



# Application Program Development

Segment : Collection Framework

Mahboob Ali



# Outcomes

- Understanding of Collections
    - Lists
    - Sets
    - Maps
- 

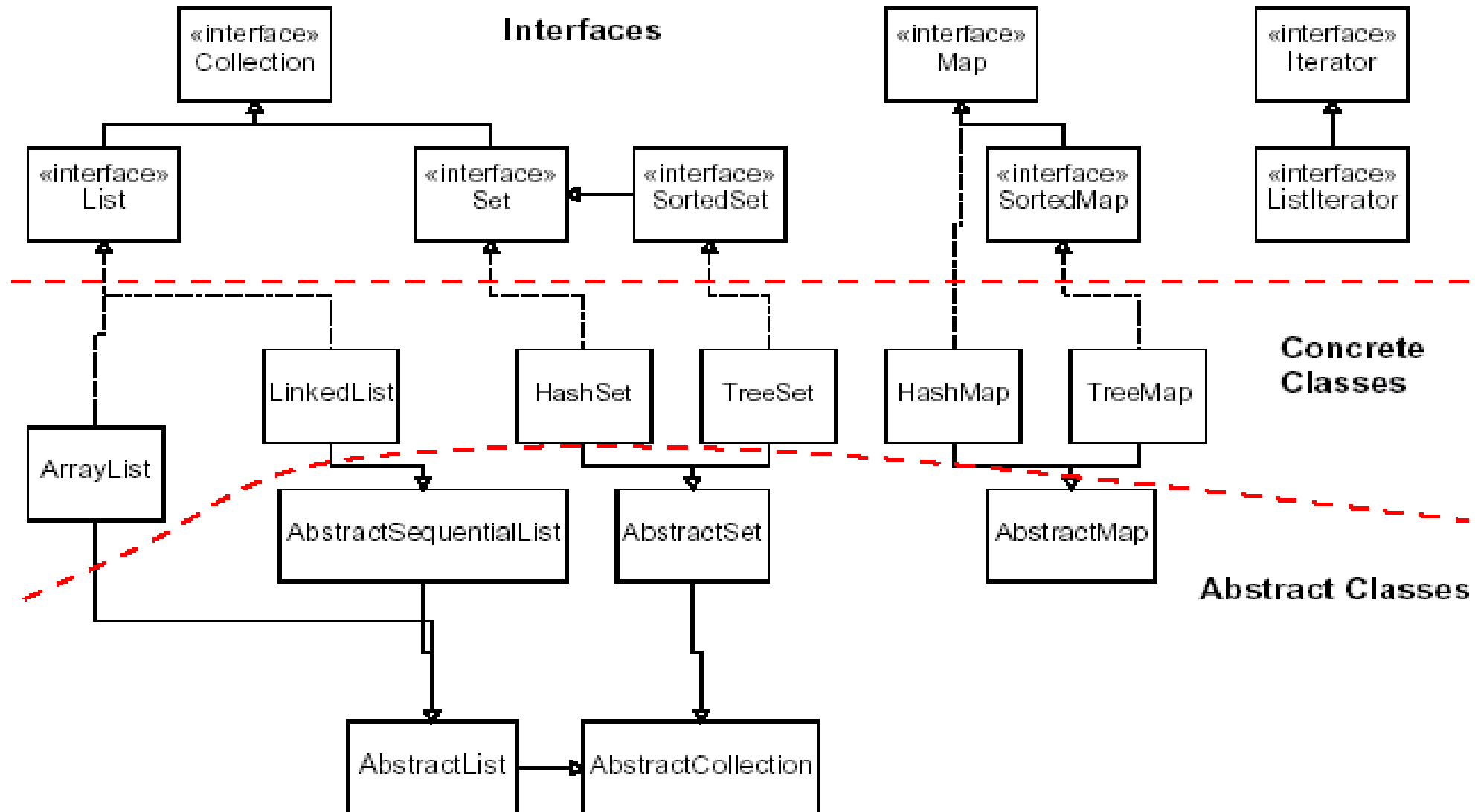
# What is a Data Structure?

- A data structure is a collection of data organized in some fashion.
- The structure not only stores data, but also supports operations for accessing and manipulating the data.

# The Collection

- A *collection* is a container object that represents a group of objects, often referred to as *elements*.
- Collections are used to store, retrieve, manipulate and communicate with the aggregated data.
- *Examples*
  - A poker hand (collection of cards)
  - A mail folder (collection of letters)
  - A telephone directory (mapping of names to phone numbers)

# Java collections framework



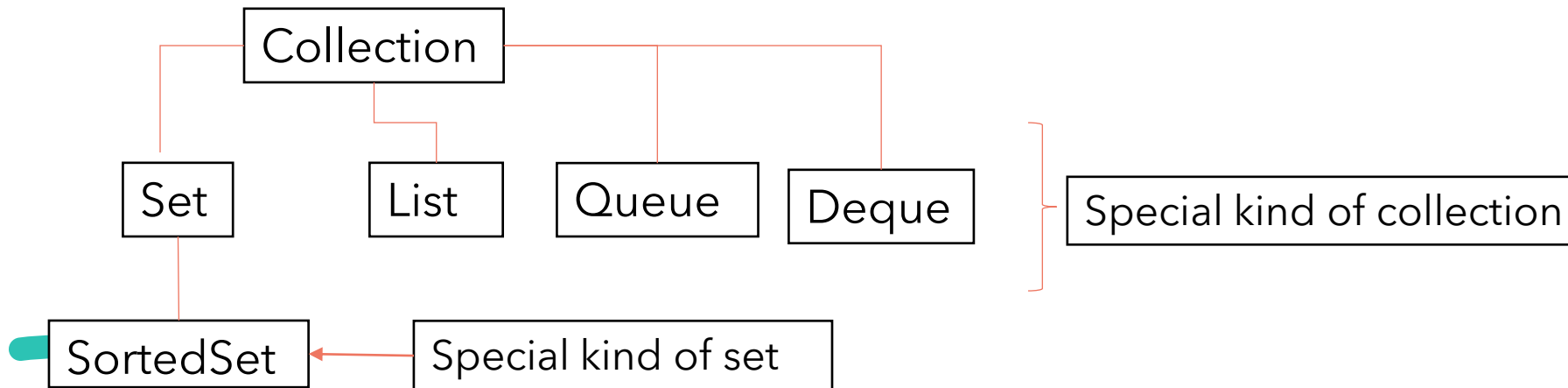
# Interfaces

- Interfaces provides the abstract data type to represent collection.

*java.util.Collection*

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

- *java.util* contains all the collections framework interfaces.



# Syntax

```
public interface Collection<E> extends Iterable<E>{  
  
    // any collection object to be used in foreach looks  
}
```

- <E> tells you that the interface is generic.
- You must specify the type of object when you instantiate the collection.
- **Iterable<T>** is from java.lang package with one method **iterator()**.

# Collection Interface

«interface»  
*java.util.Collection<E>*

+*add(o: E): boolean*  
+*addAll(c: Collection<? extends E>): boolean*  
+*clear(): void*  
+*contains(o: Object): boolean*  
+*containsAll(c: Collection<?>): boolean*  
+*equals(o: Object): boolean*  
+*hashCode(): int*  
+*isEmpty(): boolean*  
+*iterator(): Iterator*  
+*remove(o: Object): boolean*  
+*removeAll(c: Collection<?>): boolean*  
+*retainAll(c: Collection<?>): boolean*  
+*size(): int*  
+*toArray(): Object[]*

Adds a new element o to this collection.

Adds all the elements in the collection c to this collection.

Removes all the elements from this collection.

Returns true if this collection contains the element o.

Returns true if this collection contains all the elements in c.

Returns true if this collection is equal to another collection o.

Returns the hash code for this collection.

Returns true if this collection contains no elements.

Returns an iterator for the elements in this collection.

Removes the element o from this collection.

Removes all the elements in c from this collection.

Retains the elements that are both in c and in this collection.

Returns the number of elements in this collection.

Returns an array of Object for the elements in this collection.

«interface»  
*java.util.Iterator<E>*

+*hasNext(): boolean*  
+*next(): E*  
+*remove(): void*

Returns true if this iterator has more elements to traverse.

Returns the next element from this iterator.

Removes the last element obtained using the next method.



# Collection Example (Basic)

```
import java.util.*;

public class TestCollection {
    public static void main(String[] args) {
        ArrayList<String> collection1 = new ArrayList<>();
        collection1.add("New York");
        collection1.add("Atlanta");
        collection1.add("Dallas");
        collection1.add("Madison");

        System.out.println("A list of cities in collection1:");
        System.out.println(collection1);

        System.out.println("\nIs Dallas in collection1? "
            + collection1.contains("Dallas"));
```

A list of cities in collection1:  
[New York, Atlanta, Dallas, Madison]

Is Dallas in collection1? true

```
collection1.remove("Dallas");  
System.out.println("\n" + collection1.size() +  
    " cities are in collection1 now");
```

3 cities are in collection1 now

```
Collection<String> collection2 = new ArrayList<>();  
collection2.add("Seattle");  
collection2.add("Portland");  
collection2.add("Los Angeles");  
collection2.add("Atlanta");
```

A list of cities in collection2:  
[Seattle, Portland, Los Angeles, Atlanta]

```
System.out.println("\nA list of cities in collection2:");  
System.out.println(collection2);
```

```
ArrayList<String> c1 = (ArrayList<String>)(collection1.clone());  
c1.addAll(collection2);  
System.out.println("\nCities in collection1 or collection2: ");  
System.out.println(c1);
```

Cities in collection1 or collection2:  
[New York, Atlanta, Madison, Seattle, Portland, Los Angeles, Atlanta]


```
c1 = (ArrayList<String>)(collection1.clone());  
c1.retainAll(collection2);  
System.out.print("\nCities in collection1 and collection2: ");  
System.out.println(c1);
```

Cities in collection1 and collection2: [Atlanta]

```
c1 = (ArrayList<String>)(collection1.clone());  
c1.removeAll(collection2);  
System.out.print("\nCities in collection1, but not in 2: ");  
System.out.println(c1);
```

Cities in collection1, but not in 2: [New York, Madison]

```
}
```



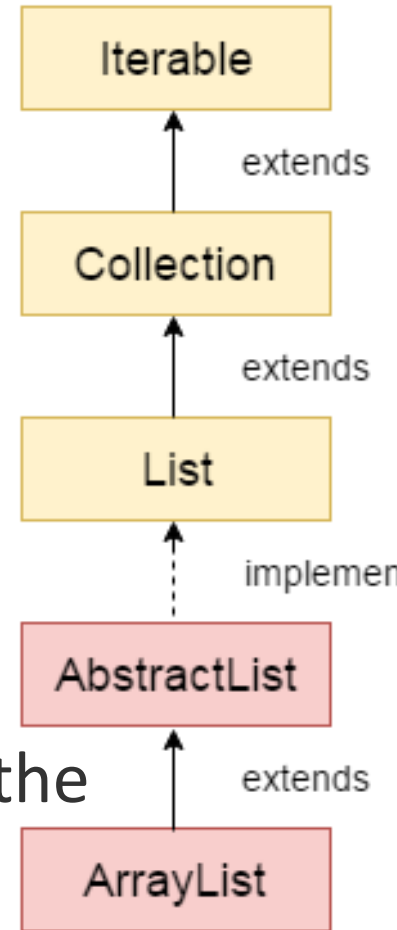
S.No.	Collection in Java	Collections in Java
1.	In Java, collection is an interface.	In Java, collections is a utility class.
2.	It showcases a set of individual objects as a single unit.	It represents multiple utility processes and methods that are utilised to operate on collection.
3.	This interface includes a static method since java8. It can also have both default and methods.	It only includes static methods.

## Difference between Collection and Collections



# Java ArrayList

- Java **ArrayList** class uses a *dynamic array* for storing the elements.
- There is *no size limit*.
- We can add or remove elements anytime.
- Much more flexible than the traditional array.
- It is found in the *java.util* package.
- It is like the Vector in C++.
- The ArrayList in Java can have the duplicate elements also.
- It implements the List interface so we can use all the methods of the List interface here.
- The ArrayList maintains the insertion order internally.
- It inherits the AbstractList class and implements **List interface**.



# Java Non-generic Vs. Generic Collection

- Java generic collection allows you to have only one type of object in a collection.
- Now it is type-safe, so typecasting is not required at runtime.
- Let's see the old non-generic example of creating a Java collection.

```
ArrayList list=new ArrayList(); //creating old non-generic arraylist
```

- Let's see the new generic example of creating java collection.

```
ArrayList<String> list=new ArrayList<String>() //creating new generic arraylist
```

- In a generic collection, we specify the type in angular braces.
- Now ArrayList is forced to have the only specified type of object in it. If you try to add another type of object, it gives a *compile-time error*.

# Java ArrayList Example

```
import java.util.*;

public class ArrayListExample1{

    public static void main(String args[]){

        //Creating arraylist

        ArrayList<String> list=new ArrayList<String>();

        list.add("Mango");//Adding object in arraylist

        list.add("Apple");

        list.add("Banana");

        list.add("Grapes");

        //Printing the arraylist object

        System.out.println(list);

    }

}
```

# Java Vectors

- **Vector** is like the *dynamic array* which can grow or shrink its size.
- Unlike array, we can store n-number of elements in it as there is no size limit.
- It implements the *List* interface, so we can use all the methods of List interface here.
- \*\*\* It is recommended to use the Vector class in the thread-safe implementation only.\*\*\*
- If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.
- The Iterators returned by the Vector class are *fail-fast*.
- In case of concurrent modification, it fails and throws the `ConcurrentModificationException`.
- It is similar to the ArrayList, but with two differences-
  - Vector is synchronized.
  - Java Vector contains many legacy methods that are not the part of a collections framework.

# Java Vectors

```
public class VectorExample {  
    public static void main(String args[]) {  
        //Create a vector  
        Vector<String> vec = new Vector<String>();  
        //Adding elements using add() method of List  
        vec.add("Tiger");  
        vec.add("Lion");  
        vec.add("Dog");  
        vec.add("Elephant");  
        //Adding elements using addElement() method of Vector  
        vec.addElement("Rat");  
        vec.addElement("Cat");  
        vec.addElement("Deer");  
  
        System.out.println("Elements are: "+vec);  
    }  
}
```

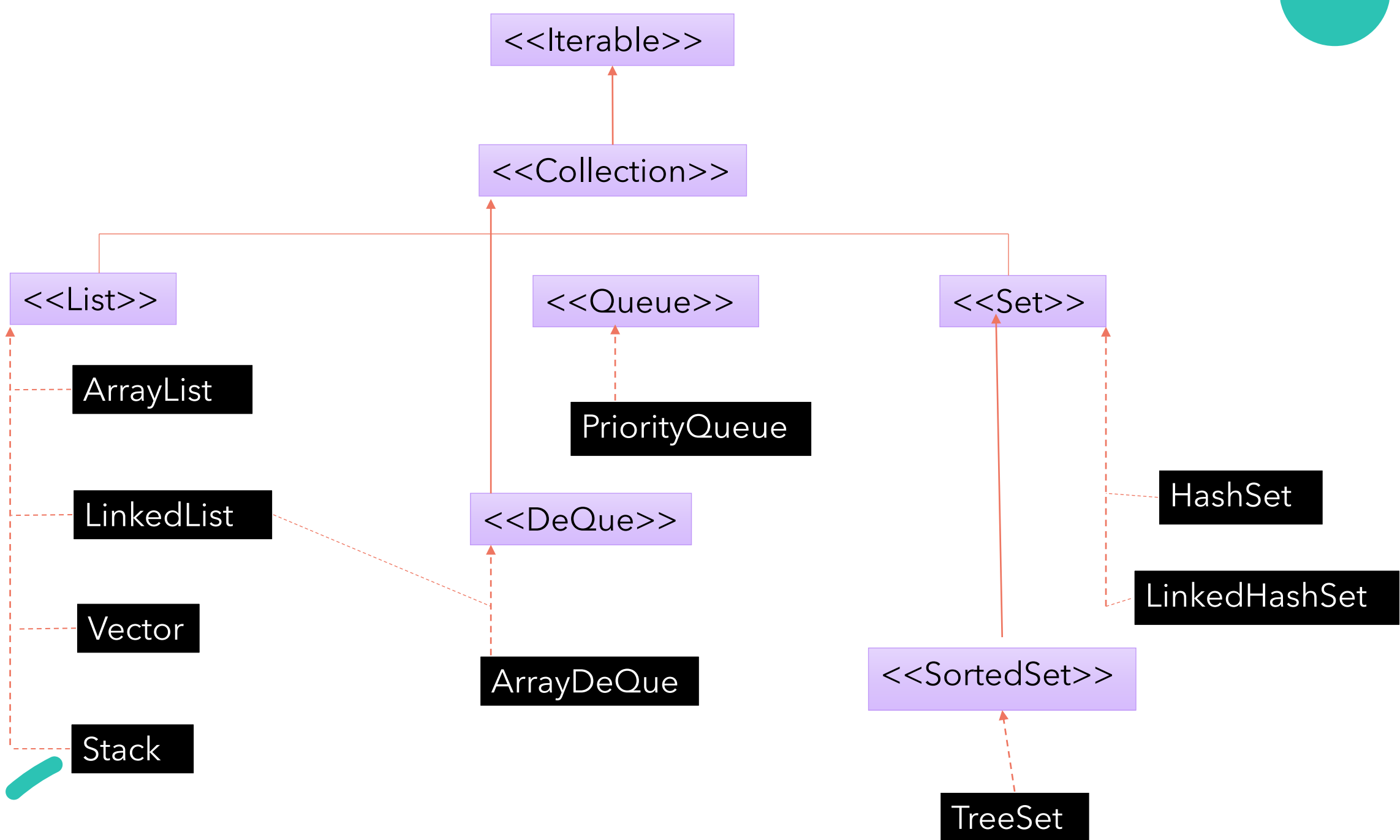


# ArrayList VS Vectors

ArrayList	Vector
1) ArrayList is <b>not synchronized</b> .	Vector is <b>synchronized</b> .
2) ArrayList <b>increments 50%</b> of current array size if the number of elements exceeds from its capacity.	Vector <b>increments 100%</b> means doubles the array size if the total number of elements exceeds than its capacity.
3) ArrayList is <b>not a legacy</b> class. It is introduced in JDK 1.2.	Vector is a <b>legacy</b> class.
4) ArrayList is <b>fast</b> because it is non-synchronized.	Vector is <b>slow</b> because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.
5) ArrayList uses the <b>Iterator</b> interface to traverse the elements.	A Vector can use the <b>Iterator</b> interface or <b>Enumeration</b> interface to traverse the elements.



List



# The List

- A List stores duplicate elements.
- 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.

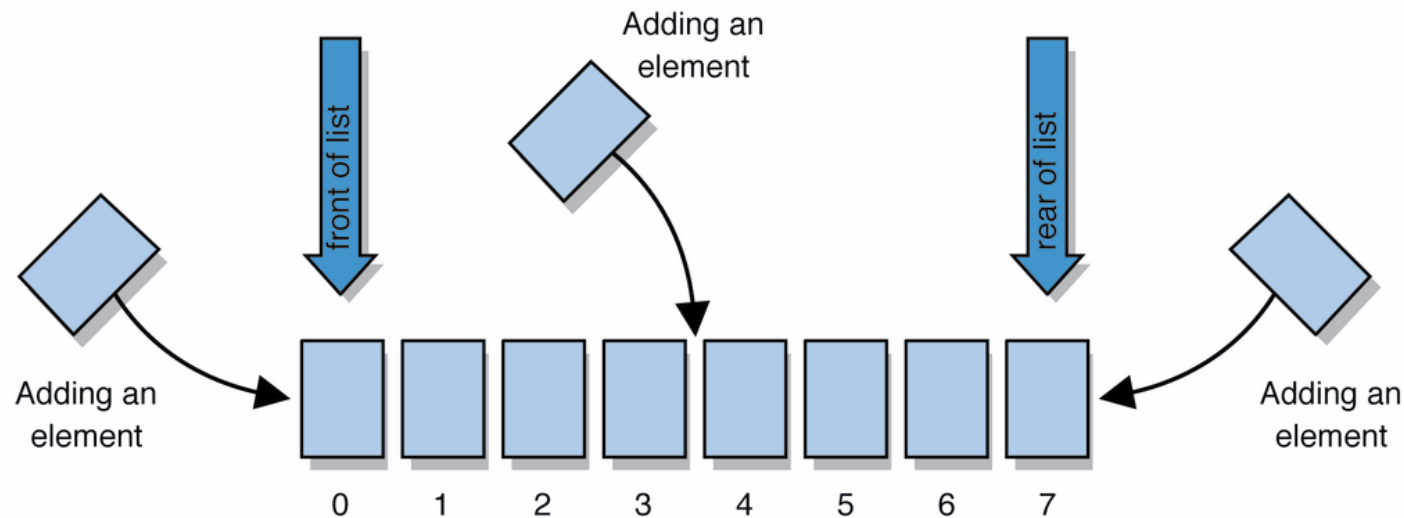
# Syntax

```
public interface List<E> extends Collection<E>{  
    }
```

```
List<Integer> arrayList = new ArrayList<>();  
List<Integer> linkedList = new LinkedList<>();  
List<Integer> vector = new Vector<>();  
List<Integer> stack = new Stack<>();
```

# Lists

- **list**: a collection storing an ordered sequence of elements
  - each element is accessible by a 0-based **index**
  - a list has a **size** (number of elements that have been added)
  - elements can be added to the front, back, or elsewhere
  - in Java, a list can be represented as an **ArrayList** object



# Idea of a list

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

`[]`

- You can add items to the list.
  - The default behavior is to add to the end of the list.

`[hello, ABC, goodbye, okay]`

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
  - Think of an "array list" as an automatically resizing array object.
  - Internally, the list is implemented using an array and a size field.

# ArrayList methods

<code>add(<b>value</b>)</code>	appends value at end of list
<code>add(<b>index</b>, <b>value</b>)</code>	inserts given value just before the given index, shifting subsequent values to the right
<code>clear()</code>	removes all elements of the list
<code>indexOf(<b>value</b>)</code>	returns first index where given value is found in list (-1 if not found)
<code>get(<b>index</b>)</code>	returns the value at given index
<code>remove(<b>index</b>)</code>	removes/returns value at given index, shifting subsequent values to the left
<code>set(<b>index</b>, <b>value</b>)</code>	replaces value at given index with given value
<code>size()</code>	returns the number of elements in list
<code>toString()</code>	returns a string representation of the list such as "[3, 42, -7, 15]"



# ArrayList methods 2

<code>addAll(<b>list</b>)</code> <code>addAll(<b>index</b>, <b>list</b>)</code>	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
<code>contains(<b>value</b>)</code>	returns true if given value is found somewhere in this list
<code>containsAll(<b>list</b>)</code>	returns true if this list contains every element from given list
<code>equals(<b>list</b>)</code>	returns true if given other list contains the same elements
<code>iterator()</code> <code>listIterator()</code>	returns an object used to examine the contents of the list (seen later)
<code>lastIndexOf(<b>value</b>)</code>	returns last index value is found in list (-1 if not found)
<code>remove(<b>value</b>)</code>	finds and removes the given value from this list
<code>removeAll(<b>list</b>)</code>	removes any elements found in the given list from this list
<code>retainAll(<b>list</b>)</code>	removes any elements <i>not</i> found in given list from this list
<code>subList(<b>from</b>, <b>to</b>)</code>	returns the sub-portion of the list between indexes <b>from</b> (inclusive) and <b>to</b> (exclusive)
<code>toArray()</code>	returns the elements in this list as an array

# Positional operations

```
public interface List<E> extends Collection<E>{  
    E get(int index);  
    E set(int index, E element); //optional  
    void add(int index, E element); //optional  
    boolean add(E element); //optional  
    E remove(int index); // optional  
    boolean addAll(int index, Collection<? extends E> c; //  
    optional  
}
```

# Using positional operations

```
public class ListDemo{  
    public static void main(String[] args){  
        List <Integer> arrayList = new ArrayList<>();  
        arrayList.add(0, 1) // adding at 0 index  
        arrayList.add(1, 5) // adding at 1 index  
        List <Integer> arrayList1 = new ArrayList<>();  
        arrayList1.add(1) // adding at 0 index  
        arrayList1.add(2) // adding at 1 index  
        arrayList1.add(3) // adding at 2 index  
        arrayList.addAll(0, arrayList1); //adding arrayList1 at 0 index  
        arrayList.remove(1);  
    }  
}
```

# Search operations

```
public interface List<E> extends Collection<E>{
    int indexOf(Object o);
    int lastIndexOf(Object o);
}

...

List<String> arrayList = new ArrayList<>(5); //Size of 5
arrayList.add("Hello");
arrayList.add("World");
System.out.println("Hello is at index: " +
                    arrayList.indexOf("Hello"));
System.out.println("World is at index: " +
                    arrayList.lastIndexOf("World"));
```

# Range-view operations

```
public interface List<E> extends Collection<E>{  
    List<E> subList(int fromIndex, int toIndex);  
}
```

# Operations on the List

- **Access:** manipulates the elements based on the provided index (numerical) position in the list.
- **Search:** for specified object in the list and returns its index (position).
- **Iteration:** extends from the Iterator Interface, provide advantage of the List logical order.
- **Range:** the subList method performs arbitrary range operations on the list.

# Two Ways to Implement Lists

There are two ways to implement a list.

## Using arrays:

- One is to use an array to store the elements.
- The array is dynamically created.
- If the capacity of the array is exceeded, create a new larger array and copy all the elements from the current array to the new array.
- Default size is 10, uses ***System.arraycopy*** when increasing size.

## Using linked list:

- 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.

# List Implementation

- **ArrayList:**

- A resizable array implementation of a List interface.
- Default capacity = 10, increased by 50%.
- *ArrayList(int initialCapacity)* OR  
*ensureCapacity(int)*
- Allow duplicates and nulls.



# Typical uses


- Simple iteration of elements.
- Fast random access  $\sim O(1) \sim$  constant time
  - So size doesn't matter here.
- Appending elements or deleting elements  $\sim O(1) \sim$  constant time

# add & remove Methods

- *add(index, element)*
  - Following elements **shifted right** by one position.
  - $O(n)$  ~ Linear time
- *remove(index)*
  - Following elements **shifted left** by one position.
  - $O(n)$  ~ Linear time

# Search methods

- *contains()*
- *indexOf()*
  - $O(n)$  ~ Linear time
  - Uses *equals()*
  - Frequent search operations then consider using **Set** implementation, e.g., HashSet ~  $O(1)$  ~ constant time




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

```
public class CreateArrayListExample {
    public static void main(String[] args) {
        // Creating an ArrayList of String
        List<String> animals = new ArrayList<>();
        // Adding new elements to the
        ArrayList animals.add("Lion");
        animals.add("Tiger");
        animals.add("Cat");
        animals.add("Dog");
```

[Lion, Tiger, Cat, Dog]

```
System.out.println(animals);
```



```
// Adding an element at a particular index in an ArrayList
animals.add(2, "Elephant");
System.out.println(animals); } }
```

[Lion, Tiger, Elephant, Cat, Dog]



# Linked Lists

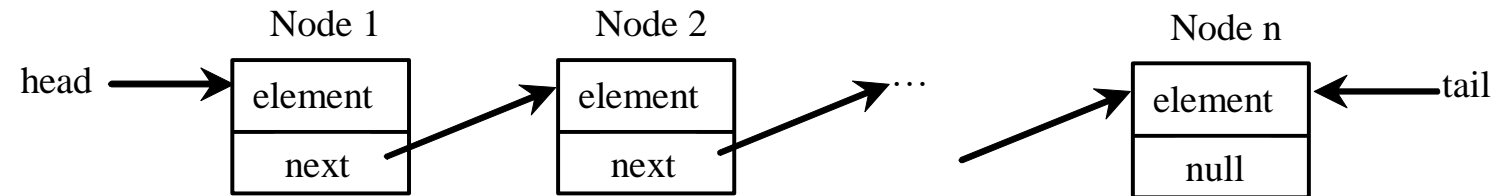
- In ArrayList the methods
  - get(int index)
  - set(int index, Object o)
  - for accessing and modifying an element through an index and the add(Object o) for adding an element at the end of the list are efficient.
- However, the methods
  - add(int index, Object o)
  - remove(int index)
  - are inefficient because it requires shifting potentially a large number of elements.
- You can use a linked structure to implement a list to improve efficiency for adding and removing an element anywhere in a list.

# Nodes in Linked Lists

- A linked list consists of nodes.
- Each node contains an element, and each node is linked to its next neighbor.
- Thus a node can be defined as a class, as follows:

```
class Node<E> {  
    E element;  
    Node<E> next;
```

```
    public Node(E o) {  
        element = o;  
    }  
}
```



# Adding Three Nodes

- The variable
  - head refers to the first node in the list
  - tail refers to the last node in the list.
- If the list is empty, both are null. For example, you can create three nodes to store three strings in a list, as follows:

Step 1: Declare head and tail:

```
Node<String> head = null;  
Node<String> tail = null;
```

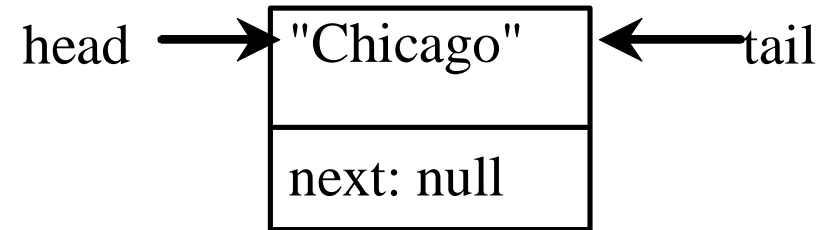
The list is empty now

## Adding Three Nodes, cont.

Step 2: Create the first node and insert it to the list:

```
head = new Node<> ("Chicago");  
tail = head;
```

After the first node is inserted



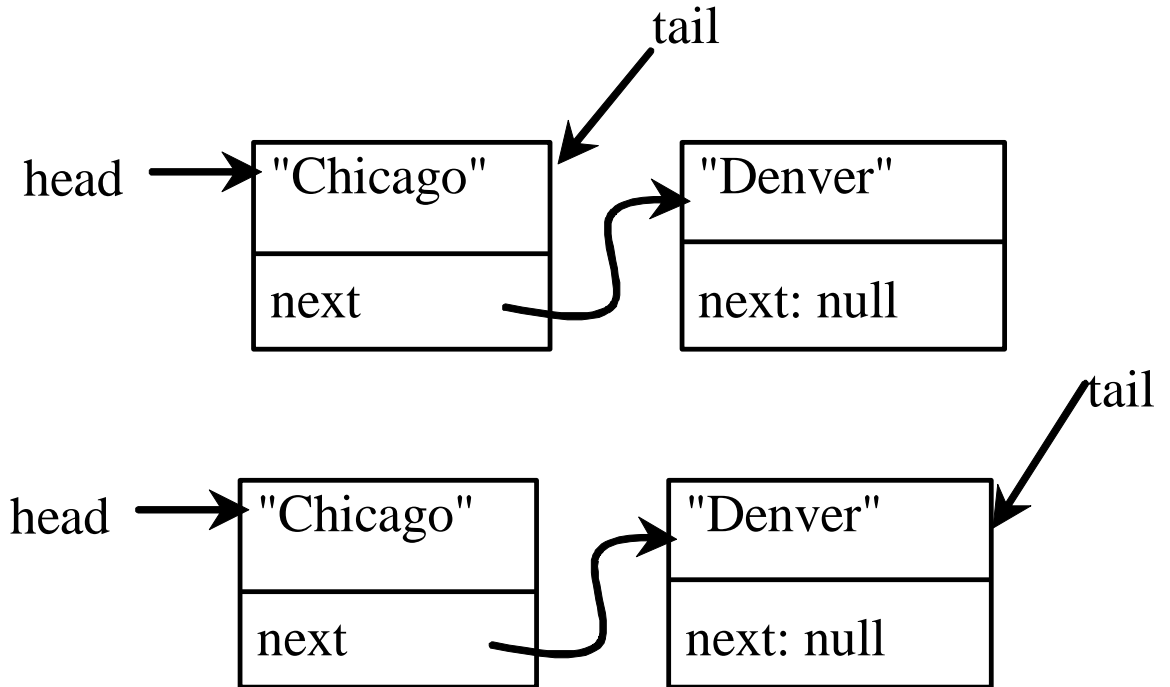


## Adding Three Nodes, cont.

Step 3: Create the second node and insert it to the list:

```
tail.next = new Node<>("Denver");
```

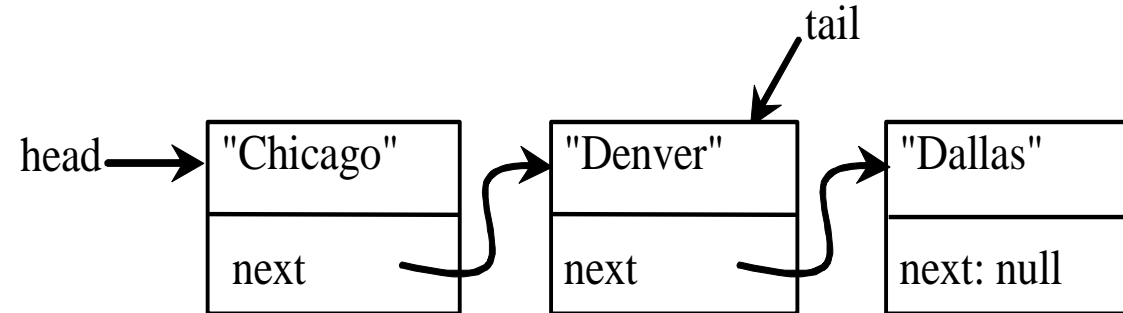
```
tail = tail.next;
```



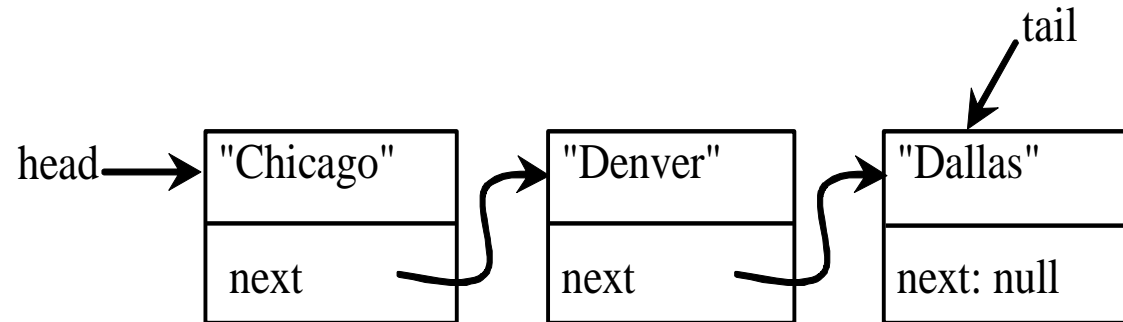
## Adding Three Nodes, cont.

Step 4: Create the third node and insert it to the list:

```
tail.next =  
  new Node<>("Dallas");
```



```
tail = tail.next;
```



# Examining List

- elements of Java Lists, Sets and Maps can't be accessed by index
  - must use a "foreach" loop:

```
List<Integer> scores = new ArrayList<Integer>();  
for (int score : scores) {  
    System.out.println("The score is " + score);  
}
```

- Problem: foreach is read-only; cannot modify set while looping

```
for (int score : scores) {  
    if (score < 60) {  
        // throws a ConcurrentModificationException  
        scores.remove(score);  
    }  
}
```

# Sets

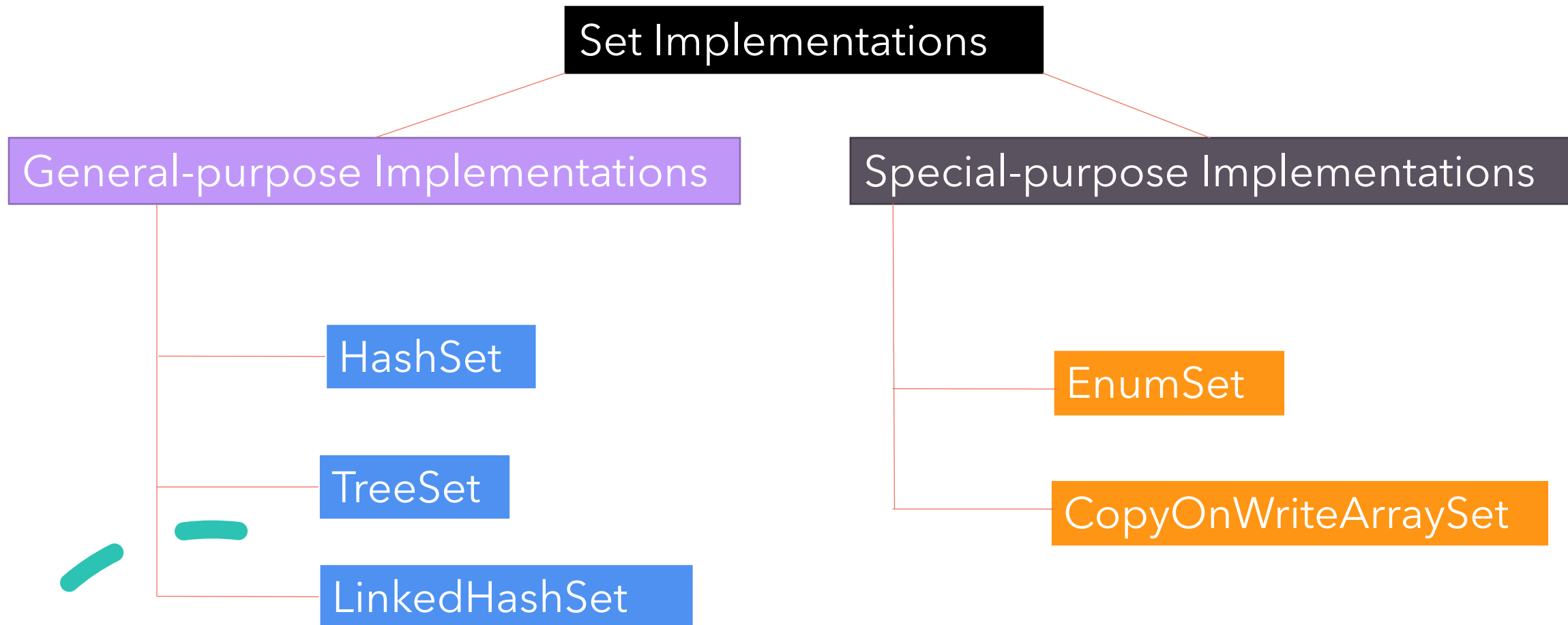


# Set *Interface*

- The Set interface extends the Collection interface.
- It does not introduce new methods or constants, but it stipulates that an instance of Set contains no duplicate elements.
- The concrete classes that implement Set must ensure that no duplicate elements can be added to the set.
- That is no two elements  $e_1$  and  $e_2$  can be in the set such that  $e_1.equals(e_2)$  is true.

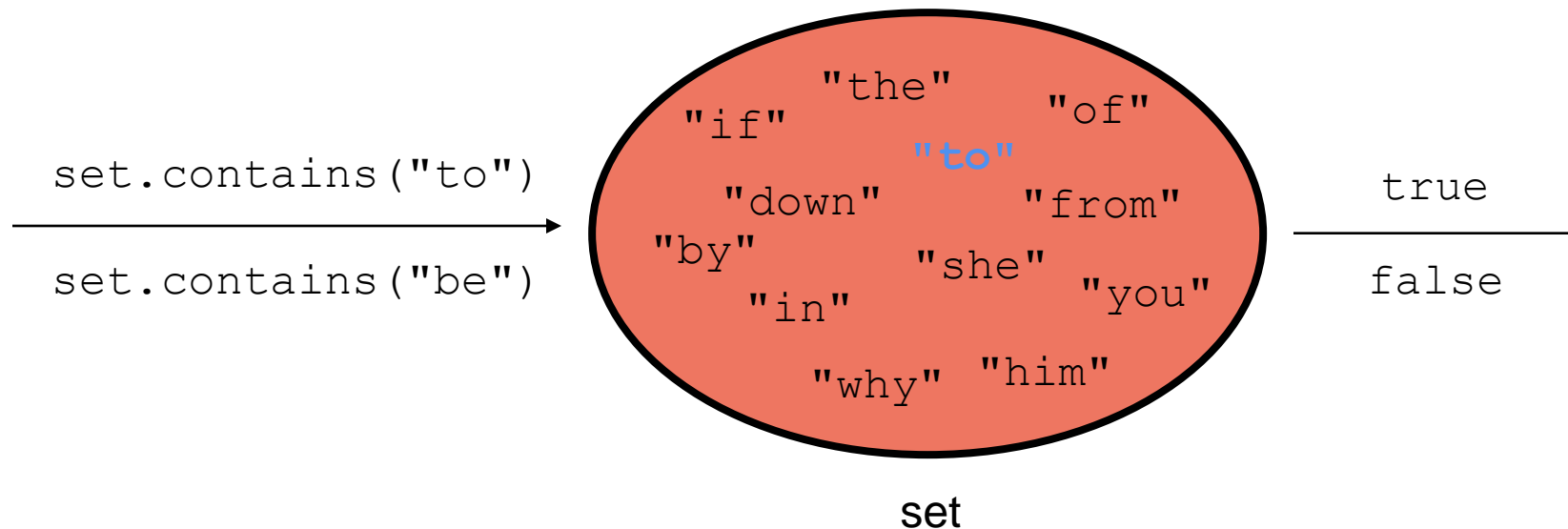
# Set Implementation

- Set implementations are grouped into two groups



# Set

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)
- We don't think of a set as having indexes; we just add things to the set in general and don't worry about order



# Set implementation

- in Java, sets are represented by Set interface in `java.util`
- Set is implemented by HashSet and TreeSet classes
  - HashSet: implemented using a "hash table" array;  
very fast:  **$O(1)$**  for all operations  
elements are stored in unpredictable order
  - TreeSet: implemented using a "binary search tree";  
pretty fast:  **$O(\log N)$**  for all operations  
elements are stored in sorted order
  - LinkedHashSet:  **$O(1)$**  but stores in order of insertion



# Set methods

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

```
...
```

```
Set<Integer> set = new TreeSet<Integer>(); // empty
```

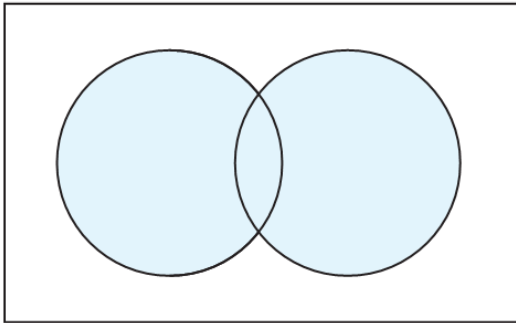
```
Set<String> set2 = new HashSet<String>(list);
```

- can construct an empty set, or one based on a given collection

add ( <b>value</b> )	adds the given value to the set
contains ( <b>value</b> )	returns <code>true</code> if the given value is found in this set
remove ( <b>value</b> )	removes the given value from the set
clear()	removes all elements of the set
size()	returns the number of elements in list
isEmpty()	returns <code>true</code> if the set's size is 0
toString()	returns a string such as "[3, 42, -7, 15]"

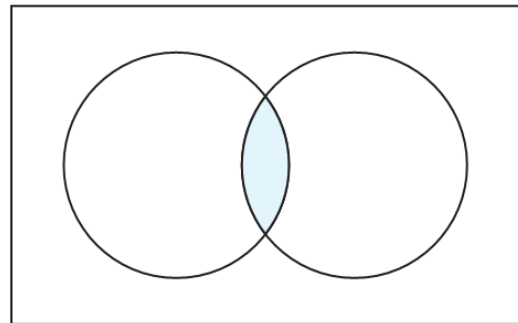
# Set operations

$A \cup B$  Union



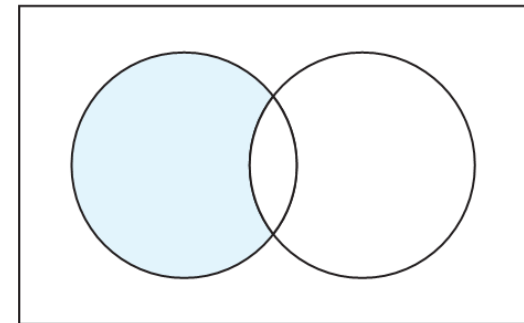
`addAll`

$A \cap B$  Intersection



`retainAll`

$A - B$  Difference



`removeAll`

<code>addAll (<b>collection</b>)</code>	adds all elements from the given collection to this set
<code>containsAll (<b>coll</b>)</code>	returns <code>true</code> if this set contains every element from given set
<code>equals (<b>set</b>)</code>	returns <code>true</code> if given other set contains the same elements
<code>iterator()</code>	returns an object used to examine set's contents ( <i>seen later</i> )
<code>removeAll (<b>coll</b>)</code>	removes all elements in the given collection from this set
<code>retainAll (<b>coll</b>)</code>	removes elements <i>not</i> found in given collection from this set
<code>toArray()</code>	returns an array of the elements in this set

# Hash Set

- No duplicates.
- Useful when *uniqueness & fast lookup* matters
- HashSet is much faster than TreeSet (constant-time vs long-time for most operations)
- Insertion order does not matter (no guarantee of order).
- Default capacity is 16.
- Capacity increases with power of 2. ( $X^2$ )
- It is a Hash Table implementation of a Set interface.
- Internally uses **HashMap**.

# Typical Uses and Useful methods

- **Quick** *lookup, insertion, and deletion*  $\sim O(1)$
- **Insertion** order is not important
- **Better** for *removeAll()* and *retainAll()*
- **add()** *//To add elements to the set*
- **contains()** *//To check if a particular element present in the HashSet*
- **remove()** *//To remove the specified element*
- **clear()** *//To remove all the elements*
- **size()** *//To check number of elements*
- **isEmpty()** *//To check if an instance of HashSet is empty or not*
- **iterator()** *//Return an iterator over the elements, elements visited in no order.*

```
import java.util.*;

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

        // 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 set
        for (String s: set) {
            System.out.print(s.toUpperCase() + " ");
        }

        // Process the elements using a forEach method
        System.out.println();
        set.forEach(e -> System.out.print(e.toLowerCase() + " "));
    }
}
```

Added twice

[San Francisco, New York, Paris, Beijing, London]

Lambda Expression replacing the Inner class Implementation



```
import java.util.HashSet;
import java.util.Set;
```

```
public class SetDemo {
```

```
    private static void hashSetDemo() {
```

```
        Book book1 = new Book("Walden", "Henry Thoreau", 1854);
```

```
        Book book2 = new Book("Walden", "Henry Thoreau", 1854);
```

```
        Set<Book> set2 = new HashSet<>();
```

```
        set2.add(book1);
```

```
        set2.add(book2);
```

```
        System.out.println("set2: " + set2);
```

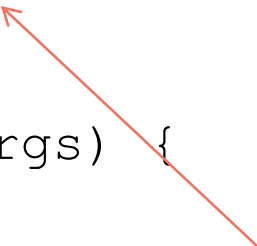
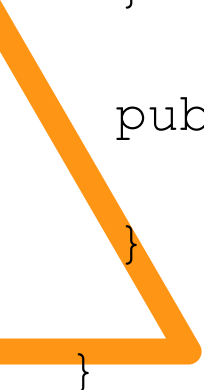
```
    }
```

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

```
        hashSetDemo();
```

```
    }
```

```
}
```



```
set2: [Book [title=Walden, author=Henry Thoreau
, year=1854], Book [title=Walden, author=Henry
Thoreau, year=1854]]
```

```
class Book {  
    private String title;  
    private String author;  
    private int year;  
    public String getTitle() {  
        return title;    }  
    public void setTitle(String title) {  
        this.title = title;    }  
    public String getAuthor() {  
        return author;    }  
    public void setAuthor(String author) {  
        this.author = author;    }  
    public int getYear() {  
        return year;    }  
    public void setYear(int year) {  
        this.year = year;    }  
    public Book(String title, String author, int year) {  
        super();  
        this.title = title;  
        this.author = author;  
        this.year = year;  
    }  
}
```

```
@Override  
public String toString() {  
    return "Book [title=" + title + ",  
    author=" + author + ", year=" + year + "];"  
}  
  
public int hashCode() {  
    return title.hashCode();  
}  
  
public boolean equals(Object o) {  
    return (year == (((Book)o).getYear())) &&  
    (author.equals((((Book)o).getAuthor())));  
}  
}
```

After implementing the Hashcode:  
set2: [Book [title=Walden, author=Henry Thoreau, year=1854]]

# LinkedHashSet

- Extends HashSet with a linked-list implementation.
- Supports the order of the elements in which they are entered.
- Provides insertion-ordered iteration (least recently inserted to most recently) and run almost as fast as HashSet.
- If you don't need to maintain the order of the elements then use HashSet. (which is more efficient)
- The other tuning parameters works just same as HashSet, but the iteration time is not affected by the capacity.



```
import java.util.*;

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

        // 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 elements: set) {
            System.out.print(elements.toLowerCase() + " ");
        }
    }
}
```

Output:

[London, Paris, New York, San Francisco, Beijing]  
london paris new york san francisco beijing

# TreeSet

- Guarantees that the elements in the set are sorted.
- Sorts the elements in ascending order.
- A TreeSet should be our primary choice if we want to keep our entries sorted as a TreeSet may be accessed and traversed in either ascending or descending order.
- Operations like add, remove and search takes longer time than in the HashSet.
- first() and last() methods, return the first and last elements in the set.

# Set and List Performance

```
import java.util.*;

public class SetListPerformanceTest {
    static final int N = 5000;

    public static void main(String[] args) {
        // Add numbers 0, 1, 2, ..., N - 1 to the array list
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < N; i++)
            list.add(i);
        Collections.shuffle(list); // Shuffle the array list
```

## **// Create a hash set, and test its performance**

```
Collection<Integer> set1 = new HashSet<>(list);
System.out.println("Member test time for hash set is " + getTestTime(set1) + " milliseconds");
System.out.println("Remove element time for hash set is " + getRemoveTime(set1) + " milliseconds");
```

Member test time for hash set is 20 milliseconds

Remove element time for hash set is 27 milliseconds

## **// Create a linked hash set, and test its performance**

```
Collection<Integer> set2 = new LinkedHashSet<>(list);
System.out.println("Member test time for linked hash set is " + getTestTime(set2) + " milliseconds");
System.out.println("Remove element time for linked hash set is " + getRemoveTime(set2) + " milliseconds");
```

Member test time for linked hash set is 27 milliseconds

Remove element time for linked hash set is 26 milliseconds

**// Create a tree set, and test its performance**

```
Collection<Integer> set3 = new TreeSet<>(list);  
System.out.println("Member test time for tree set is " + getTestTime(set3) + " milliseconds");  
System.out.println("Remove element time for tree set is " +  
    getRemoveTime(set3) + " milliseconds");
```

Member test time for tree set is 47 milliseconds

Remove element time for tree set is 34 milliseconds

**// Create an array list, and test its performance**

```
Collection<Integer> list1 = new ArrayList<>(list);  
System.out.println("Member test time for array list is "  
    getTestTime(list1) + " milliseconds");  
System.out.println("Remove element time for array list is " +  
    getRemoveTime(list1) + " milliseconds");
```

Member test time for array list is 39802 milliseconds

**// Create a linked list, and test its performance**

```
Collection<Integer> list2 = new LinkedList<>(list);  
System.out.println("Member test time for linked list is " +  
    getTestTime(list2) + " milliseconds");  
System.out.println("Remove element time for linked list is " +  
    getRemoveTime(list2) + " milliseconds");  
}
```

Remove element time for array list is 16196 milliseconds

Member test time for linked list is 52197 milliseconds

Remove element time for linked list is 14870 milliseconds

```
public static long getTestTime(Collection<Integer> c) {
    long startTime = System.currentTimeMillis();

    // Test if a number is in the collection
    for (int i = 0; i < N; i++)
        c.contains((int) (Math.random() * 2 * N));

    return System.currentTimeMillis() - startTime;
}

public static long getRemoveTime(Collection<Integer> c)
{
    long startTime = System.currentTimeMillis();

    for (int i = 0; i < N; i++)
        c.remove(i);

    return System.currentTimeMillis() - startTime;
}
}
```

Conclusion: **Sets** are more efficient than **List** for testing whether an element is in a set or list.



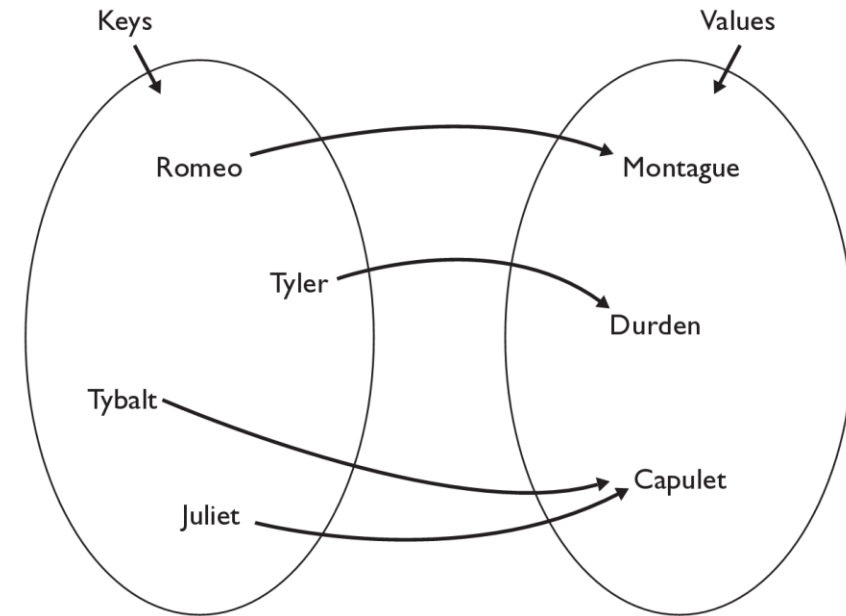
Maps

# The Map Interface

- Map is a container object that stores a collection of key/ value pairs.
- 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.
- Cannot contain duplicate keys.
- Each key maps to one value.
- HashMap and HashSet both are based on HashTable.

# The Map

- **map**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
  - a.k.a. "dictionary", "associative array", "hash"
- basic map operations:
  - **put**(key, value ): Adds a mapping from a key to a value.
  - **get**(key ): Retrieves the value mapped to the key.
  - **remove**(key ): Removes the given key and its mapped value.



`myMap.get("Juliet")` returns "Capulet"



# Map Interface

*java.util.Map<K, V>*

*+clear(): void*

Removes all mappings from this map.

*+containsKey(key: Object): boolean*

Returns true if this map contains a mapping for the specified key.

*+containsValue(value: Object): boolean*

Returns true if this map maps one or more keys to the specified value.

*+entrySet(): Set*

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 mappings.

*+keySet(): Set<K>*

Returns a set consisting of the keys in this map.

*+put(key: K, value: V): V*

Puts a mapping in this map.

*+putAll(m: Map): void*

Adds all the mappings from m to this map.

*+remove(key: Object): V*

Removes the mapping for the specified key.

*+size(): int*

Returns the number of mappings in this map.

*+values(): Collection<V>*

Returns a collection consisting of the values in this map.

# Hash Table

- Implements an **associative array**.
  - It is an array where each element basically an association between a key and value.  
<key, value> → <name, phone>
  - Each <key, value> pair is also referred as **mapping**.
- A Hash Table is also referred as a **Dictionary**.

# Key Operations

- Insert<key, value>
- Search by key
- Remove by key

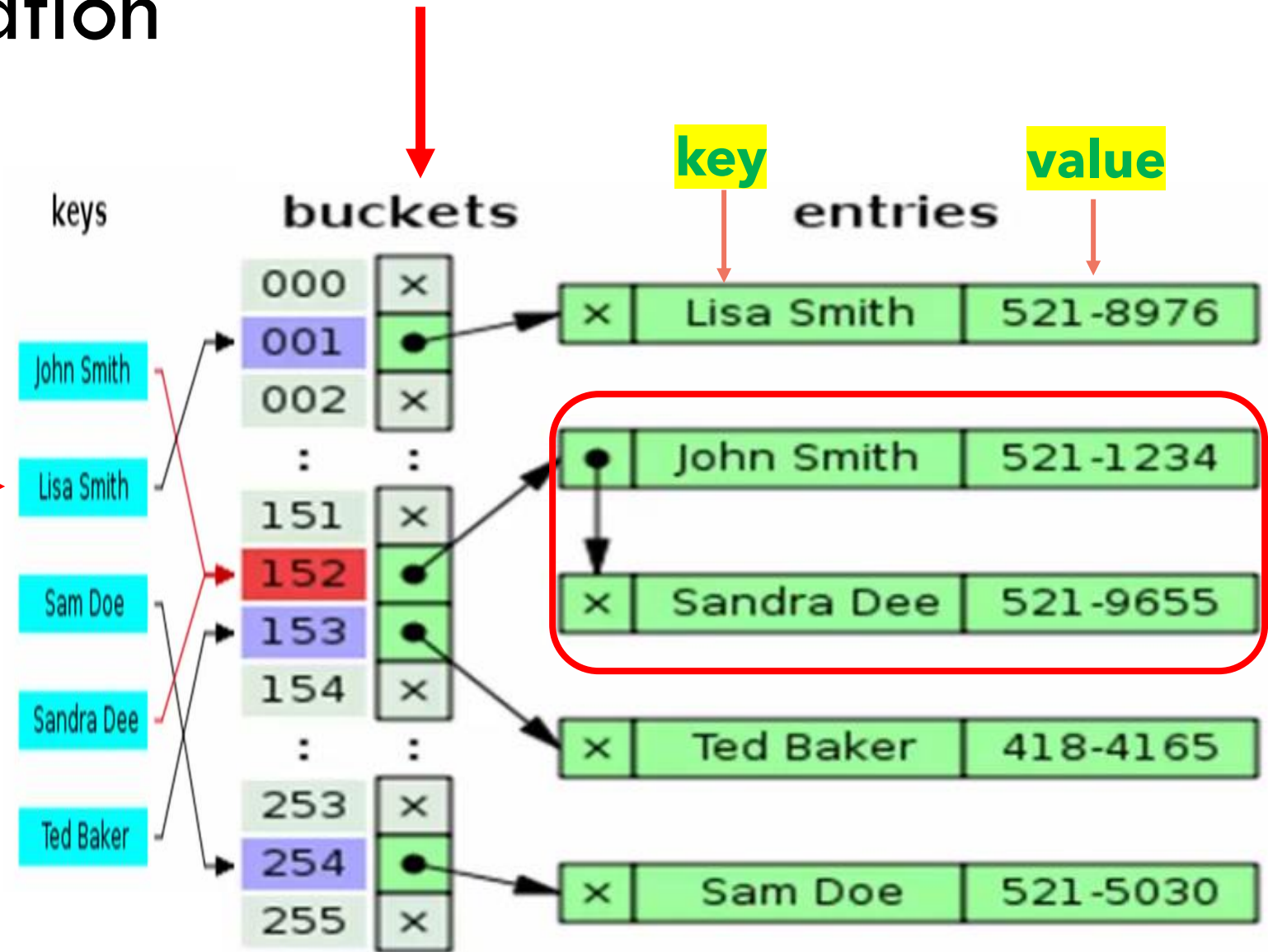
**$O(1)$**

# Hash Table Characteristics

- **No duplicate keys.**
- Duplicate values is fine.
- 1 key -> 1 value (at most)
- nulls (depending on the implementation)
  - 1 null key, null values are fine. ~ (HashMap)
  - No null values at all. ~ (TreeHashMap)

# Implementation

- Each element of an **Array** is referencing to a Linkedlist.
- Array does not store the <key, value> mapping.
- Linkedlist actually stores the actual <key,value> mapping.
- Each slot in an array is commonly referred to as a bucket.
- Each Linkedlist can have multiple mappings.
- For quick search in the bucket for a particular mapping, **Hash Function** is applied.



# Hash Function

Hash Function is the function of key and the array-size

$$\text{bucket \#} = f(\text{key, array-size})$$

e.g.,  $\text{key \% array-size}$



Transforms large space to smaller one

$$15 = 315 \% 25$$



Target **bucket#** for the mapping with key 315

## 2 – Step Hashing in Java

Step -1: hashCode() function is invoked, which returns the hash code belongs to the key. hashCode() belongs to the Object class and returns the hash code (which basically is a memory address of an object converted into **int** value).

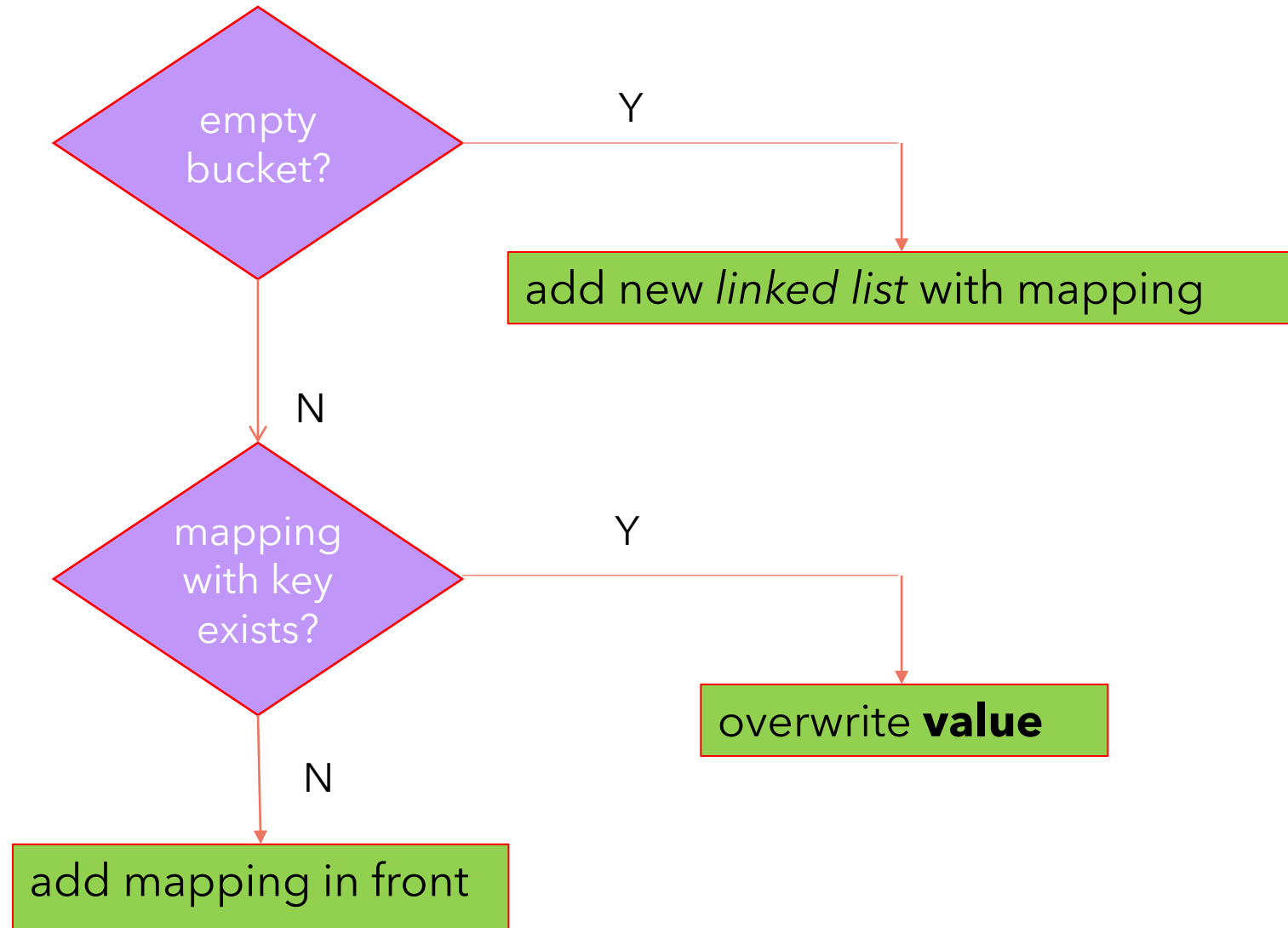
Step -2: Once getting the hash a bitwise AND is applied on it in order to offset into the hash table (basically finding the bucket number)

hash = **hash**(key.hashCode())  
bucket # = hash **&** (length - 1)

### Properties of Hash function:

- Quickly locate bucket
- Uniformly disperse elements

# Insertion ~ Separate Chaining





# Other Performance Factors

capacity = # buckets

load factor -> decides on *resizing*

*(how full the Hash Table can get before its capacity is automatically increased.)*

For HashSet: default capacity ~ 16

load factor ~ 0.75

*resizing involves expensive **rehashing***

**High load factor** -> less freq. **rehashing** -> more lookup time

# Applications

- Database indexing.
- NoSql databases.
- Switch statements.

# Implementation

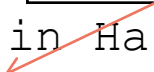
- HashMap:
  - The HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping.
  - Use it when you want speed and not the order.
  - Entries in the **HashMap** are in random order.
- TreeMap:
  - The TreeMap class, implementing SortedMap, is efficient for traversing the keys in a sorted order.
  - Entries in **TreeMap** are in increasing order of the keys.
- LinkedHashMap:
  - extends HashMap with a linked list implementation, that supports an ordering of the entries in the map.
  - Use it when you need speed nearly as good as HashMap and also give you order.



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



```
public class TestMap {  
    public static void main(String[] args) {  
        // Create a HashMap  
        Map<String, Integer> hashMap = new HashMap<>();  
        hashMap.put("Smith", 30);  
        hashMap.put("Anderson", 31);  
        hashMap.put("Lewis", 29);  
        hashMap.put("Cook", 29);
```

Display entries in HashMap  
{Cook=29, Smith=30, Lewis=29, Anderson=31}

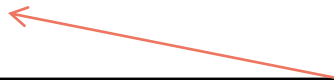


```
        System.out.println("Display entries in HashMap");  
        System.out.println(hashMap + "\n");
```

```
        // Create a TreeMap from the preceding HashMap  
        Map<String, Integer> treeMap = new TreeMap<>(hashMap);  
        System.out.println("Display entries in ascending order of key");  
        System.out.println(treeMap);
```



Display entries in ascending order of key  
{Anderson=31, Cook=29, Lewis=29, Smith=30}



## // Create a LinkedHashMap

```
Map<String, Integer> linkedHashMap = new LinkedHashMap<>();  
linkedHashMap.put("Smith", 30);  
linkedHashMap.put("Anderson", 31);  
linkedHashMap.put("Lewis", 29);  
linkedHashMap.put("Cook", 29);
```

## // Display the age for Lewis

```
System.out.println("\nThe age for " + "Lewis is " +  
    linkedHashMap.get("Lewis"));
```

The age for Lewis is 29

```
System.out.println("Display entries in LinkedHashMap");
```

```
System.out.println(linkedHashMap);
```

Display entries in LinkedHashMap  
{Smith=30, Anderson=31, Cook=29, Lewis=29}

```
// Display each entry with name and age
```

```
System.out.print("\nNames and ages are ");
```

```
treeMap.forEach(  
    (name, age) -> System.out.print(name + ": " + age + " ");
```

```
    (name, age) -> System.out.print(name + ": " + age + " ");
```

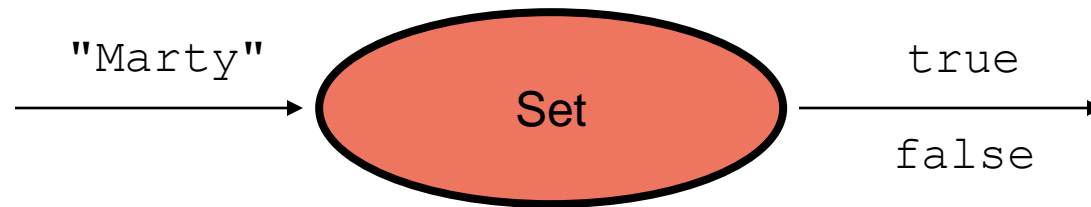
```
}
```

```
}
```

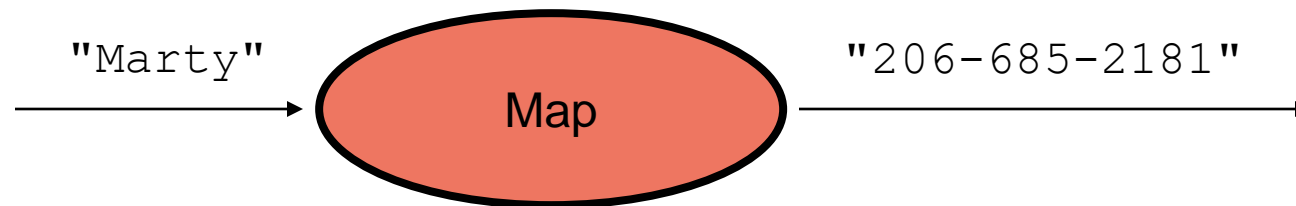
Names and ages are Anderson: 31 Cook: 29 Lewis: 29 Smith: 30

# Maps vs. sets

- A set is like a map from elements to boolean values.
  - Set: *Is "Marty" found in the set? (true/false)*



- Map: *What is "Marty" 's phone number?*



# keySet and values

- `keySet` method returns a `Set` of all keys in the map
  - can loop over the keys in a `foreach` loop
  - can get each key's associated value by calling `get` on the map

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2); // ages.keySet() returns Set<String>
ages.put("Vicki", 57);
for (String name : ages.keySet()) { // Geneva -> 2
    int age = ages.get(name); // Marty -> 19
    System.out.println(name + " -> " + age); // Vicki -> 57
}
```

- `values` method returns a collection of all values in the map
  - can loop over the values in a `foreach` loop
  - no easy way to get from a value to its associated key(s)

# Problem: opposite mapping

- It is legal to have a map of sets, a list of lists, etc.
- Suppose we want to keep track of each TA's GPA by name.

```
Map<String, Double> taGpa = new HashMap<String, Double>();  
taGpa.put("Jared", 3.6);  
taGpa.put("Alyssa", 4.0);  
taGpa.put("Steve", 2.9);  
taGpa.put("Stef", 3.6);  
taGpa.put("Rob", 2.9);  
...  
System.out.println("Jared's GPA is " +  
                    taGpa.get("Jared"));    // 3.6
```

- This doesn't let us easily ask which TAs got a given GPA.
  - How would we structure a map for that?



# Reversing a map

- We can reverse the mapping to be from GPAs to names.

```
Map<Double, String> taGpa = new HashMap<Double, String>();  
taGpa.put(3.6, "Jared");  
taGpa.put(4.0, "Alyssa");  
taGpa.put(2.9, "Steve");  
taGpa.put(3.6, "Stef");  
taGpa.put(2.9, "Rob");  
...  
System.out.println("Who got a 3.6? " +  
                    taGpa.get(3.6));    // ???
```

- What's wrong with this solution?
  - More than one TA can have the same GPA.
  - The map will store only the last mapping we add.

# Proper map reversal

- Really each GPA maps to a *collection* of people.

```
Map<Double, Set<String>> taGpa =  
    new HashMap<Double, Set<String>> ();  
taGpa.put (3.6, new TreeSet<String>());  
taGpa.get (3.6) .add ("Jared");  
taGpa.put (4.0, new TreeSet<String>());  
taGpa.get (4.0) .add ("Alyssa");  
taGpa.put (2.9, new TreeSet<String>());  
taGpa.get (2.9) .add ("Steve");  
taGpa.get (3.6) .add ("Stef");  
taGpa.get (2.9) .add ("Rob");  
...  
System.out.println ("Who got a 3.6? " +  
    taGpa.get (3.6));    // [Jared, Stef]
```

- must be careful to initialize the set for a given GPA before adding

# Maps and tallying

- a map can be thought of as generalization of a tallying array
  - the "index" (key) doesn't have to be an `int`
- recall previous tallying examples from CSE 142
  - count digits: 22092310907

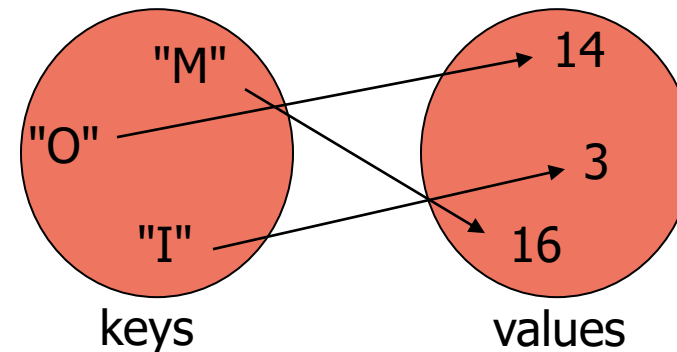
→

index	0	1	2	3	4	5	6	7	8	9
value	3	1	3	0	0	0	0	1	0	2

// (M)cCain, (O)bama, (I)ndependent

- count votes: "MOOOOOOOMMMMMMOOOOOOOMOMMIOMMMIOMMMIO"

key	"M"	"O"	"I"
value	16	14	3



# Map implementation

- Map is implemented by the `HashMap` and `TreeMap` classes
  - `HashMap`: implemented using an array called a "hash table"; extremely fast:  **$O(1)$** ; keys are stored in unpredictable order
  - `TreeMap`: implemented as a linked "binary tree" structure; very fast:  **$O(\log N)$** ; keys are stored in sorted order
- A map requires 2 type parameters: one for keys, one for values.

**// maps from String keys to Integer values**

```
Map<String, Integer> votes = new HashMap<String, Integer>();
```

# Map methods

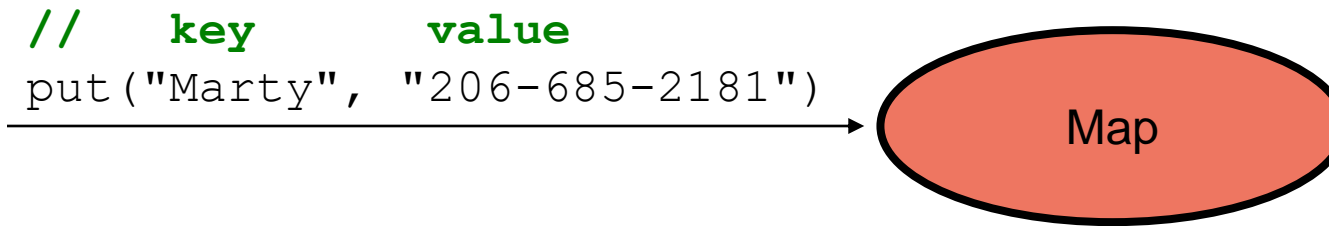
<code>put(<b>key</b>, <b>value</b>)</code>	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
<code>get(<b>key</b>)</code>	returns the value mapped to the given key ( <code>null</code> if not found)
<code>containsKey(<b>key</b>)</code>	returns <code>true</code> if the map contains a mapping for the given key
<code>remove(<b>key</b>)</code>	removes any existing mapping for the given key
<code>clear()</code>	removes all key/value pairs from the map
<code>size()</code>	returns the number of key/value pairs in the map
<code>isEmpty()</code>	returns <code>true</code> if the map's size is 0
<code>toString()</code>	returns a string such as " <code>{a=90, d=60, c=70}</code> "

<code>keySet()</code>	returns a set of all keys in the map
<code>values()</code>	returns a collection of all values in the map
<code>putAll(<b>map</b>)</code>	adds all key/value pairs from the given map to this map
<code>equals(<b>map</b>)</code>	returns <code>true</code> if given map has the same mappings as this one

# Using maps

- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).



- Later, we can supply only the key and get back the related value:  
Allows us to ask: *What is Marty's phone number?*

