



## 第十二章 容器和泛型

# 讲授思路

- 容器概述
- 集合 ( List、Set、Queue、Map )
- 迭代器
- 泛型

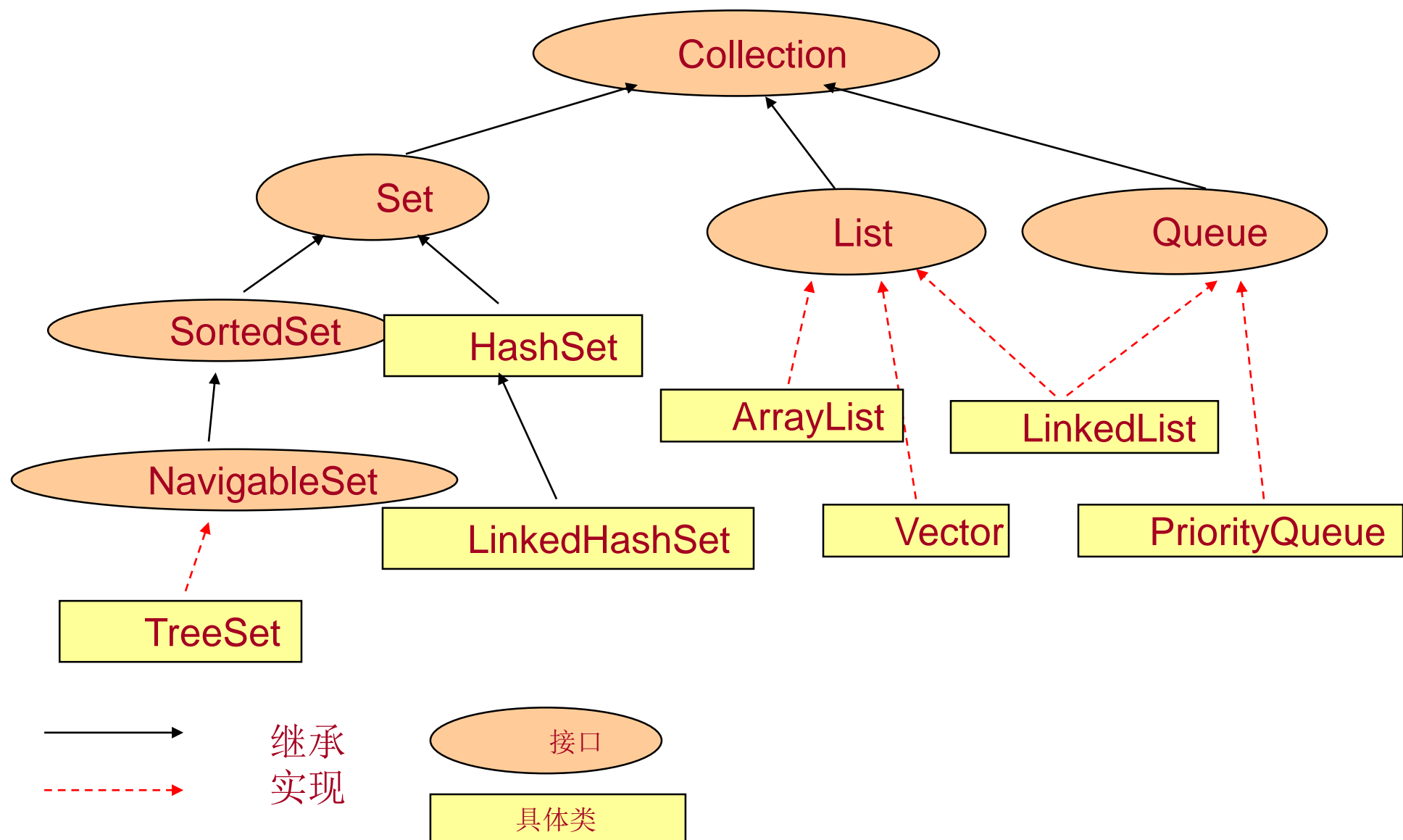
# 引入

- 如果一个程序只包含固定数量的且其生命期都是已知的对象，那么这是一个非常简单的程序。
- 通常，程序总是根据运行时才知道的某些条件去创建新对象，在此之前，不会知道所需对象的数量，甚至不知道确切类型。--Think In Java
- Java使用类库提供了一组相当完整的容器类来解决这个问题。

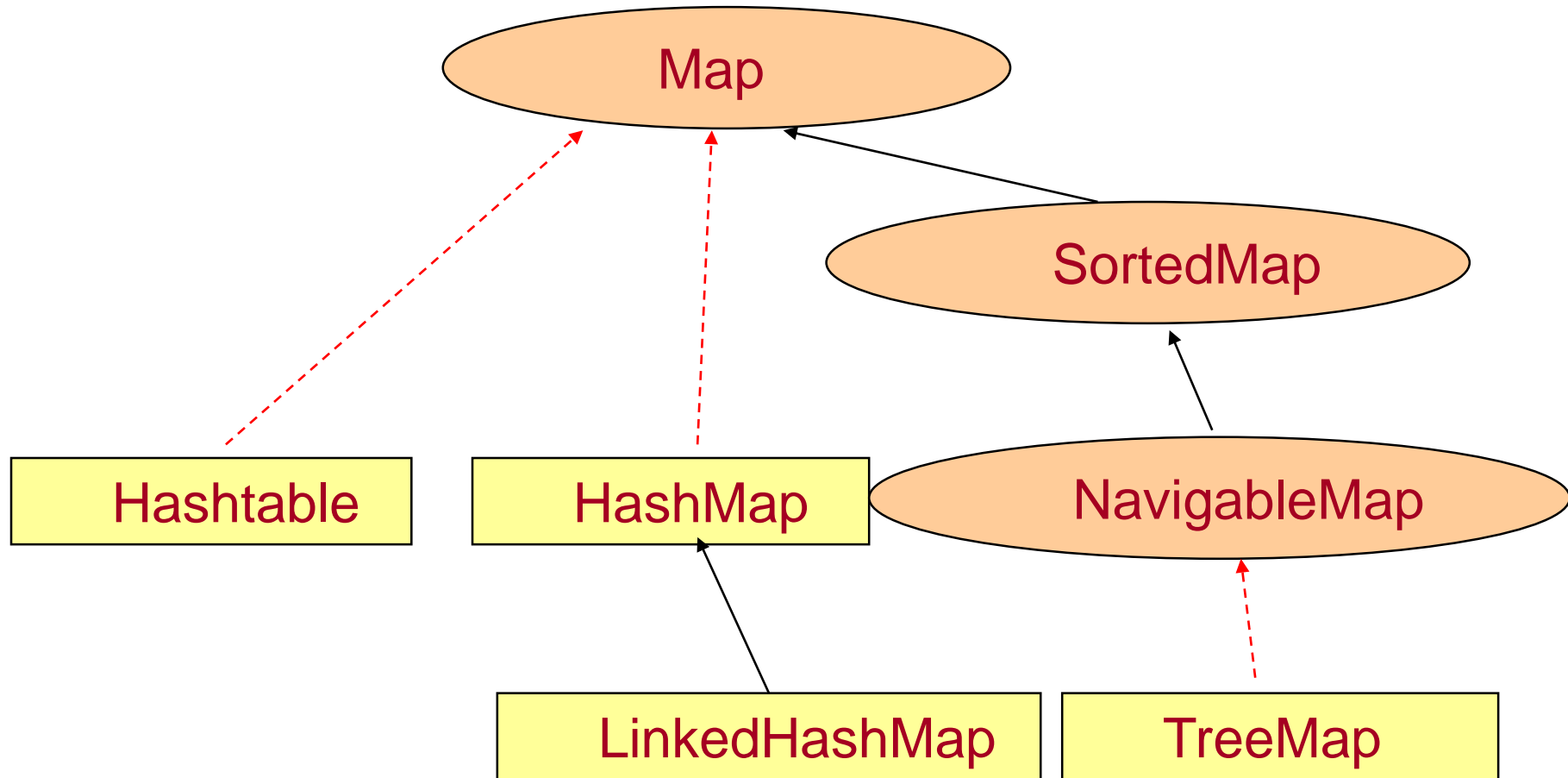
# 什么是集合

- Think In Java :
- Java提供的这一套容器类，其中基本类型是List、Set、Queue和Map，这些对象类型也称之为集合类。

# Java的Collection



# Java的Map



# 集合框架中的重点接口和类

- 需要了解的9个核心接口

Collection	Set	Map
List	SortedSet	SortedMap
Queue	NavigableSet	NavigableMap

- 需要了解的13个核心具体实现类

Map	Set	List	Queue	实用工具
HashMap	HashSet	ArrayList		Collections
Hashtable	LinkedHashSet	Vector	PriorityQueue	Arrays
TreeMap	TreeSet	LinkedList		
LinkedHashMap				

# List

- 列表（ List ）：关心的是索引
  - 对象按索引存储
  - 可以存储重复元素
  - 具有与索引相关的一套方法
- 主要实现类
  - ArrayList：动态数组
    - 快速迭代，少量插入删除
  - LinkedList：链表
    - 迭代速度慢，快速插入删除



# ArrayList

- 什么是ArrayList ?
  - ArrayList就是动态数组，动态的增加和减少元素，可灵活的设置数组的大小
- ArrayList的使用方法
  - 创建ArrayList的对象
  - 向该对象中添加元素
  - 根据需要修改该对象中的元素

# ArrayList的方法

- 构造方法

- ArrayList() : 构造一个初始容量为 10 的空列表
- ArrayList (int initialCapacity) : 构造一个具有指定初始容量的空列表

- 其他方法

- add(E e) : 将指定的元素添加到此列表的尾部
- add(int index, E element) : 将指定的元素插入此列表中的指定位置
- remove(int index) : 移除此列表中指定位置上的元素
- get(int index) : 返回此列表中指定位置上的元素
- set(int index, E element) : 用指定的元素替代此列表中指定位置上的元素
- size() : 返回此列表中的元素数

# 课堂练习

- ArrayListDemo

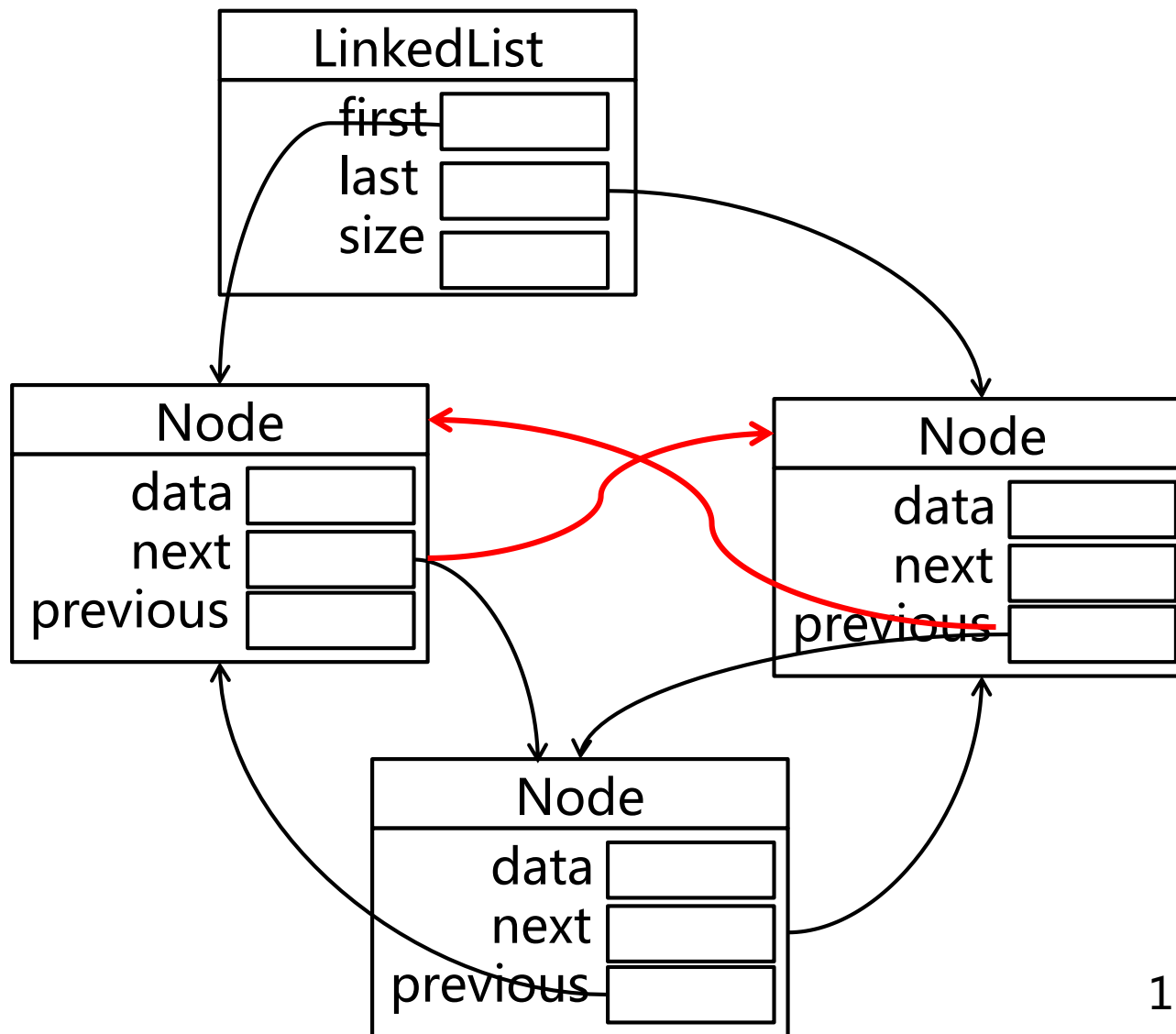
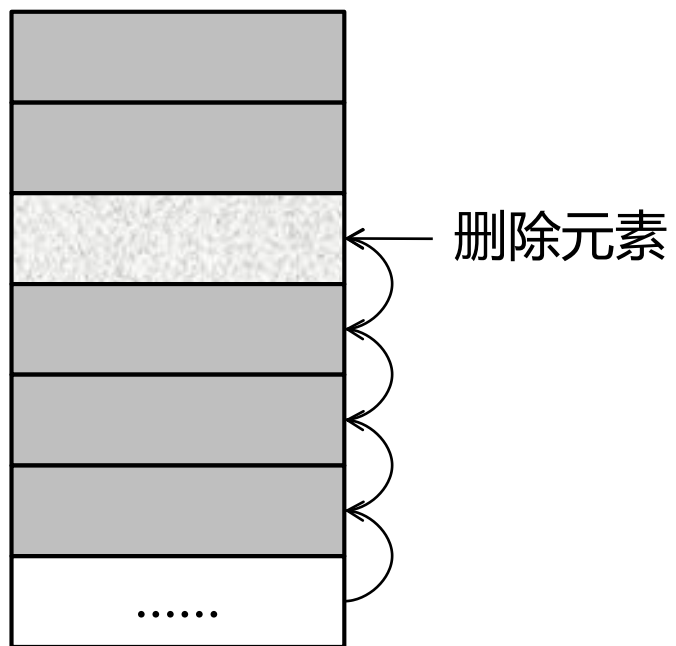
```
ArrayList list = new ArrayList<>();  
for(int i=0; i<10; i++){  
    //给数组增加10个Int元素  
    list.add(i);  
}  
list.remove(5); //将第6个元素移除  
for(int i=0; i<3; i++){  
    //再增加3个元素  
    list.add(i+20);  
}  
for(int i=0; i<list.size(); i++){  
    System.out.print(list.get(i) + "\t");  
}
```

# ArrayList取代Vector类？

- 对于动态数组，有人可能会使用Vector类
- 为什么用ArrayList取代Vector类？
  - Vector类的方法是同步的，同步操作将耗费大量时间
  - ArrayList类的方法不是同步的，故建议在不需要同步时使用

# LinkedList

- 双向链表
- 为什么使用LinkedList ?



# LinkedList方法

- 使用方法
  - 创建LinkedList对象
  - 添加元素
  - 维护对象中的元素（添加、更新、删除）
- 构造方法
  - LinkedList()：构造一个空列表
  - LinkedList(Collection<? extends E> c)：构造一个包含指定 collection 中的元素的列表
- 其他方法
  - add(E e)：将指定的元素添加到此列表的尾部
  - add(int index, E element)：在此列表中指定的位置插入指定的元素
  - remove(int index)：移除此列表中指定位置上的元素
  - size()：返回此列表中的元素数

# 课堂练习

- LinkedListDemo

```
LinkedList<String> link = new LinkedList<String>();  
link.add("Tom");  
link.add("LiLy");  
link.add("John");  
link.add(2, "Linda");  
link.remove(1);  
for(int i = 0; i < link.size(); i++){  
    System.out.print(link.get(i) + "\t");  
}
```



# Set

- 集（Set）是最简单的一种集合：关心唯一性
  - 对象无序存储
  - 不能存储重复元素
- 主要实现类
  - HashSet：使用被插入对象的Hash码
  - LinkedHashSet：HashSet的ordered版本
  - TreeSet：二叉树结构，保证元素按照元素的自然顺序进行升序排序



# HashSet

- 基于 HashMap 实现的，HashSet 底层采用 HashMap 来保存所有元素。
- 不允许有重复元素。
- 不关心集合中元素的顺序。

# HashSet的方法

- 构造方法
  - HashSet()：构造一个空散列集，其底层 HashMap 实例的默认初始容量是 16
  - HashSet(Collection<? extends E> c)：构造一个散列集，并将集合中的所有元素添加到这个散列集中
  - HashSet(int initialCapacity)：构造一个空的具有指定容量(桶数)的散列集
- 其他方法
  - add(E e)：如果此 set 中尚未包含指定元素，则添加指定元素
  - clear()：从此 set 中移除所有元素
  - remove(Object o)：如果指定元素存在于此 set 中，则将其移除
  - size()：返回此 set 中的元素的数量（set 的容量）
  - isEmpty()：如果此 set 不包含任何元素，则返回 true

# HashSet

- 注意事项

- HashSet不能重复存储equals相同的数据。原因就是equals相同，数据的散列码也就相同（ hashCode必须和equals兼容）。大量相同的数据将存放在同一个散列单元所指向的链表中，造成严重的散列冲突，对查找效率是灾难性的。
- HashSet的存储是无序的，没有前后关系，他并不是线性结构的集合。
- hashCode必须和equals必须兼容

# 课堂练习

- HashSetDemo

```
HashSet set = new HashSet();  
set.add("aaa");  
set.add("bbb");  
set.add("ccc");  
set.add("bbb");  
System.out.print("输出结果: \n" + set);
```

输出结果:  
[aaa, ccc, bbb]

# TreeSet

- TreeSet ( 树集 ) 类似HashSet ( 散列集 ) 。
- 可以以任意顺序将元素插入到集合中。
- 对集合遍历时，每个值会自动的按照排序后的顺序呈现。
- 添加操作速率比散列集慢 ( 因为迭代器总是以排好序的顺序访问每个元素 ) 。

# TreeSet方法

- 构造方法
  - TreeSet() : 构造一个新的空 set , 该 set 根据其元素的自然顺序进行排序
- 其他方法
  - add() : 将指定的元素添加到此 set(如果该元素尚未存在 set 中)
  - remove(Object o) : 将指定的元素从 set 中移除 ( 如果该元素存在于此 set 中 )
  - first() : 返回此 set 中当前第一个 ( 最低 ) 元素
  - last() : 返回此 set 中当前最后一个 ( 最高 ) 元素
  - isEmpty() : 如果此 set 不包含任何元素 , 则返回 true
  - size() : 返回 set 中的元素数 ( set 的容量 )
  - .....

# 课堂练习

- TreeSetDemo



# LinkedHashSet

- 什么是LinkedHashSet
  - 在Hash的实现上添加了Linked的支持，在每个节点上通过一个链表串联起来，有确定的顺序。适用于有常量复杂度的高效存取性能要求、同时又要求排序的情况。
- 非同步。
- 继承于HashSet、又基于LinkedHashMap来实现。



# LinkedHashSet方法

- 构造方法
  - `LinkedHashSet()` : 构造一个带默认初始容量 (16) 和加载因子 (0.75) 的新空链接哈希 set
  - .....
- 其他方法
  - 包含继承自 `HashSet` 的方法 : `add`, `clear`, `isEmpty`, `remove`, `size`
  - .....

# 课堂练习

- LinkedListDemo

# 三个类的比较

- HashSet
  - 不能保证元素的排列顺序，顺序有可能发生变化
  - 不是同步的，集合元素可以是null,但只能放入一个null
  - 哈希表是通过使用称为散列法的机制来存储信息的，元素并没有以某种特定顺序来存放；
- LinkedHashSet
  - 以元素插入的顺序来维护集合的链接表，允许以插入的顺序在集合中迭代；
  - 遍历性能比HashSet好，但是插入时性能稍微逊色于HashSet
- TreeSet
  - 提供一个使用树结构存储Set接口的实现，对象以升序顺序存储，访问和遍历的时间很快；

# 课堂练习

- SetDemo
- 分别使用TreeSet、LinkedHashSet、LinkedSet三个类，在其中依次添加元素 "B "、 " A "、 " D "、 " E "、 " C "、 " F "，查看输出结果顺序区别。

```
HashSet<String> hs = new HashSet<String>();  
hs.add("B");  
hs.add("A");  
hs.add("D");  
hs.add("E");  
hs.add("C");  
hs.add("F");  
System.out.println("HashSet 顺序:\n"+hs);
```

# 课堂练习

```
LinkedHashSet<String> lhs = new LinkedHashSet<String>();  
lhs.add("B");  
lhs.add("A");  
lhs.add("D");  
lhs.add("E");  
lhs.add("C");  
lhs.add("F");  
System.out.println("LinkedHashSet 顺序:\n"+lhs);  
TreeSet<String> ts = new TreeSet<String>();  
ts.add("B");  
ts.add("A");  
ts.add("D");  
ts.add("E");  
ts.add("C");  
ts.add("F");  
System.out.println("TreeSet 顺序:\n"+ts);
```

# 课堂练习

- 结果对比（元素添加顺序：B、A、D、E、C、F）

HashSet 顺序:

[D, E, F, A, B, C]

LinkedHashSet 顺序:

[B, A, D, E, C, F]

TreeSet 顺序:

[A, B, C, D, E, F]

# Queue接口

- java.util.Queue
- 队列是一种特殊的线性表，只允许在表的前端（front，队头）进行删除操作，而在表的后端（rear，队尾）进行插入操作
- 继承了Collection接口
- LinkedList实现了Queue接口



# Queue接口常用方法

- `add(E e)`: 增加一个元素。成功时返回`true`，如果队列已满，则抛出一个 `IllegalStateException`异常
- `remove()`: 移除并返回队列头部的元素。如果队列为空，则抛出一个 `NoSuchElementException`异常
- `Element()`: 返回队列头部的元素。如果队列为空，则抛出一个 `NoSuchElementException`异常
- `offer(E e)`: 添加一个元素并返回`true`。如果队列已满，返回`false`
- `poll()`: 移除并返回队列头部的元素。如果队列为空，则返回`null`
- `peek()`: 返回队列头部的元素。如果队列为空，则返回`null`



# 其他方法

- `put(E e)`: 添加一个元素。如果队列满，则阻塞
- `take()`: 移除并返回队列头部的元素。如果队列空，则阻塞
- 注：
  - 由于`add()`和`remove()`方法在失败的时候会抛出异常，推荐使用`offer()`来加入元素，使用`poll()`来获取并移出元素。
  - `LinkedList`类实现了`Queue`接口，通常使用`LinkedList`代替`Queue`

# 课堂练习

```
public class TestQueue {  
    public static void main(String[] args) {  
        Queue<String> queue = new LinkedList<String>();  
        queue.offer("Hello");  
        queue.offer("World!");  
        queue.offer("你好！");  
        System.out.println(queue.size());  
        String str;  
        while((str=queue.poll())!=null){  
            System.out.print(str);  
        }  
        System.out.println();  
        System.out.println(queue.size());  
    }  
}
```

# MAP

- Map介绍
- 实现Map接口的常用类

# Map接口

- 映射 ( Map )
  - 对象以键 - 值对 ( key-value ) 存储
  - key不允许有重复, value允许有重复
- Map中元素, 可以将key序列、value序列单独抽取出来
  - 使用keySet()抽取key序列, 将map中的所有keys生成一个Set。
  - 使用values()抽取value序列, 将map中的所有values生成一个Collection。

# HashMap

- 基于哈希表的 Map 接口的实现
- HashMap是非线程安全的
- 常用方法：
  - Object put(K key,V value)
  - Object get(Object K)
  - containsKey(Object K)
  - containsValue(Object v)
- 遍历HashMap

# TreeMap、LinkedHashMap

- TreeMap
  - 基于红黑树实现
  - 按照元素的自然顺序排序
- LinkedHashMap
  - HashMap的ordered版本

# 迭代器 ( Iterator )

- Iterator : “轻量级” 对象
- iterator()方法是java.lang.Iterable接口,被Collection继承。
- 主要功能：用于对容器的遍历
- 主要方法
  - boolean hasNext():判断是否有可以元素继续迭代
  - Object next() : 返回迭代的下一个元素
  - void remove() : 从迭代器指向的集合中移除迭代器返回的最后一个元素

# 迭代器 ( Iterator )

```
Set<String> name = new HashSet<String> ();
name.add("LL");
name.add("VV");
name.add("WW");

Iterator<String> it = name.iterator();
while(it.hasNext()){
    String n = it.next();
    if(n.equals("WW")){
        it.remove();
        //name.add( "ww" );//运行时异常, ConcurrentModificationException
    }
}
System.out.println(name);
```



# 迭代器 ( Iterator )

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "LL");
map.put(2, "LW");
//map.remove(1);
Iterator it = map.entrySet().iterator();
while(it.hasNext()){
    Map.Entry m = (Entry) it.next();
    if(m.getValue().equals("LL")){
        //it.remove();
        m.setValue("PP");
    }
    System.out.println(m.getKey());
    System.out.println(m.getValue());
}
System.out.println(map);
```

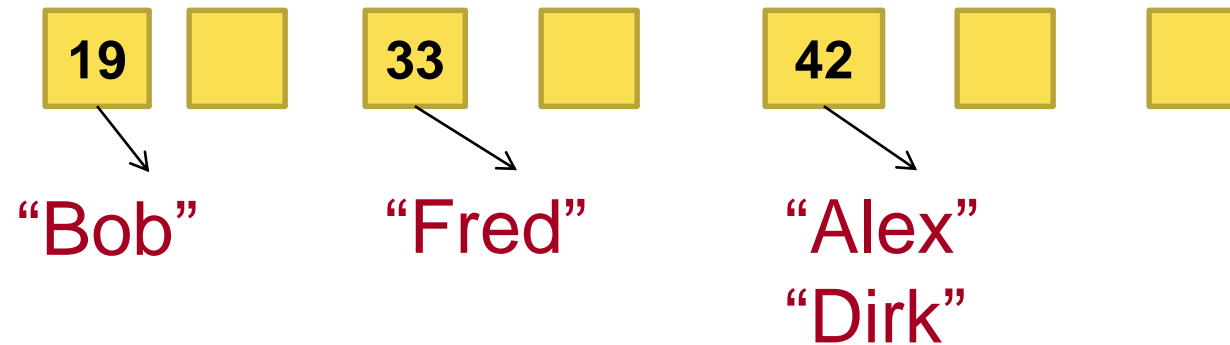
# 泛型

- 重写hashCode和equals方法
- 泛型

# Hash算法

键	哈希算法	哈希码
Alex	$A(1)+L(12)+E(5)+X(24)$	$=42$
Bob	$B(2)+O(15)+B(2)$	$=19$
Dirk	$D(4)+I(9)+R(18)+K(11)$	$=42$
Fred	$F(6)+R(18)+E(5)+(D)$	$=33$

哈希桶



# Hash算法

```
class Cat{
    private String name;
    public Cat(String name){
        this.name=name;  }
    public boolean equals(Object o){
        if(o instanceof Cat  && (Cat)o.getName().equals(this.getName())){
            return true;
        }else{
            return false;
        }
    }
    public int hashCode(){
        return name.hashCode()*11;
    }
}
```

# hashCode契约

- equals方法比较相等的两个对象hashCode返回值必须相同。
- equals方法比较不相等的两个对象hashCode返回值可以不相同，也可以相同。
- 如果没有修改对象的equals比较内的任何属性信息，则这个对象多次调用hashCode返回相同结果。

# 重写equals方法

首先，确保检测  
对象类型正确



其次，比较我们  
所关心的属性

```
class Cat{  
    private String name;  
    public Cat(String name){  
        this.name=name;  
    }  
    public boolean equals(Object o){  
        if(o instanceof Cat &&  
        (Cat)o.getName().equals(this.getName())){  
            return true;  
        }else{  
            return false;  
        }  
    }  
}
```

# Equals契约

## 自反的

- 对任意的x引用，`x.equals(x)`都应返回true

## 对称的

- 对任意的x、y引用，当且仅当`x.equals ( y )`时，`y.equals(x)`才返回true

## 传递的

- 对任意的x、y、z引用,如果`x.equals(y)`和`y.equals(z)`返回true，则`z.equals(x)`返回true

## 一致的

- 对任意的x、y引用，对象中的信息没有做休息的前提下多次调用`x.equals(y)`返回一致的结果

## 非空

- 对任意非NULL引用x,`x.equals(null)`返回false




# 泛型

- 重写hashCode和equals方法
- 泛型

# 引入

- 两个模块的功能非常相似，一个是处理int数据，另一个是处理String数据，或者其他自定义的数据类型
- 解决方案
  - 写多个方法处理每个数据类型
- 例子

```
class Num {  
    private int var;  
    public int getVar() {  
        return var;  
    }  
    public void setVar(int var) {  
        this.var = var;  
    }  
}
```



如果是String类型的数据呢？

# 引入

- 为了提高代码的重用性，用通用的数据类型Object来实现

- 优点

- 灵活
- 通用性强

- 缺点

- 处理值类型时，会出现装箱、折箱操作，性能损失非常严重
- 处理引用类型时，虽然没有装箱和折箱操作，但将用到数据类型的强制转换操作，增加处理器的负担
- 如果处理数据是数组，数组中数据类型不一致
  - 运行时类型转换异常
  - 编译器无法检查出来

```
class Obj{  
    private Object var;  
    public Object getVar() {  
        return var;  
    }  
    public void setVar(Object var) {  
        this.var = var; //隐式装箱操作  
    }  
}
```

```
Node1 x = new Node1();  
stack.Push(x);  
Node2 y = (Node2)stack.Pop();
```

# 为什么使用泛型？

- 既增强代码通用性，又避免编译器无法检查编译错误的问题——泛型。
- 泛型用一个通用的数据类型T来代替Object，在类实例化时指定T的类型，运行时自动编译为本地代码，运行效率和代码质量都有很大提高，并且保证数据类型安全。
- 泛型的作用就是提高代码的重用性，避免强制类型转换，减少装箱拆箱提高性能，减少错误。

```
class Info<T> {  
    private T var;  
    public T getVar() {  
        return var;  
    }  
    public void setVar(T var) {  
        this.var = var;  
    }  
}
```

# 泛型的概念

- 泛型 ( Generics )
  - 所谓泛型，即通过参数化类型来实现在同一份代码上操作多种数据类型。
  - 泛型编程是一种编程范式，它利用“参数化类型”将类型抽象化，从而实现更为灵活的复用。
  - 泛型赋予了代码更强的类型安全，更好的复用，更高的效率，更清晰的约束。

# Java泛型与C++中模板的比较

- 泛型的语法在表面上与C++中的模板非常类似，但是二者之间有着本质的区别。
- Java 中的泛型**只接受引用类型**作为类型参数。
  - 如：可以定义 `List<Integer>`，不可以定义 `List<int>`。
- C++中`List<A>`和`List<B>`实际上是两个不同的类，而java中`ArrayList<Integer>`和 `ArrayList<String>`共享相同的类。
  - `ArrayList<T>`

# 泛型的声明

- 语法
  - class 名称<泛型列表>
  - 如：class ArrayList<E>
  - 参数E是泛型，它可以是任何类或接口（除基本数据类型外）



# 泛型的应用

- 集合中使用泛型
  - `List<E>`
- 方法参数
  - `void do(List<Dog> dogs){...}`
- 返回类型
  - `List<Dog> getDogs(){...}`
- 变量声明的类型必须匹配传递给实际对象的类型
  - `List<Animal> animals = new ArrayList<Animal>();`
  - `List<Animal> animals = new ArrayList<Dog>();`
- 声明一个类型参数为<Object>的List，相当于非泛型集合（可将任何Object放入集合中）



# 使用通配符 ( ? )

- 接受所声明变量类型的任何子类型
  - `void addAnimal(List<? extends Animal> animals)`
  - `Animal`可以是类或接口
- 接受父类型的变量
  - `void addAnimal(List<? super Dog> animals)`
  - 接受`super`右边类型或其超类型
- `List<?>` 与 `List<? extends Object>` 完全相同
- `List<Object>` 与 `List<?>` 完全不同

# 通配符使用限制

- 泛型通配符只能用于引用的声明中，不可以在创建对象时使用
  - `Fruit<?> fruit=new Fruit<?>();` ❌
- 不可以使用采用了泛型通配符的引用调用使用了泛型参数的方法

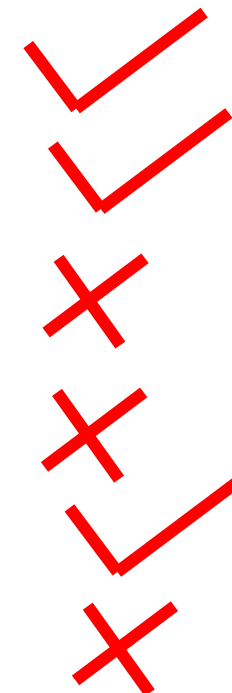
```
public class Fruit {  
    public static void main(String[] args) {  
        Colors<?> fruit = new Colors<String>();  
        fruit.setColor("red");  
    }  
}
```

 ❌

```
class Colors<T> {  
    private T color;  
    public void setColor(T color) {  
        this.color = color;  
    }  
    public String getColor() {  
        return this.color.toString();  
    }  
}
```

# 判断

- `List<?> list = new ArrayList<Dog>();`
- `List<? extends Animal> list = new ArrayList<Dog>();`
- `List<?> list = new ArrayList<? extends Animal>();`
- `List<? extends Animal> list = new ArrayList<Integer>();`
- `List<? super Dog> list = new ArrayList<Animal>();`
- `List<? super Animal> list = new ArrayList<Dog>();`



# 总结

- 集合使用方法
- 泛型

# Java容器的4种基本形式





**Thank You**