Java Collections Framework (JCF)

Università di Modena e Reggio Emilia
Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)

ING

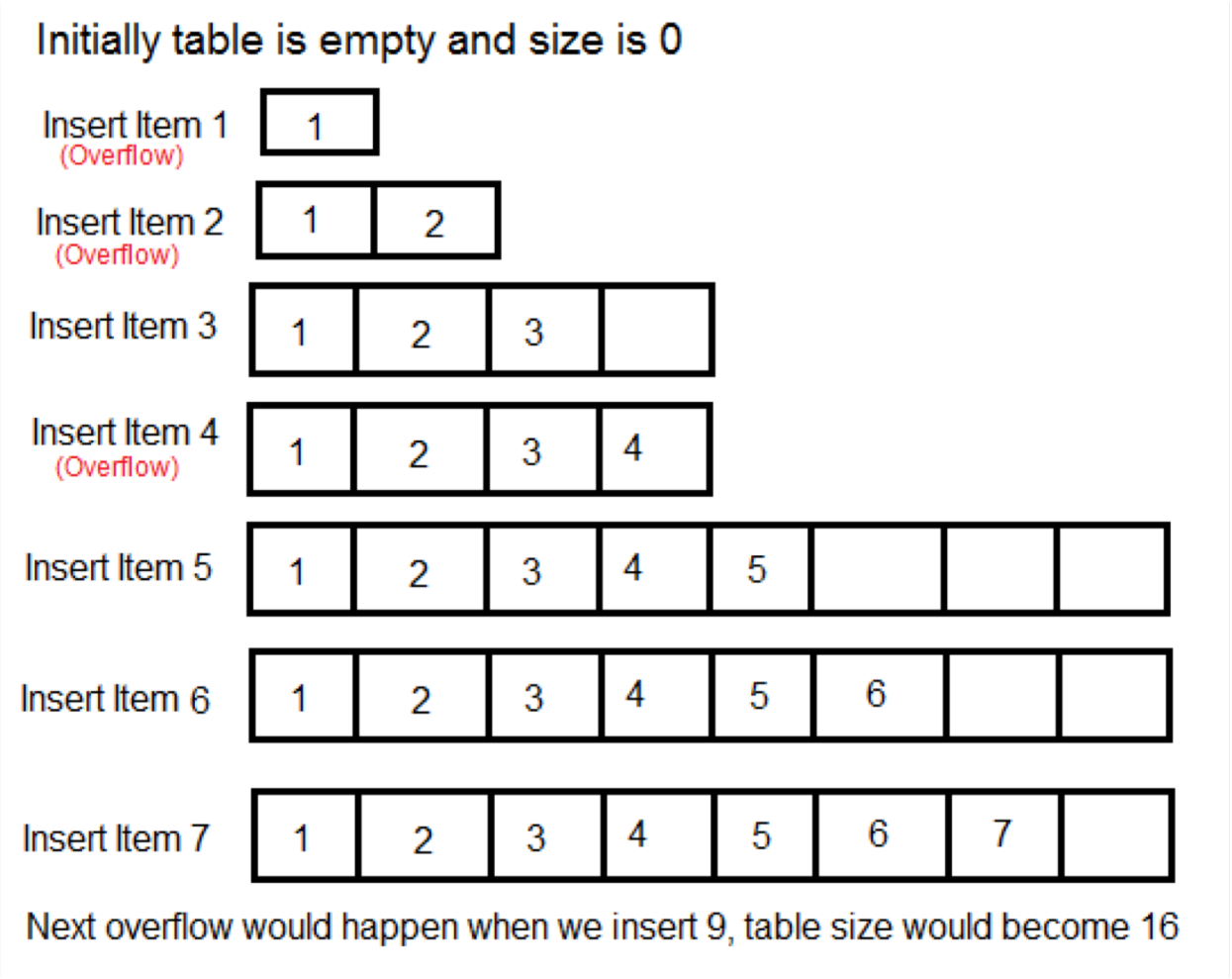
Framework

- The Java Collection Framework (JCF) is a set of classes and interfaces implementing commonly reusable data structures.
- The JCF (package java.util) provides
  - interfaces defining functionalities
  - abstract classes for shared code aggregation
  - concrete classes implementing functionalities
  - algorithms (java.util.Collections)

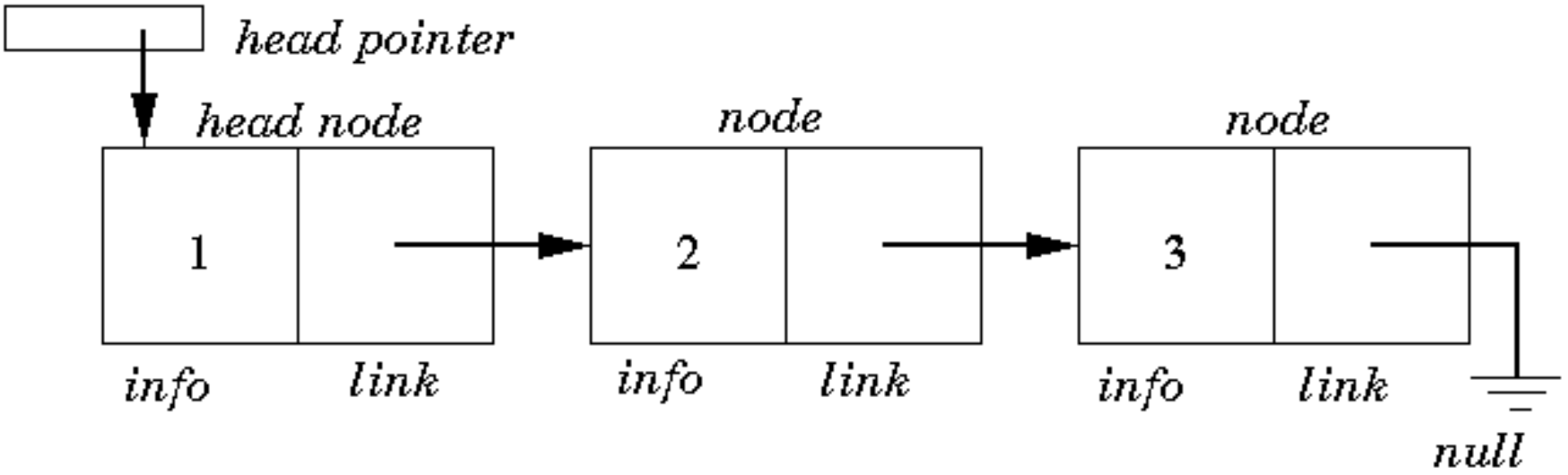
Key Concepts

- Resizable Array
- Linked List
- Balanced Tree
- Hash Table

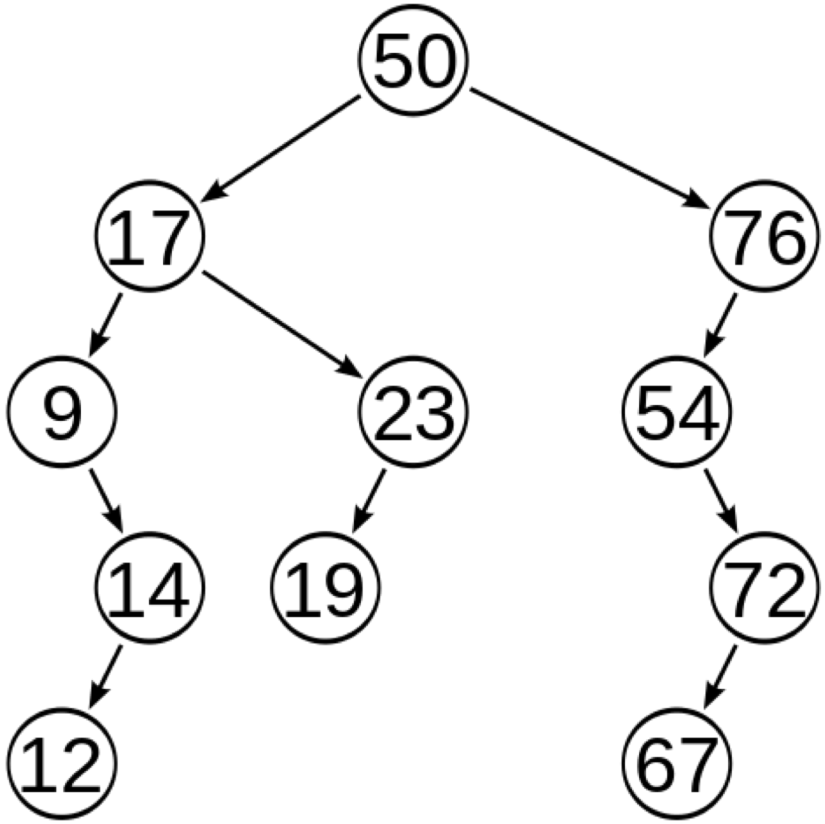
Resizable Array ~O(n)

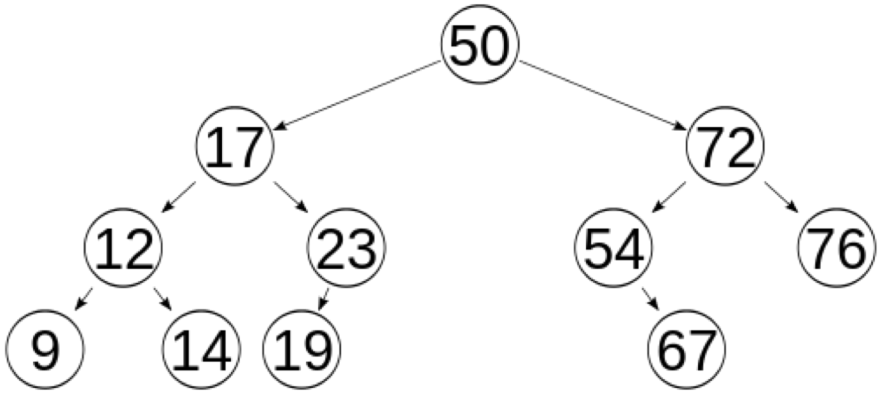


Linked List ~O(n)



Balanced Tree ~O(log(n))

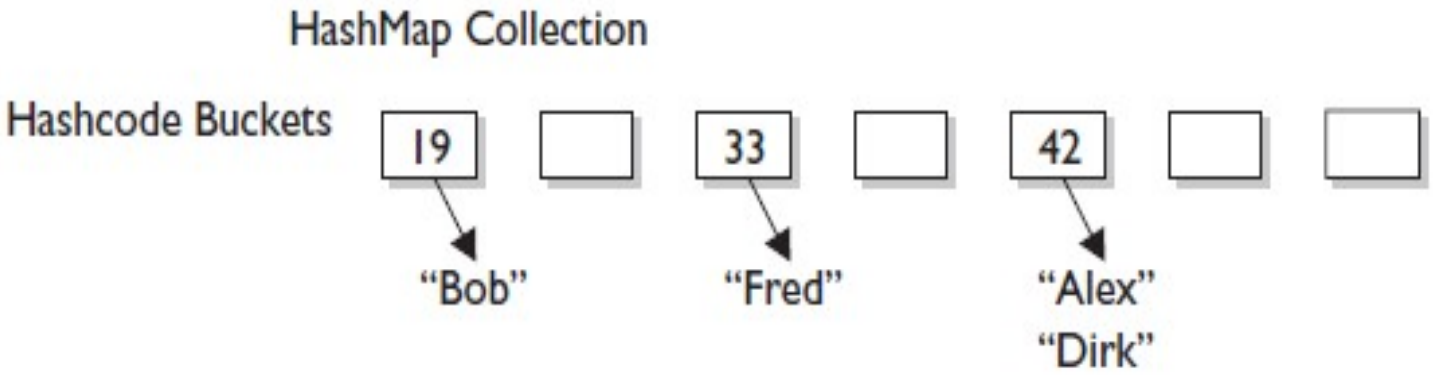




** A binary tree is balanced if, for each node it holds that, the number of inner nodes in the left subtree and the number of inner nodes in the right subtree differ by at most 1. A binary tree is balanced if for any two leaves the difference of the depth is at most 1.*

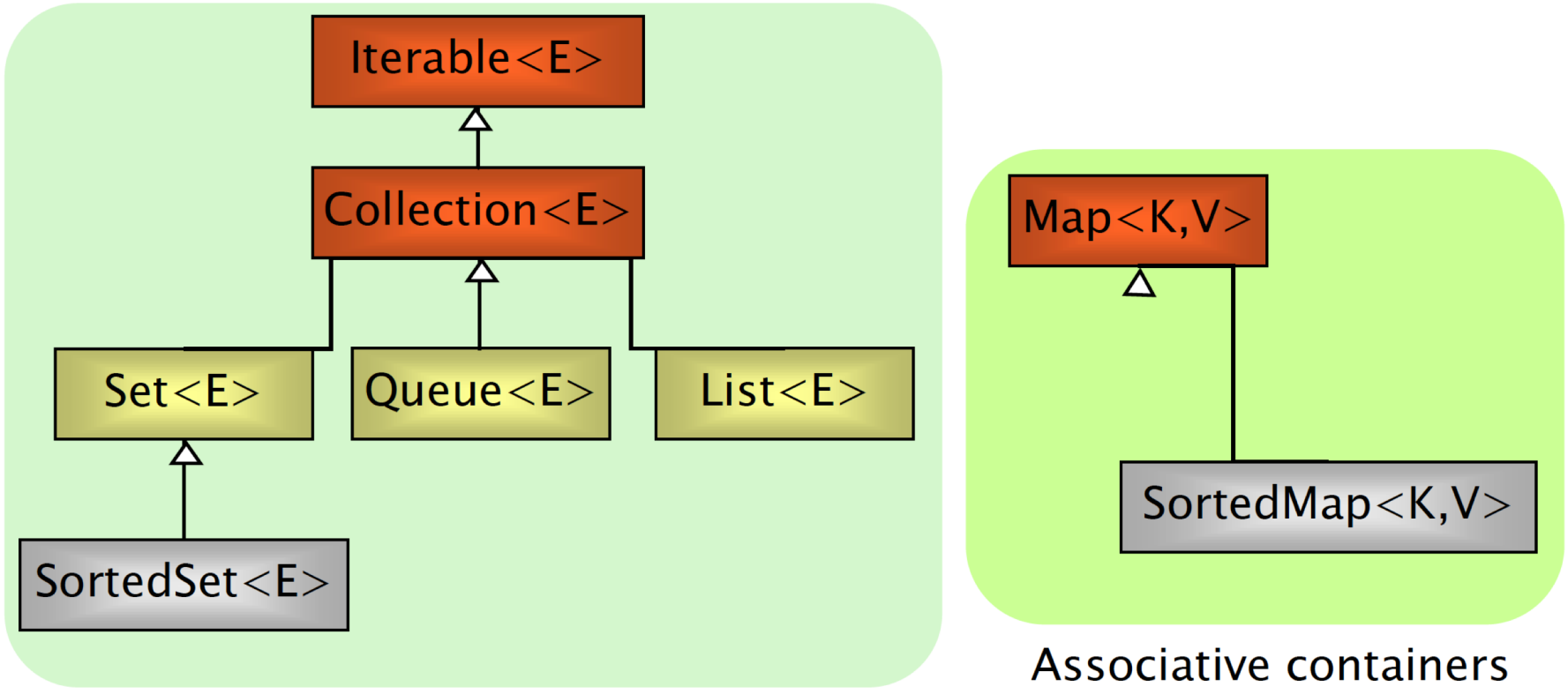
Hash Table ~O(1)

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

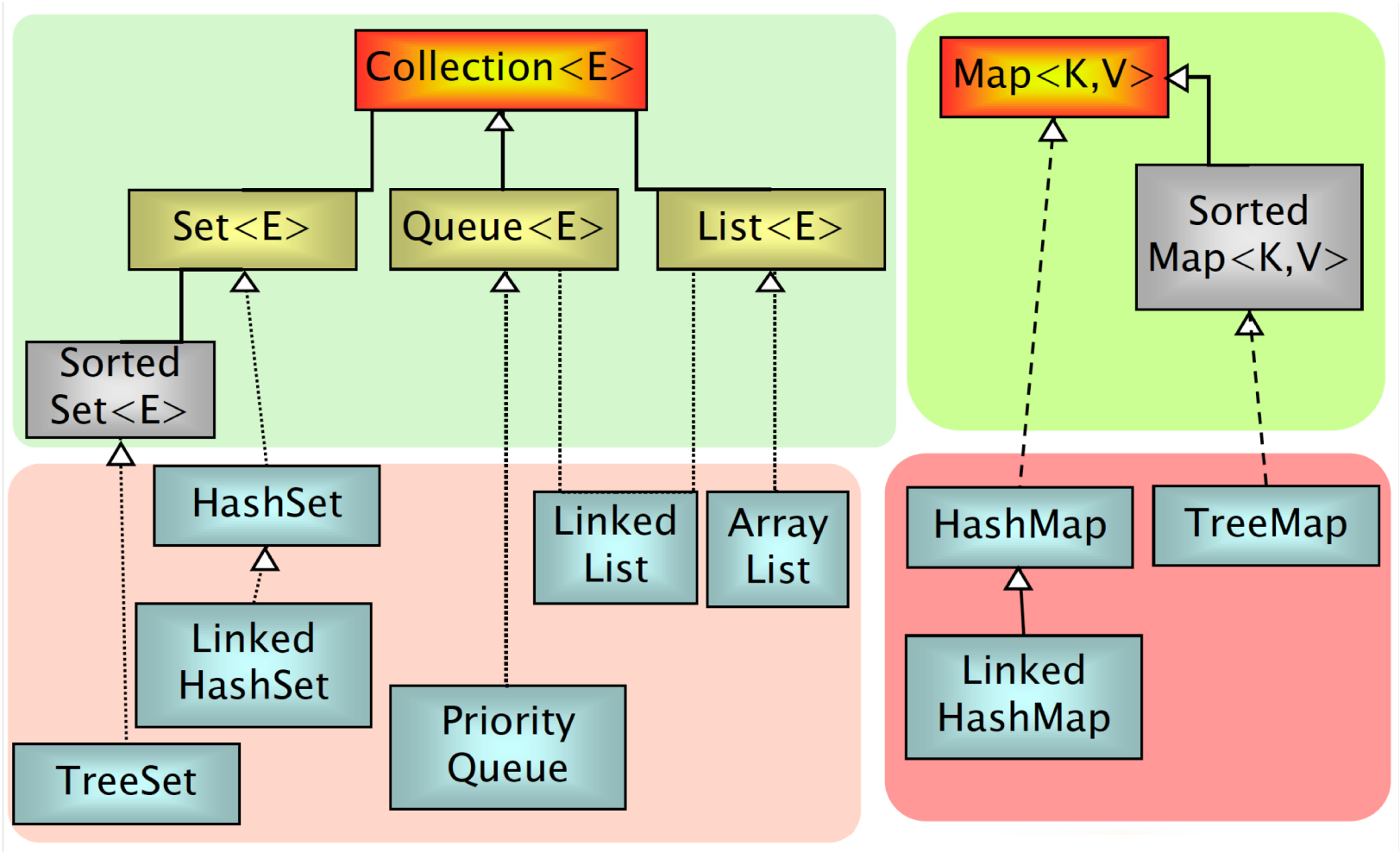


ING

Interfaces

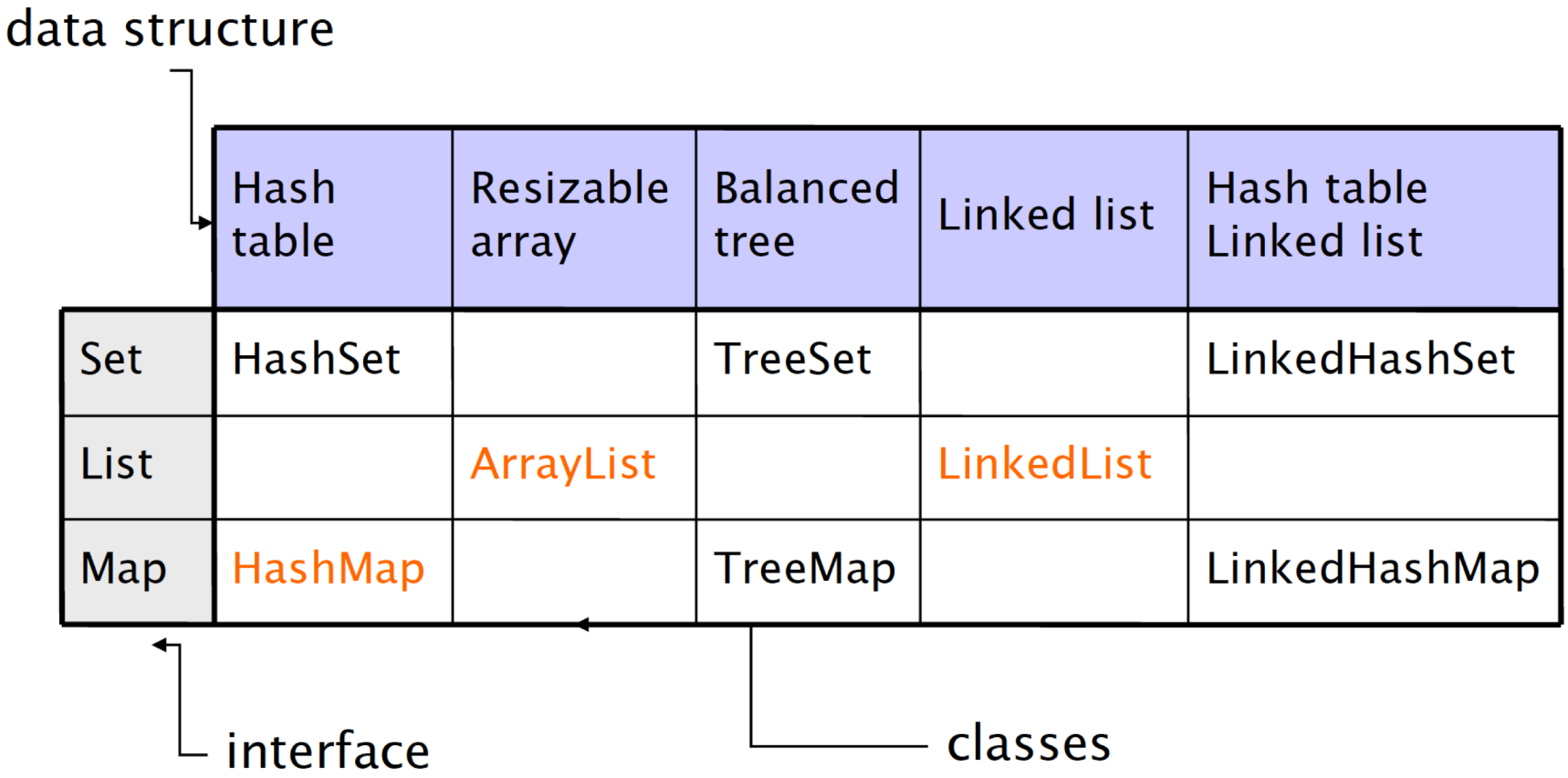


Implementations



ING

Internals


data structure
Hash table
Resizable array
Balanced tree
Linked list
Hash table Linked list
Set
HashSet
TreeSet
LinkedHashSet
List
ArrayList
LinkedList
Map
HashMap
TreeMap
LinkedHashMap
interface
classes

Iterable Interface

```
public interface Iterable<T> {
  public Iterator<T> iterator();
}

ArrayList<Object> l = new ArrayList<Object>();
for(Object o : l){
    //do something;
}
```

The Iterable interface (java.lang.Iterable) is the root interface of the Java collection framework. Iterable, literally, means that “can be iterated”. Technically, it means that an Iterator can be returned. Iterable objects (objects implementing the iterable interface) can be used with the for-each loop

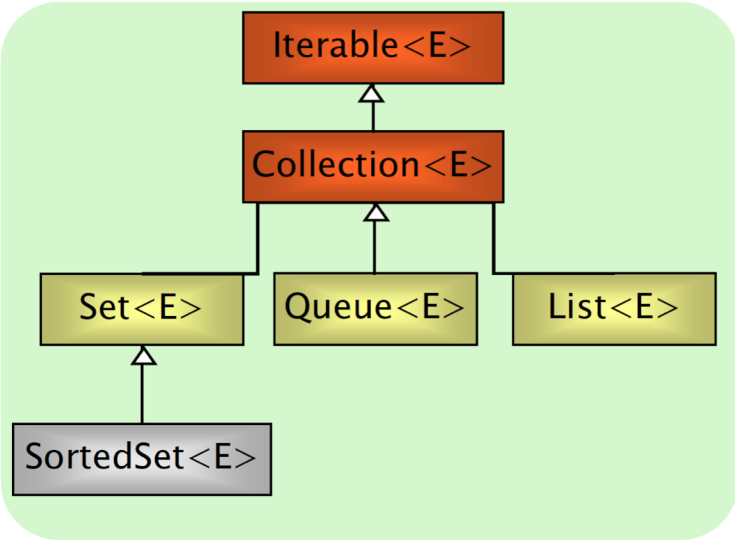
Iterator Interface

- boolean hasNext()
- object next()
- void remove()

```
ArrayList<Object> l = new ArrayList<Object>();
for (Iterator<Object> i = l.iterator(); i.hasNext();) {
    Object o = i.next();
    // do something
}
```

Collection Interface

- Group of elements (references to objects)
- It is not specified whether they are
  - Ordered / not ordered
  - Duplicated / not duplicated
- Common constructors
  - Collection()
  - Collection(Collection c)



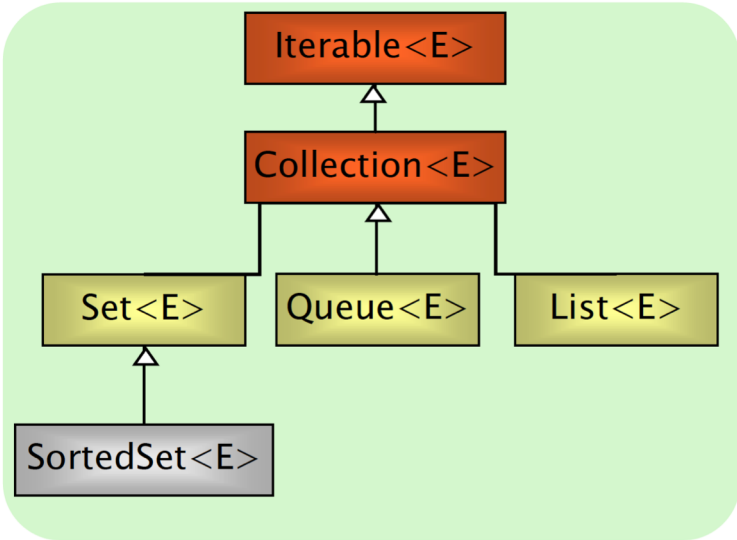
ING

Collection Interface

- `int size()`
- `boolean isEmpty()`
- `boolean contains(Object element)`
- `boolean containsAll(Collection c)`
- `boolean add(Object element)`
- `boolean addAll(Collection c)`
- `boolean remove(Object element)`
- `boolean removeAll(Collection c)`
- `void clear()`
- `Object[] toArray()`
- `Iterator iterator()`

List Interface

- Can contain duplicate elements
- Insertion order is preserved
- User can select arbitrary insertion points
- Elements can be accessed by position



ING

List additional methods

- `Object get(int index)`
- `Object set(int index, Object o)`
- `Object remove(int index)`
- `void add(int index, Object o)`
- `boolean addAll(int index, Collection c)`
- `int indexOf(Object o)`
- `int lastIndexOf(Object o)`
- `List subList(int fromIndex, int toIndex)`

List Initialization

```
/* plain, simple, long */
List<Integer> l = new ArrayList<Integer>();
l.add(14);
l.add(73);
l.add(18);
…

/* more compact version */
List<Integer> l = new
ArrayList<Integer>(Arrays.asList(14, 73, 18));
```

List Implementations

```
List<Car> garage =
new ArrayList<Car>();

garage.add(new Car());
garage.add(new SDCar());
garage.add(new SDCar());
garage.add(new Car());

for(Car c : garage) {
     c.turnOn();
}

/* Decoupling references from actual
objects allows to change implementation
(and related performance!) by changing
a single line of code! */
```

```
List<Car> garage =
new LinkedList<Car>();

garage.add(new Car());
garage.add(new SDCar());
garage.add(new SDCar());
garage.add(new Car());

for(Car c : garage) {
     c.turnOn();
}
```

List Implementations

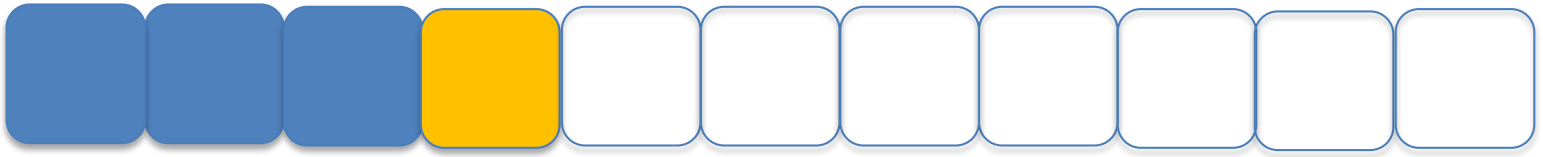
- ArrayList implements List
  - get(index) -> Constant time
  - add(index, obj) -> Linear time

- LinkedList implements List, Queue
  - get(index) -> Linear time
  - add(index, obj) -> Linear time (but more lightweight)

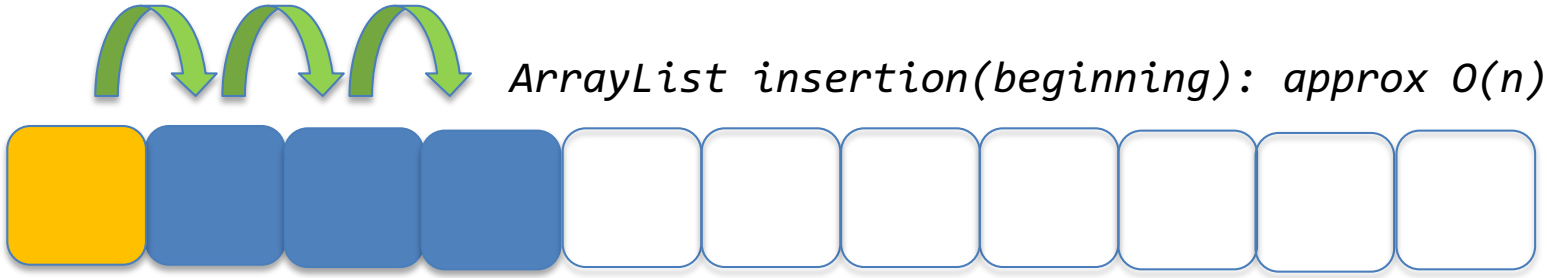
List Implementations

ArrayList retrieval: approx O(1)

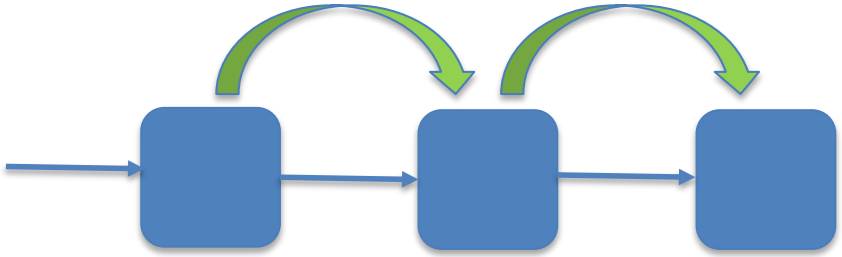
ArrayList insertion(end): approx O(1)



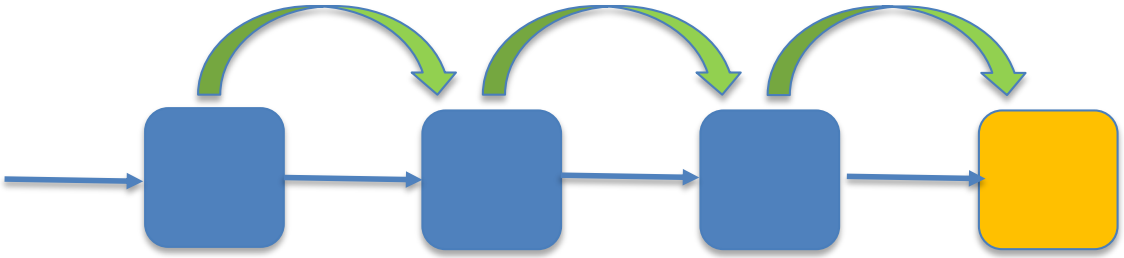
ArrayList insertion(beginning): approx O(n)



List Implementations



LinkedList retrieval: approx O(n)

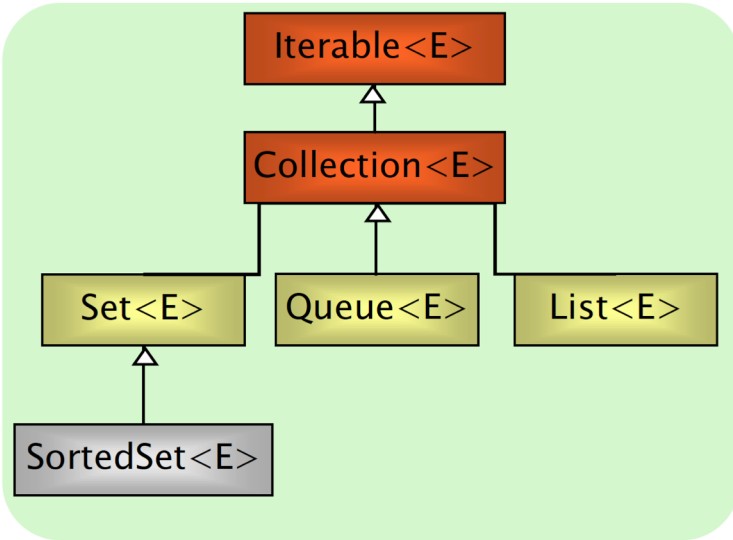


LinkedList insertion(end): approx O(n)

LinkedList insertion(beginning): approx O(1)

Set Interface

- Contains no methods other than those inherited from Collection
- No duplicate elements are allowed



ING

Set Implementations

- HashSet implements Set
  - Hash tables as internal data structure (fast!)
  - Insertion order not preserved
- LinkedHashSet extends HashSet
  - Insertion order preserved
- TreeSet implements SortedSet (an extension of Set)
  - R-B trees as internal data structure
  - User definable internal ordering
  - Slow when compared to hash-based implementations

TreeSet Internal Ordering

- Depending on the constructor used, SortedSet implementations can use different orderings
- TreeSet()
  - Natural ascending ordering
  - Elements must implement the Comparable Interface
- TreeSet(Comparator c)
  - Ordering is defined by the Comparator c

HashSet Example

```
ArrayList<String> l = new ArrayList<String>(
  Arrays.asList(“Nicola”, “Agata”, “Marzia”, “Agata”));
Set<String> hs = new HashSet<String>(l);

System.out.println(l);
[Nicola, Agata, Marzia, Agata]

System.out.println(hs);
[Marzia, Nicola, Agata]
```

LinkedHashSet Example

```
ArrayList<String> l = new ArrayList<String>(
  Arrays.asList(“Nicola”, “Agata”, “Marzia”, “Agata”));
Set<String> lhs = new LinkedHashSet<String>(l);

System.out.println(l);
[Nicola, Agata, Marzia, Agata]

System.out.println(lhs);
[Nicola, Agata, Marzia]
```

TreeSet Example

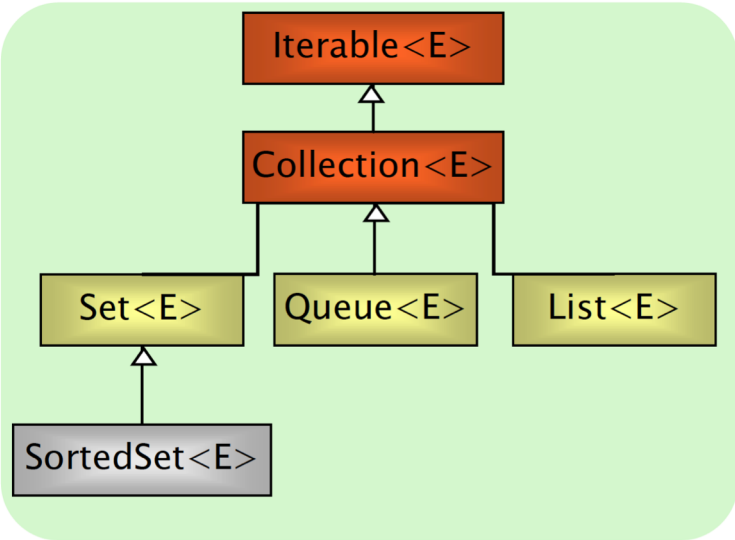
```
ArrayList<String> l = new ArrayList<String>(
  Arrays.asList(“Nicola”, “Agata”, “Marzia”, “Agata”));
Set<String> ts = new TreeSet<String>(l);

System.out.println(l);
[Nicola, Agata, Marzia, Agata]

System.out.println(ts);
[Agata, Marzia, Nicola]
```

Queue Interface

- A collection designed for holding elements prior to processing
- Provides additional insertion, extraction, and inspection operations. It also defines a head (first element) and a tail (last element)



ING

Queue additional methods

- `boolean add(Object o)`

```
/* not throwing exception on error */
```

- `Object peek()`
- `Object poll()`

```
/* throwing exception on error */
```

- `Object element()`
- `Object remove()`

Queue Implementations

- LinkedList implements List, Queue
  - Insertion order conserved
  - Head is the first element of the list
  - FIFO (Fist-In-First-Out) policy
- PriorityQueue implements Queue
  - Internal ordering policy. Default is natural ascending ordering, if defined. Can be modified by implementing the Comparable interface

Queue Example

```
ArrayList<Integer> l = new ArrayList<Integer>(
  Arrays.asList(3, 1, 2));
Queue<Integer> fifo = new LinkedList<Integer>(l);
Queue<Integer> pqueue = new PriorityQueue<Integer>(l);

System.out.println(fifo.peek());     // 3
System.out.println(pqueue.peek());   // 1
```

PriorityQueue or TreeSet?!?

- **Similarities**
  - Both provide O(log(N)) time complexity for adding, removing, and searching elements
  - Both provide elements in sorted order

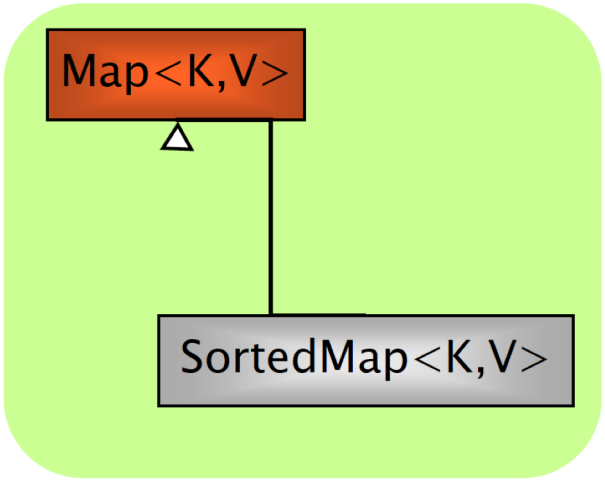
- **Differences**
  - TreeSet is a Set and doesn't allow a duplicate element, while PriorityQueue is a queue and doesn't have such restriction.
  - Another key difference between TreeSet and PriorityQueue is *iteration order*, though you can access elements from the head in a sorted order e.g. head always give you lowest or highest priority element depending upon your Comparable or Comparator implementation but iterator returned by PriorityQueue doesn't provide any ordering guarantee.

Map Interface

- An object storing pairs of (key, value)

(e.g., key: surname, value: phone number)

  - Keys and values must be objects
  - Keys must be unique
- Common constructors:
  - Map()
  - Map(Map m)



ING

Map Interface

- `Object put(Object key, Object value)`
- `Object get(Object key)`
- `Object remove(Object key)`
- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`
- `public Set keySet()`
- `public Collection values()`
- `int size()`
- `boolean isEmpty()`
- `void clear()`

Map Implementations

- HashMap implements Map
  - Hash tables as internal data structure (fast!)
  - Insertion order not preserved
- LinkedHashMap extends HashMap
  - Insertion order preserved
- TreeMap implements SortedMap
  - R-B trees as internal data structure
  - User definable internal ordering
  - Slow when compared to hash-based implementations

* Similar to Set implementations

HashMap

- Get/set takes constant time (without considering collisions)
- Automatic re-allocation when load factor reached
- Constructor optional arguments
  - load factor(default = .75)
  - initial capacity(default = 16)

Map Example I

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

m.put(“Agata”,  2);
m.put(“Marzia”, 3);
m.put(“Agata”,  4);
m.put(“Nicola”, 1);

System.out.println(m);
{Agata=4, Nicola=1, Marzia=3}
```

Map Example II

```
Map<String, Integer> m = new HashMap<String, Integer>();
…
// looping keys and accessing values
List<String> keys = m.keySet();
for(String key : keys) {
    System.out.println(key + " -> " + m.get(key));
}

// contains key
if (m.containsKey(key)) {
    System.out.println(m.get(key));
}
```

Collections and Iterators

It is unsafe to modify (add or remove elements) a Collection while iterating over it!

```
List<Double> l = new LinkedList<Double>(
   Arrays.asList(10.8, 11.1, 13.2, 30.2));

int count = 0;
for (double i : l) {
   if (count == 1) l.remove(count);
   if (count == 2) l.add(22.3);
   count++;
} // Wrong! We modify the list while iterating
```

Collections and Iterators

- Interface Iterator provides a transparent means for cycling through all elements of a Collection (forward only) and removing elements
- Interface ListIterator provides a transparent means for cycling through all elements of a Collection (forward and backward) and removing and adding elements

Iterator Interface

- `boolean hasNext()`
- `Object next()`
- `void remove()`

ListIterator Interface

- boolean hasNext()
- boolean hasPrevious()
- object next()
- object previous()
- void add()
- void set()
- void remove()
- int nextIndex()
- int previousIndex()

Iterator Example

```
List<Double> l = new LinkedList<Double>(
    Arrays.asList(10.8, 11.1, 13.2, 30.2));

int count = 0;
for (Iterator<Double> i = l.iterator(); i.hasNext();) {
    double d = i.next();
    if (count == 1) i.remove();
    count++;
}
```

ListIterator Example

```
List<Double> l = new LinkedList<Double>(
    Arrays.asList(10.8, 11.1, 13.2, 30.2));

int count = 0;
for (ListIterator<Double> i = l.listIterator(); i.hasNext();) {
    double d = i.next();
    if (count == 1) i.remove();
    if (count == 2) i.add(22.3);
    count++;
}
```

for() and Iterators

```
List<Person> pl = new ArrayList<Person>();

/* C style */
for (int i = 0; i < pl.size(); i++)
     System.out.println(pl.get(i))

/* Java style */
for (Person p : pl)
     System.out.println(p);

/* Iterator style */
for(Iterator<Person> i = pl.iterator(); i.hasNext();) {
     Person p = i.next();
     System.out.println(p);
}

/* While style */
Iterator i = pl.iterator();
while (i.hasNext())
     System.out.println((Person)i.next());
```

Algorithms

- java.util.Collections provide frequently used utilities for manipulating collections (lists, sets, queues)
- java.util.Arrays provide frequently used utilities for manipulating arrays
- Both of them contain only static methods!

- `sort()` - merge sort implementation, n log(n)
- `binarySearch()` - requires ordered collection
- `shuffle()` - unsort
- `reverse()` - requires ordered collection
- `rotate()` – rotate elements of a given distance
- `min()`, `max()` - in a collection

Algorithms

```
ArrayList<String> l = new ArrayList<String>(
  Arrays.asList("Nicola", "Agata", "Marzia", "Agata"));

Collections.sort(l);
System.out.println(l);     // [Agata, Agata, Marzia, Nicola]

Collections.reverse(l);
System.out.println(l);     // [Nicola, Marzia, Agata, Agata]

Collections.shuffle(l);
System.out.println(l);     // [Marzia, Agata, Agata, Nicola]

Collections.rotate(l, 1);
System.out.println(l);     // [Nicola, Marzia, Agata, Agata]
```

Algorithms

```
ArrayList<String> l = new ArrayList<String>(
  Arrays.asList("Nicola", "Agata", "Marzia", "Agata"));

Collections.sort(l);
System.out.println(l);   // [Agata, Agata, Marzia, Nicola]

Collections.binarySearch(l, "Nicola");  // 3
Collections.binarySearch(l, "Zuck"));   // -5
```

The list must be sorted into ascending order according to the natural ordering of its elements (as by the sort(List) method) prior to making this call. If it is not sorted, the results are undefined.

The Comparable Interface

How can we order collections of generic objects according to a policy we define?

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

public interface Comparator<T> {
  public int compare(T obj1, T obj2);
}
```

The Comparable Interface

The Comparable interface must be implemented for making objects comparable to each other. Thus, a generic collection of T can be sorted if T implements Comparable. compareTo() compares the object with the object passed as a parameter. Return value must be:

< 0 if ***this object*** precedes ***obj***

== 0 if ***this object*** has the same position as ***obj***

> 0 if ***this object*** follows ***obj***

The Comparable Interface

- The Comparable Interface is implemented for language common types in packages java.lang and java.util. For example:
  - String objects are lexicographically ordered
  - Date objects are chronologically ordered
  - Number and sub-classes are ordered numerically

- Let's implement Comparable in the following class

```
class Person {
  protected String name;
  protected String lastname;
  protected int age;
  …
}
```

The Comparable Interface

```
class Person implements Comparable<Person> {
  protected String name;
  protected String lastname;
  protected int age;

  public int compareTo(Person p) {
    // order by surname
    cmp = lastname.compareTo(p.lastname);
    if(cmp == 0)
        // if equal surnames, order by name
        cmp = firstname.compareTo(s.firstname);
    return cmp;
}
```

The Comparator Interface

Given a class already implementing Comparable<E>, we can sort it using alternative orders using a Comparator<E>

```
public class SortByAge implements
Comparator<Person> {
  @Override
  public int compare(Person o1, Person o2) {
    return o1.age - o2.age;
  }
}
class Person implements Comparable<Person> {
  protected String name;
  protected String lastname;
  protected int age;

  public int compareTo(Person p) {
    return lastname.compareTo(p.lastname);
  }
}
```

```
public static void main(String[] args) {
  ArrayList<Person> l = new ArrayList<Person>();
  l.add(new Person("Mario", "Rossi", 68));
  l.add(new Person("Luca", "Bianchi", 28));
  l.add(new Person("Carlo", "Antoni", 34));

  // natural ordering (Comparable)
  Collections.sort(l)

  // special ordering (Comparator)
  Collections.sort(l, new SortByAge());

  // Comparator concise form
  Collections.sort(l, new Comparator<Person>() {
  @Override
    public int compare(Person o1, Person o2) {
      return o1.age - o2.age;
    }
  });
}
```