Advanced Programming
CSE 201
Instructor: Sambuddho

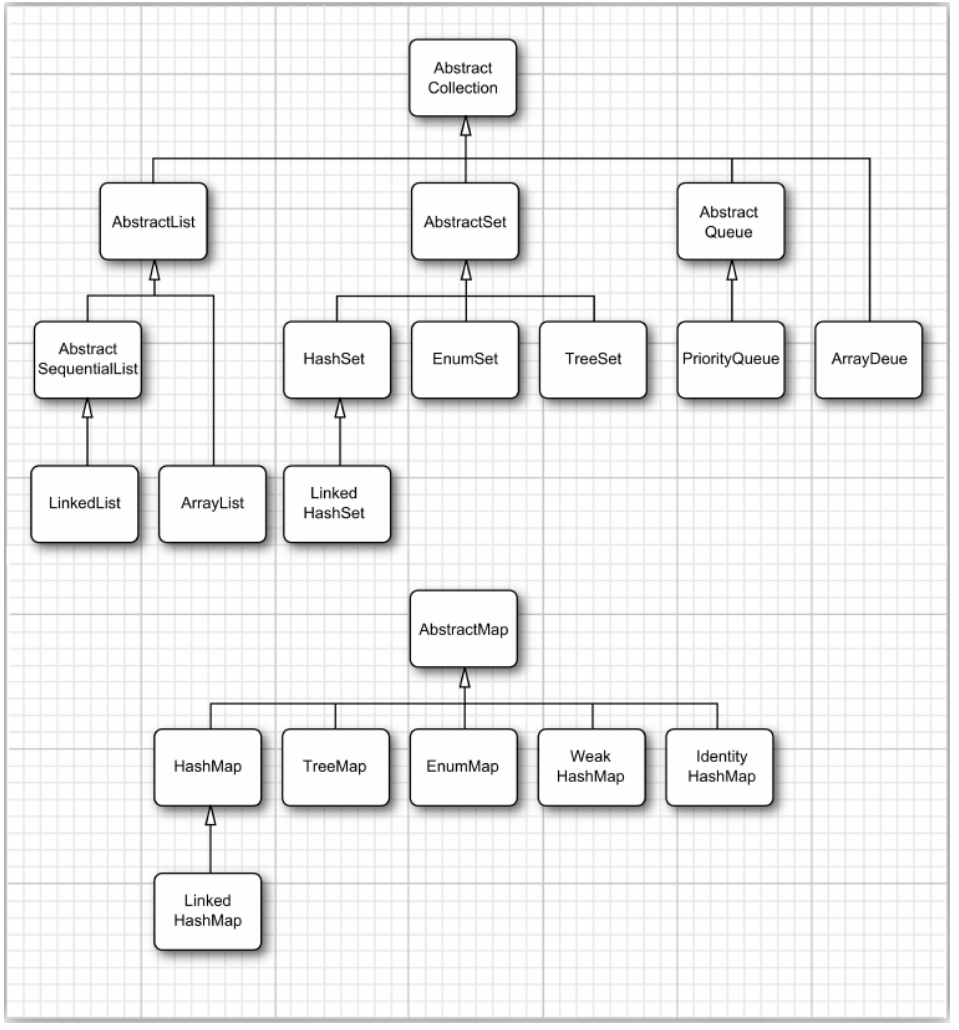
(Semester: Monsoon 2025)
Week 5 - Collections

Concrete Collections

- *Concrete generic classes, useful for various purposes.*

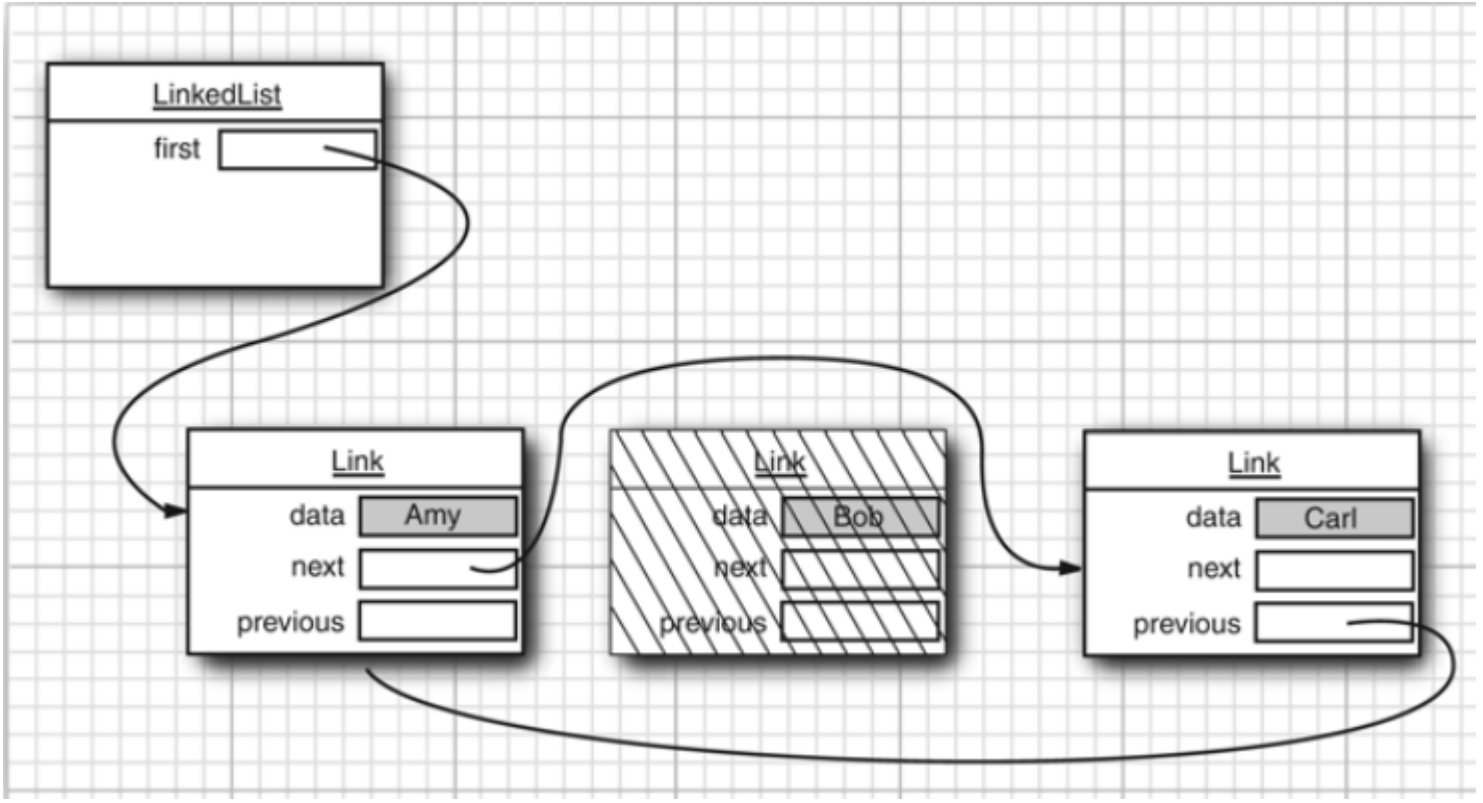
Collection Type	Description
`ArrayList`	An indexed sequence that grows and shrinks dynamically
`LinkedList`	An ordered sequence that allows efficient insertion and removal at any location
`ArrayDeque`	A double-ended queue that is implemented as a circular array
`HashSet`	An unordered collection that rejects duplicates
`TreeSet`	A sorted set
`EnumSet`	A set of enumerated type values
`LinkedHashSet`	A set that remembers the order in which elements were inserted
`PriorityQueue`	A collection that allows efficient removal of the smallest element
`HashMap`	A data structure that stores key/value associations
`TreeMap`	A map in which the keys are sorted
`EnumMap`	A map in which the keys belong to an enumerated type
`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
`IdentityHashMap`	A map with keys that are compared by ==, not `equals`

Concrete Collections



Concrete Collections

- *LinkedList <T>: In Java everything is a doubly linked list*



Concrete Collections

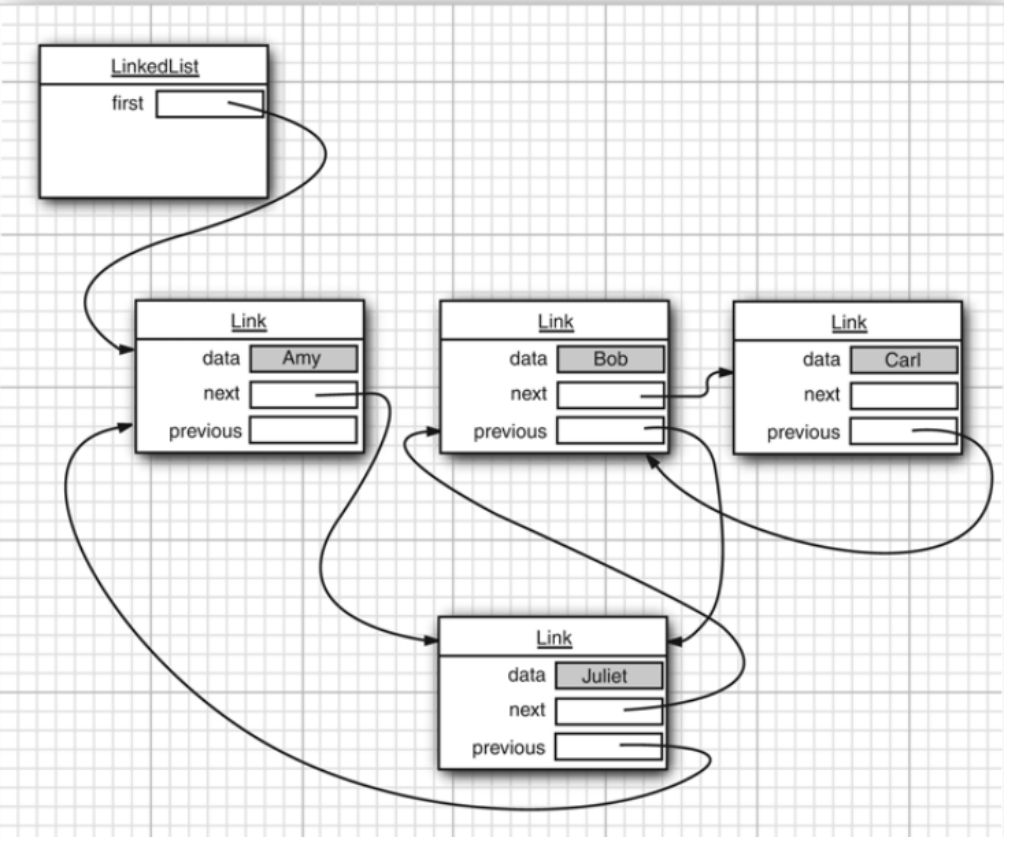
```
var staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
Iterator<String> iter = staff.iterator();
String first = iter.next(); // visit first element
String second = iter.next(); // visit second element
iter.remove(); // remove last visited element
```

```
interface ListIterator<E> extends Iterator<E>
{
   void add(E element);
   . . .
}
```

```
var 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");
```

- *LinkedList.add() always adds at the end. What if you want to add in the middle ?*

Concrete Collections



- *LinkedList.add() always adds at the end. The ListIterator<E> has the responsibility to add() anywhere.*

Concrete Collections: ArrayLists

Ordered list of indexable items; much like an generic array of variable length.

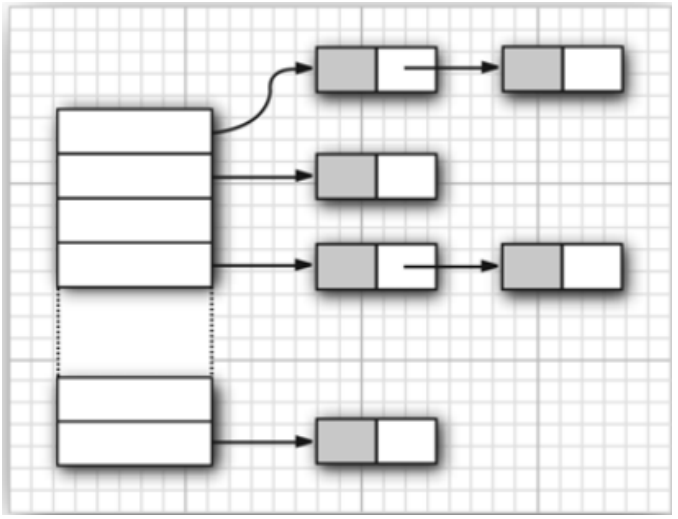
ArrayList Methods

- *The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array*
  - `Object get(int index)`
  - `Object set(int index, Object element)`
  - `May throw IndexOutOfBoundsException`
- *Indexed add and remove are provided, but can be costly if used frequently*
  - `void add(int index, Object element)`
  - `Object remove(int index)`
  - `May throw IndexOutOfBoundsException`
- *May want to resize in one shot if adding many elements*
  - `void ensureCapacity(int minCapacity)`
- *ArrayList allows adding duplicate elements*

Sets

- *Sets keep unique elements only*
  - *Like lists but no duplicates*
- *HashSet<E>*
  - *Keeps a set of elements in a hash tables*
  - *The elements are randomly ordered by their hash code*
- *TreeSet<E>*
  - *Keeps a set of elements in a red-black ordered*
  - *search tree The elements are ordered incrementally*

Concrete Collections: HashSet



Modifier and Type	Method and Description
boolean	**add**(E e) Adds the specified element to this set if it is not already present.
void	**clear**() Removes all of the elements from this set.
Object	**clone**() Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.
boolean	**contains**(Object o) Returns true if this set contains the specified element.
boolean	**isEmpty**() Returns true if this set contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present.
int	**size**() Returns the number of elements in this set (its cardinality).

Set Interface

- *Same methods as Collection*
  - *different contract – no duplicate entries*
- *Provides an Iterator to step through the elements in the Set*
  - *No guaranteed order in the basic Set interface*

HashSet

- Find and add elements very quickly
  - uses hashing
- Hashing uses an array of linked lists
  - The **hashCode()** is used to index into the array
  - Then **equals()** is used to determine if element is in the (short) list of elements at that index
- No order imposed on elements
- Behaves like a set of unique elements

HashSet

```
import java.util.HashSet;

HashSet<String> students = new HashSet<String>();

students.add("Alice");
students.add("Bob");
students.add("Charlie");
System.out.println(students);


students.contains("Bob");  //boolean outcome.

students.remove("Alice"); //boolean “could it be
                          //removed or not “

students.clear();      //clear the set

students.size();       //the size in terms of elements.
```

Concrete Collections: TreeSet

- *Similar to HashSet but all tree values are a sorted collection of*
- *generic object types. The object types must implement Comparable.*

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

String sorter tree

Concrete Collections: TreeSet

```
package treeSet;

import java.util.*;

/**
 * This program sorts a set of Item objects by comparing their descriptions.
 * @version 1.13 2018-04-10
 * @author Cay Horstmann
 */
public class TreeSetTest
{
   public static void main(String[] args)
   {
      var parts = new TreeSet<Item>();
      parts.add(new Item("Toaster", 1234));
      parts.add(new Item("Widget", 4562));
      parts.add(new Item("Modem", 9912));
      System.out.println(parts);

      var sortByDescription = new TreeSet<Item>(Comparator.comparing(Item::getDescription));

      sortByDescription.addAll(parts);
      System.out.println(sortByDescription);
   }
}
```

```
public class Item implements Comparable<Item>
{
   private String description;
   private int partNumber;

   /**
    * Constructs an item.
    * @param aDescription the item's description
    * @param aPartNumber the item's part number
    */
   public Item(String aDescription, int aPartNumber)
   {
      description = aDescription;
      partNumber = aPartNumber;
   }

   /**
    * Gets the description of this item.
    * @return the description
    */
   public String getDescription()
   {
      return description;
   }

   public String toString()
   {
      return "[description=" + description + ", partNumber=" + partNumber + "]";
   }

   public boolean equals(Object otherObject)
   {
      if (this == otherObject) return true;
      if (otherObject == null) return false;
     if (getClass() != otherObject.getClass()) return false;
     var other = (Item) otherObject;
     return Objects.equals(description, other.description) && partNumber == other.partNumber;
  }

  public int hashCode()
  {
     return Objects.hash(description, partNumber);
  }

  public int compareTo(Item other)
  {
     int diff = Integer.compare(partNumber, other.partNumber);
     return diff != 0 ? diff : description.compareTo(other.description);
  }
}
```

Concrete Collections: PQs

Similar to TreeSet. Insert in arbitrary order, remove the smallest fist - much like 'heap' data structure.

```
package priorityQueue;

import java.util.*;
import java.time.*;

/**
 * This program demonstrates the use of a priority queue.
 * @version 1.02 2015-06-20
 * @author Cay Horstmann
 */
```

```
public class PriorityQueueTest
{
   public static void main(String[] args)
   {
      var pq = new PriorityQueue<LocalDate>();
      pq.add(LocalDate.of(1906, 12, 9)); // G. Hopper
      pq.add(LocalDate.of(1815, 12, 10)); // A. Lovelace
      pq.add(LocalDate.of(1903, 12, 3)); // J. von Neumann
      pq.add(LocalDate.of(1910, 6, 22)); // K. Zuse

      System.out.println("Iterating over elements . . .");
      for (LocalDate date : pq)
         System.out.println(date);
      System.out.println("Removing elements . . .");
      while (!pq.isEmpty())
         System.out.println(pq.remove());
   }
}
```

Maps

- *Stores unique key/value pairs.*
- *Maps from the key to the value.*
- *Keys are unique.*
  - *a single key only appears once in the Map.*
  - *a key can map to only one value.*
  - *Values do not have to be unique.*

- *HashMap<K, V>: Keeps a map of elements in a hash table; elements randomly ordered*
- *TreeMap<K, V>: Keep a set of elements in a red-black ordered search tree.*

HashMap examples

```
import java.util.HashMap;

public class Main {
  public static void main(String[] args) {


    HashMap<String, Integer> namesRollNos = new HashMap<String,
Integer>();

    namesRollNos.put("Alice", new Integer(100));
    namesRollNos.put("Bob", new Integer (101));
    namesRollNos.put("Charlie", new Integer(102));
    System.out.println(namesRollNos);
    System.out.println(namesRollNos.get(“Charlie”);
  }
}
```

HashMap examples

void	**clear()** Removes all of the mappings from this map.
Object	**clone()** Returns a shallow copy of this `HashMap` instance: the keys and values themselves are not cloned.
boolean	**containsKey(Object key)** Returns `true` if this map contains a mapping for the specified key.
boolean	**containsValue(Object value)** Returns `true` if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	**entrySet()** Returns a **Set** view of the mappings contained in this map.
V	**get(Object key)** Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
boolean	**isEmpty()** Returns `true` if this map contains no key-value mappings.
Set<K>	**keySet()** Returns a **Set** view of the keys contained in this map.
V	**put(K key, V value)** Associates the specified value with the specified key in this map.
void	**putAll(Map<? extends K,? extends V> m)** Copies all of the mappings from the specified map to this map.
V	**remove(Object key)** Removes the mapping for the specified key from this map if present.
int	**size()** Returns the number of key-value mappings in this map.
Collection<V>	**values()** Returns a **Collection** view of the values contained in this map.

TreeMap

public class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap <K,V>, Cloneable, Serializable

RB Tree sorted according to the natural ordering of the keys or by the Comparator provided at the map creation time.

log(n) time insertion and searching

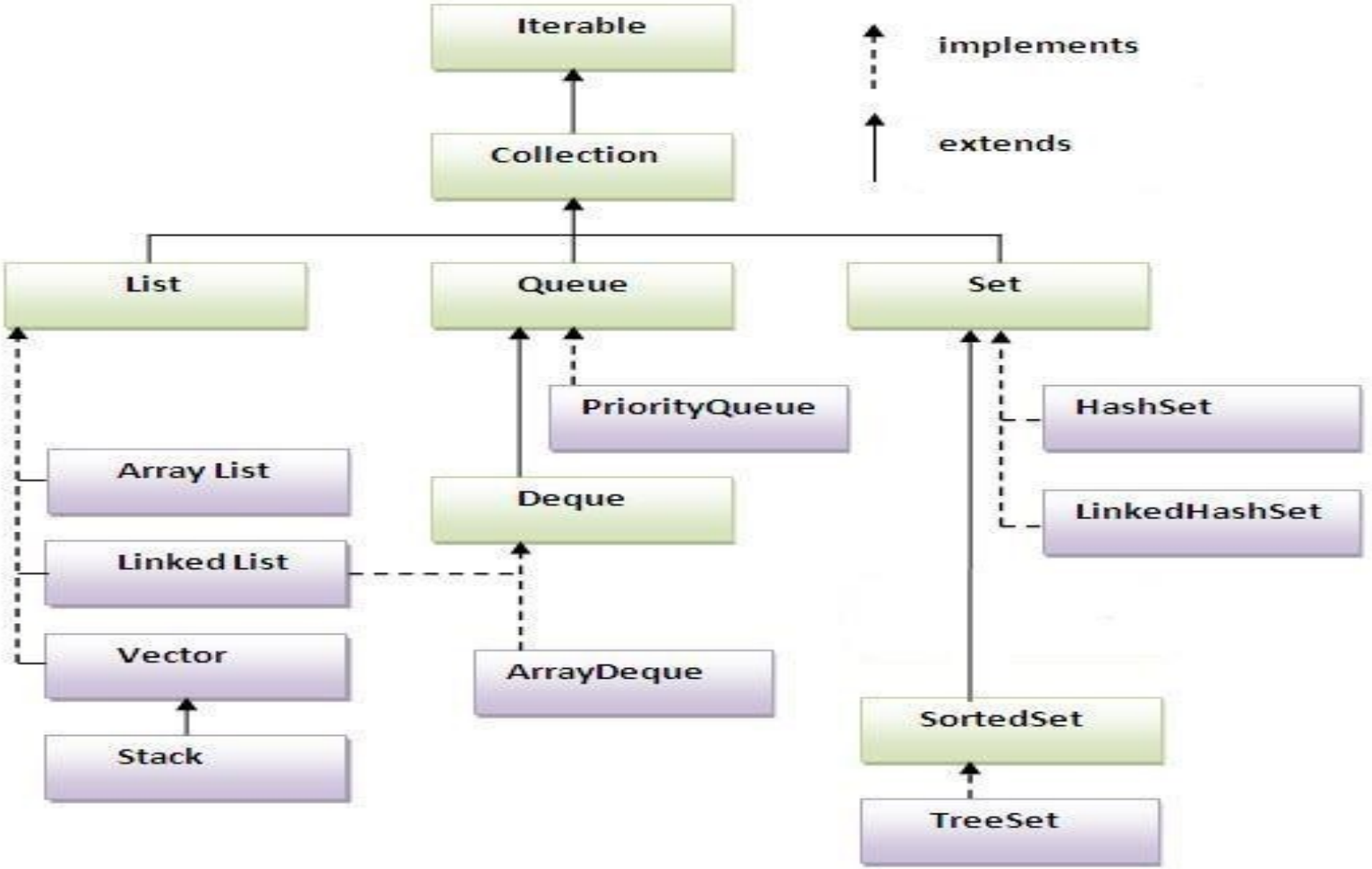
TreeMap Example

```
import java.util.TreeMap;

public class Main {
  public static void main(String[] args) {


    TreemapMap<String, Integer> namesRollNos = new TreeMap<>();

    namesRollNos.put("Charlie", new Integer(102));
    namesRollNos.put("Alice", new Integer(100));
    namesRollNos.put("Bob", new Integer (101));

    System.out.println(namesRollNos);

  }
}
```

Output:

```
{Alice=100, Bob=101, Charlie=102}
```

Collection Hierarchy

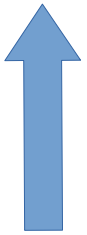


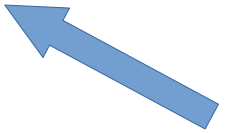
Collection Interfaces

- *Interfaces for building complex data structures.*
- *Minimal number of own method declarations.*
- *Operations like add(), remove(), iterator() [To traverse complex data structures].*
- *Can be used for any form of complex data structures.*

Collection and Interfaces

```
public interface Queue<E> // a simplified form of the interface in the standard library
{
   void add(E element);
   E remove();
   int size();
}
```





```
public class LinkedListQueue<E> implements Queue<E>
{
   private Link head;
   private Link tail;

   LinkedListQueue() { . . . }
   public void add(E element) { . . . }
   public E remove() { . . . }
   public int size() { . . . }
}
```

```
public class CircularArrayQueue<E> implements Queue<E>
{
   private int head;
   private int tail;

   CircularArrayQueue(int capacity) { . . . }
   public void add(E element) { . . . }
   public E remove() { . . . }
   public int size() { . . . }
   private E[] elements;
}
```

```
Queue<Customer> expressLane = new CircularArrayQueue<>(100);
expressLane.add(new Customer("Harry"));
```

```
Queue<Customer> expressLane = new LinkedListQueue<>();
expressLane.add(new Customer("Harry"));
```

Collection

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

Iterators

```
public interface Iterator<E>
{
   E next();
   boolean hasNext();
   void remove();
   default void forEachRemaining(Consumer<? super E> action);
}
```

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

Iterators - forEach()/forEachRemaining - Lambda functions

```
iterator.forEachRemaining(element -> do something with element);
```

- *General Lambda functions/expressions:*
- *Single-line code snippets that can be used as without a function name - only behaviour.*

```
class LengthComparator implements Comparator<String>
{
   public int compare(String first, String second)
   {
      return first.length() - second.length();
   }
}
```

```
(String first, String second) ->
   {
      if (first.length() < second.length()) return -1;
      else if (first.length() > second.length()) return 1;
      else return 0;
   }
```

More on Lambda functions/expressions.

```
() -> { for (int i = 100; i >= 0; i--) System.out.println(i); }
```

- *Parameterless lambda function*

```
ActionListener listener = event ->
   System.out.println("The time is "
      + Instant.ofEpochMilli(event.getWhen()));
      // instead of (event) -> . . . or (ActionEvent event) -> . . .
```

- *Single parameter lambda function*

More on Lambda functions/expressions.

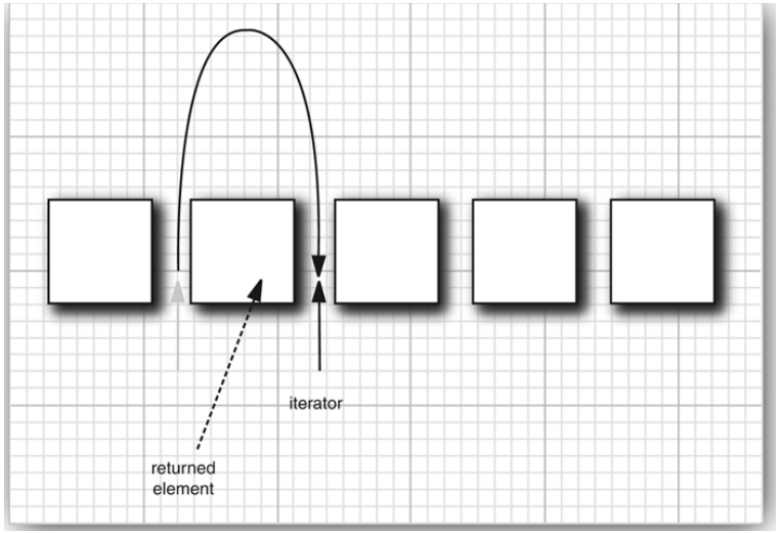
```
1  package lambda;
2
3  import java.util.*;
4
5  import javax.swing.*;
6  import javax.swing.Timer;
7
8  /**
9   * This program demonstrates the use of lambda expressions.
10  * @version 1.0 2015-05-12
11  * @author Cay Horstmann
12  */
13 public class LambdaTest
14 {
15    public static void main(String[] args)
16    {
17       var planets = new String[] { "Mercury", "Venus", "Earth", "Mars",
18          "Jupiter", "Saturn", "Uranus", "Neptune" };
19       System.out.println(Arrays.toString(planets));
20       System.out.println("Sorted in dictionary order:");
21       Arrays.sort(planets);
22       System.out.println(Arrays.toString(planets));
23       System.out.println("Sorted by length:");
24       Arrays.sort(planets, (first, second) -> first.length() - second.length());
25       System.out.println(Arrays.toString(planets));
26
27       var timer = new Timer(1000, event ->
28          System.out.println("The time is " + new Date()));
29       timer.start();
30
31       // keep program running until user selects "OK"
32       JOptionPane.showMessageDialog(null, "Quit program?");
33       System.exit(0);
34    }
35 }
```

Back to Iterators

- *Overall semantics - Iterators are "in-between" elements.*

- *Iterator.jump() -- Jump past the next element and return the value that you jumped passed.*

- *Iterator.remove() -- Remove the element that you just jumped passed.*




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

```
it.remove();
it.remove(); // ERROR
```

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

Remove consecutive elements.