Chapter 2
Java Collections Framework (JCF)

Happy Buddy

SB: What is Collections in Java?
HB: Collections are used to store, retrieve, manipulate, and communicate aggregate data.
SB: How many types of Collections are available in java?
HB: Collections come in four basic flavors: List, Map, Set, Queue
SB: What about Generics?
HB: It allows developers to create custom variations of their code for different types.
Lets see all things in detail...Are you ready for the RIDE? ;-)

Sad Buddy

Introduction

Choosing the best *data structures* and *algorithms* for a particular task is one of the keys to developing high-performance software.

Software version/update!!

Cont..

A **data structure** is a collection of data organized in some fashion.

The structure not only stores data but also support operations for accessing and manipulating data.

Cont..

- **In object-oriented thinking**, a data structure, also known as a *container* or *container object*, is an object that stores other objects, referred to as data or elements.
- **To define** a data structure is essentially to define a class for it...
- **To create** a data structure is therefore to create an instance from the class…

Java Collections Framework

- Java provides several data structures that can be used to organize and manipulate data efficiently.
- These are commonly known as

Java Collections Framework (JCF)

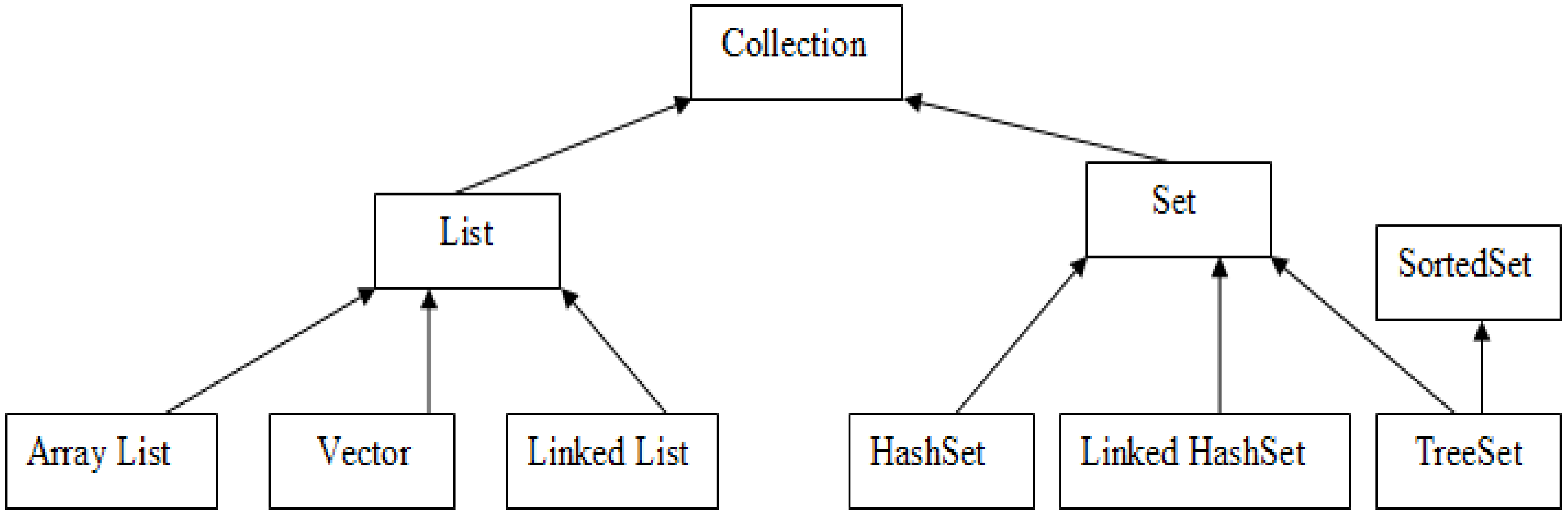
- All the interfaces and classes defined in the Java Collections Framework are grouped in the **`java.util`** package.

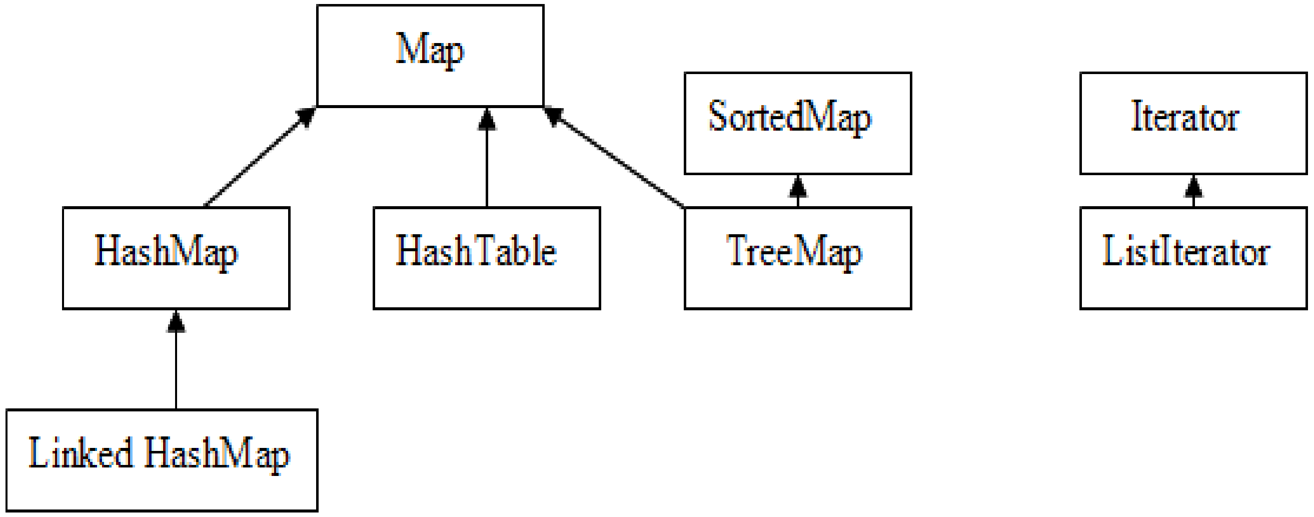
What JCF is made of?

- ☞ JCF supports TWO types of containers:

(1) One for storing a collection of elements is simply called a *collection*.

(2) The other, for storing key/value pairs, is called a *map*.

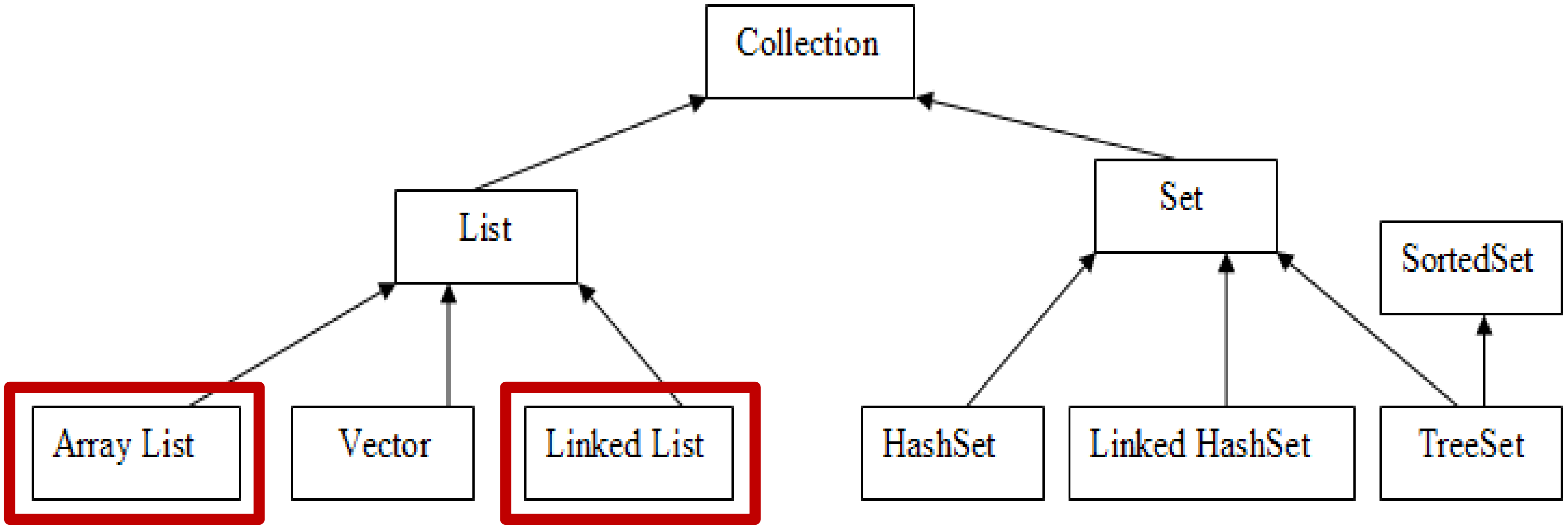

Collection
List
Set
SortedSet
Array List
Vector
Linked List
HashSet
Linked HashSet
TreeSet

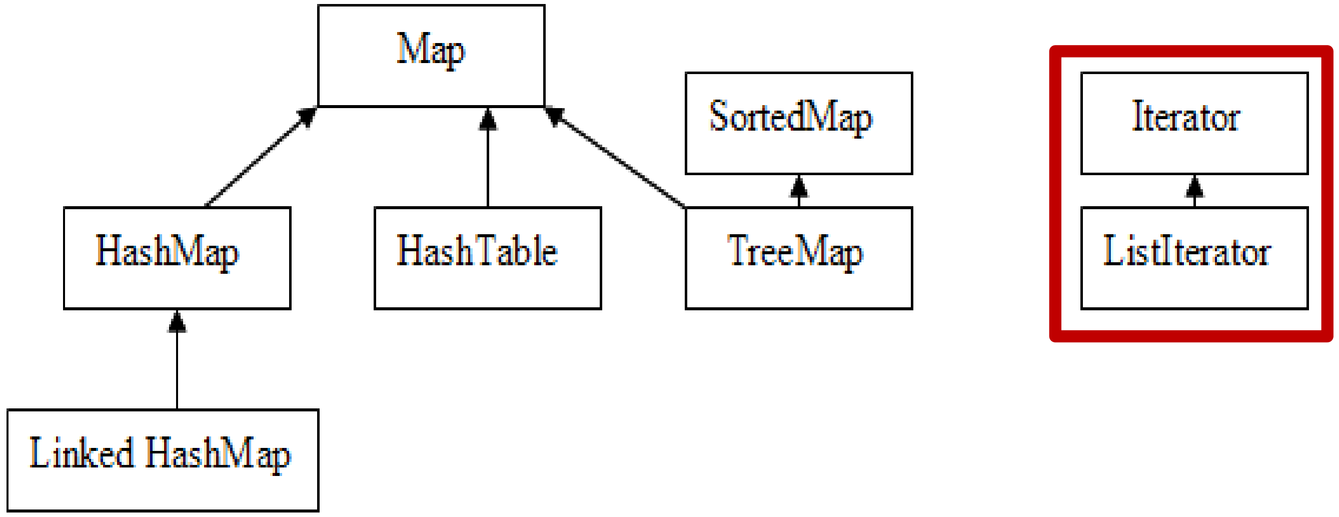

Map
SortedMap
Iterator
HashMap
HashTable
TreeMap
ListIterator
Linked HashMap

Kinds of Collections

- Collection--a group of objects, called elements
  - List--an ordered collection, duplicates are allowed
  - Set--An <u>unordered</u> collection with no duplicates
    - SortedSet--An <u>ordered</u> collection with no duplicates
- Map--a collection that maps keys to values
  - SortedMap--a collection ordered by the keys
- Note that there are two distinct hierarchies

Collection (Part 1)
ArrayList & LinkedList


Collection
List
Set
SortedSet
Array List
Vector
Linked List
HashSet
Linked HashSet
TreeSet


Map
SortedMap
Iterator
HashMap
HashTable
TreeMap
ListIterator
Linked HashMap

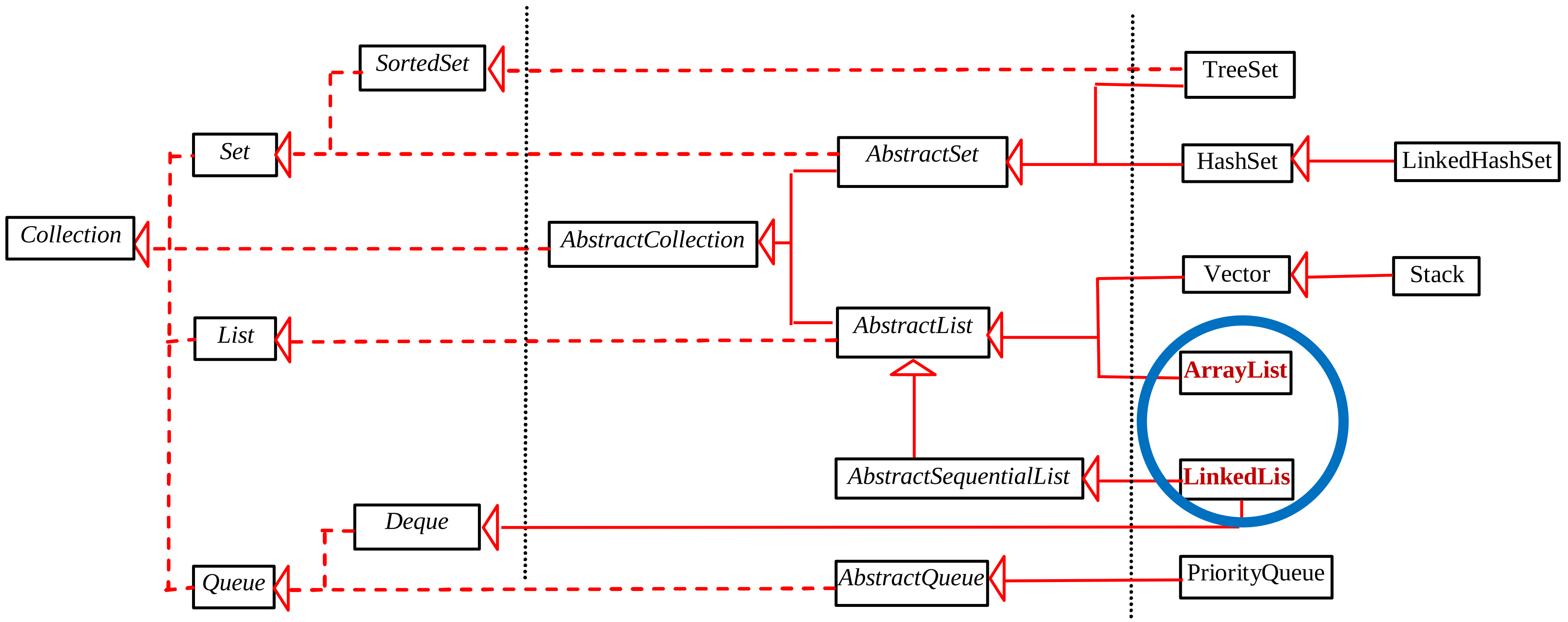
Collection

☞ There are different kinds of collections:

- ■ **Lists** store an ordered collection of elements.
- ■ **Sets** store a group of non-duplicate elements.
- ■ **Queues** store objects that are processed in first-in, first-out fashion.

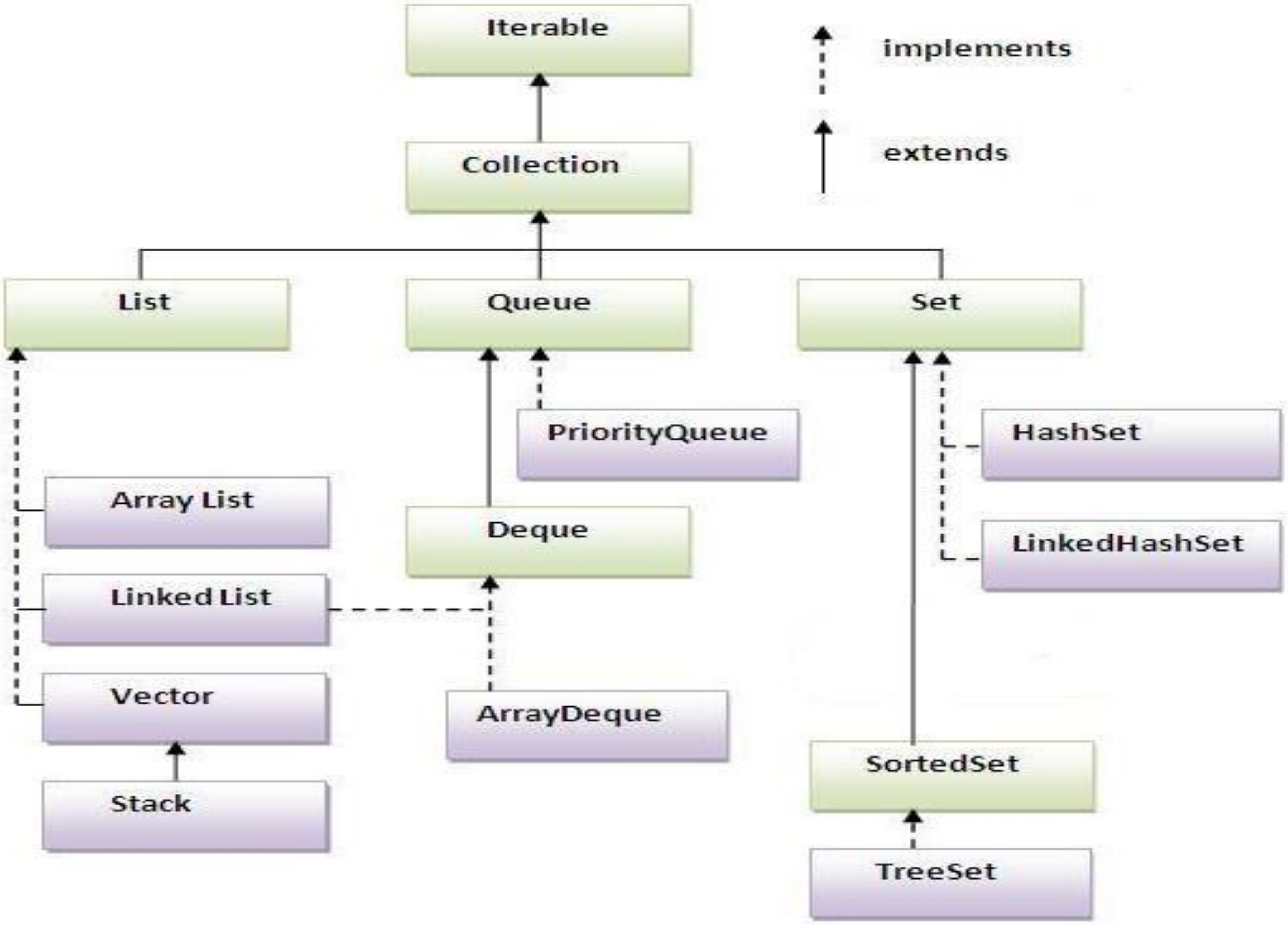
The common features of these collections are defined in the interfaces, and implementations are provided in concrete classes, as shown in the next slide.

Hierarchy of Collection Framework

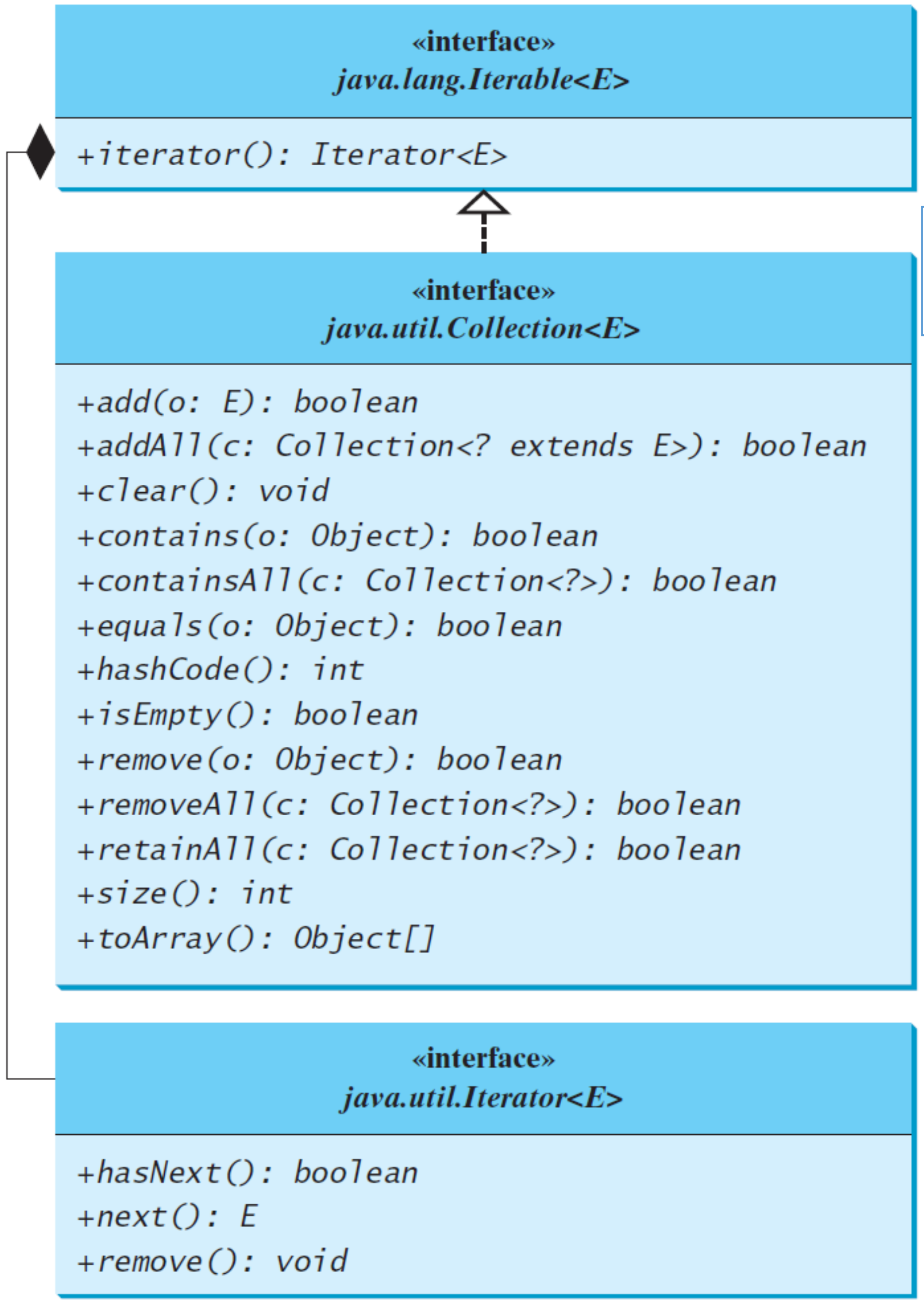


Interfaces **Abstract Classes** **Concrete Classes**

Hierarchy of Collection Framework



The Collection Interface



«interface»
java.lang.Iterable<E>

```
+iterator(): Iterator<E>
```

Returns an iterator for the elements in this collection.

The Collection interface is the root interface for manipulating a collection of objects.

«interface»
java.util.Collection<E>

`+add(o: E): boolean`	Adds a new element `o` to this collection.
`+addAll(c: Collection<? extends E>): boolean`	Adds all the elements in the collection `c` to this collection.
`+clear(): void`	Removes all the elements from this collection.
`+contains(o: Object): boolean`	Returns true if this collection contains the element `o`.
`+containsAll(c: Collection<?>): boolean`	Returns true if this collection contains all the elements in `c`.
`+equals(o: Object): boolean`	Returns true if this collection is equal to another collection `o`.
`+hashCode(): int`	Returns the hash code for this collection.
`+isEmpty(): boolean`	Returns true if this collection contains no elements.
`+remove(o: Object): boolean`	Removes the element `o` from this collection.
`+removeAll(c: Collection<?>): boolean`	Removes all the elements in `c` from this collection.
`+retainAll(c: Collection<?>): boolean`	Retains the elements that are both in `c` and in this collection.
`+size(): int`	Returns the number of elements in this collection.
`+toArray(): Object[]`	Returns an array of `Object` for the elements in this collection.

«interface»
java.util.Iterator<E>

`+hasNext(): boolean`	Returns true if this iterator has more elements to traverse.
`+next(): E`	Returns the next element from this iterator.
`+remove(): void`	Removes the last element obtained using the next method.

Lists

A list is a popular data structure to store data in sequential order.

For example, a list of students, a list of available rooms, a list of cities, and a list of books, etc. can be stored using lists.

The common operations on a list are usually the following:

- Retrieve an element from this list.
- Insert a new element to this list.
- Delete an element from this list.
- Find how many elements are in this list.
- Find if an element is in this list.
- Find if this list is empty.

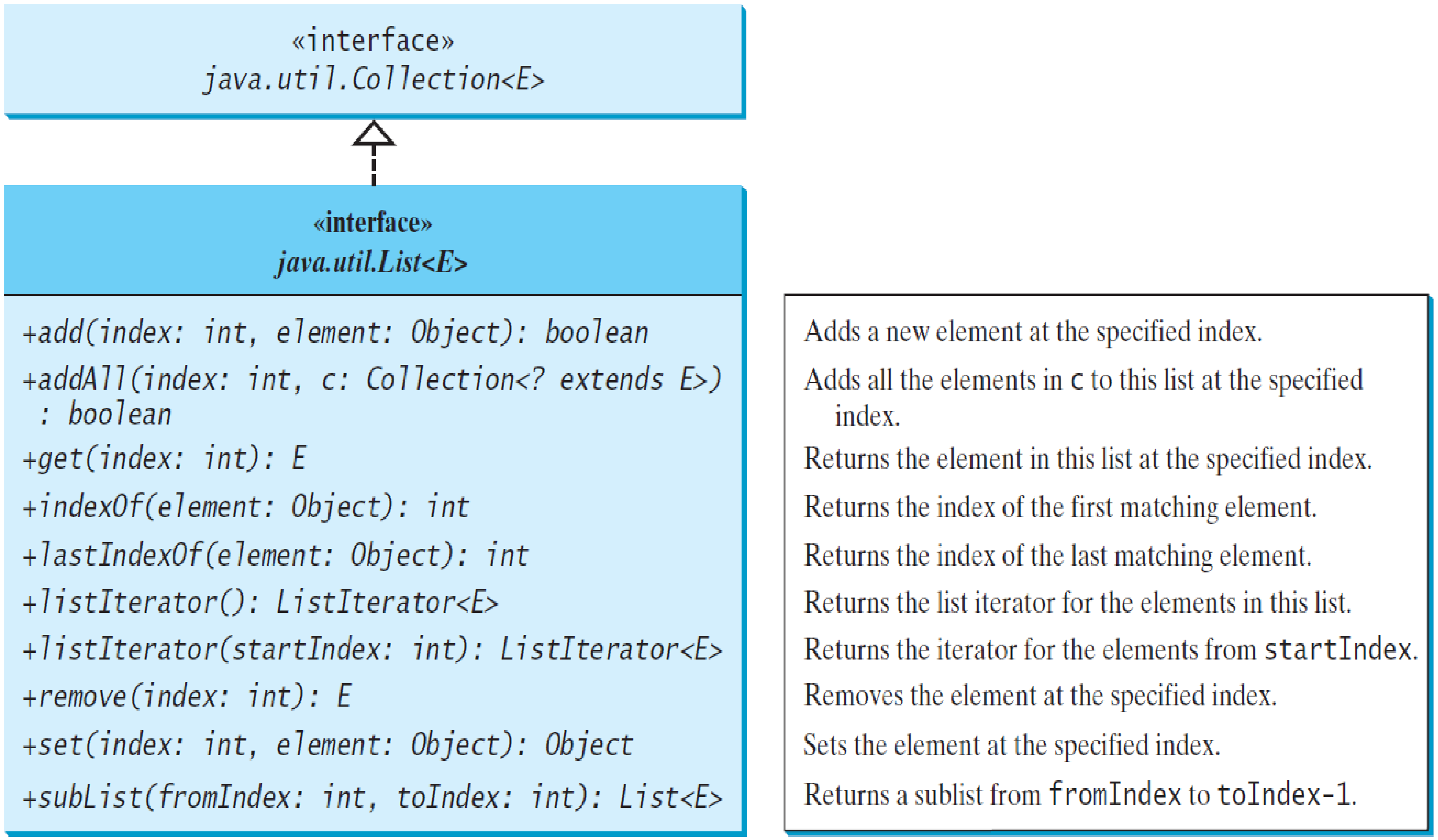
The List<E> Interface

A set stores non-duplicate elements.

To allow duplicate elements to be stored in a collection, you need to use a list.

A list can not only store duplicate elements, but can also allow the user to specify where the element is stored. The user can access the element by index.

The List Interface



+add(index: int, element: Object): boolean	Adds a new element at the specified index.
+addAll(index: int, c: Collection<? extends E>) : boolean	Adds all the elements in c to this list at the specified index.
+get(index: int): E	Returns the element in this list at the specified index.
+indexOf(element: Object): int	Returns the index of the first matching element.
+lastIndexOf(element: Object): int	Returns the index of the last matching element.
+listIterator(): ListIterator<E>	Returns the list iterator for the elements in this list.
+listIterator(startIndex: int): ListIterator<E>	Returns the iterator for the elements from startIndex.
+remove(index: int): E	Removes the element at the specified index.
+set(index: int, element: Object): Object	Sets the element at the specified index.
+subList(fromIndex: int, toIndex: int): List<E>	Returns a sublist from fromIndex to toIndex-1.

Three Ways to Implement Lists in Java

1. Using arrays (simulation)

2. Using ArrayList class

3. Using LinkedList class

1. Using Arrays

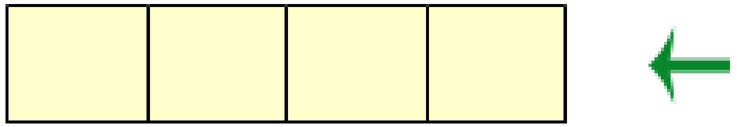
Array is a fixed-size data structure. Once an array is created, its size cannot be changed.

Nevertheless, **you can still use array to implement dynamic data structures**.

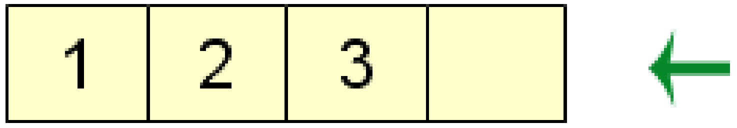
What is the trick?

The trick is to create a new larger array to replace the current array if the current array cannot hold new elements in the list.

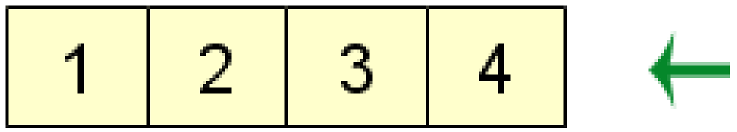
Disadvantage of Array



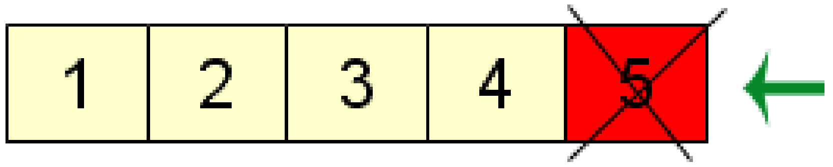
Create an empty integer array.



Fill it partially with some data. The array component without a number indicates allocated but unused space. This is space you could have used for something better.



Add another element. We now have a full array to which we can not add any more elements. We can delete or replace, but we can not add.



If the array is full, you can not add more elements, because arrays have a fixed size. Linked Lists do not.

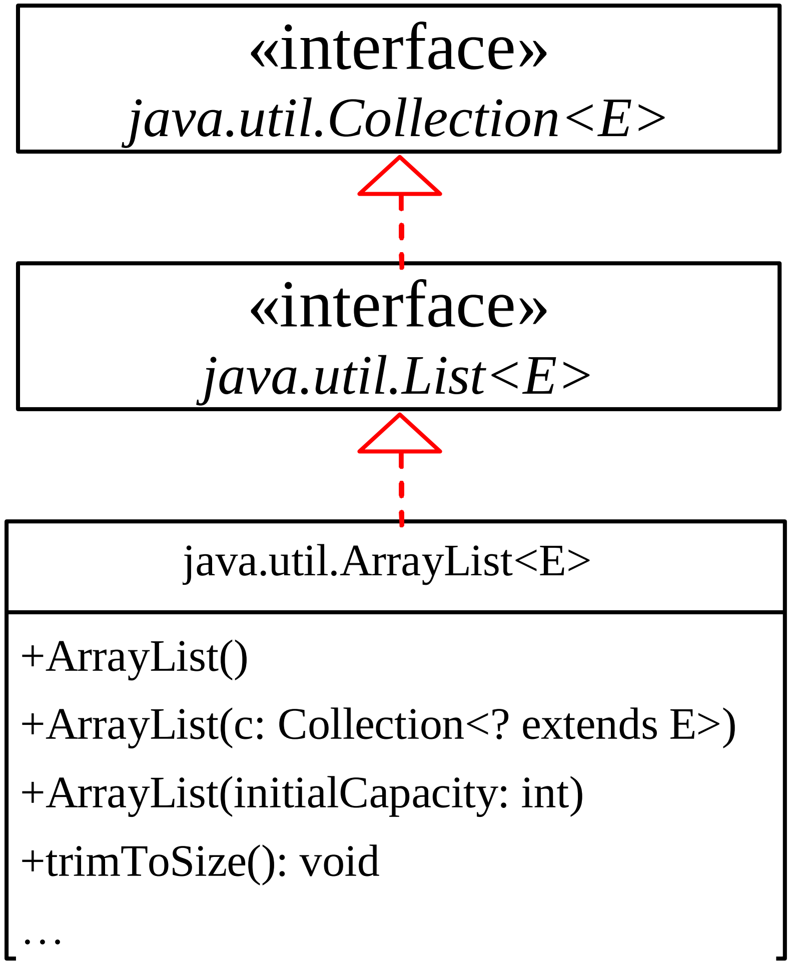
Do like the first way?

- Not really!
- Two concrete implementations of the List API in the Java Collections:
  - java.util.ArrayList
  - java.util.LinkedList
- These concrete classes that were developed at Sun.

2. Using ArrayList class

- ArrayList supports dynamic arrays that can grow as needed.
- Store unlimited number of objects

java.util.ArrayList


«interface»
java.util.Collection<E>
«interface»
java.util.List<E>
java.util.ArrayList<E>
+ArrayList()
+ArrayList(c: Collection<? extends E>)
+ArrayList(initialCapacity: int)
+trimToSize(): void
…

+ArrayList()	Creates an empty list with the default initial capacity.
+ArrayList(c: Collection<? extends E>)	Creates an array list from an existing collection.
+ArrayList(initialCapacity: int)	Creates an empty list with the specified initial capacity.
+trimToSize(): void	Trims the capacity of this ArrayList instance to be the list's current size.

How to use ArrayList

1. Import the package `java.util.ArrayList;`

2. Instantiate the ArrayList object (to create list)

E.g.,

```
ArrayList<String> mylist= new ArrayList<String>();
```

3. Call methods on created object list

E.g.,

```
 mylist.add (“Red”);
```

Creating a ArrayList

- Specify, in angle brackets after the name, the type of object that the class will hold
- Examples:
  - `ArrayList<String> vec1 = new ArrayList<String>();`
  - `ArrayList<String> vec2 = new ArrayList<String>(10);`

Adding elements to a ArrayList

- boolean add(Object ***obj***)
  - Appends the object ***obj*** to the end of this ArrayList
  - With generics, the ***obj*** must be of the correct type, or you get a compile-time (syntax) error
  - Always returns true
    - This is for consistency with other, similar classes
- void add(int ***index***, Object ***element***)
  - Inserts the ***element*** at position ***index*** in this ArrayList
  - The ***index*** must be greater than or equal to zero and less than or equal to the number of elements in the ArrayList
  - With generics, the ***obj*** must be of the correct type, or you get a compile-time (syntax) error

Removing elements

- boolean remove(Object ***obj***)
  - Removes the first occurrence of ***obj*** from this ArrayList
  - Returns true if an element was removed
  - Uses equals to test if it has found the correct element
- void remove(int ***index***)
  - Removes the element at position ***index*** from this ArrayList
- void clear()
  - Removes all elements

Accessing elements

- Object get(int ***index***)
  - Returns the component at position ***index***

  ```
  - ArrayList<String> myList = new
    ArrayList<String>();
    myList.add("Some string");
    String s = myList.get(0);
  ```

- Object sublist(int ***fromIndex,*** int ***toIndex***)
  - Returns the component at position ***fromIndex*** to ***toIndex***

Searching a ArrayList

- boolean contains(Object ***element***)
  - Tests if ***element*** is a component of this ArrayList
  - Uses equals to test if it has found the correct element

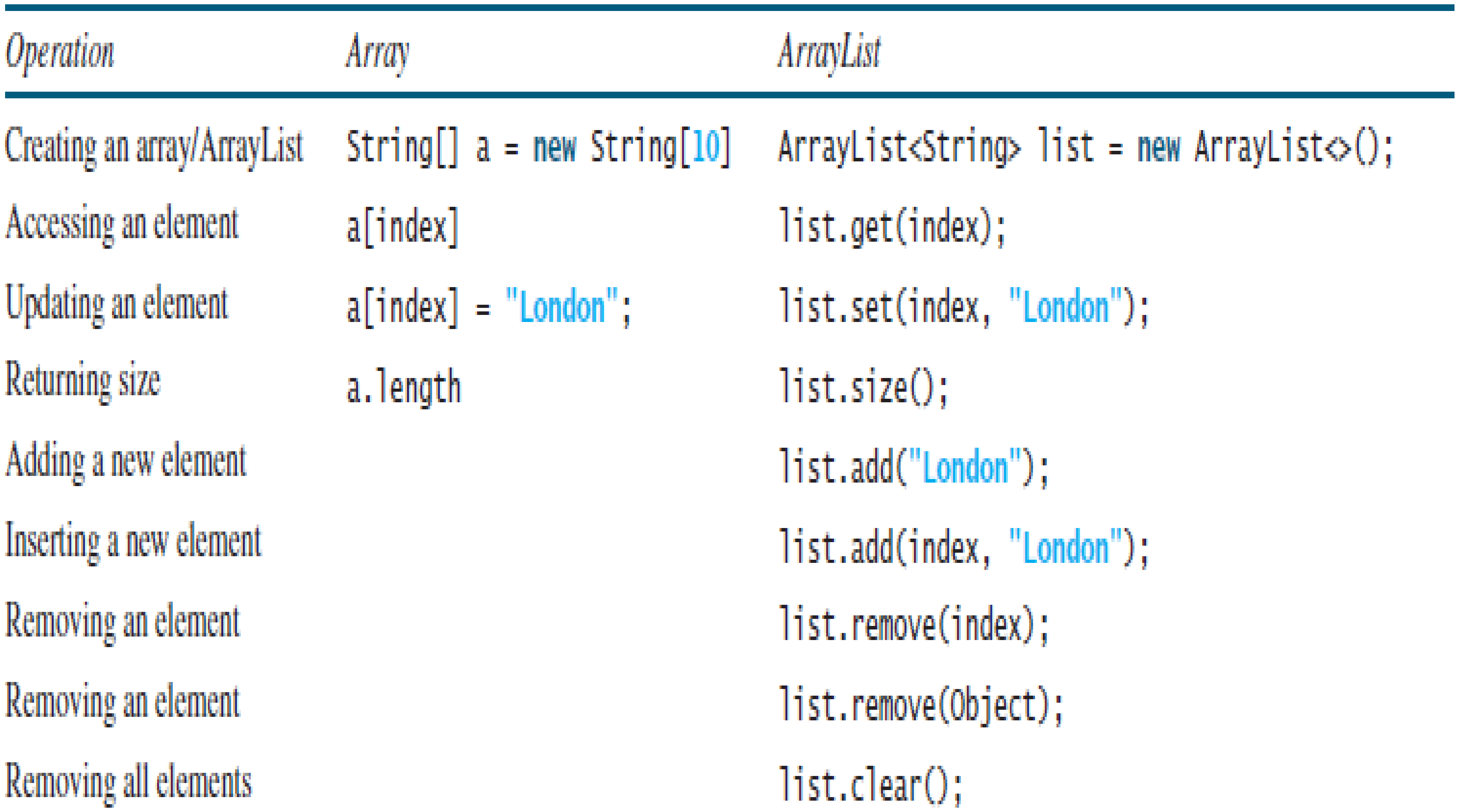
- int indexOf(Object ***element***)
  - Returns the index of the first occurrence of ***element*** in this ArrayList
  - Uses equals to test if it has found the correct element
  - Returns -1 if ***element*** was not found in this ArrayList

- int lastIndexOf(Object ***element***)
  - Returns the index of the last occurrence of ***element*** in this ArrayList
  - Uses equals to test if it has found the correct element
  - Returns -1 if ***element*** was not found in this ArrayList

Getting information

- boolean isEmpty()
  - Returns true if this ArrayList has no elements
- int size()
  - Returns the number of elements currently in this ArrayList
- Object[ ] toArray()
  - Returns an array containing all the elements of this ArrayList in the correct order

Differences and Similarities between **Arrays** and **ArrayList**



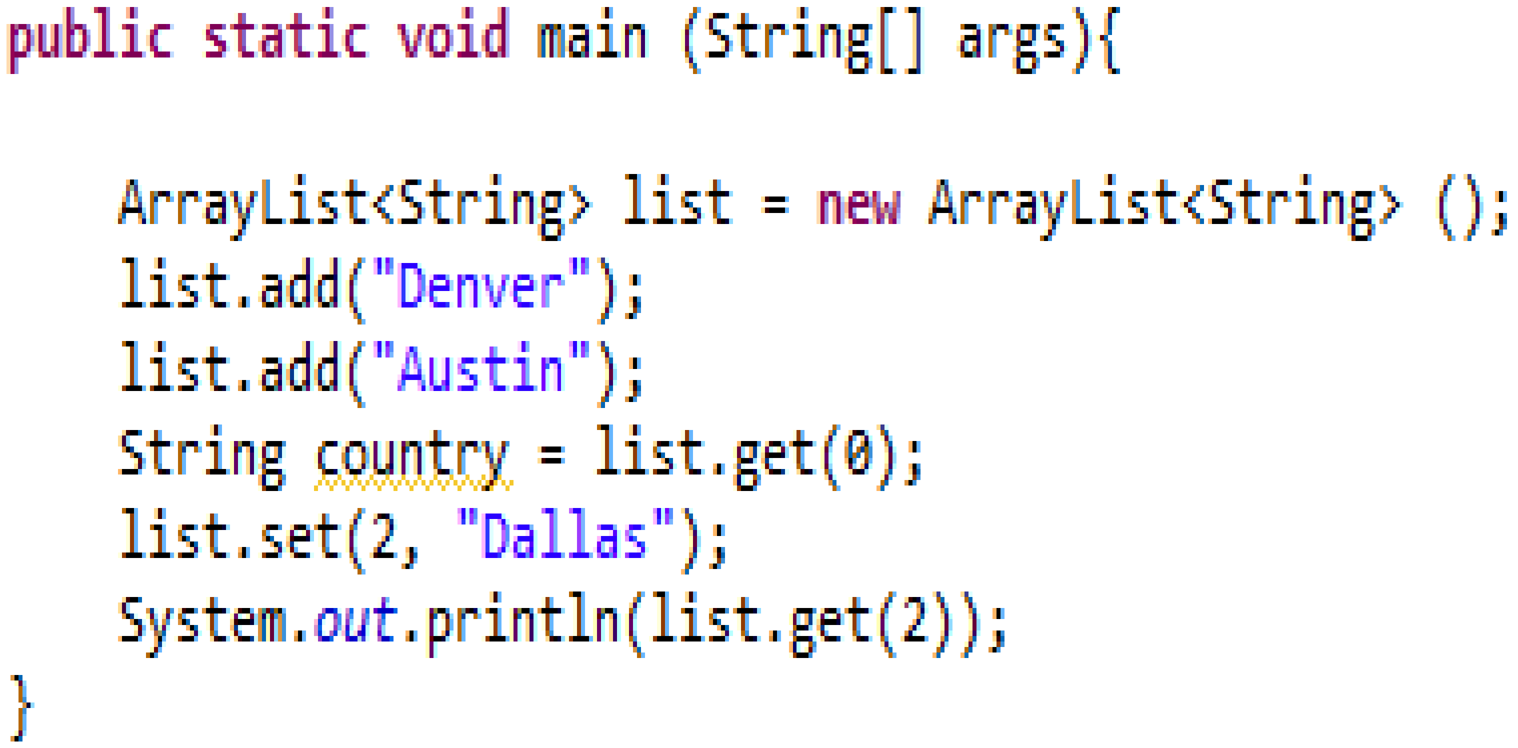
Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

Warning: size of List dynamically changes

(Java checks before and after any operation)

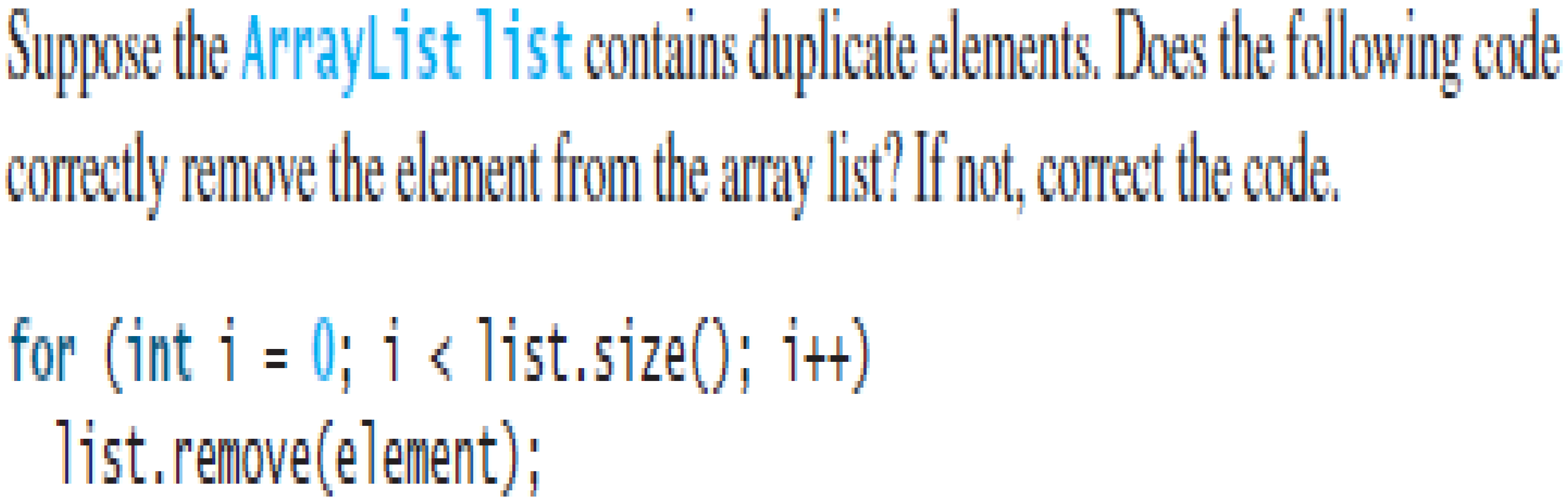
- `add (e)` → appends e to the end of list.
- `add (index, e)` → inserts e to the specified index.
- `remove (e)` → remove the first occurrence of e in the list (start from index 0)
- `remove (index)` → removes the element at the specified index in this list.
- ~~`remove (index, e)`~~
- `set (index, e)` → replaces the element at the specified position in this list with the specified element

Identify the errors in the following code?



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

    ArrayList<String> list = new ArrayList<String> ();
    list.add("Denver");
    list.add("Austin");
    String country = list.get(0);
    list.set(2, "Dallas");
    System.out.println(list.get(2));
}
```

Another question?



Suppose the `ArrayList list` contains duplicate elements. Does the following code correctly remove the element from the array list? If not, correct the code.

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

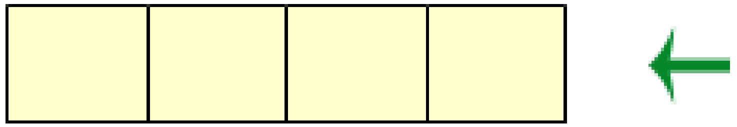
Same stands correct when use remove (index)

A linked list is like a conga line!

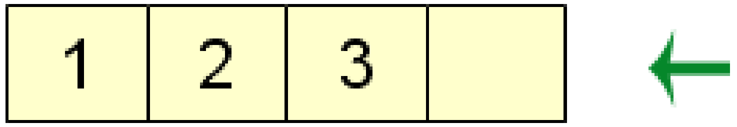
3. Using LinkedList class

- Linked Lists are a very common way of storing arrays of data.
- The major benefit of linked lists is that you do not specify a fixed size for your list. The more elements you add to the chain, the bigger the chain gets.
- Data structures (e.g. Stacks, Queues, Binary Trees) are often implemented using the concept of linked lists.

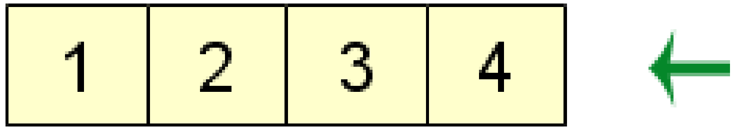
Disadvantage of Array



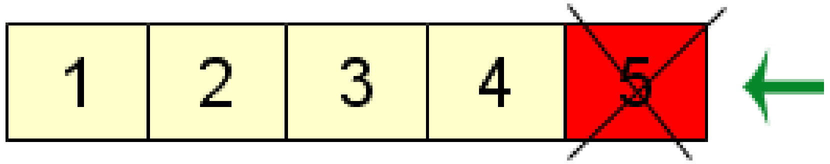
Create an empty integer array.



Fill it partially with some data. The array component without a number indicates allocated but unused space. This is space you could have used for something better.

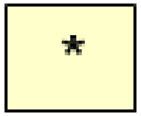


Add another element. We now have a full array to which we can not add any more elements. We can delete or replace, but we can not add.

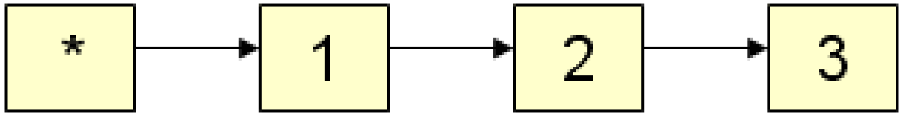


If the array is full, you can not add more elements, because arrays have a fixed size. Linked Lists do not.

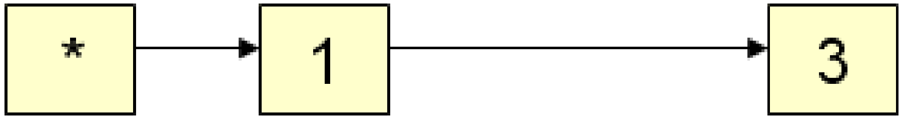
Advantage of LinkedList



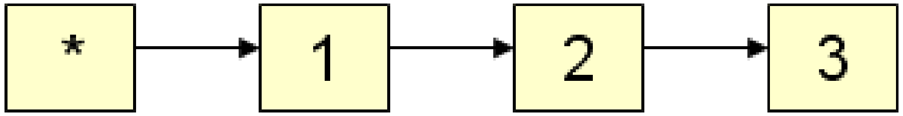
This is what an empty Linked List looks like. The * is an empty node (each element in a Linked List is called a node) which has its next-node reference set to the first node in the list. Since we don't have a first node, its next-node is a null pointer.



This is a Linked List with three nodes. Each node points to the next node in the chain. As mentioned above, * is an empty node with a reference to the first node. [3] is the last node in the chain with next == null.



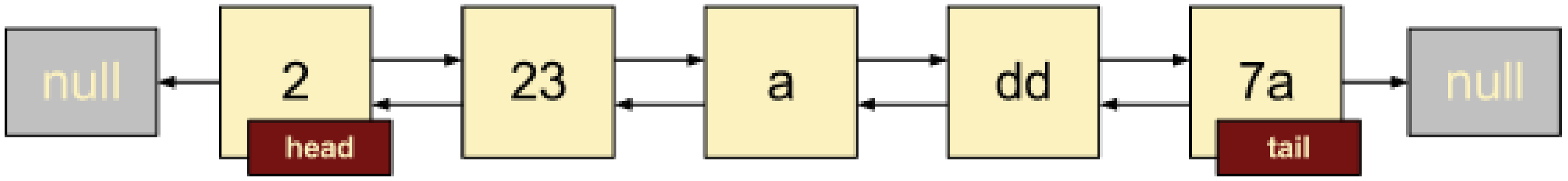
Here we have deleted node [2], so node [1] (previously pointing to [2]) now points to [3]. If we didn't change the reference, node [3] and any nodes behind it would have no references from your program and get lost. If this happens and you're using C or C++, you have a memory leak. If you're using Java, the nodes would get automatically garbage collected. Either way, make sure you update the references!



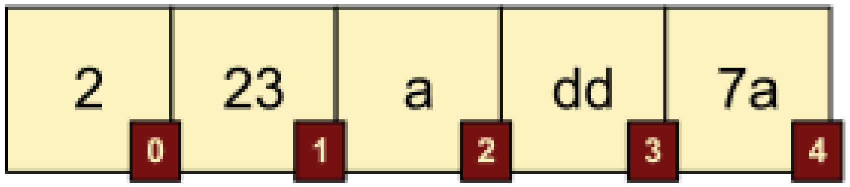
For whatever reasons, we've decided to add node [2] back between nodes [1] and [3]. The reference from [1] is set to [2], and the reference from [2] is set to the old reference of [1], which is [3].

Array vs. Linked List

Linked List



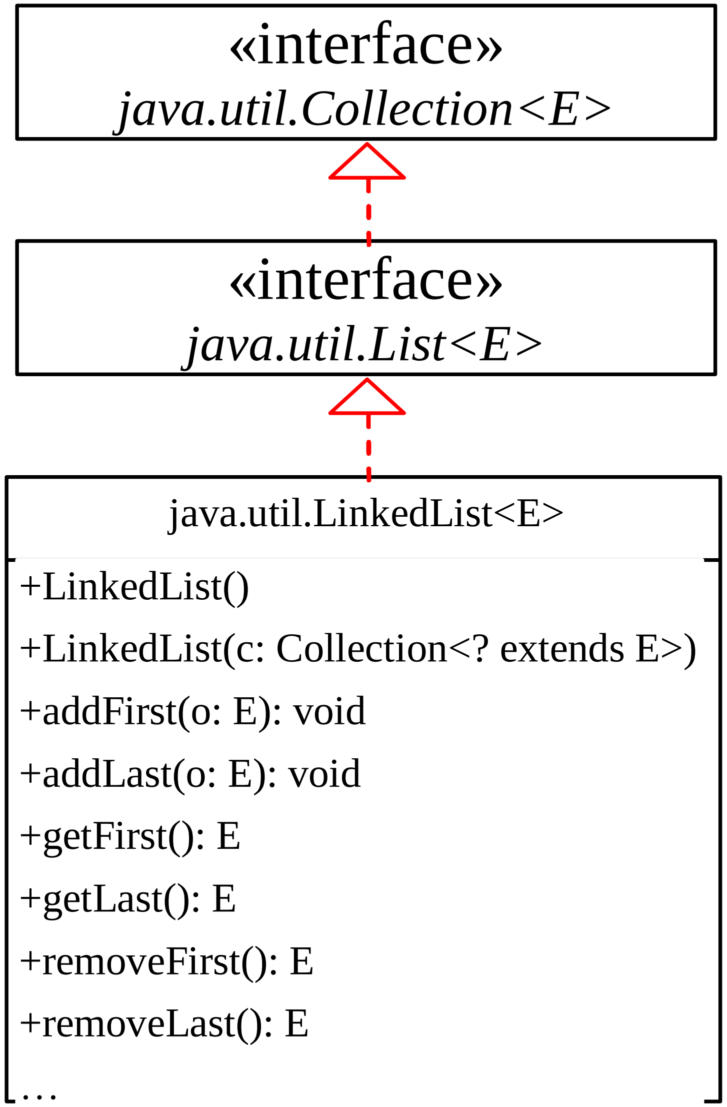
Array



LinkedList

- The other approach is to use a linked structure.
- A linked structure consists of nodes.
- Each node is dynamically created to hold an element.
- All the nodes are linked together to form a list.

java.util.LinkedList



Creates a default empty linked list.

Creates a linked list from an existing collection.

Adds the object to the head of this list.

Adds the object to the tail of this list.

Returns the first element from this list.

Returns the last element from this list.

Returns and removes the first element from this list.

Linked Lists

You can use a linked structure to implement a list to improve efficiency for adding and removing an elements from both ends of the list.

Note, ArrayList (by default) allows from the end of the list only.

They are similar to use. The real **difference** is their underlying implementation and their operation complexity.

How to use?

1. Import the package

```
java.util.LinkedList;
```

2. Instantiate the LinkedList object (to create list)

E.g.,

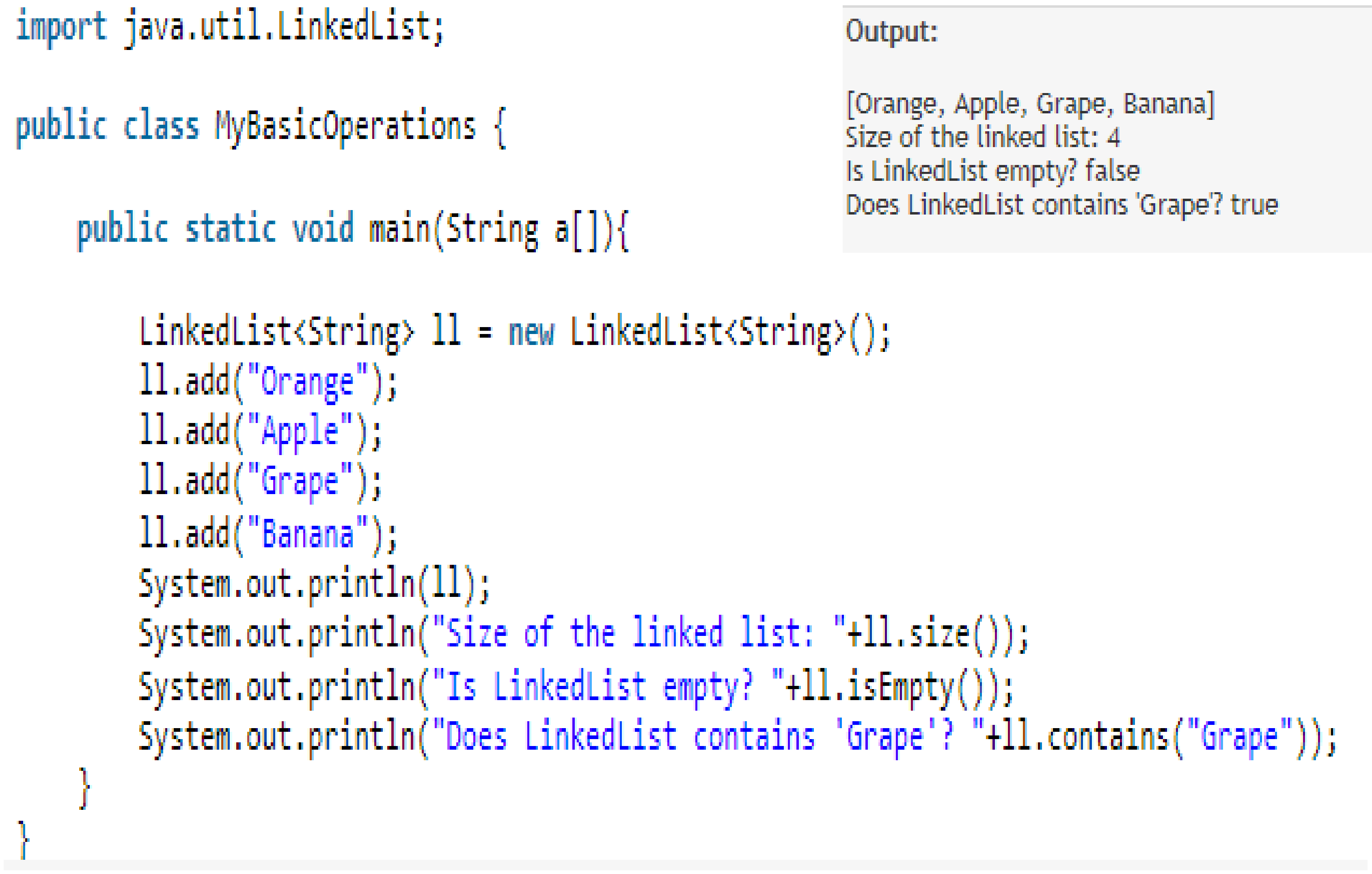
```
LinkedList<String> mylist2= new
LinkedList<String>();
```

3. Call methods on created object list

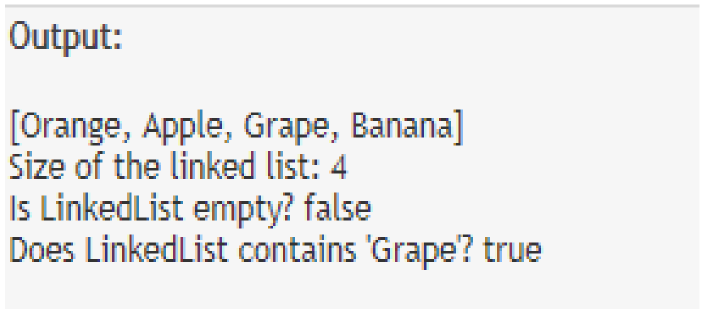
E.g.,

```
 mylist2.addFirst (“Blue”);
```

Example



```
import java.util.LinkedList;

public class MyBasicOperations {

    public static void main(String a[]){

        LinkedList<String> ll = new LinkedList<String>();
        ll.add("Orange");
        ll.add("Apple");
        ll.add("Grape");
        ll.add("Banana");
        System.out.println(ll);
        System.out.println("Size of the linked list: "+ll.size());
        System.out.println("Is LinkedList empty? "+ll.isEmpty());
        System.out.println("Does LinkedList contains 'Grape'? "+ll.contains("Grape"));
    }
}
```



Output:

```
[Orange, Apple, Grape, Banana]
Size of the linked list: 4
Is LinkedList empty? false
Does LinkedList contains 'Grape'? true
```

Sample application of LinkedList

- One of the applications of LinkList is in implementing “undo” functionality in software systems such as Photoshop or Word, where a linked list of state is kept.

Array, ArrayList, LinkedList

- Which of the classes you use depends on your specific needs.
- If you need to support random access through an index without inserting or removing elements from any place other than the end, ArrayList offers the most efficient collection.
- If, however, your application requires the insertion or deletion of elements from any place in the list, you should choose LinkedList. A list can grow or shrink dynamically.
- An array is fixed once it is created. If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

ArrayList vs. LinkedList

- As mentioned, the difference lies in their performance NOT usage.
- LinkedList is faster in add and remove, but slower in get.
- In brief, LinkedList should be preferred if:
  - there are no large number of random access of element
  - there are a large number of add/remove operations

One interesting feature in JCF

- ListIterator
  - How do you move forward and backward through a collection (e.g. List)?

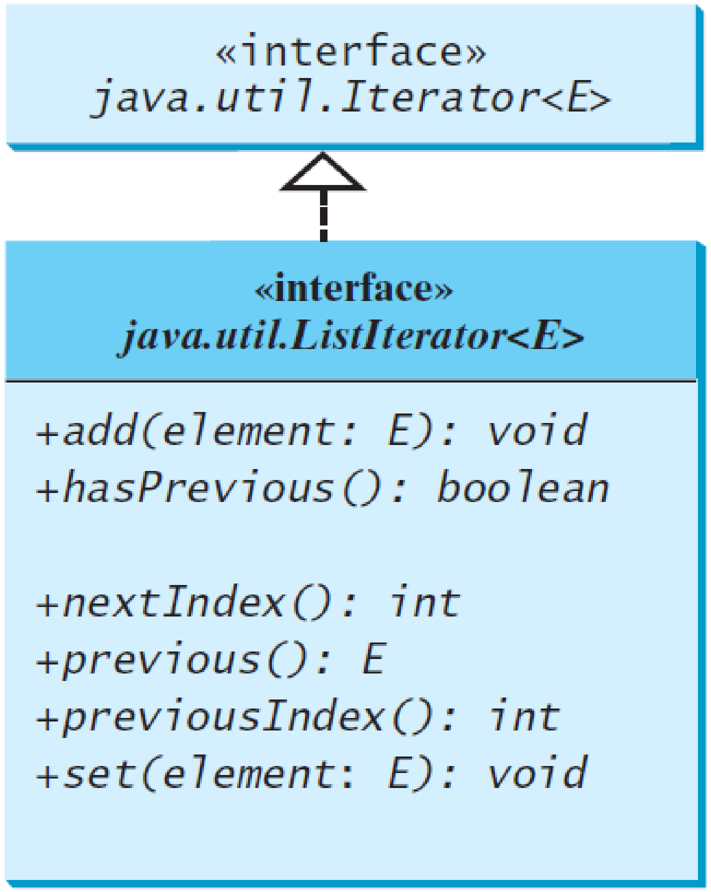
Iterator

An **iterator** for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.

```
                     Element(0)   Element(1)   Element(2)   ... Element(n-1)
cursor positions:  ^            ^            ^            ^                  ^
```

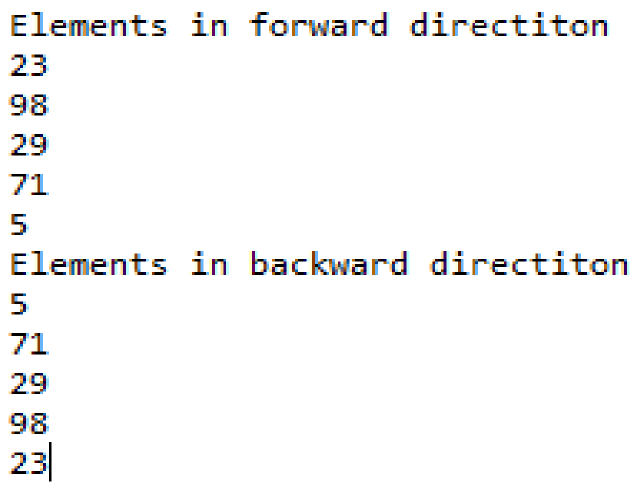
How would it be used to replace “loops”?

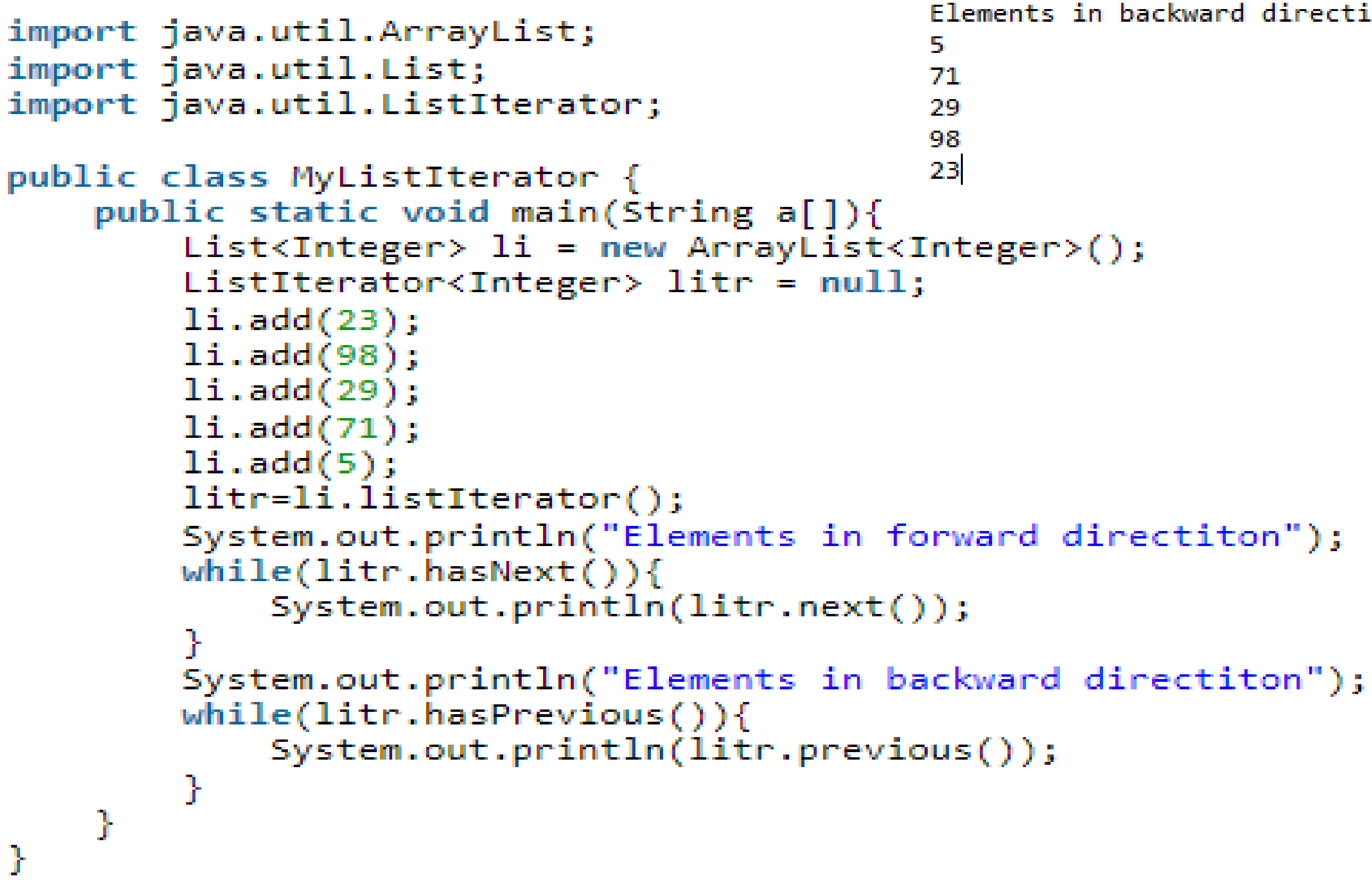
The List Iterator



+add(element: E): void	Adds the specified object to the list.
+hasPrevious(): boolean	Returns true if this list iterator has more elements when traversing backward.
+nextIndex(): int	Returns the index of the next element.
+previous(): E	Returns the previous element in this list iterator.
+previousIndex(): int	Returns the index of the previous element.
+set(element: E): void	Replaces the last element returned by the previous or next method with the specified element.

Example


```
Elements in forward directiton
23
98
29
71
5
Elements in backward directiton
5
71
29
98
23
```


```java
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class MyListIterator {
    public static void main(String a[]){
        List<Integer> li = new ArrayList<Integer>();
        ListIterator<Integer> litr = null;
        li.add(23);
        li.add(98);
        li.add(29);
        li.add(71);
        li.add(5);
        litr=li.listIterator();
        System.out.println("Elements in forward directiton");
        while(litr.hasNext()){
            System.out.println(litr.next());
        }
        System.out.println("Elements in backward directiton");
        while(litr.hasPrevious()){
            System.out.println(litr.previous());
        }
    }
}
```

Exercise

Write a program that lets the user enter numbers from a console and displays them in a Joptionpane message box. Use a linked list to store the numbers. **Do not** store duplicate numbers.

Collection (Part 2)
Maps

Review of JCF hierarchy (1)

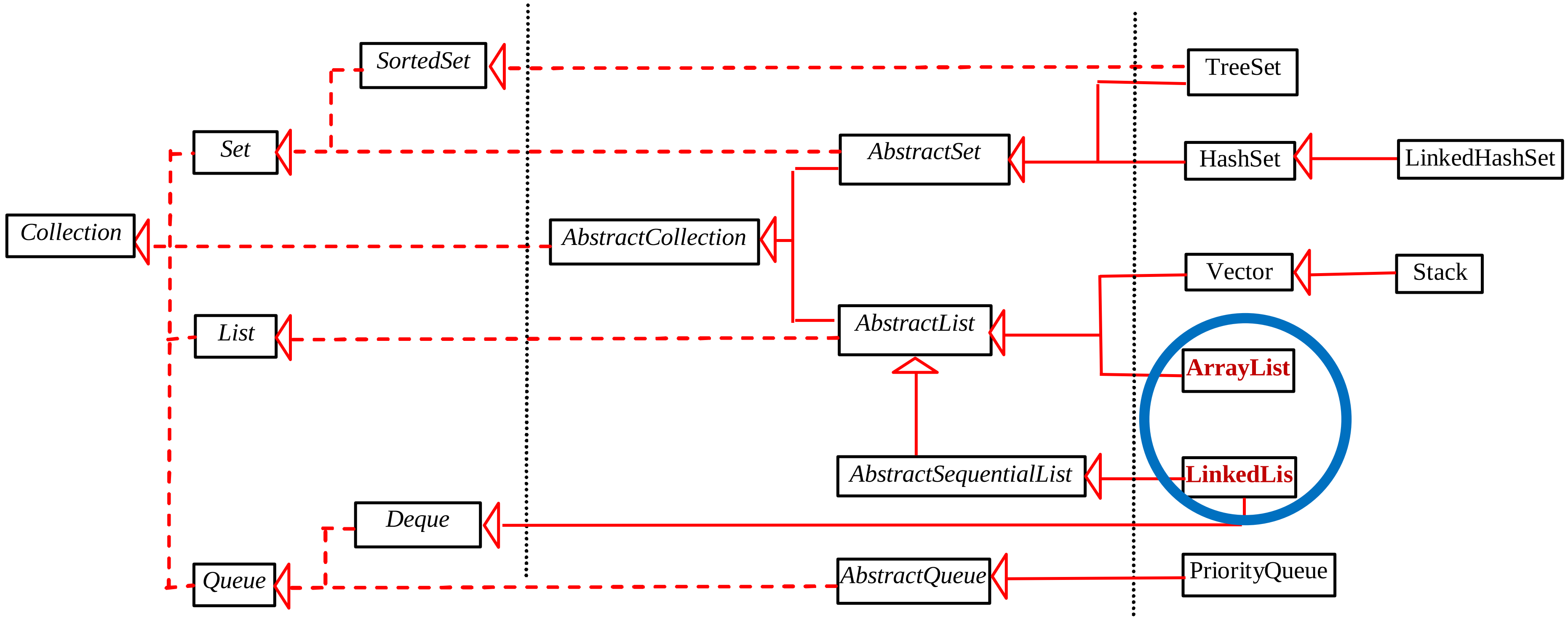
- ☞ The Java Collections Framework supports two types of containers:

(1) One for storing a collection of elements is simply called a *collection*.

(2) The other, for storing key/value pairs, is called a *map*.

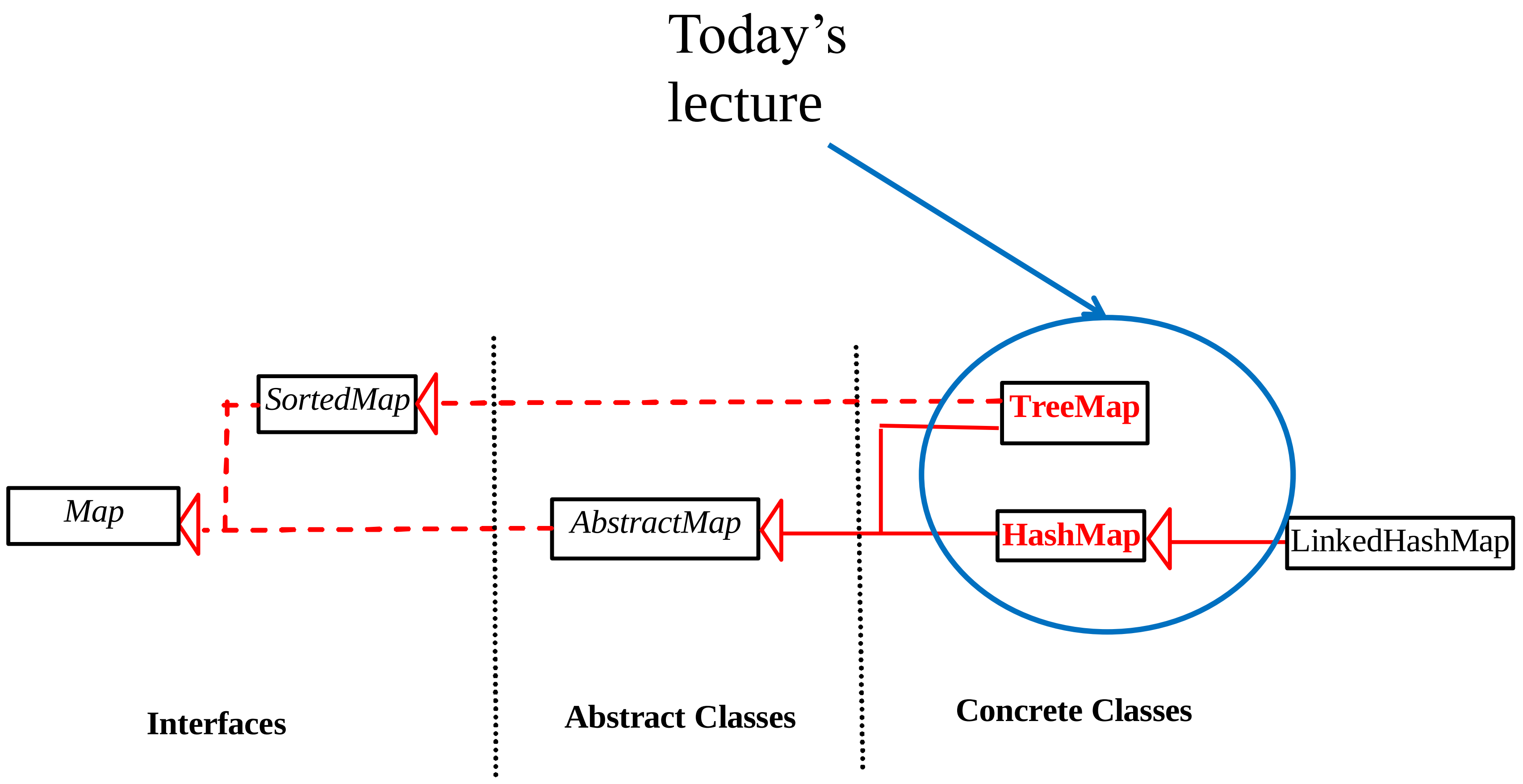
Review of JCF hierarchy (2)

Set and List are subinterfaces of Collection.



Interfaces **Abstract Classes** **Concrete Classes**

Review of JCF hierarchy (2)



Popular implementation of JCF

(commonly used classes)

- List - ArrayList, LinkedList and Vector
- Set - HashSet, TreeSet and LinkedHashSet
- Map – HashMap and TreeMap

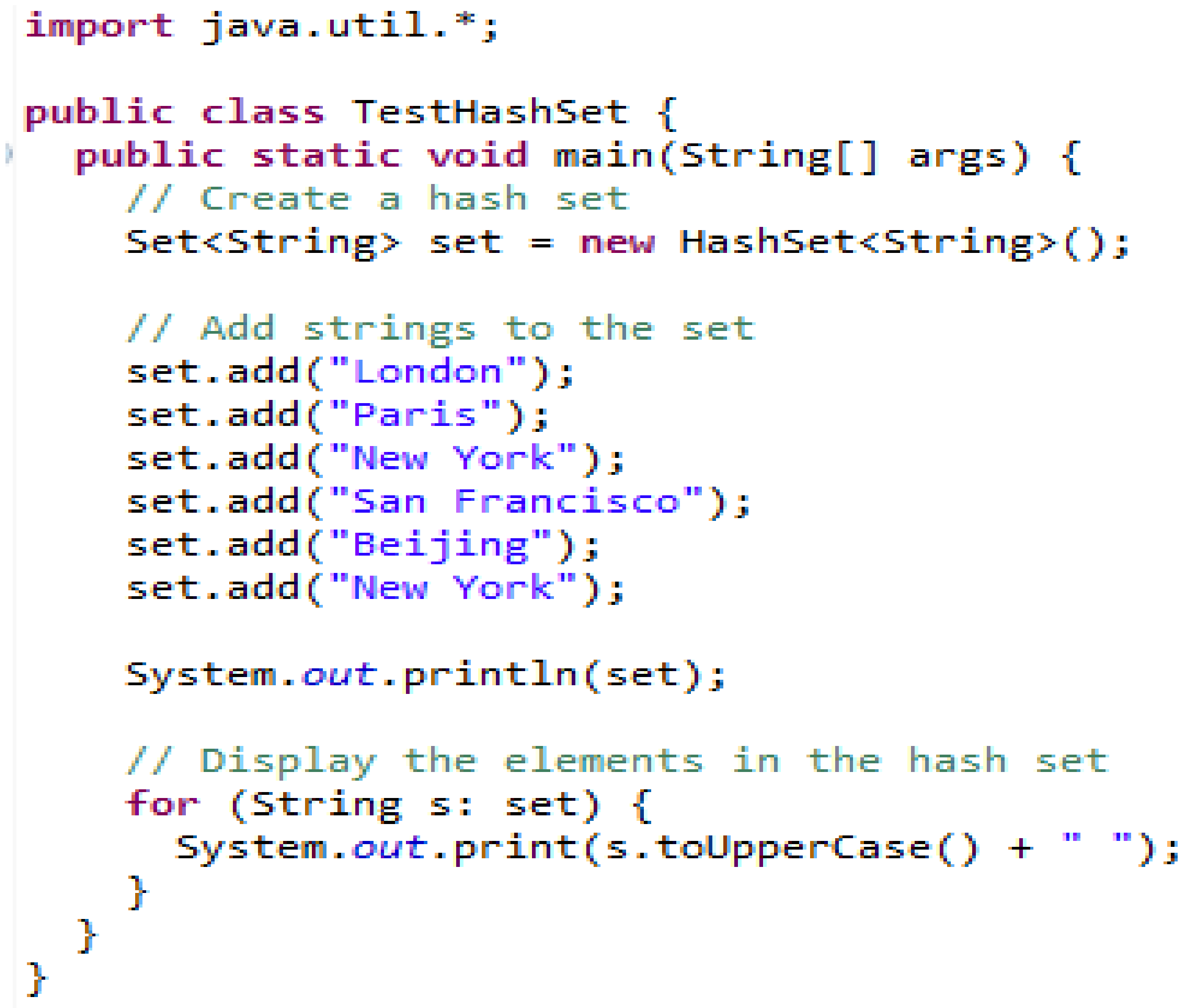
No-fly list- Motivation example

The “**No-Fly**” list is a list, created and maintained by the U.S. government’s Terrorist Screening Center, of people who are not permitted to board a commercial aircraft for travel in or out of the United States. **Suppose we need to write a program that checks whether a person is on the No-Fly list.**

Q:What is the efficient data structure to use?

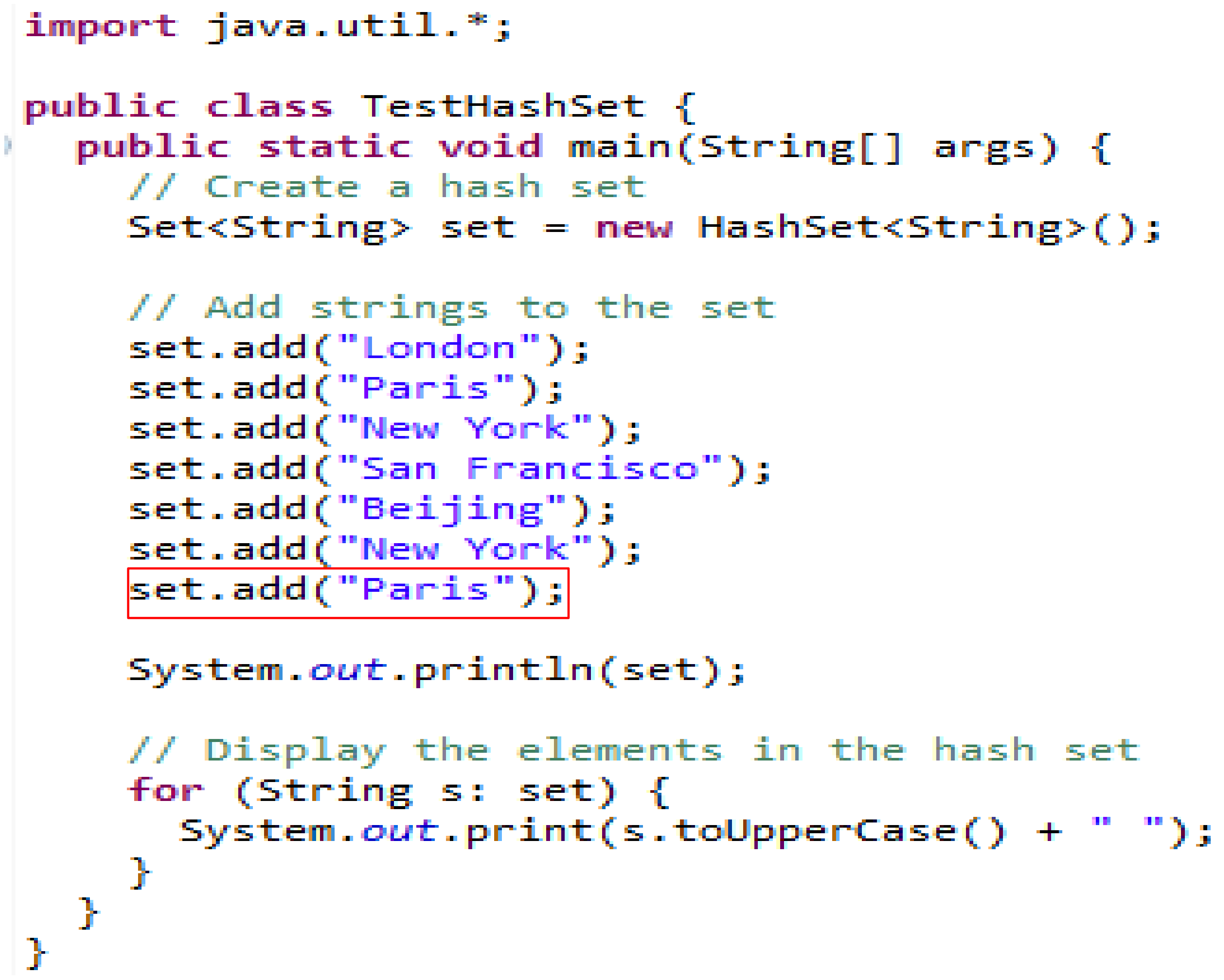
A: You can use a list to store names in the No-Fly list. However, a more efficient data structure for this application is a ***set***.

Set Example: what is the output?



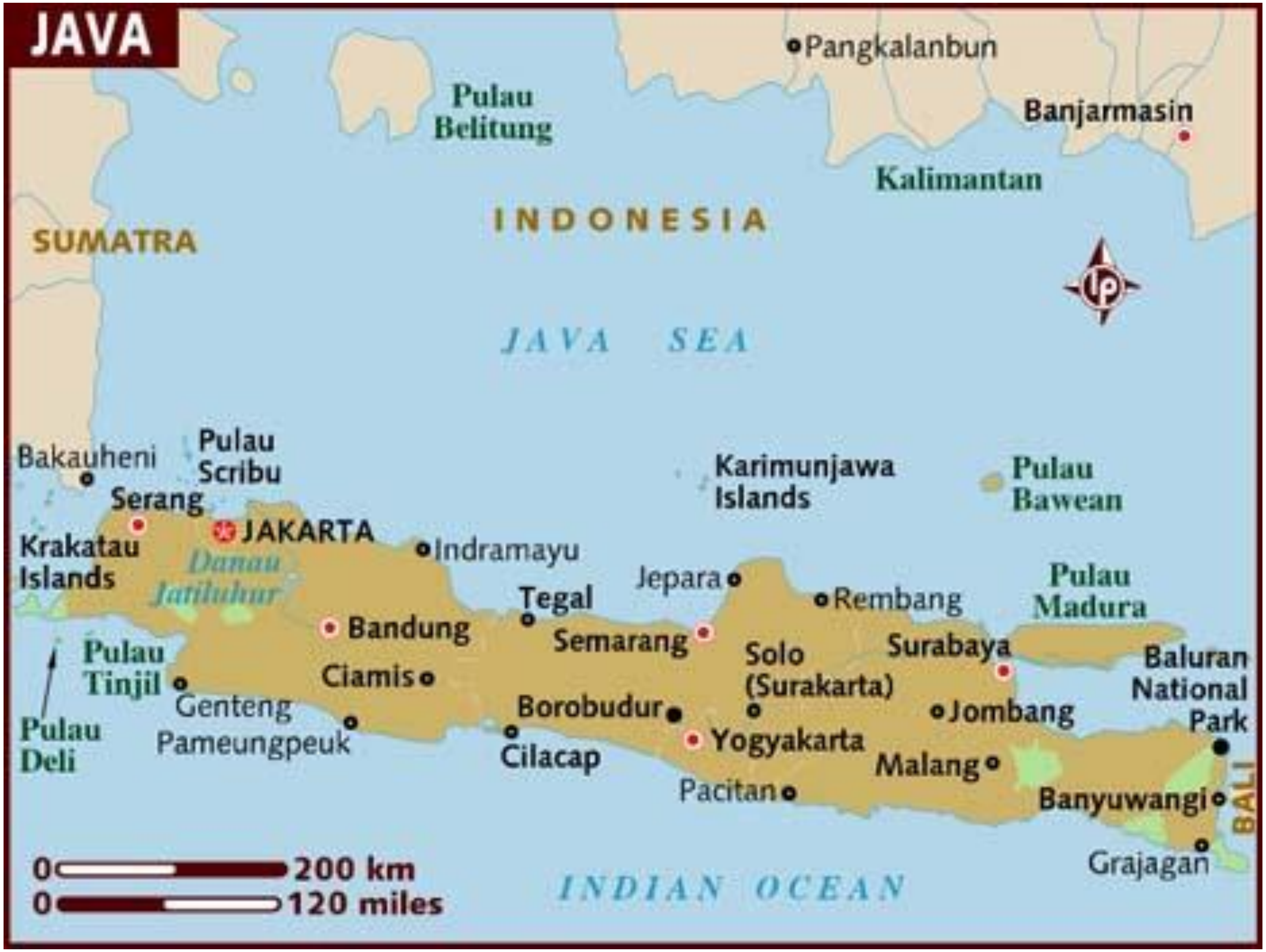
```
import java.util.*;

public class TestHashSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<String>();

    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");

    System.out.println(set);

    // Display the elements in the hash set
    for (String s: set) {
      System.out.print(s.toUpperCase() + " ");
    }
  }
}
```

Set Example: Does it compile or run?



```java
import java.util.*;

public class TestHashSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<String>();

    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");
    set.add("Paris");

    System.out.println(set);

    // Display the elements in the hash set
    for (String s: set) {
      System.out.print(s.toUpperCase() + " ");
    }
  }
}
```

Map


JAVA
Pangkalanbun
Pulau Belitung
Banjarmasin
Kalimantan
SUMATRA
INDONESIA
JAVA SEA
Bakauheni
Pulau Scribu
Serang
Karimunjawa Islands
Pulau Bawean
JAKARTA
Krakatau Islands
Indramayu
Danau Jatiluhur
Jepara
Tegal
Rembang
Pulau Madura
Bandung
Semarang
Solo (Surakarta)
Surabaya
Pulau Tinjil
Ciamis
Baluran National Park
Genteng
Borobudur
Jombang
Pulau Deli
Pameungpeuk
Cilacap
Yogyakarta
Malang
Pacitan
Banyuwangi
BALI
Grajagan
0 200 km
0 120 miles
INDIAN OCEAN

No-fly list again!

- ☞ Suppose your program also needs to store detailed information about terrorists in the No-Fly list. The detailed information such as gender, height, weight, and nationality can be retrieved using the *name* as the *key*.
- ☞ Q: *What is the efficient data structure to use?*
- ☞ A: *map* is an efficient data structure for such a task.

Motivation

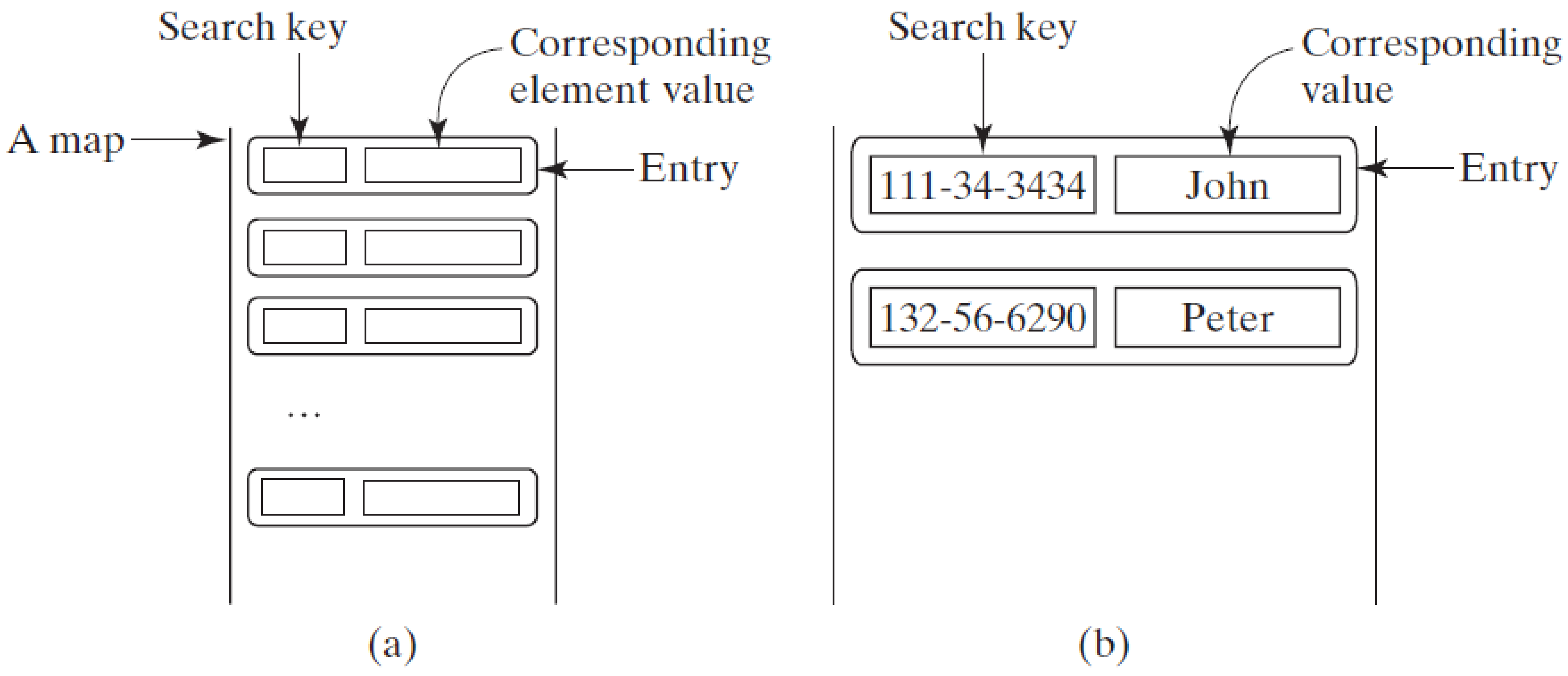
- Can we have object as index for a list of elements?
  - Yes!
- Data structures such as Array, ArrayList, and LinkedList indexes are integer numbers staring at 0
- What data structure can help us to do so?
  - Map

What is a map?

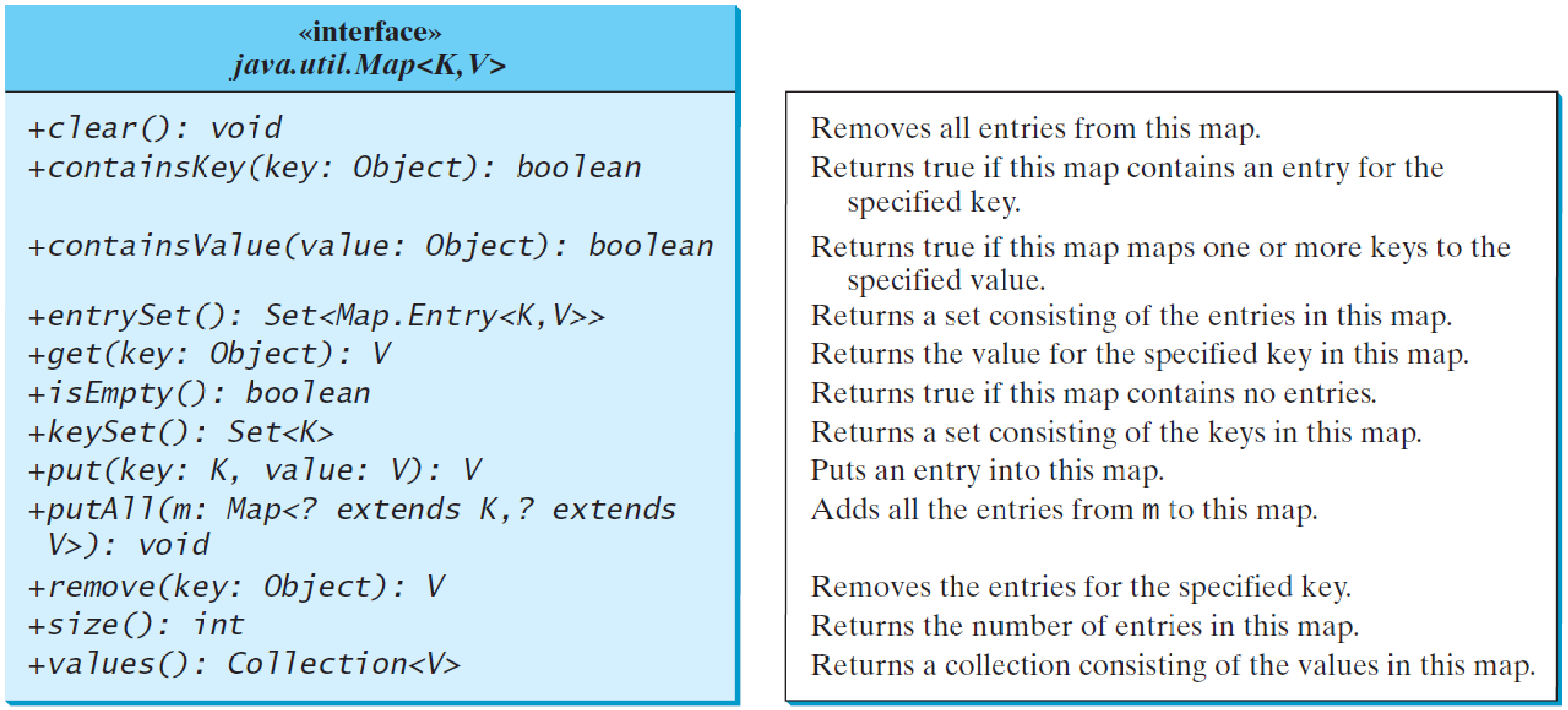
- A Map is an object that maps *keys* to *values*.
- A map cannot contain duplicate keys: Each key can map to at most one value. But, it can have duplicate values.

The Map Interface

The Map interface maps keys to the elements. The keys are like indexes. **In List**, the indexes are integer. **In Map**, the keys can be any objects.



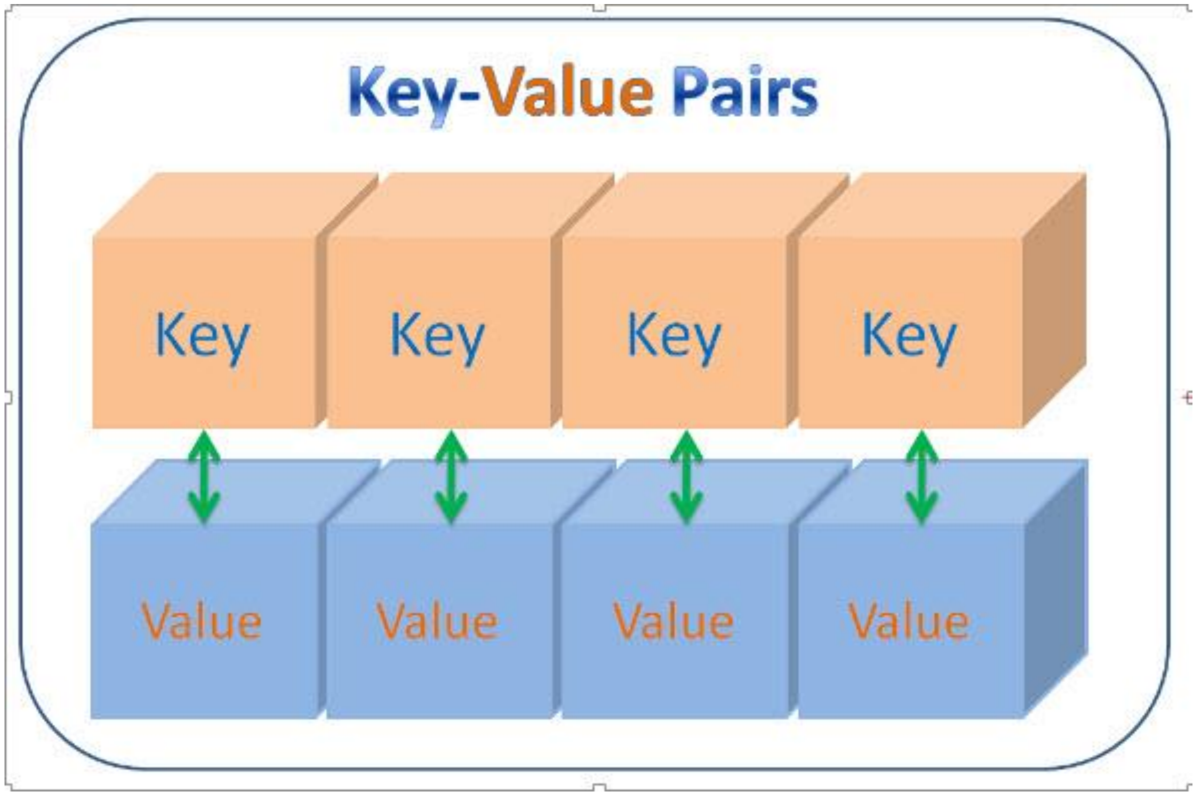
The Map Interface UML Diagram



«interface»
java.util.Map<K,V>

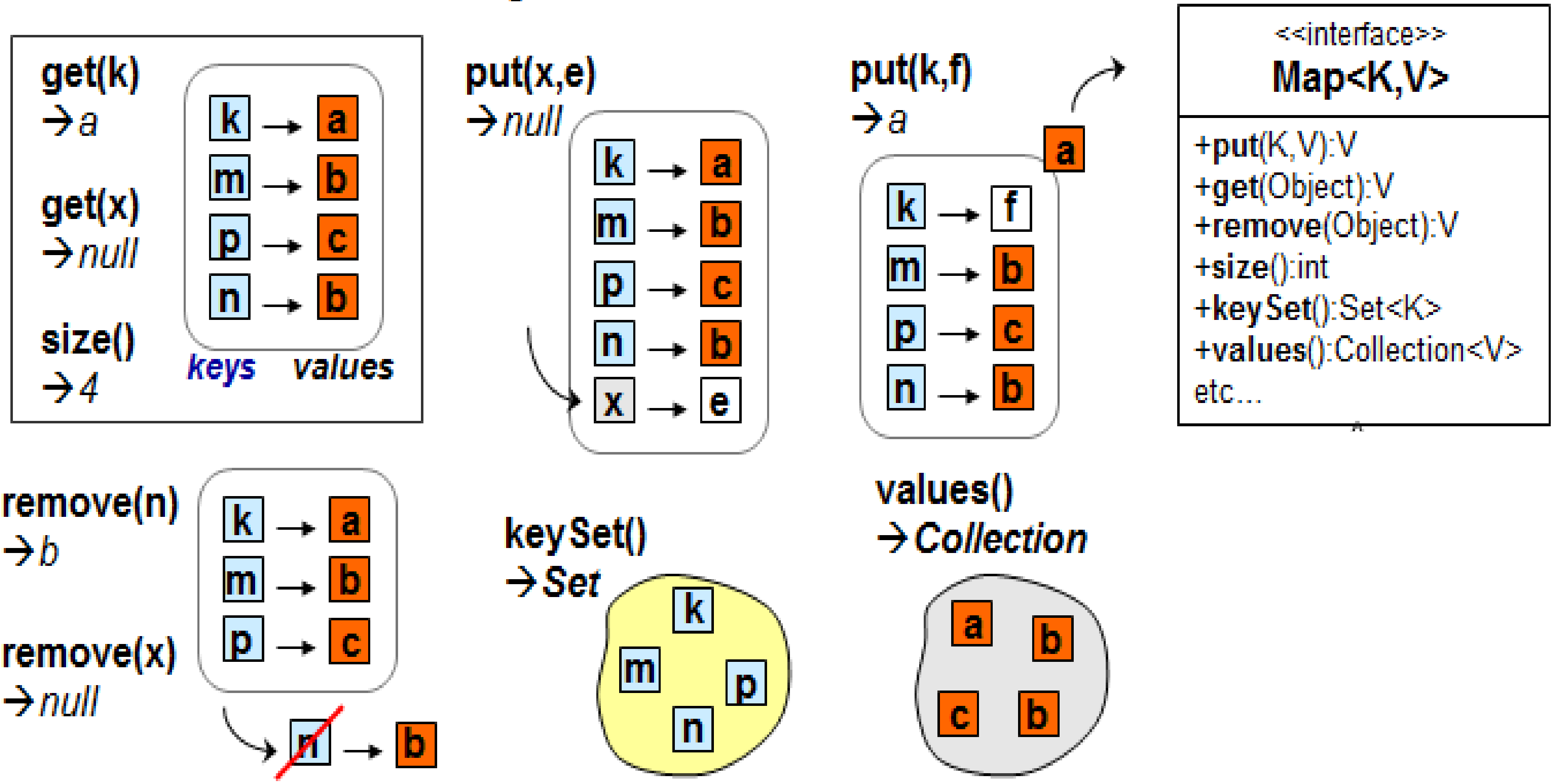
`+clear(): void`	Removes all entries from this map.
`+containsKey(key: Object): boolean`	Returns true if this map contains an entry for the specified key.
`+containsValue(value: Object): boolean`	Returns true if this map maps one or more keys to the specified value.
`+entrySet(): Set<Map.Entry<K,V>>`	Returns a set consisting of the entries in this map.
`+get(key: Object): V`	Returns the value for the specified key in this map.
`+isEmpty(): boolean`	Returns true if this map contains no entries.
`+keySet(): Set<K>`	Returns a set consisting of the keys in this map.
`+put(key: K, value: V): V`	Puts an entry into this map.
`+putAll(m: Map<? extends K,? extends V>): void`	Adds all the entries from `m` to this map.
`+remove(key: Object): V`	Removes the entries for the specified key.
`+size(): int`	Returns the number of entries in this map.
`+values(): Collection<V>`	Returns a collection consisting of the values in this map.

Map: Basic operations



Map<K,V>

- Stores mappings from (unique) keys (type `K`) to values (type `V`)
  - See, you can have more than one type parameters!
- Think of them as "arrays" but with objects (keys) as indexes
  - Or as "directories": e.g. `"Bob"` → `021999887`



More about `put`

- ☞ If the map already contains a given key, `put(key, value)` replaces the value associated with that key, the old value replaced by the new value associated with the key is returned
- ☞ This means Java has to do equality testing on keys

Map: Update methods

- void putAll(Map t);
  - Copies one Map into another
  - Example: newMap.putAll(oldMap);
- void clear();
  - Example: oldMap.clear();
- remove(Object key) method removes the entry for the specified key from the map.

Map: Query methods

- ☞ containsKey, containsValue, isEmpty, and size

Map: Collection views

- public Set keySet( );
  - returns a set of keys in the map

    myMap.KeySet()
- public Collection values( );
  - returns the values in the map

    myMap.values()
- public Set entrySet( );
  - returns a set of (key-value) pairs

    myMap.entrySet()
  -

How to iterate over a Map?

- The previous views provide the *only* way to iterate over a Map
- In Map and its all classes we don’t have iterator() method to traverse, but we can use three views to do so [see example in later slides]
- You can create iterators for the key set, the value set, or the entry set (the set of entries, that is, key-value pairs)

Popular implementations of Map <K, V> interface

- `Map` is an interface; you can't say `new Map();`

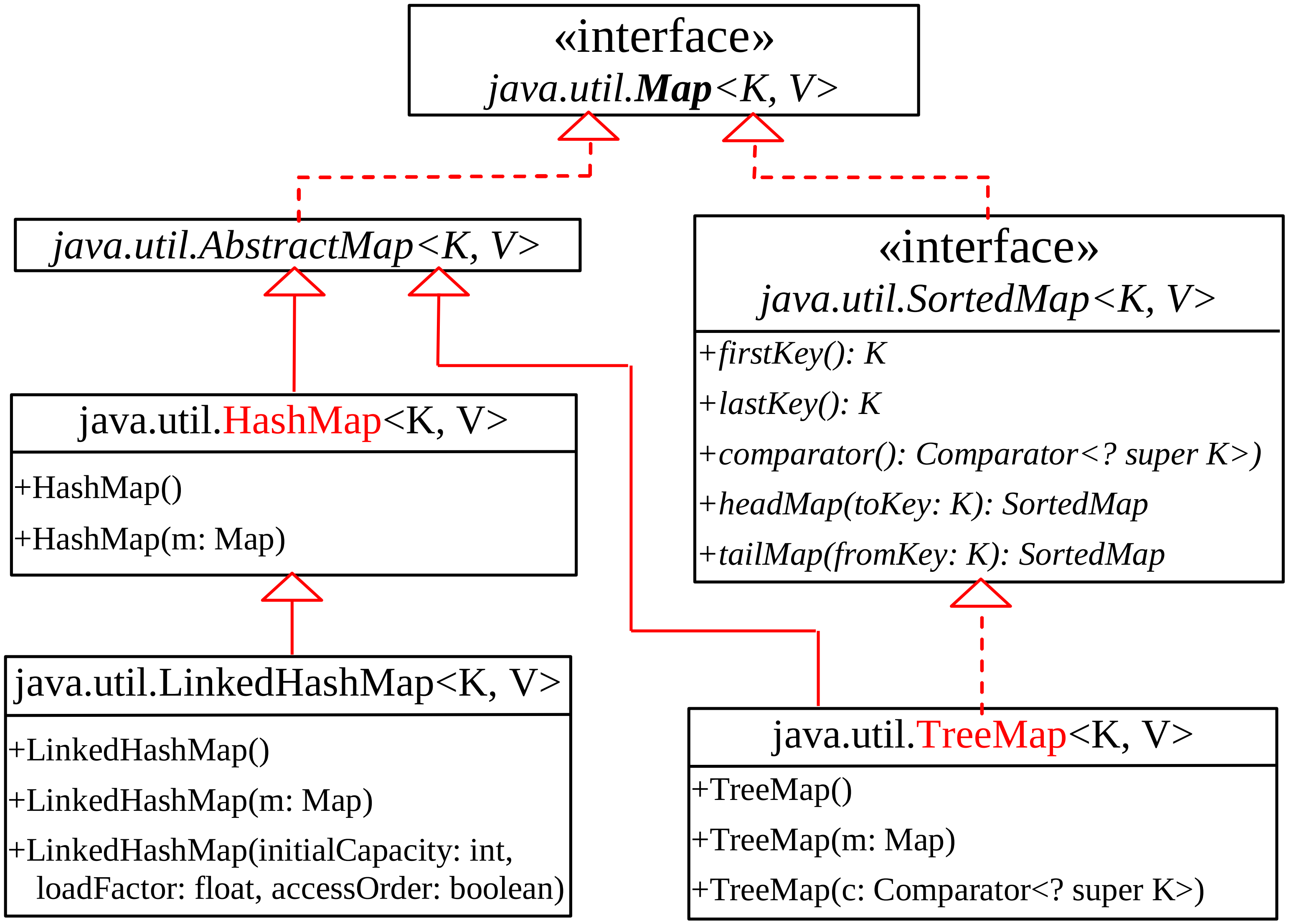
- The Java platform contains three general-purpose Map implementations:

HashMap class

TreeMap class

LinkedHashMap class

Concrete Map related Classes


«interface»
*java.util.**Map**<K, V>*
java.util.AbstractMap<K, V>
«interface»
java.util.SortedMap<K, V>
+firstKey(): K
+lastKey(): K
+comparator(): Comparator<? super K>)
+headMap(toKey: K): SortedMap
+tailMap(fromKey: K): SortedMap
java.util.HashMap<K, V>
+HashMap()
+HashMap(m: Map)
java.util.LinkedHashMap<K, V>
+LinkedHashMap()
+LinkedHashMap(m: Map)
+LinkedHashMap(initialCapacity: int, loadFactor: float, accessOrder: boolean)
java.util.TreeMap<K, V>
+TreeMap()
+TreeMap(m: Map)
+TreeMap(c: Comparator<? super K>)

HashMap<K,V>

- aka Hashtable
- keys are hashed using **`Object.hashCode()`**
  - i.e. no guaranteed ordering of keys
- **`keySet()`** returns a **`HashSet`**
- **`values()`** returns an unknown Collection

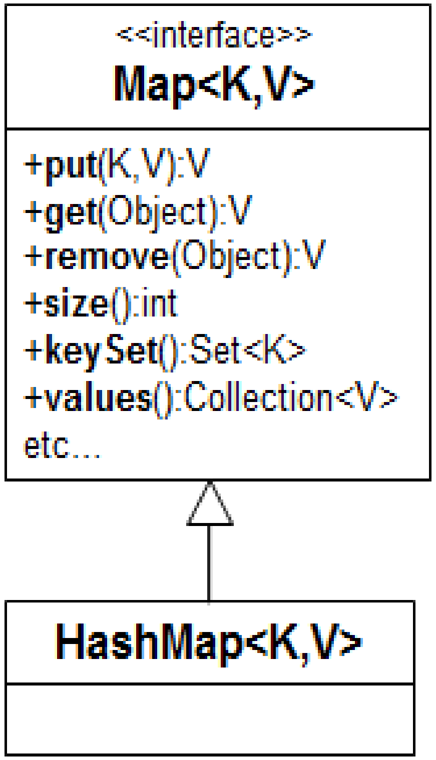
```
Map<String, Integer> directory
                        = new HashMap<String, Integer>();
directory.put("Mum", new Integer(9998888));
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);
System.out.println(directory.size());
for (String key : directory.keySet()) {
  System.out.print(key+"'s number: ");
  System.out.println(directory.get(key));
}
System.out.println(directory.values());
```

"autoboxing"

4 or 5?

`Set<String>`

What's Bob's number?



Example (how to use)

```java
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        //Create HashMap
        Map<String , String> studentGrades = new HashMap<String, String>();

        //Add Key/Value pairs
        studentGrades.put("Alvin", "A+");
        studentGrades.put("Alan", "A");
        studentGrades.put("Becca", "A-");
        studentGrades.put("Sheila", "B+");

        //find element using key
        System.out.println("Becca's Marks:: "+studentGrades.get("Becca"));

        //remove element
        studentGrades.remove("Becca");

        //Iterate over HashMap
        for(String key: studentGrades.keySet()){
            System.out.println(key  +" :: "+ studentGrades.get(key));
        }

    }
}
```

Map (HashMap) is an unordered collection

What do you notice?

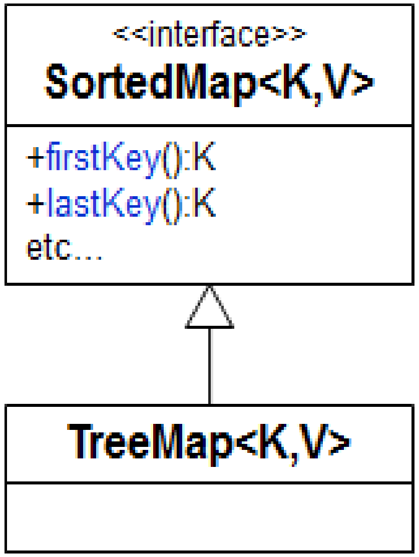
```
Becca's Marks:: A-
Alan :: A
Sheila :: B+
Alvin :: A+
```

TreeMap

- The TreeMap class is efficient for traversing the keys in a sorted order
- (because it implements SortedMap interface, refer back to JCF hierarchy)
- the elements are automatically sorted by natural order of Keys and NOT by values.
- TreeMap API

TreeMap<K,V>

- Guaranteed ordering of keys (like TreeSet)
  - In fact, TreeSet is implemented using TreeMap ☺
  - Hence `keySet()` returns a `TreeSet`
- **`values()`** returns an unknown Collection – ordering depends on ordering of **keys**



Empty constructor
→ natural ordering

```
Map<String, Integer> directory
                       = new TreeMap<String, Integer>();
directory.put("Mum", new Integer(9998888));
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);
System.out.println(directory.size());
for (String key : directory.keySet()) {
  System.out.print(key+"'s #: ");
  System.out.println(directory.get(key));
}
System.out.println(directory.values());
```

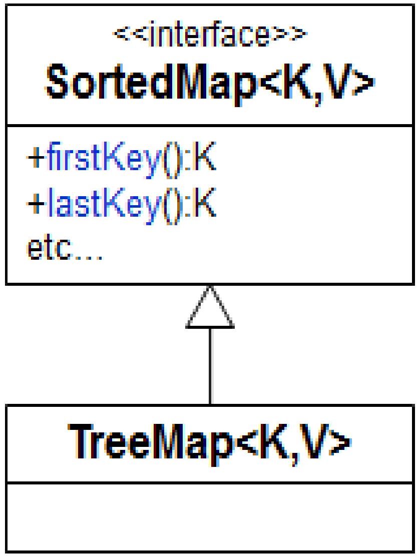
4

Loop output?

?

TreeMap<K,V>

- Guaranteed ordering of keys (like TreeSet)
  - In fact, TreeSet is implemented using TreeMap ☺
  - Hence `keySet()` returns a `TreeSet`
- **`values()`** returns an unknown Collection – ordering depends on ordering of **keys**



Empty constructor
→ natural ordering

```java
Map<String, Integer> directory
                    = new TreeMap<String, Integer>();
directory.put("Mum", new Integer(9998888));
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);
System.out.println(directory.size());
for (String key : directory.keySet()) {
  System.out.print(key+"'s #: ");
  System.out.println(directory.get(key));
}
System.out.println(directory.values());
```

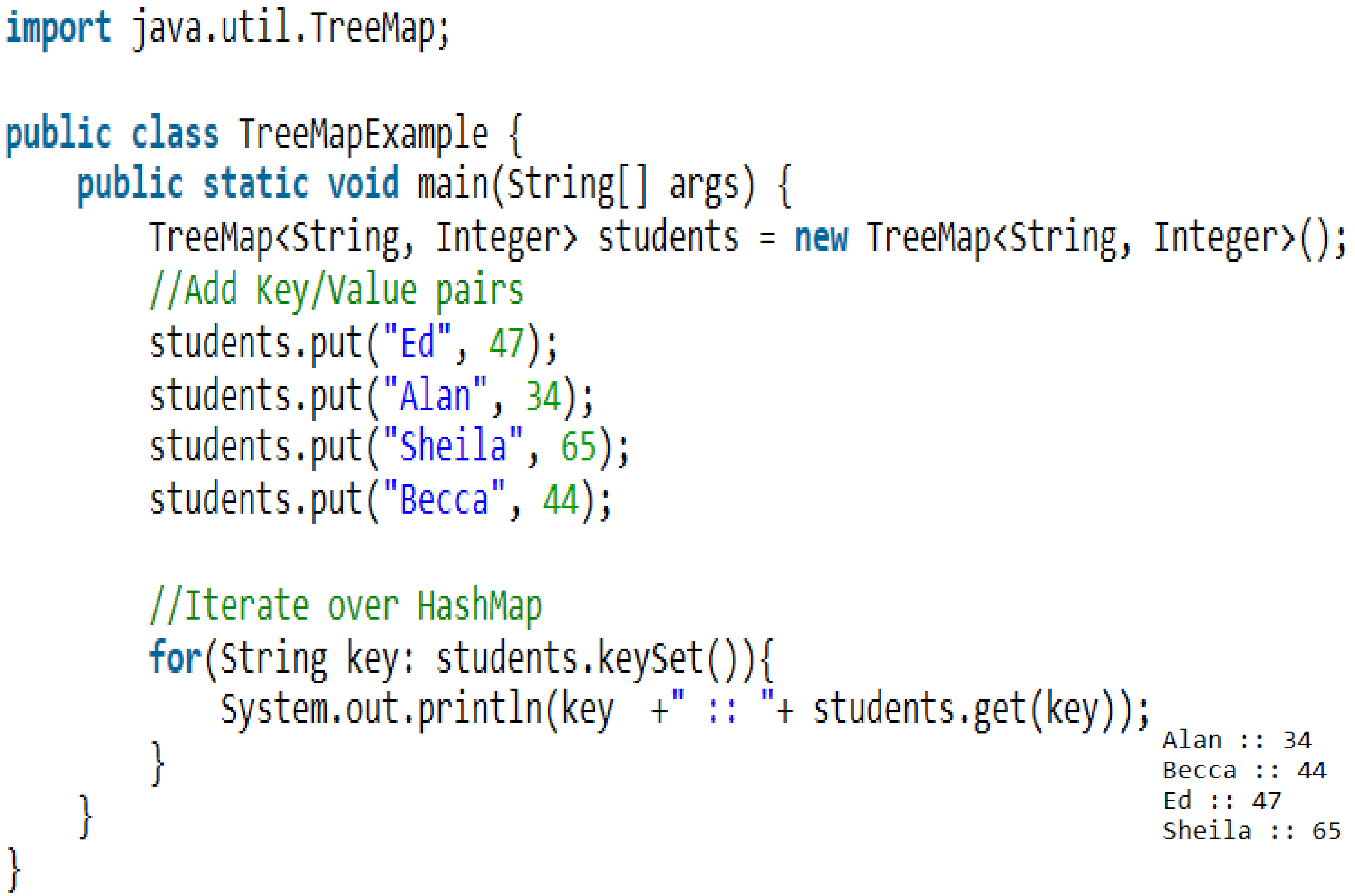
4

Loop output?

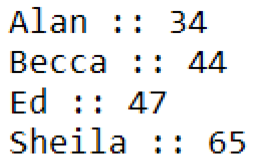
```
Bob's #: 1000000
Dad's #: 9998888
Edward's #: 5553535
Mum's #: 9998888
```

?

Example (how to use)



```java
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<String, Integer> students = new TreeMap<String, Integer>();
        //Add Key/Value pairs
        students.put("Ed", 47);
        students.put("Alan", 34);
        students.put("Sheila", 65);
        students.put("Becca", 44);

        //Iterate over HashMap
        for(String key: students.keySet()){
            System.out.println(key  +" :: "+ students.get(key));
        }
    }
}
```



```
Alan :: 34
Becca :: 44
Ed :: 47
Sheila :: 65
```

HashMap vs. TreeMap

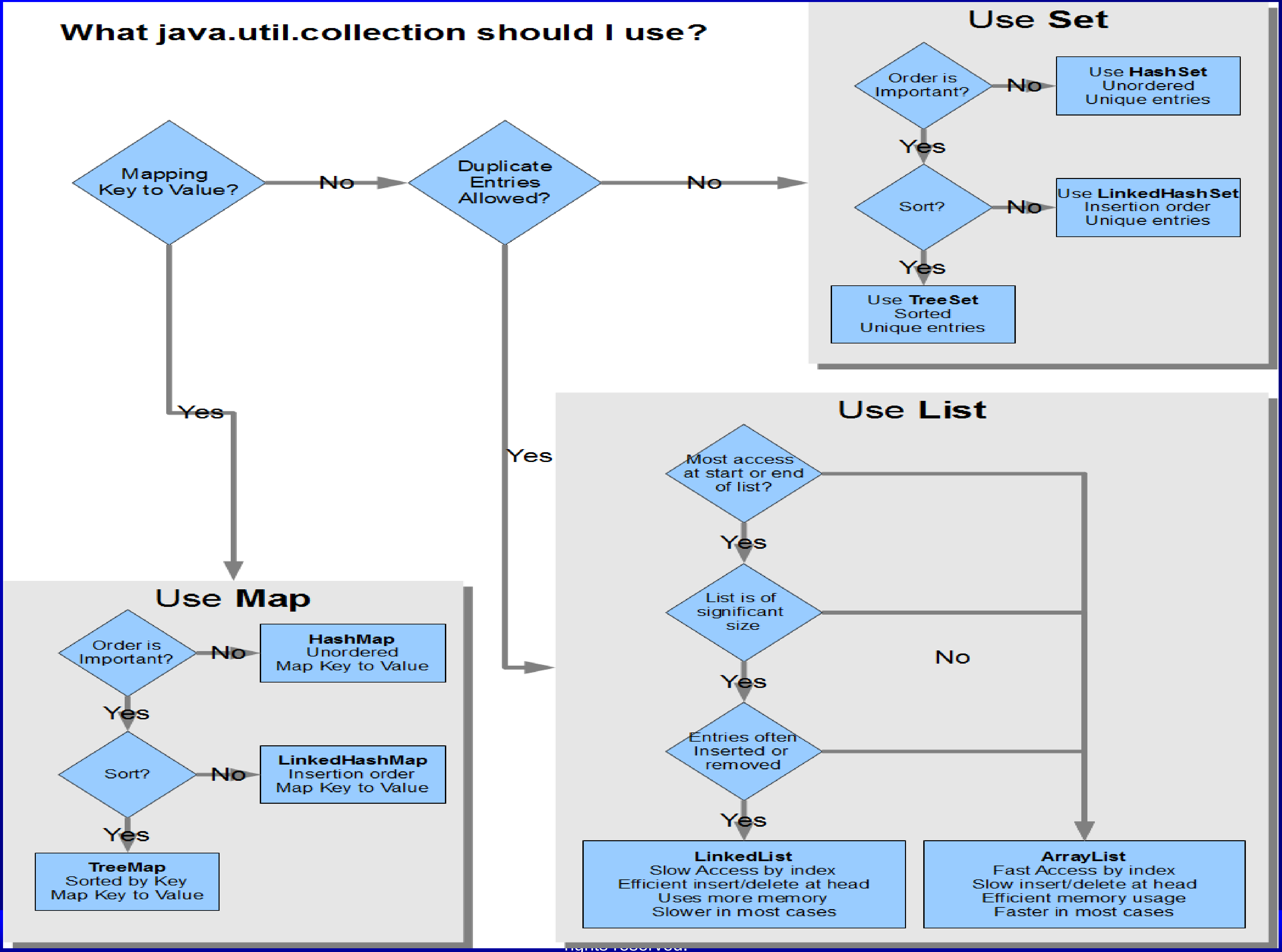
- HashMap is the faster
- TreeMap guarantees the order of iteration

- so use HashMap whenever you don't care about the order of the keys.

Application of Maps

Examples: dictionary, phone book, etc.

What java.util.collection should I use?



Mapping Key to Value?
No
Duplicate Entries Allowed?
No
Yes
Yes
Use Set
Order is Important?
No
Use **HashSet** Unordered Unique entries
Yes
Sort?
No
Use **LinkedHashSet** Insertion order Unique entries
Yes
Use **TreeSet** Sorted Unique entries
Use List
Most access at start or end of list?
Yes
List is of significant size
Yes
Entries often Inserted or removed
Yes
No
LinkedList Slow Access by index Efficient insert/delete at head Uses more memory Slower in most cases
ArrayList Fast Access by index Slow insert/delete at head Efficient memory usage Faster in most cases
Use Map
Order is Important?
No
HashMap Unordered Map Key to Value
Yes
Sort?
No
LinkedHashMap Insertion order Map Key to Value
Yes
TreeMap Sorted by Key Map Key to Value

Last word

- Now you know the main interfaces and implementations from the Collections framework, look them up on the Java API documentation to find out in more detail on how to use them

Quiz: Counting the Occurrences of Words in a Text

Write a program to count the occurrences of words in a text and displays the words and their occurrences.

The program uses a **hash map** to store a pair consisting of a word and its count. For each word, check whether it is already a key in the map. If not, add the key and value 1 to the map. Otherwise, increase the value for the word (key) by 1 in the map.