320341 Programming in Java

JACOBS
UNIVERSITY

Fall Semester 2014

Lecture 9: Collections

Instructor: Jürgen Schönwälder

Slides: Bendick Mahleko

Objectives

JACOBS UNIVERSITY

This lecture introduces the following

- The Collections Framework
- Collections Interfaces
- Collection Concrete Classes
- Introduction to generic programming

The Collections Framework

A collections framework is a unified architecture for representing and manipulating collections.

All collections framework contain the following:

-**Interfaces**: allow collections to be manipulated independent of details about their representation.

-**Implementations**: concrete implementations of collection interfaces.

-**Algorithms**: methods that polymorphically perform useful computations

The Collections Framework

JACOBS UNIVERSITY

Benefits of Java Collections Framework

Benefit	Notes
Reduced programming effort	Key data structures already implemented
Better program quality & performance	Well-tested code written & tested by experts
Better software reuse	Implemented data structures can be reused
Reduced effort to design new APIs	No need to reinvent the wheel when designing new APIs that rely on collections

Collections Framework Interface Overview

JACOBS UNIVERSITY

There are *two fundamental interfaces* for collections

- **Collection** – The root interface for the Collections hierarchy
- The following interfaces extend the Collection:

  `Set, List, SortedSet, Queue, Deque, BlockingQueue and BlockingDeque`

- **Map** (holds key: value pairs) – represent mappings rather than Collections

  `Map, SortedMap, ConcurrentMap`

Collections Framework Interface Overview

JACOBS UNIVERSITY

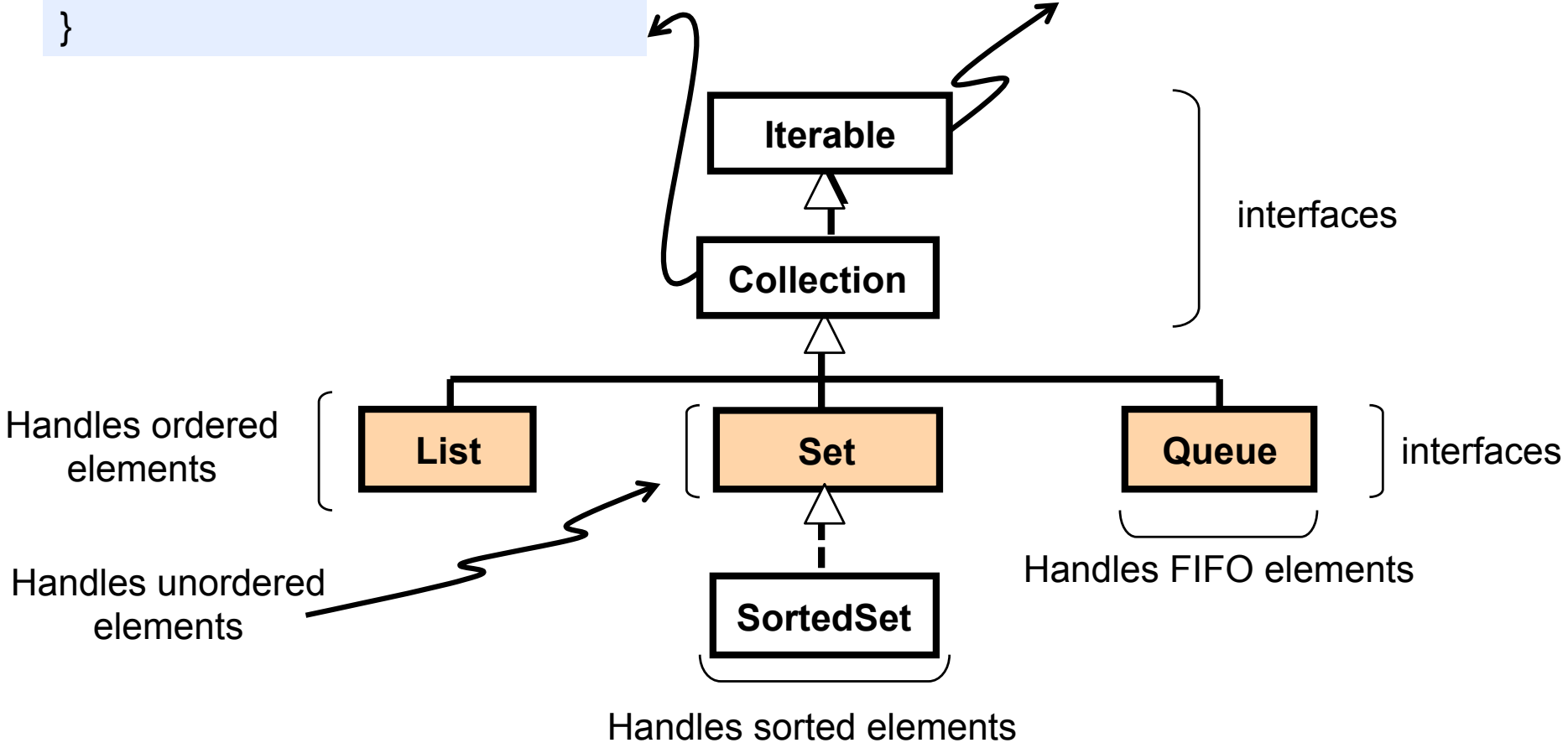
Interface	Description
Collection	The root interface in the collections hierarchy from which interfaces **Set**, **Queue**, and **List** are derived
Set	A collection that does not contain duplicates (models the mathematical set abstraction)
List	An ordered collection that can contain duplicate elements
Map	Associates keys to values and cannot be contain duplicate keys
Queue	Typically a first-in, first-out collection that models a waiting line; other orders can be specified e.g., priority queues

The classes and interfaces of the **collections framework** are members of package **java.util**.

The Collections Framework

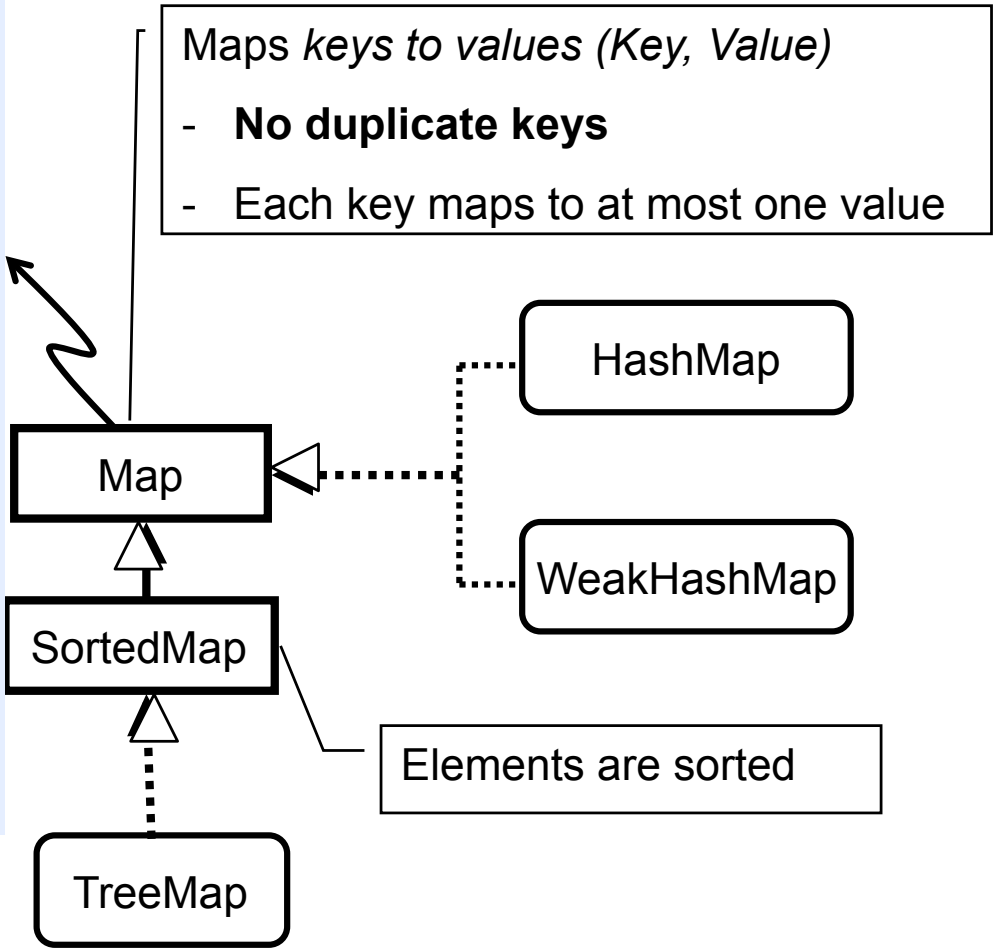
```
public interface Collection<E> {
    boolean add() ;
    Iterator <E> iterator ();
    …
}
```

```
public interface Iterable {
    Iterator iterator();
}
```



The Collections Framework

```
public interface Map<K, V> {
        void clear()
        boolean containsKey(Object key)
        boolean containsValue(Object value)
        Set entrySet()
        boolean equals(Object o)
        V get(Object key)
        int hashCode()
        boolean isEmpty()
        Set keySet()
        V put(K key, V value)
        void putAll(Map t)
        V remove(Object key)
        int size()
        Collection values()
}
```



Concrete class

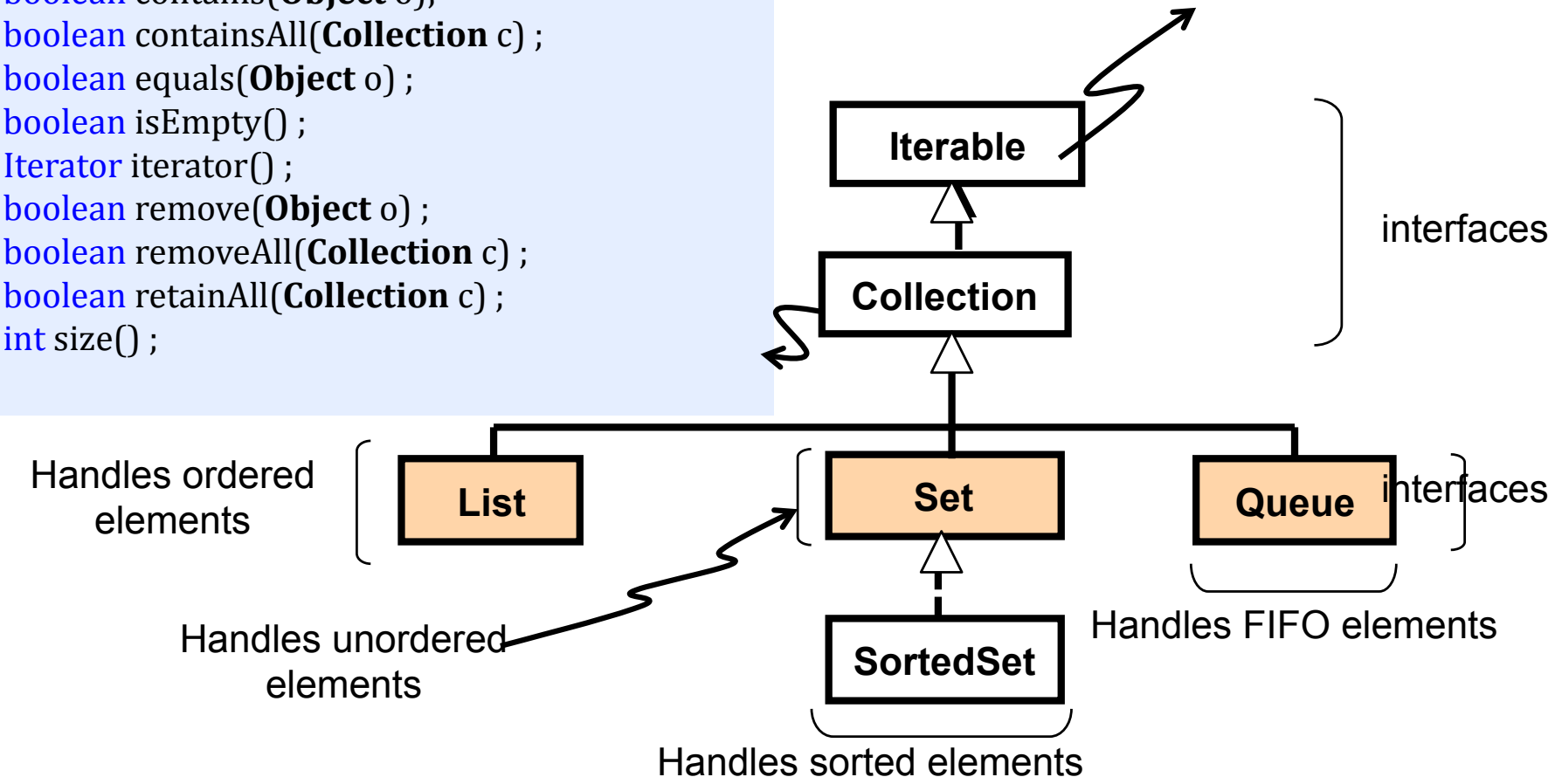
interface class

A map models a mathematical function abstraction

The Collections Framework

```
public interface Collection<E> extends Iterable<E> {
        boolean add(E o) ;
        boolean addAll(Collection c) ;
        void clear() ;
        boolean contains(Object o);
        boolean containsAll(Collection c) ;
        boolean equals(Object o) ;
        boolean isEmpty() ;
        Iterator iterator() ;
        boolean remove(Object o) ;
        boolean removeAll(Collection c) ;
        boolean retainAll(Collection c) ;
        int size() ;
}
```

```
public interface Iterable {
   Iterator iterator();
}
```



The Collections Framework

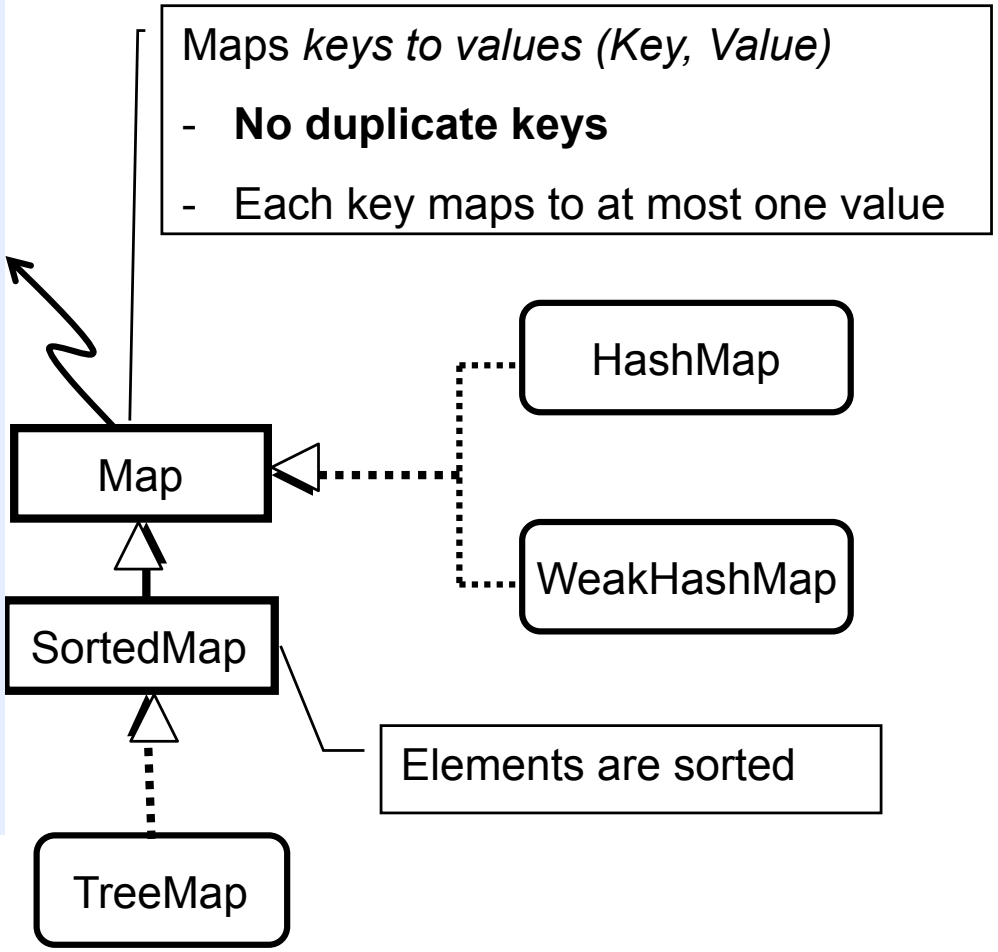
JACOBS UNIVERSITY

Bulk operations perform an operation on an entire Collection

Bulk operations	Notes
containsAll	true if target Collection contains all elements in specified Collection
addAll	Adds all elements in specified Collection to target Collection
removeAll	Removes all elements from the target Collection that are also contained in the specified Collection
retainAll	Removes all elements from target Collection that are not contained in the specified Collection
clear	Removes all elements from the Collection

The Collections Framework

```
public interface Map<K, V> {
        void clear()
        boolean containsKey(Object key)
        boolean containsValue(Object value)
        Set entrySet()
        boolean equals(Object o)
        V get(Object key)
        int hashCode()
        boolean isEmpty()
        Set keySet()
        V put(K key, V value)
        void putAll(Map t)
        V remove(Object key)
        int size()
        Collection values()
}
```



Concrete class

interface class

A map models a mathematical function abstraction

The Collections Framework

JACOBS UNIVERSITY

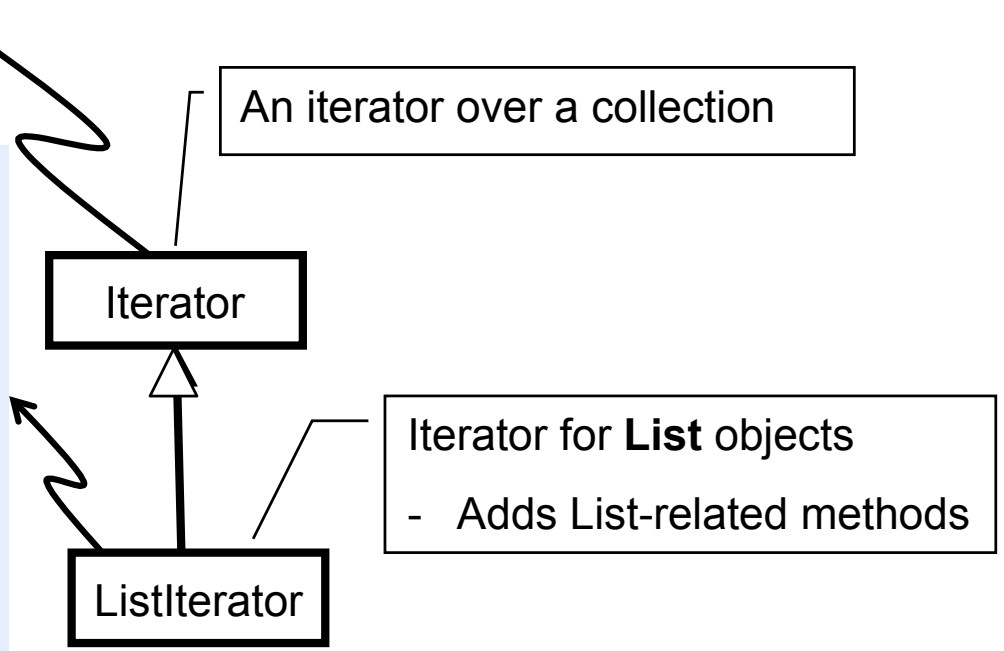
Map bulk operations perform an operation on an entire Collection

Bulk operations	Notes
putAll	Adds all elements in specified Map to target Map
clear	Removes all elements from the Map

The Collections Framework: The Iterator

```
public interface  Iterator {
        boolean hasNext()
        E next()
        void remove()
}
```

```
public ListIterator<E> extends Iterator<E> {
       void add(E o)
        boolean hasNext()
        boolean hasPrevious()
        E next()
        int nextIndex()
        E previous()
        int previousIndex()
        void remove()
        void set(E o)
}
```



Framework Overview

JACOBS UNIVERSITY

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Generic Programming - Introduction

JACOBS UNIVERSITY

"**Generics** provides a way to communicate the type of a collection to the compiler, so that it can be checked" taken from Java Doc

```
public class ArrayList { // before JDK 5.0
   public Object get(int i) { . . . }
   public void add(Object o) { . . . }
   . . .
   private Object[] elementData;
}
```

Two problems

- A cast is necessary to retrieve a value

```
ArrayList files = new ArrayList();
String filename = (String) names.get(0);
```

- There is no error checking. Values of any class can be added

```
files.add(new File(". . ."));
```

Example

```
// Removes 4-letter words from c. Elements must be strings
static void expurgate(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
        if (((String) i.next()).length() == 4)
            i.remove();
}
```

```
// Removes the 4-letter words from c
static void expurgate(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
        if (i.next().length() == 4)
            i.remove();
}
```

Type parameter

Example using Generics

Read <Type> as “Type of” e.g., “Collection of String c”

- An unsafe cast has been eliminated
- The code is clearer and safer (type errors are detected at compile-time)

Iterating over Elements in a Collection

JACOBS UNIVERSITY

iterator method returns an object that implements the **Iterator** interface

- Use the iterator object to visit elements in **Collection** one by one

```
Collection<String> c = . . .;
Iterator<String> iter = c.iterator();
while (iter.hasNext()) {
   String element = iter.next();
   do something with element
}
```

The “for each” loop (JDK 5.0) works with any objects implementing the **Iterable** interface

c is a **Collection**

```
for (String element : c) {
   do something with element
}
```

Removing Elements from a Collection

JACOBS UNIVERSITY

The remove method of the **Iterator** interface removes the element that was returned by the last call to *next*

```
Iterator<String> it = c.iterator();
it.next(); // skip over the first element
it.remove(); // now remove it
```

- Calling `remove` before `next` causes an **IllegalStateException**

```
it.remove();
it.remove(); // Error!
```

- First call next to jump over the element to be removed

```
it.remove();
it.next();
it.remove(); // Ok
```

Concrete Collections

JACOBS UNIVERSITY

All classes in the table below (except those with names ending in **Map**) implement the **Collection** interface

Collection type	Description
ArrayList	An indexed sequence that grows and shrinks dynamically
LinkedList	An ordered sequence that allows *efficient insertions* and *removal at any location*
HashSet	An unordered collection that rejects duplicates
TreeSet	A sorted set
LinkedHashSet	A set that remembers the order in which elements were inserted
HashMap	A data structure that stores key/value associations
TreeMap	A map in which the keys are sorted
LinkedHashMap	A map that remembers the order in which entries were added
WeakHashMap	A map with values that can be reclaimed by the garbage collector if they are not used elsewhere

Concrete Collections cont…

JACOBS UNIVERSITY

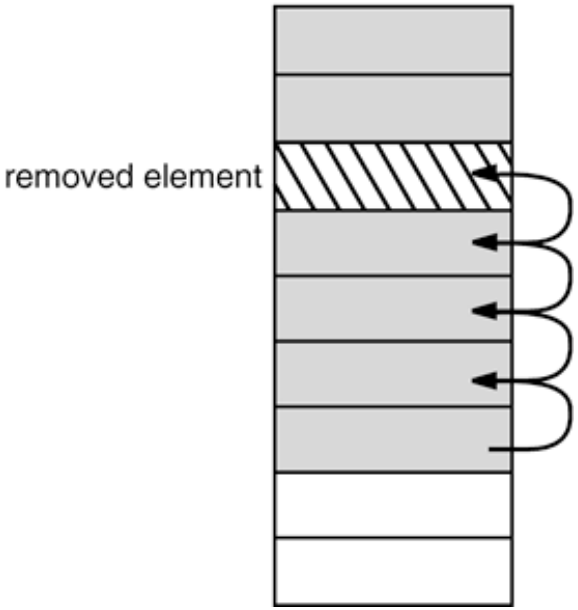
Collection type	Description
ArrayDeque	An double-ended queue that is implemented as a circular array
EnumSet	A set of enumerated type values
PriorityQueue	A collection that allows efficient removal of the smallest element
EnumMap	A map in which the keys belong to an enumerated type
IdentityHashMap	A map with keys that can be compared by ==, not equals

Concrete Collections: Linked Lists

JACOBS UNIVERSITY

Disadvantages of Arrays/ **ArrayList**

- Removing an element from the middle of an array is expensive

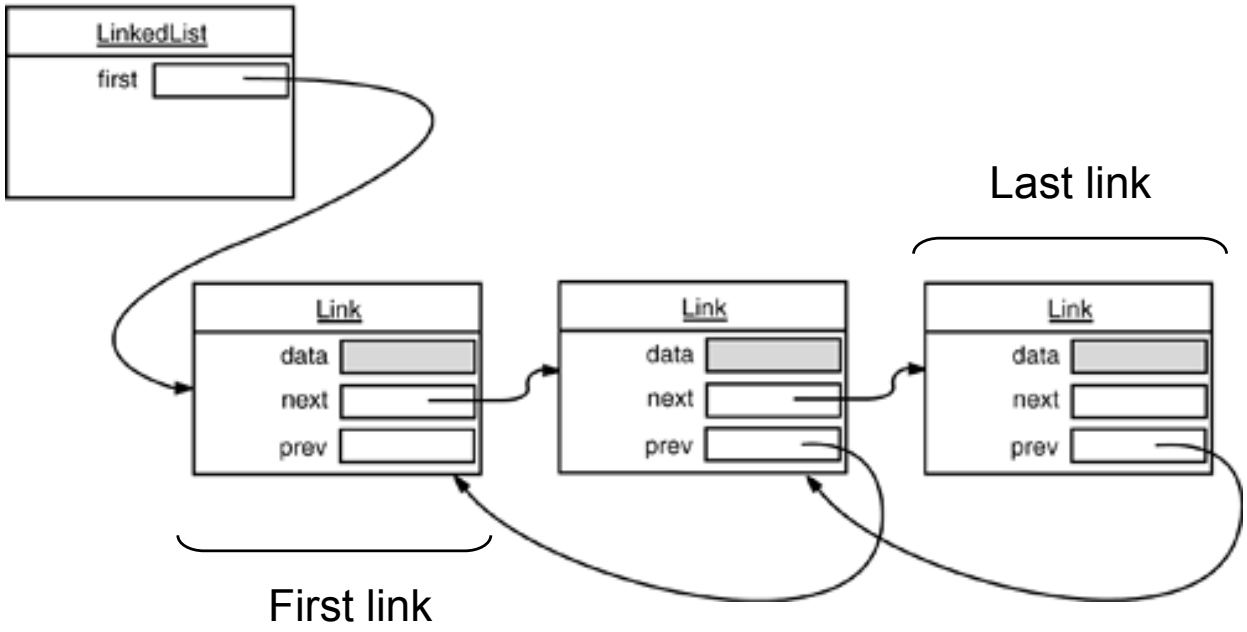


- The same is true for inserting elements in the middle

Concrete Collections: Linked Lists

The **LinkedList** data structure solves the Array/ **ArrayList** problem

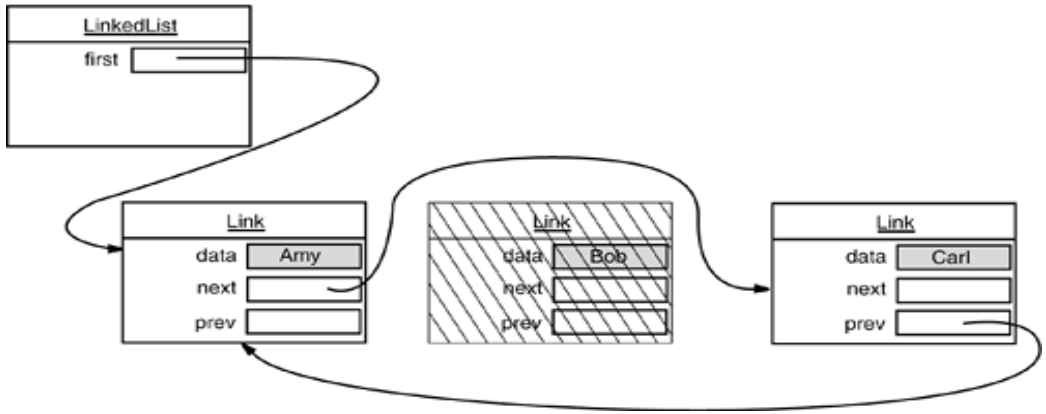
- An array stores object references in consecutive memory locations
- A **LinkedList** stores each object in a separate link
- Each link also stores a reference to the next link in the sequence
- *In Java, all links are doubly linked*



Concrete Collections: Linked Lists

Removing an element from the middle of a **LinkedList** is inexpensive

- Only the links around the element to be removed need to be updated



Example: adding and removing elements

```
List<String> staff = new LinkedList<String>(); // LinkedList implements List
staff.add("Amy"); // LinkedList add method adds to the end of the collection
staff.add("Bob");
staff.add("Carl");
Iterator iter = staff.iterator();
String first = iter.next();      // visit first element
String second = iter.next();  // visit second element
iter.remove();                          // remove last visited element
```

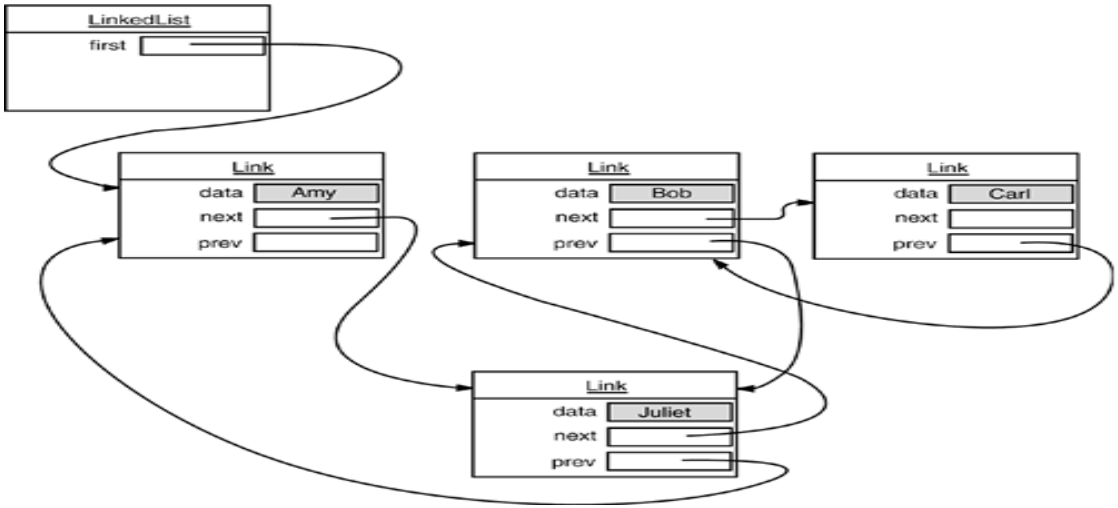
Concrete Collections: Linked Lists

Adding Element at the Middle

- Achieved via the *add* method of the **ListIterator** interface

```
List<String> staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.listIterator();
iter.next();                    // skip past first element
iter.add("Juliet");
```

ListIterator has *add* method that adds a new element before the iterator position



Concrete Collections: Linked Lists

A `set` method replaces the last element returned by a call to `next` or `previous` with a new element

```
ListIterator<String> iter = list.listIterator();
String oldValue = iter.next();          // returns first element
iter.set(newValue);                     // sets first element to newValue
```

Linked List Summary

- Use **ListIterator** to traverse elements of a **LinkedList** in either direction
- Use an array or **ArrayList** for random access into a collection
- A **LinkedList** is used to minimize the cost of insertion/ removal at the middle of a list

Concrete Collections: Array Lists

JACOBS UNIVERSITY

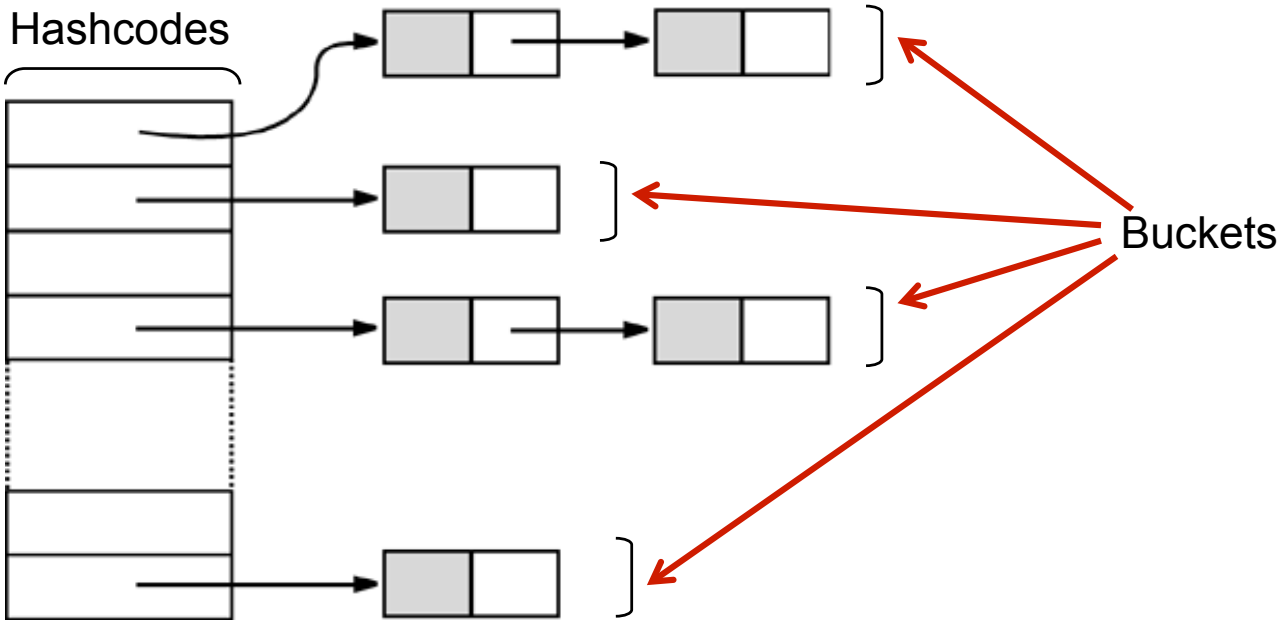
Like the **LinkedList**, the **ArrayList** class implements the **List** interface

- Random access of collection elements is not appropriate for **LinkedList**
- It is appropriate for the **ArrayList**
- The `get` and `set` methods are used for random access to the collection
- An **ArrayList** encapsulates a dynamically reallocated array of objects

Concrete Collections: HashSet

This is a **Set** implemented using a **Hashtable**

- Duplicates not allowed
- A **Hashtable** computes a hash code for each object that is used to store the object
- A **Hashtable** is an array of **LinkedLists**
- Each list is called a *bucket*



Concrete Collections: HashSet

JACOBS UNIVERSITY

Example

```
Set<String> words = new HashSet<String>(); // HashSet implements Set
Scanner in = new Scanner(System.in);
while (in.hasNext()) {
    String word = in.next();
    words.add(word);
}
…
Iterator<String> iter = words.iterator();
for (int i = 1; i <= 20; i++)
        System.out.println(iter.next());
```

Concrete Collections: TreeSet

JACOBS UNIVERSITY

A **TreeSet** is similar to a **HashSet** with one improvement

- *A TreeSet is an sorted collection*
- Elements are inserted in any order, but sorted automatically when iterating

```
SortedSet<String> sorter = new TreeSet<String>(); // TreeSet implements SortedSet
sorter.add("Bob");
sorter.add("Amy");
sorter.add("Carl");
for (String s : sorter)
        System.out.println(s);
```

- The values are printed in sorted order (lexicographic order): Amy Bob Carl
- Sorting is accomplished by a tree data structure
- The *red-black tree* is used in the current implementation

Concrete Collections: TreeSet

JACOBS UNIVERSITY

Object Comparison

- **TreeSet** assumes that elements implement the **Comparable** interface

```
public interface Comparable<T> {
   int compareTo(T other);
}
```

The call `a.compareTo(b)` returns

- 0 if and b are equal
- A negative integer if a comes before b
- A positive integer if a comes after b in the sort order

Define a sort order by implementing **Comparable<T>** for your objects

Concrete Collections: Maps

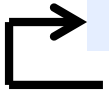
JACOBS UNIVERSITY

- A map stores key/ value pairs; retrieves a value by providing a key
- **HashMap** and **TreeMap** are concrete classes implementing maps

HashMap

- Hashes the keys to organize storage

```
// HashMap implements Map
Map<String, Employee> staff = new HashMap<String, Employee>();
Employee harry = new Employee("Harry Hacker");
staff.put("987-98-9996", harry);
```



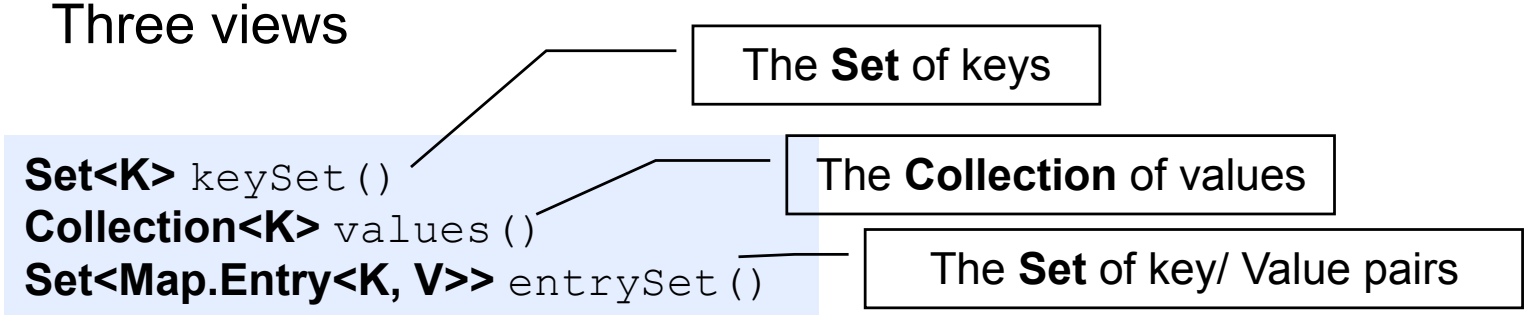
Key/ value pairs are added to the map using the `put(key, value)` method

- Retrieving an object

```
String s = "987-98-9996";
e = staff.get(s);                    // gets harry
```

Concrete Collections: Maps

Three views



TreeMap

- Uses a total ordering on keys to organize them into a search tree
- Use if need for sorting is necessary
- Look at **TreeMap** API

Concrete Collections: Weak Hash Maps

JACOBS UNIVERSITY

The **WeakHashMap** class

- Removes key/value pairs when the only reference to the key is the one from the hash table entry

LinkedHashSet and **LinkedHashMap** classes

- Remember the order in which items were added
- Elements joined using doubly linked list as they are being added

Legacy Container Classes

JACOBS UNIVERSITY

The following legacy classes exist since JDK 1.0

Enumeration	Analogous to **Iterator**
Vector	Analogous to **ArrayList**, but all methods are synchronized
Stack	Subclass of **Vector** that added methods to push and pop elements
Dictionary	Analogous to **Map** interface, **Dictionary** is an abstract class
Hashtable	Analogous to **HashMap**
Properties	Subclass of **HashMap**, maintains key/ value pairs

Examine the API for these classes

Reading Assignment

JACOBS UNIVERSITY

- Core Java 2 Volume I, 9th Ed. Chapter 13. Collections by Horstmann and Cornell

- *Java Collections Framework (n.d) http://docs.oracle.com/javase/7/docs/technotes/guides/collections/ Last visited: 16 October 2013*