("cat"); //无法插入  
set.remove("cat");  
System.out.println(set);
```



**dog**





# Iterator（迭代器）

```
List<String> a = new ArrayList<String>();
```

```
...
```

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

```
    ...
```

```
    String s = a.get(i); //取出元素
```

```
    ...
```

```
}
```

```
Set<String> a = new HashSet<String>();
```

```
...
```

```
for(Iterator<String> it = a.iterator(); it.hasNext(); ) {
```

```
    ...
```

```
    String s = it.next(); //取出元素
```

```
    ...
```

```
}
```



# Iterator（迭代器）

◆一种设计模式，它是一个对象。它可以遍历并选择序列中的对象。

➤使用 **next()** 获得序列中的下一个元素，每成功调用一次迭代器向后移动一个元素。

➤使用 **hasNext()** 检查序列中是否还有元素。

➤使用 **remove()** 将迭代器

```
public interface Iterator<E>
Java.util.Iterator
```

```
import java.util.Iterator;

public class MyIterator implements Iterator{
    @Override
    public boolean hasNext() {
        // TODO Auto-generated method stub
        return false;
    }
    @Override
    public Object next() {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    public void remove() {
        // TODO Auto-generated method stub
    }
}
```

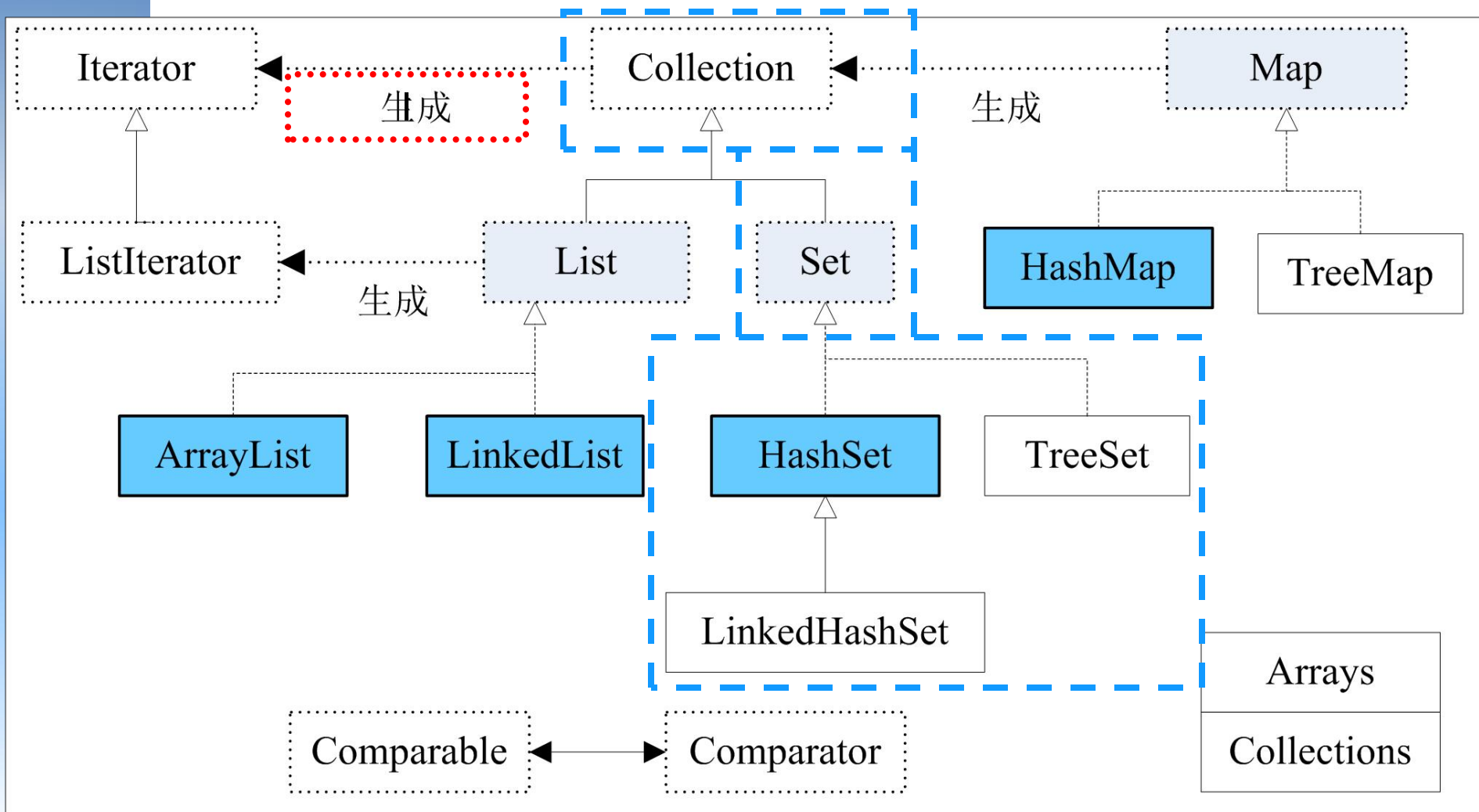


# Iterator（迭代器）

```
Set<String> a = new HashSet<String>();  
...  
for(Iterator<String> it= a.iterator(); it.hasNext(); ) {  
    ...  
    String s = it.next(); //取出元素  
    ...  
}
```



# 容器类关系





# Collection

方 法	描 述
<code>add()</code>	向类集中添加一个元素
<code>addAll()</code>	将另一个类集的所有元素添加到本类集
<code>clear()</code>	清空本类集
<code>contains()</code>	判断类集中是否存在某元素
<code>containsAll()</code>	判断一个类集是否为本类集的子集
<code>equals()</code>	判断两个类集是否相等
<code>isEmpty()</code>	判断类集是否为空
<code>remove()</code>	从类集中移除一个元素
<code>size()</code>	查看类集中元素的数量
<code>toArray()</code>	讲本类集转化为一个同类型数组并返回数组引用
<code>iterator()</code>	生成一个本类集的迭代器





## (3) 容器的泛型

### ◆接口声明

---

Collection    interface Collection<E> extends Iterable<E>

Map            interface Map<K, V>

List           Interface List<E> extends Collection<E>

Set            interface Set<E> extends Collection<E>

---

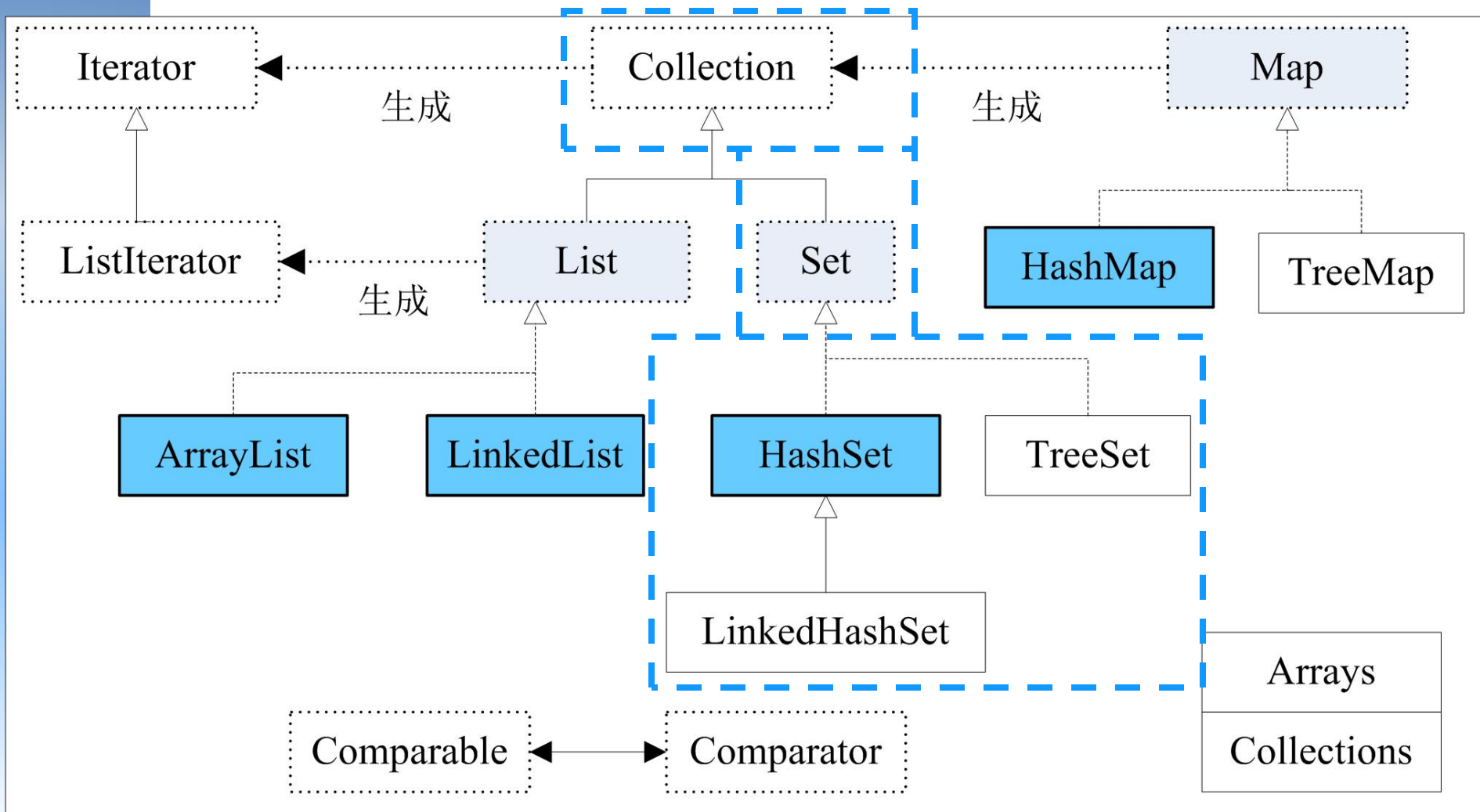
### ◆接口中的方法也是泛型的

➤ boolean    add( E e);



# 容器类关系

Iterable





# Iterator（迭代器）

## ◆使用过程：

- 调用方法**iterator()** 返回一个迭代器。第一次调用Iterator的**next()**方法时，它返回序列的第一个元素。
- 使用**next()**获得序列中的下一个元素，每成功调用一次迭代器向后移动一个元素。
- 使用**hasNext()**检查序列中是否还有元素。
- 使用**remove()**将迭代器新返回的元素删除。

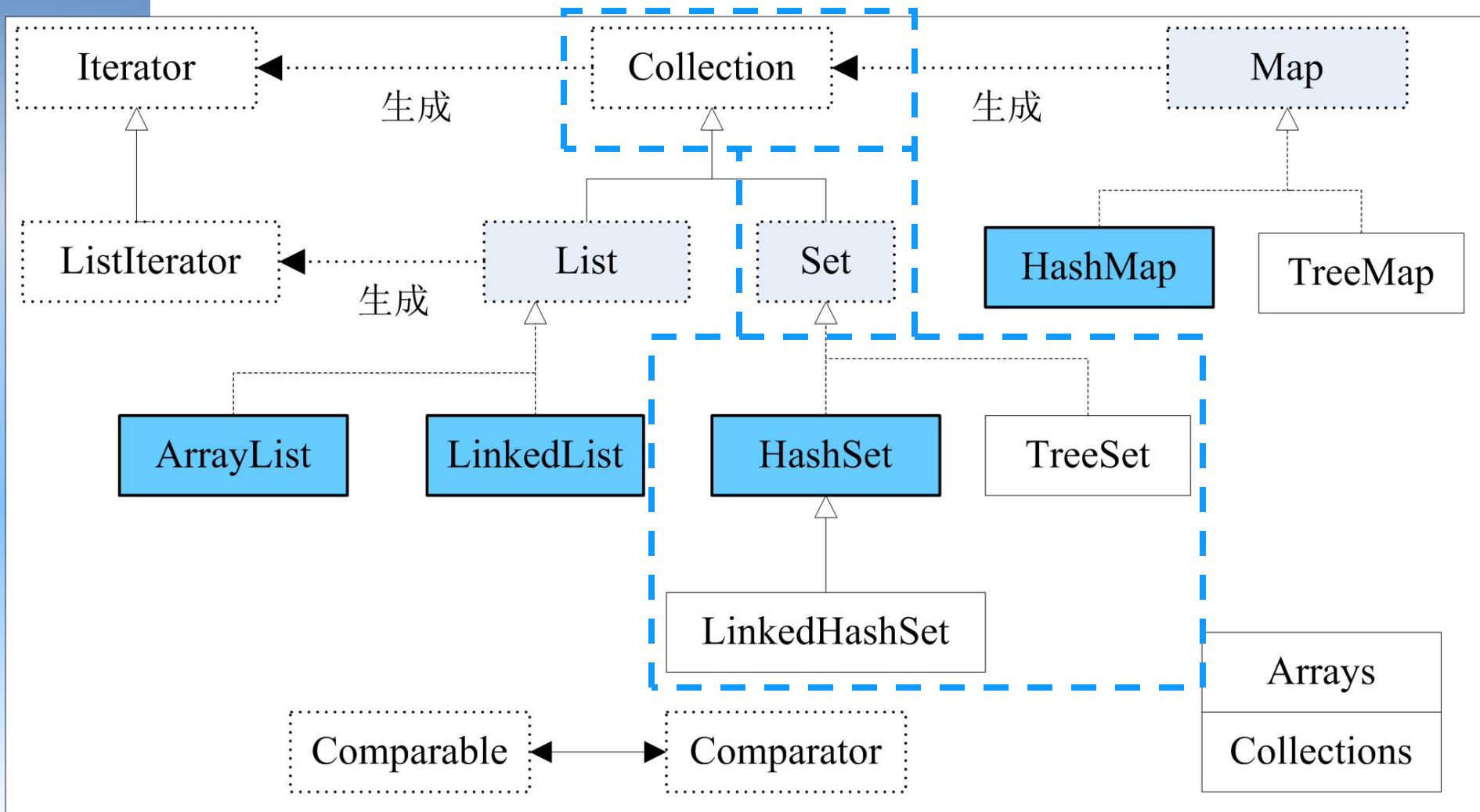
```
public interface Iterable<E>
```

```
Java: public class MyIterator implements Iterable{  
    @Override  
    public Iterator iterator() {  
        // TODO Auto-generated method stub  
        return null;  
    }  
}
```



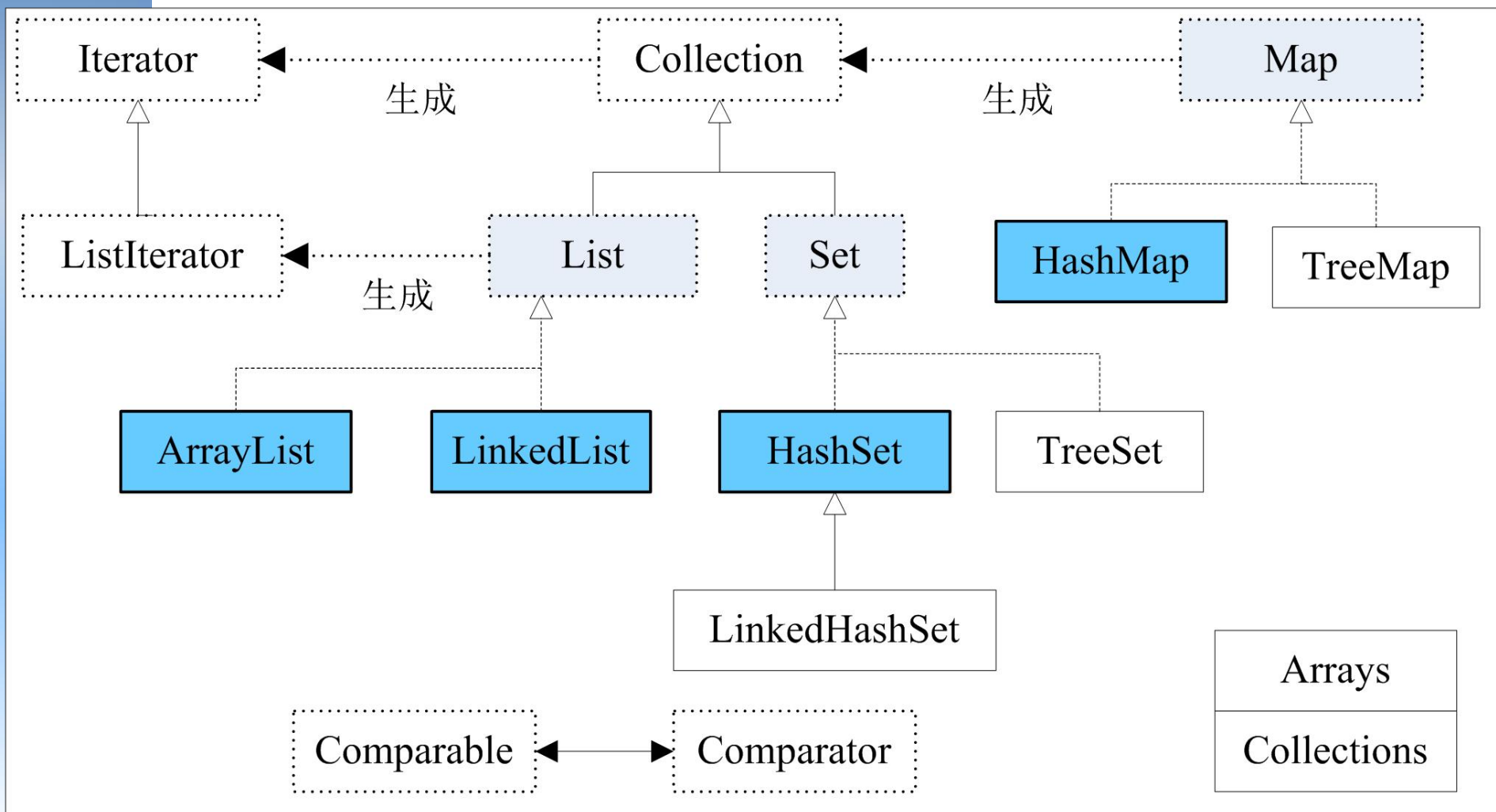
# 容器类关系

Iterable





# 容器类关系





数组

Arrays.sort()

Comparable和Comparator

Arrays.copy()

动态数组ArrayList、LinkedList

Collection

List

泛型

Set 和 Map

Set

Iterator



# Iterator（迭代器）

```
List<String> a = new ArrayList<String>();  
...  
for(Iterator<String> it= a.iterator(); it.hasNext(); ) {  
    ...  
    String s = it.next(); //取出元素  
    ...  
}  
  
Set<String> a = new HashSet<String>();  
...  
for(Iterator<String> it= a.iterator(); it.hasNext(); ) {  
    ...  
    String s = it.next(); //取出元素  
    ...  
}
```



# Iterator (迭代器)

- ◆ Collection 接口都可以使用迭代器
- ◆ 迭代器把存取逻辑从不同类型的集合类中
- ◆ ListIterator

```
import java.util.ListIterator;

public class MyIterator implements ListIterator{

    @Override
    public void add(Object arg0) {
        // TODO Auto-generated method stub
    }

    @Override
    public boolean hasNext() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean hasPrevious() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public Object next() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public int nextIndex() {
        // TODO Auto-generated method stub
        return 0;
    }
}
```

Iterator





# for-each遍历

◆把迭代器进行了包装，使得代码更加简洁。

**for(变量类型 变量名:集合){...}**

◆for-each只能遍历两种类型的对象：  
—数组  
—实现了java.lang.Iterable接口的类的实例

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

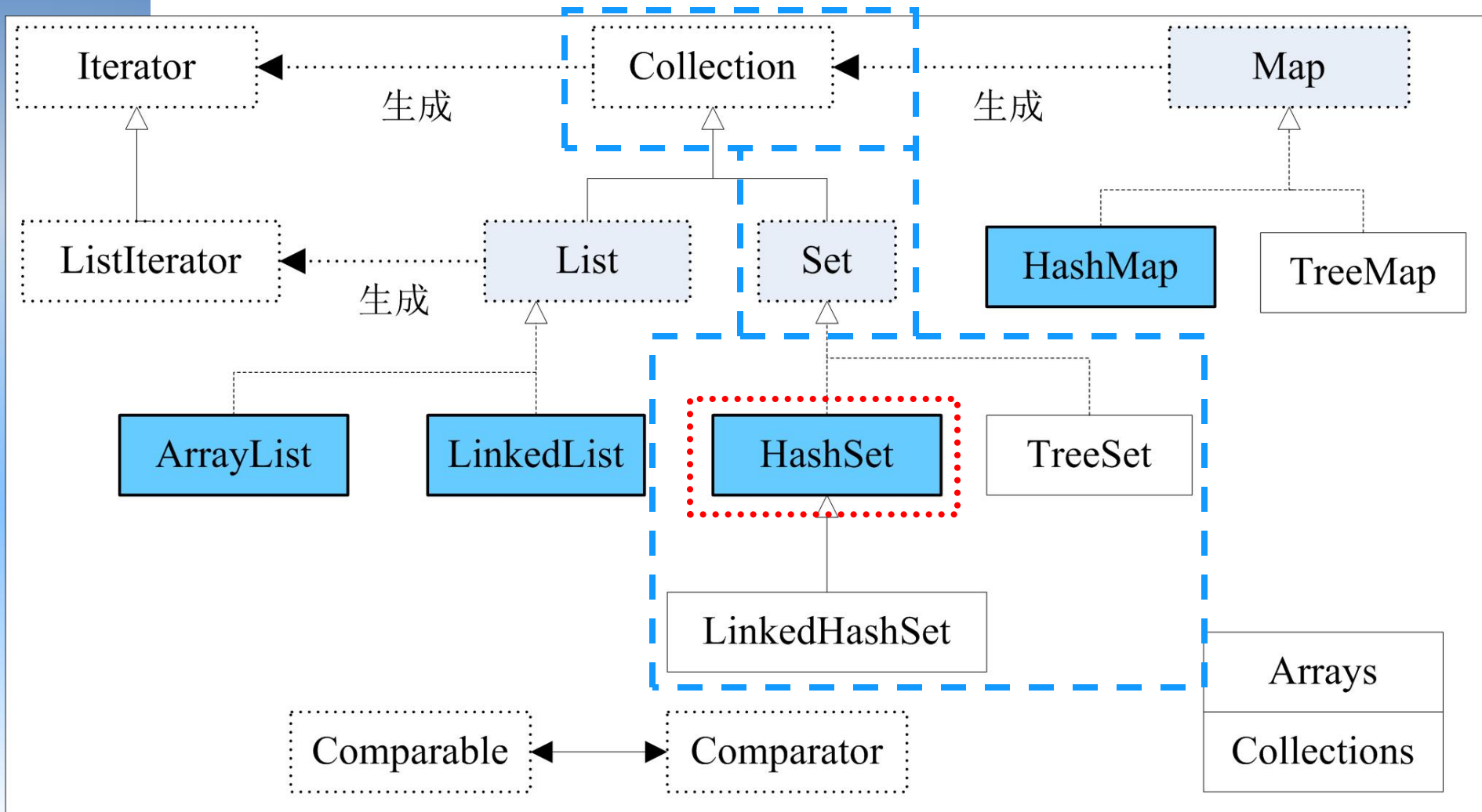
```
...
```

```
for(String s : set) {  
    System.out.println(s);  
}
```



# 容器类关系

Iterable





# HashSet (哈希集)

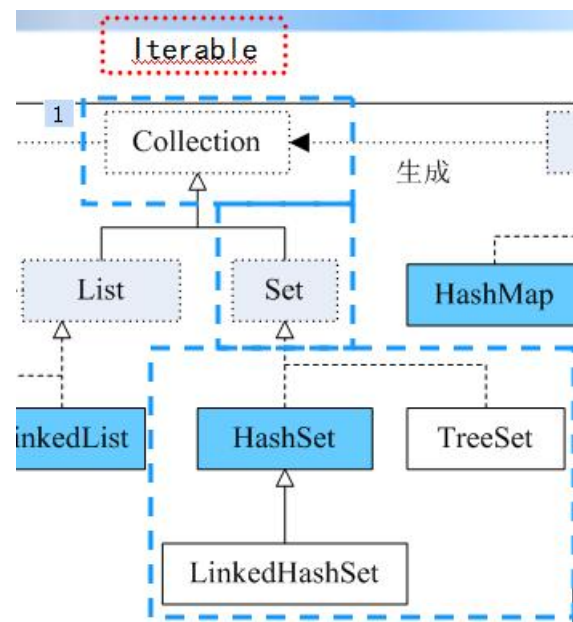
◆ 向Set中添加删除时，若循环比较则效率低下

```
Set<String> a = new HashSet<String>();  
...  
for(Iterator<String> it= a.iterator(); it.hasNext(); ) {  
    ...  
    String s = it.next(); //取出元素  
    ...  
}
```



# HashSet (哈希集)

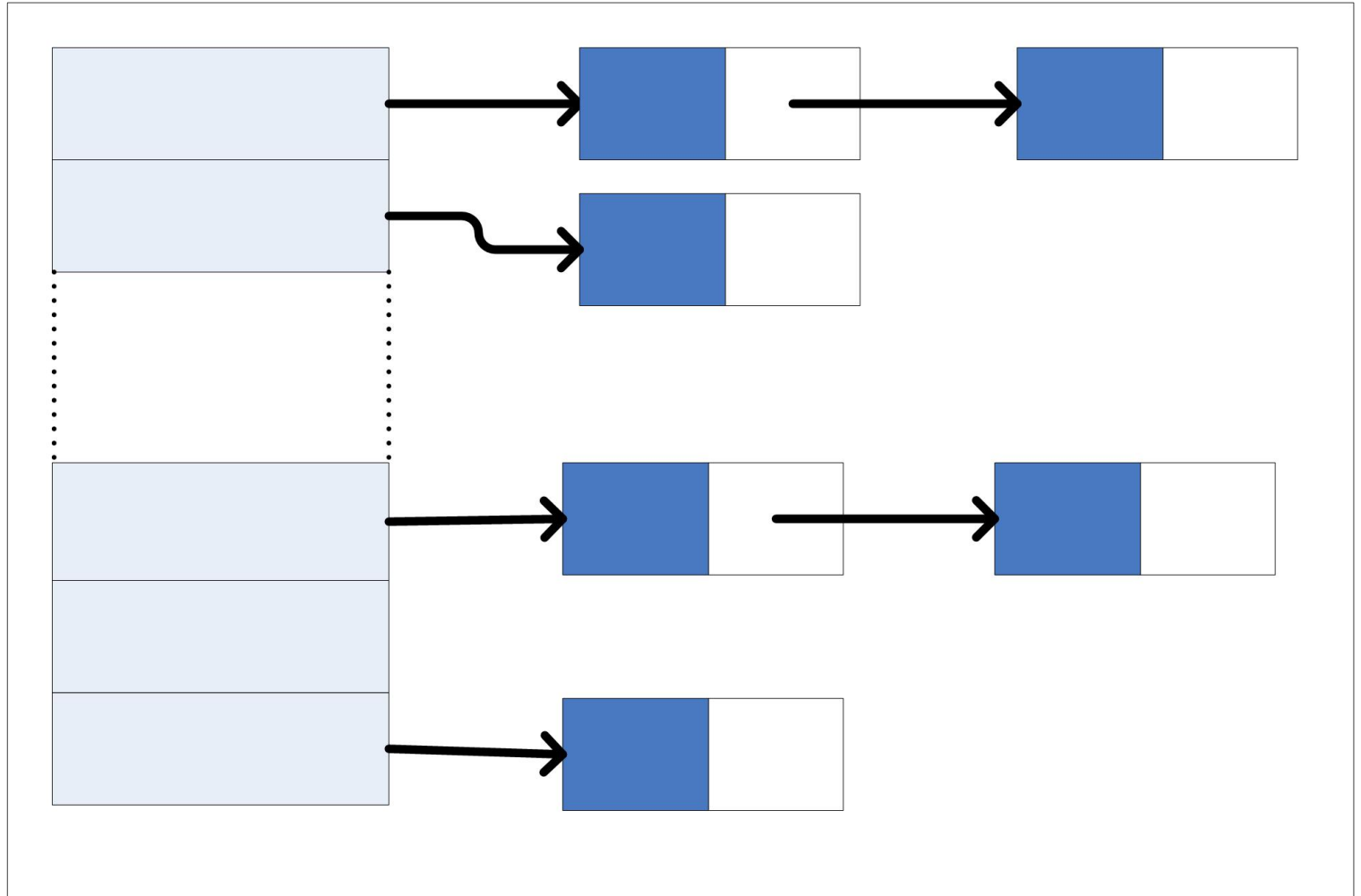
- ◆ 添加删除时，若循环比较则效率低下
- ◆ 采用“**哈希表**”与**哈希算法**实现快速查找HashSet
- ◆ 内部维护一个**链表数组**





# HashSet (哈希集)

## ◆ 链表数组





# HashSet (哈希集)

◆添加、删除之前首先进行查找

◆查找元素a过程:

- 调用a.**hashCode()**方法获得哈希码
- 由哈希码计算出a在数组中位置，即插入哪一个链表。
- 调用a.**equals()**方法和此链表中每一个元素比较。
- 如果有相同的元素，a被找到。结束。
- 否则说明未找到a。



## 3.4 Java标准类库

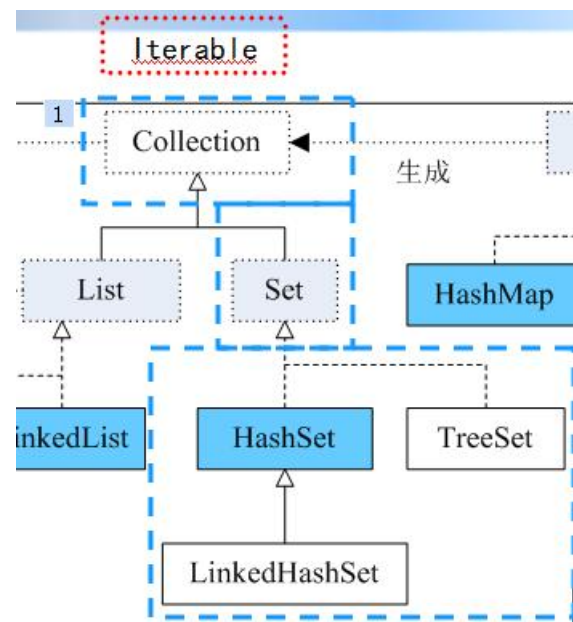
### ◆Object类

```
public class java.lang.Object{  
    public Object();                //构造方法  
    protected Object clone();      //建立当前对象的拷贝  
    public boolean equals(Object obj); //比较对象  
    protected void finalize();    //释放资源  
    public final Class getClass(); //求对象对应的类  
    public int hashCode();        //求hash码值  
    public final void notify();   //唤醒当前线程  
    public final void notifyAll(); //唤醒所有线程  
    public String toString();    //返回当前对象的字符串  
    public final void wait();    //使线程等待  
    public final void wait(long timeout);  
    public final void wait(long timeout, int nanos);  
    // 其中timeout为最长等待时间，单位为毫秒；nanos为附加时间，单位为纳秒，  
    // 取值范围为0~999999。  
}
```



# HashSet (哈希集)

- ◆ 添加删除时，若循环比较则效率低下
- ◆ 采用“**哈希表**”与**哈希算法**实现快速查找**HashSet**
- ◆ 内部维护一个**链表数组**
- ◆ 拥有常数查找时间







# HashSet (哈希集)

## ◆ 构造方法

---

HashSet()	构造空的HashSet，容量16，载入因子0.75
HashSet(Collection)	从其它集合构造HashSet
HashSet(int)	指定初始容量，载入因子0.75的HashSet
HashSet(int, float)	指定初始容量与载入因子的HashSet

---

◆ 容量为链表数组的容量，载入因子为数组重构的条件。

当（元素的数目 > 容量 \* 载入因子）时，扩展哈希集容量。如16→32



# HashSet (哈希集)

```
import java.util.*;

public class HashSetTest {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Set<Integer> set = new HashSet<Integer>();
        for(int i = 0;i<100;i++){
            Integer temp = new Integer(i%6);
            set.add(temp);
        }
        for(Iterator<Integer> it = set.iterator();it.hasNext();){
            System.out.println(it.next()+",");
        }
    }
}
```

<terminat

0,

1,

2,

3,

4,

5,



# HashSet (哈希集)

```
import java.util.*;
class Employee{
    int id;
    public Employee(int id){
        this.id = id;
    }
}
public class HashSetNoHashCode {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Set<Employee> set = new HashSet<Employee>();
        set.add(new Employee(12));
        set.add(new Employee(1));
        set.add(new Employee(1));
        for(Employee e:set)
            System.out.println("ID="+e.id+",hashcode="+e.hashCode());
    }
}
```

ID=1,hashcode=28705408

ID=12,hashcode=7971559

ID=1,hashcode=6182315



## (3) HashSet (哈希集)

- ◆系统默认hashCode() 返回对象在内存中的地址
- ◆系统默认equals() 方法比较两个对象内存中的地址
- ◆HashSet中的对象需要重写  
hashCode() 方法与equals() 方法
- ◆如果 a.equals(b) 返回true, 那么  
a.hashCode()=b.hashCode()
  - 内容equals的对象hashcode一定相等。
  - 两个对象的hashcode不相等则两个对象一定不equals



# HashSet（哈希集）

- ◆ 添加删除时，若循环比较则效率低下
- ◆ 采用“**哈希表**”与**哈希算法**实现快速查找**HashSet**
- ◆ 内部维护一个**链表数组**
- ◆ 拥有常数查找时间
- ◆ Integer, Double, String等重写了**hashCode()**方法与**equals()**方法





# HashSet (哈希集)

```
1 import java.util.*;
2 class Employee{
3     int id;
4     public Employee(int id){
5         this.id = id;
6     }
7     public int hashCode(){
8         return this.id;
9     }
10    public boolean equals(Object o){
11        Employee e = (Employee)o;
12        if(this.id==e.id)
13            return true;
14        else
15            return false;
16    }
17 }
18 public class HashSetNoHashCode {
19     public static void main(String[] args) {
20         // TODO Auto-generated method stub
21         Set<Employee> set = new HashSet<Employee>();
22         set.add(new Employee(12));
23         set.add(new Employee(1));
```

ID=1,hashCode=1  
ID=12,hashCode=12



## (3) HashSet (哈希集)

### ◆合格的`equals()`方法必须满足条件:

- **自反性**: 对任意 `x`, `x.equals(x)` 总返回 `true`。
- **对称性**: 对任意 `x` 与 `y`, `x.equals(y)` 总返回 `true` 当且仅当 `y.equals(x)` 返回 `true`。
- **传递性**: 对任意 `x`, `y`, and `z`, 如果 `x.equals(y)` 返回 `true` 而且 `y.equals(z)` 返回 `true`, 那么 `x.equals(z)` 总返回 `true`。
- **持续性**: 对任意 `x` and `y`, 多次调用 `x.equals(y)` 结果不会改变。
- 对任意非空 `x`, `x.equals(null)` 总返回 `false`。



# 其它Set

## ◆ TreeSet : 元素间有序的Set

- 实现“**红黑树**” (自平衡排序二叉树) 数据结构后推得到的顺序Set。
- 对象必须实现**Comparable**接口或者**Comparator**接口
- 对象需重写**equals()** 以保证元素的单一性
- 查找效率 $\log(n)$

## ◆ LinkedHashSet: HashSet+链表

- 在HashSet的基础上将每一个元素用链表串连起来
- 元素间有确定的顺序
- 适合于常量复杂度的高效存取性能要求, 同时又要求元素有序的情况





## 示例：比较

### HashSet, TreeSet, LinkedHashSet

```
Set<Integer> hashSet = new HashSet<Integer>();  
Set<Integer> linkedHashSet = new LinkedHashSet<Integer>();  
Set<Integer> treeSet = new TreeSet<Integer>();
```

```
//产生一个随机数s，并将其包装后放入3种Set中  
for(int i=0;i<5;i++){  
    int s=(int)(Math.random()*100);  
    //产生一个0—100的随机整数  
    Integer temp = new Integer(s);  
    hashSet.add(temp);  
    linkedHashSet.add(temp);  
    treeSet.add(temp);  
  
    System.out.println("第 "+i+" 次随机数产生为: "+s);  
}  
System.out.println("HashSet: "+hashSet);  
System.out.println("LinkedHashSet: "+linkedHashSet);  
System.out.println("TreeSet : "+treeSet);
```



# 比较HashSet, TreeSet, LinkedHashSet

## ◆输出结果:

第 0 次随机数产生为: 46

第 1 次随机数产生为: 0

第 2 次随机数产生为: 27

第 3 次随机数产生为: 70

第 4 次随机数产生为: 61

**HashSet:** [0, 70, 27, 46, 61]

**LinkedHashSet:** [46, 0, 27, 70, 61]

**TreeSet :** [0, 27, 46, 61, 70]

- HashSet的元素存放无顺序
- LinkedHashSet 保持元素的添加顺序。
- TreeSet元素按值进行排序存放。



数组

Arrays.sort()

Comparable和Comparator

Arrays.copy()

动态数组ArrayList、LinkedList

Collection

List

泛型

Set 和 Map

Set

Iterator

HashSet, TreeSet, LinkedHashSet

重写 hashCode()

Map

Map 遍历

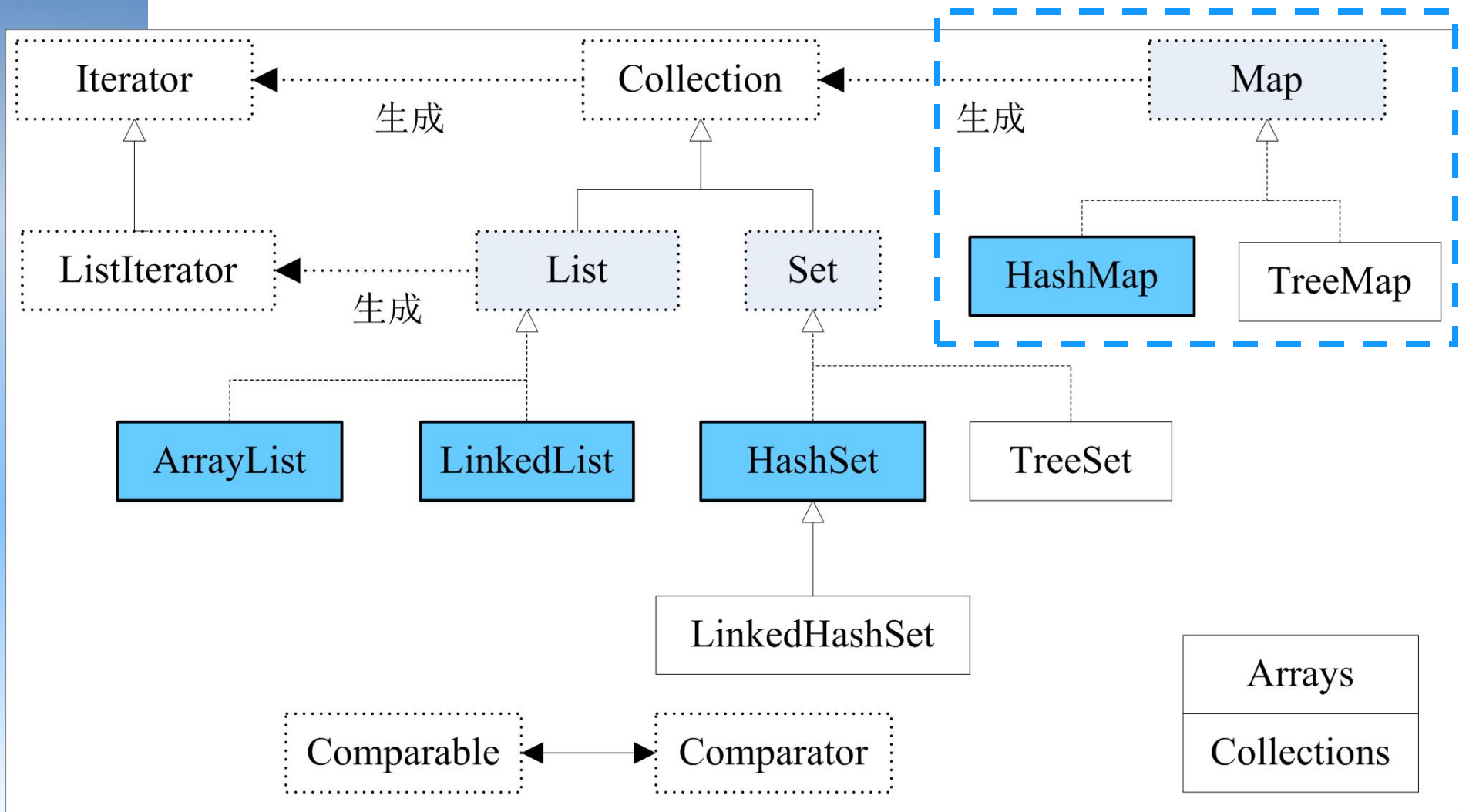
HashMap 与 HashSet

Collections



# 容器类关系

Iterable





## 4. Map（映射）

◆ 维护“键（**key**）—值(**value**)”关系对结构的无序容器

◆ `public interface Map<K,V>`

◆ 键与值都为对象，键值对也为对象

◆ 一个Map中不能包含相同的key，每个key只能映射一个value。、

◆ 常用的实现类：**HashMap** 和 **TreeMap** 以及**LinkedHashMap**。

```
Map<Integer,String> map =  
    new HashMap<Integer,String>();  
map.put(1,"abc");
```



## 4. Map（映射）

```
Map<String,String> map = new HashMap<String,String>();  
map.put("dog","yellow");  
map.put("dog","red");  
map.put("cat","red");  
System.out.println(map);  
Map.remove("cat");  
System.out.println(map);
```

```
{cat = red, dog = red}  
{dog = red}
```



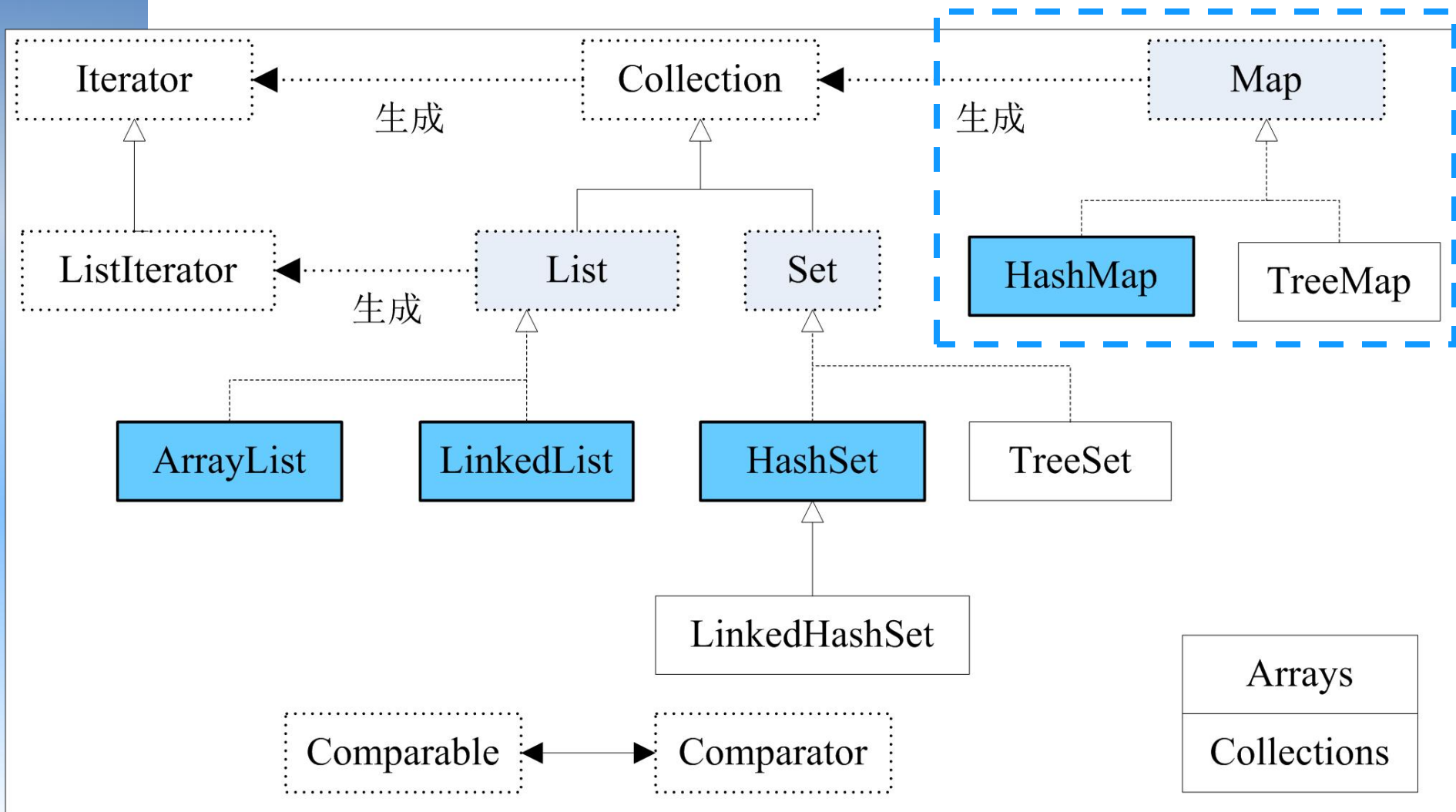
# Map常用方法

<code>put (K, V)</code>	将指定值与指定键相关联，如果此键已存在，将值替换
<code>putAll (Map)</code>	将指定 Map 中的所有映射复制到此 map
<code>clear ()</code>	删除map中所有映射
<code>remove (Object)</code>	从 Map 中删除键和关联的值
<code>get (Object)</code>	通过键查找值
<code>size ()</code>	返回键值对个数
<code>isEmpty ()</code>	判断映射集是否为空
<code>keySet ()</code>	将映射中的键打包成Set返回
<code>values ()</code>	将映射中的值打包成Collection返回
<code>entrySet ()</code>	返回键值对实体的Set
<code>containsKey (Object)</code>	判断集合中是否存在指定键
<code>containsValue (Object)</code>	判断集合中是否存在指定值



# 容器类关系

Iterable







# Map遍历

- ◆ Map接口没有继承`Iterable`接口，无法使用迭代器
- ◆ Map键值对元素无序，无法使用索引
- ◆ Map接口提供3种集合的视图：
  - `keySet()`：生成`键`的Set型集合
  - `values()`：生成`值`的Collection型集合
  - `entrySet()`：生成`键值对`的Set型集合



# (1) Map遍历

## ◆方法一（遍历值）：

```
public void byValue(Map<String, Student>map) {  
    Collection<Student> c = map.values();  
    Iterator it;  
    for (it=c.iterator(); it.hasNext();) {  
        Student s = (Student)it.next();  
    }  
}
```

## ◆方法二（通过键遍历值）：

```
public void byKey (Map<String, Student> map) {  
    Set<String> key = map.keySet();  
    Iterator it;  
    for (it = key.iterator(); it.hasNext();) {  
        String s = (String) it.next();  
        Student value = map.get(s);  
    }  
}
```



# (1) Map遍历

◆ **Map.Entry**: Map内部定义的一个接口，专门用来保存**键值对**的内容

◆ 方法三：

```
public static void byEntry(Map<String, Student> map) {  
    Set<Map.Entry<String, Student>> set = map.entrySet();  
    Iterator<Map.Entry<String, Student>> it;  
    for (it= set.iterator(); it.hasNext();) {  
        Map.Entry<String, Student> entry = it.next();  
        System.out.println(entry.getKey() + "---->" +  
entry.getValue());  
    }  
}
```



# HashMap（哈希映射）

◆ Map必须保持“键”的单一性，这和Set必须保持其元素的单一性是一致的

◆ HashSet内部维护着一个HashMap

➤ “键”就是我们要存入的对象

➤ “值”是一个常量PRESENT

◆ 例如：

➤ HashSet的add()方法调用其内部HashMap的put()方法



# HashMap (映射)

◆ HashMap的对象也需要实现 **equals()** 与 **hashCode()** 方法

◆ 构造方法:

HashMap()	构造空的HashSet, 容量16, 载入因子0.75
HashMap(Collection)	从其它集合构造HashSet
HashMap(int)	指定初始容量, 载入因子0.75的HashSet
HashMap(int, float)	指定初始容量与载入因子的HashSet

◆ 容量与载入因子作用同HashSet中的一样



# 其它Map

◆ **TreeMap** : 特点同**TreeSet**，基于红黑树数据结构的实现。按“**键**”进行排序是唯一的带有**subMap()**方法的Map，它可以返回一个子树

◆ **LinkedHashMap**将每一个键值对用双向链表串连起来，类似于**LinkedHashSet**

◆ **WeakHashMap**：如果没有map之外的引用指向某个“**键**”，则此“**键**”可以被垃圾收集器回收

◆ **IdentifyHashMap**：使用**==**代替**equals()**对“**键**”作比较的hash map



## 5. Collections（工具类）

◆ 一个包含可以操作或返回集合的专用静态类 `Java.util.Collections`

◆ 常用方法：

➤ 排序 `sort()`：对指定 `List` 按升序进行排序。列表中的所有元素都必须实现 `Comparable` 接口

➤ 二分法查找 `binarySearch()`：使用“快速查找”算法，`List` 必须已经排好序

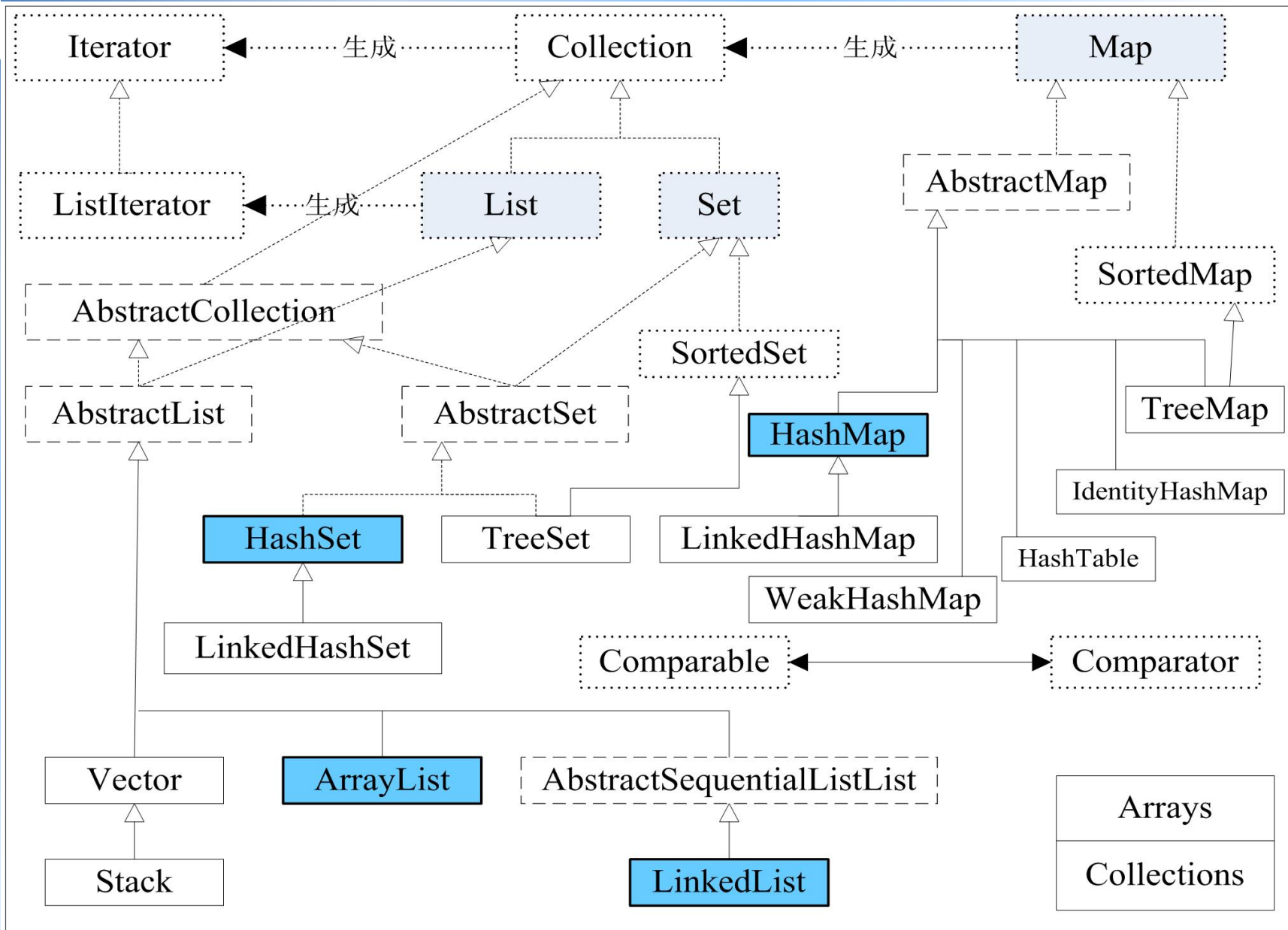
➤ 反转 `reverse()`：将 `List` 中的元素按逆序排列

➤ 拷贝 `copy()`：将源 `List` 的元素拷贝到目标，并覆盖它的内容

➤ `replaceAll()`：将 `List` 中的所有元素替换为指定的值



# 容器完整关系图







# 6. 容器类型选择

## ◆ List的选择

- ArrayList: 随机访问（即调用get()方法）以及循环反复效率最高
- LinkedList: 在列表中部进行插入和删除操作效率很高
- 两者的查找效率都相当低

## ◆ Set和Map的选择

- HashSet: “增，删，改，查”这类数据处理上的效率最高。我们大部分情况下会选择使用HashSet。
- TreeSet: 查找速度也是较快的，大概为 $\log(n)$ 级别。不过这个较快也只是相对于List来说，还是比不上HashSet。
- LinkedHashSet: 在有高效存取性能要求，同时有要求



数组

Arrays.sort()

Comparable和Comparator

Arrays.copy()

动态数组ArrayList、LinkedList

Collection

List

泛型

Set 和 Map

Set

Iterator

HashSet, TreeSet, LinkedHashSet

重写 hashCode()

Map

Map 遍历

HashMap 与 HashSet

Collections



# 总结

- ◆ 数组引用与多维数组
- ◆ 三个接口: List 、 Set 、 Map
- ◆ 两个工具类: Arrays 、 Collections
- ◆ 两个比较接口: Comparable 、 Comparator
- ◆ 四个常用类: ArrayList、LinkedList、HashSet、HashMap
- ◆ 迭代器Iterator
- ◆ 泛型



## 7.2 枚举

◆有些数据集合，它们的数值是不变化的，而且集合中的元素个数是有限的。

- 季节={春，夏，秋，冬}
- 星期={星期一，星期二，…，星期日}
- 性别={男，女}
- 音量={高，中，低}

◆创建枚举类型使用“enum”关键字，枚举类型变量只能保存此类声明时给定的某个枚举值。Java.lang.Enum

```
enum Season { SPRING, SUMMER, FALL, WINTER};  
Season s = Season.SPRING;
```



# 示例：一个简单的枚举程序

```
public class TestEnum {  
    public enum Season { SPRING, SUMMER, FALL, WINTER };  
    public static void main(String[] args) {  
        seasonOutput(Season.SPRING);  
    }  
    private static void seasonOutput(Season s) {  
        switch (s) {  
            case SPRING:  
                System.out.println("春");    break;  
            case SUMMER:  
                System.out.println("夏");    break;  
            case FALL:  
                System.out.println("秋");    break;  
            case WINTER:  
                System.out.println("冬");    break;  
            default :  
                System.out.println("default");  
        }  
    }  
}
```

春