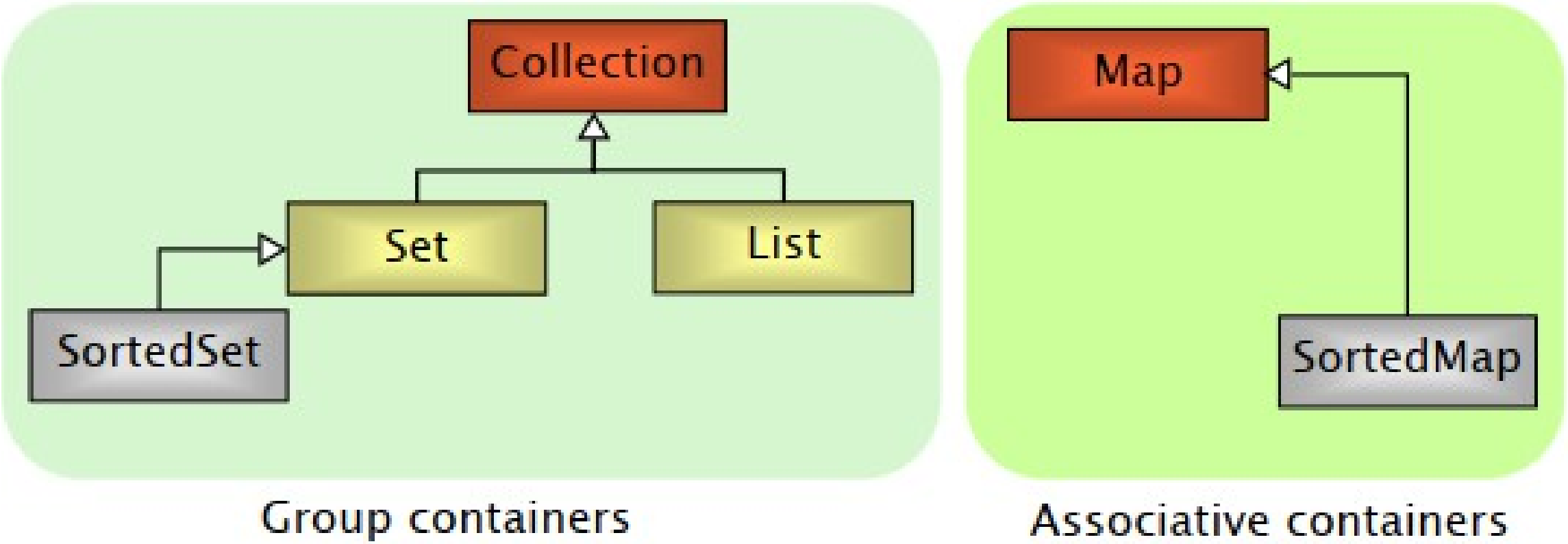
Java Collection

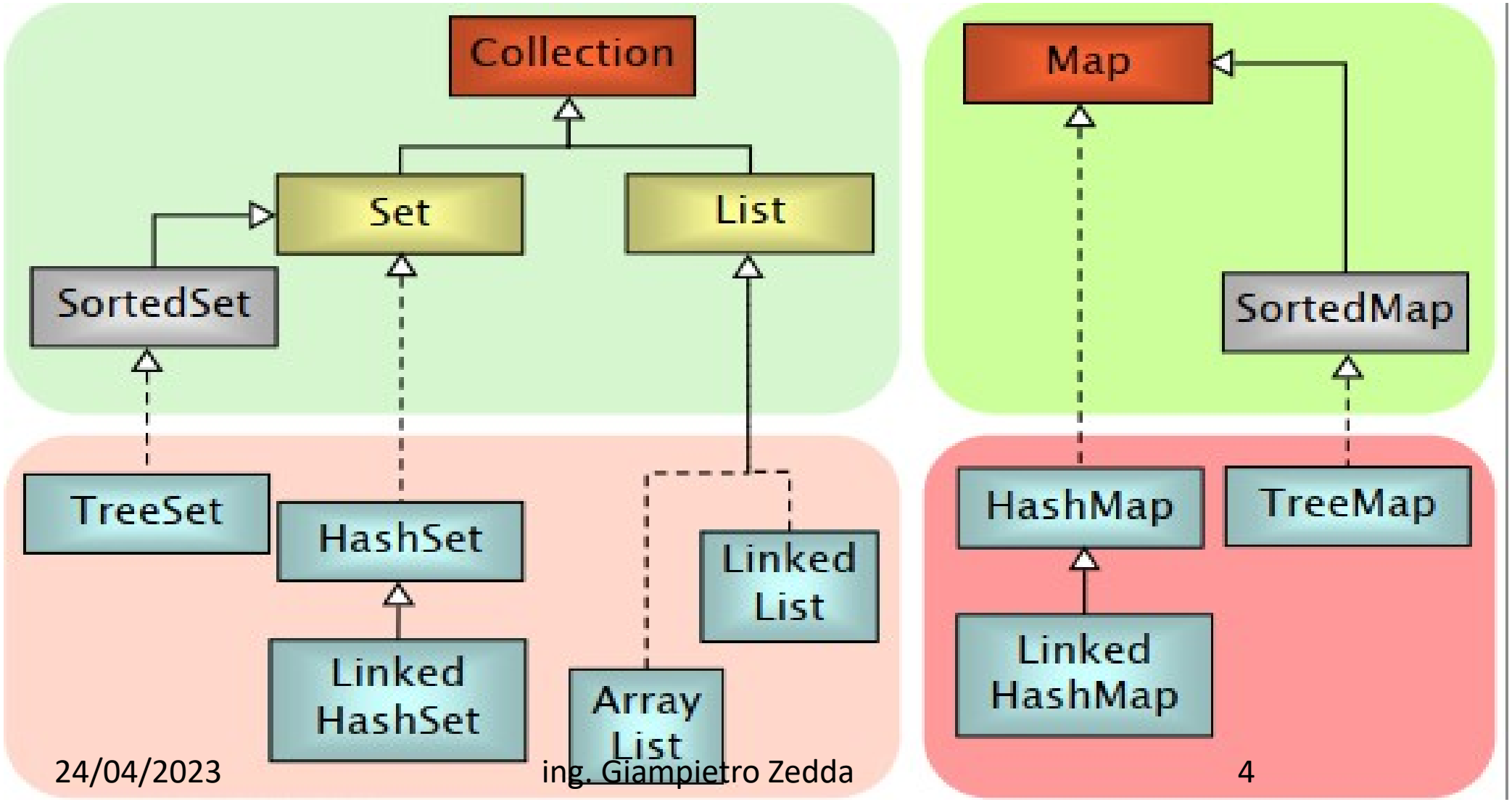
Collection

- Necessità di gestire dinamicamente insiemi di dati all'interno di particolari strutture (**collezioni di elementi**)
- Inserimento rimozione e visualizzazione di elementi
- Implementati in **java.util.***
- Interfacce di alto livello
- Classi concrete di implementazione disponibili
- In base a considerazioni di velocità e consumo di memoria

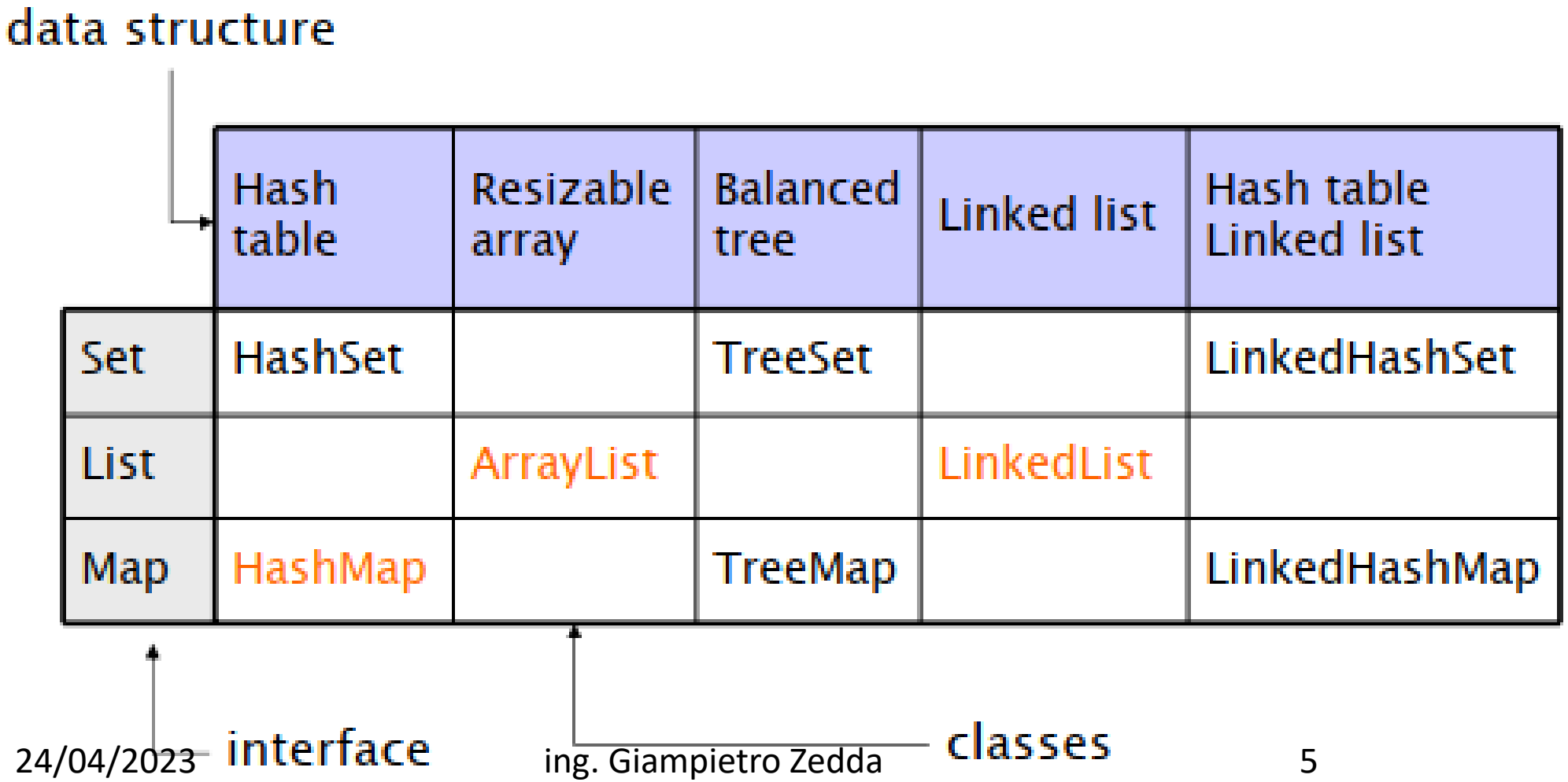
Interfaces


Collection
Set
List
SortedSet
Map
SortedMap
Group containers
Associative containers

Implementazioni



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

Collection

- Group of elements (references to Object instances)
- It is not specified whether they are
  - Ordered / not ordered
  - Duplicated / not duplicated

- Following constructors are common to all classes implementing Collection
  - T()
  - T(Collection c)

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()`

Collection example

```
Collection persons = new LinkedList();
persons.add( new Person("Alice") );
System.out.println( persons.size() );

Collection copy = new TreeSet();
copy.addAll(persons); // new TreeSet(persons)

Persons[] array = (Persons[])copy.toArray();
System.out.println( array[0] );
```

Map

- An object that associates keys to values (e.g., SSN ⇒ Person)
- Keys and values must be objects
- Keys must be unique
- Only one value per key

- Following constructors are common to all collection implementers
  - T()
  - T(Map m)

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 example

```
Map people = new HashMap();
people.put( "ALCSMT", //ssn
            new Person("Alice", "Smith") );
people.put( "RBTGRN", //ssn
            new Person("Robert", "Green") );

Person bob = (Person)people.get("RBTGRN");
if( bob == null )
   System.out.println( "Not found" );
```

Iterator

- Transparent means to cycle through all elements of a Collection

- Keeps track of last visited element of the related collection
- Each time the current element is queried, it moves on automatically

Iterator Interface

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

Iterator Examples

Print all objects in a list

```
Collection persons = new LinkedList();
…
for( Iterator i = persons.iterator(); i.hasNext(); ) {
    Person p = (Person)i.next();
    …
}
```

Iterator Examples

Print all values in a map
(variant using while)

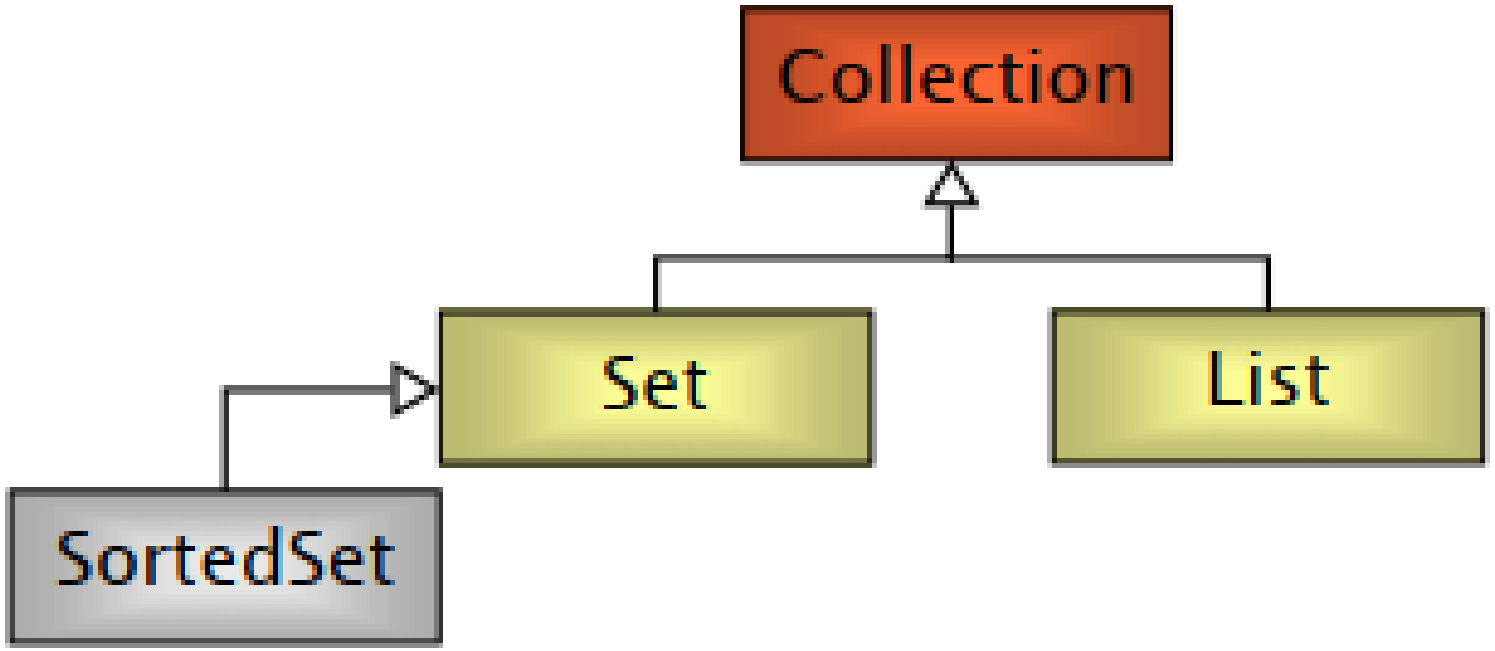
```
Map people = new HashMap();
…
Collection values = people.values();
Iterator i = values.iterator();
while( i.hasNext() ) {
    Person p = (Person)i.next();
    …
}
```

Iterator Examples

Print all keys AND values in a map

```
Map people = new HashMap();
…
Collection keys = people.keySet();
for( Iterator i = keys.iterator(); i.hasNext(); ) {
    String ssn = (String)i.next();
    Person p = (Person)people.get(ssn);
    …
}
```

Group containers (Collections)



Set

- Contains no methods other than those inherited from Collection
- `add()` has restriction that no duplicate elements are allowed
  - `e1.equals(e2) == false` $\forall$ e1,e2 $\in \Sigma$

- Iterator
  - The elements are traversed in no particular order

Sorted Set

- No duplicate elements
- Iterator
  - The elements are traversed according to the natural ordering (ascending)
- Augments Set interface
  - `Object first()`
  - `Object last()`
  - `SortedSet headSet(Object toElement)`
  - `SortedSet tailSet(Object fromElement)`
  - `SortedSet subSet(Object from, Object to)`

Set Implementations

- HashSet implements Set
  - Hash tables as internal data structure (faster)
- LinkedHashSet extends HashSet
  - Elements are traversed by iterator according to the insertion order

- TreeSet implements SortedSet
  - R Btrees as internal data structure (computationally expensive)

List

- Can contain duplicate elements
- Insertion order is preserved
- User can define insertion point
- Elements can be accessed by position
- Augments Collection interface

List Additional Method

- `Object get(int index)`
- `Object set(int index, Object element)`
- `void add(int index, Object element)`
- `Object remove(int index)`

- `boolean addAll(int index, Collection c)`
- `int indexOf(Object o)`
- `int lastIndexOf(Object o)`
- `List subList(int fromIndex, int toIndex)`

List Implementations

ArrayList

- get(n)
  - Constant time
- Insert (beginning) and delete while iterating
  - Linear

LinkedList

- get(n)
  - Linear time
- Insert (beginning) and delete while iterating
  - Constant

List Implementations

- ArrayList
  - `ArrayList()`
  - `ArrayList(int initialCapacity)`
  - `ArrayList(Collection c)`
  - `void ensureCapacity(int minCapacity)`
- LinkedList
  - `void addFirst(Object o)`
  - `void addLast(Object o)`
  - `Object getFirst()`
  - `Object getLast()`
  - `Object removeFirst()`
  - `Object removeLast()`

Example 1

```
LinkedList ll =
                new LinkedList();

ll.add(new Integer(10));
ll.add(new Integer(11));

ll.addLast(new Integer(13));
ll.addFirst(new Integer(20));

//20, 10, 11, 13
```

Example 2

```
Car[] garage = new Car[20];

garage[0] = new Car();
garage[1] = new ElectricCar();
garage[2] = new Ele
garage[3] = new Car

for(int i=0; i<gara
   garage[i].turnOn
}
```

```
List garage = new ArrayList(20);

garage.set( 0, new Car() );
garage.set( 1, new ElectricCar() );
garage.set( 2, new ElectricCar() );
garage.set( 3, new Car());

for(int i; i<garage.size(); i++){
   Car c = garage.get(i);
   c.turnOn();
}
```

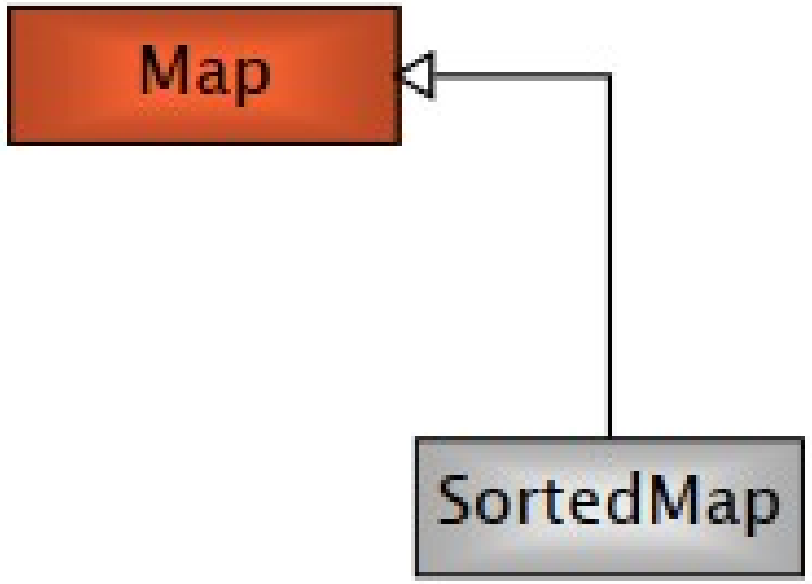
Example 3

```
List l = new ArrayList(2); // 2 refs to null

l.add(new Integer(11));     // 11 in position 0
l.add(0, new Integer(13)); // 11 in position 1
l.set(0, new Integer(20)); // 13 get replaced by 20

l.add(9, new Integer(30));  // NO: out of bounds
l.add(new Integer(30));     // OK, size extended
```

Associative containers (Maps)



SortedMap

- The elements are traversed according to the keys' natural ordering (ascending)
- Augments Map interface
  - `SortedMap subMap(Object fromKey, Object toKey)`
  - `SortedMap headMap(Object toKey)`
  - `SortedMap tailMap(Object fromKey)`
  - `Object firstKey()`
  - `Object lastKey()`

Map Implementations

- Analogous of Set

- HashMap implements Map
  - No order
- LinkedHashMap extends HashMap
  - Insertion order
- TreeMap implements SortedMap
  - Ascending key order

HashMap

- Get/set takes constant time (in case of no collisions)
- Automatic re-allocation when load factor reached
- Constructor optional arguments
  - load factor (default = .75)
  - initial capacity (default = 16)

Using HashMap

```
Map students = new HashMap();

students.put("123", new Student("123","Joe Smith"));

Studente s = (Student) students.get("123");

Iterator i = students.values().iterator();

while (i.hasNext()){
    // i.next();
}
```

Iterators

- It is unsafe to iterate over a collection you are modifying (add/del) at the same time

- Unless you are using the iterator methods
  - `Iterator:remove()`
  - `ListIterator:add()`

Delete

```
List lst = new LinkedList();

lst.add(new Integer(10));
lst.add(new Integer(11));
lst.add(new Integer(13));
lst.add(new Integer(20));

int count = 0;
for (Iterator itr = lst.iterator(); itr.hasNext(); ) {
    itr.next();
    if (count==2)
        lst.remove(count); // wrong
    count++;
}
```

Delete 2

```
List lst = new LinkedList();

lst.add(new Integer(10));
lst.add(new Integer(11));
lst.add(new Integer(13));
lst.add(new Integer(20));

int count = 0;
for (Iterator itr = lst.iterator(); itr.hasNext(); ) {
    itr.next();
    if (count==2)
        itr.remove(count); // ok
    count++;
}
```

Add

```
List lst = new LinkedList();

lst.add(new Integer(10));
lst.add(new Integer(11));
lst.add(new Integer(13));
lst.add(new Integer(20));

int count = 0;
for (Iterator itr = lst.iterator(); itr.hasNext(); ) {
    itr.next();
    if (count==2)
        lst.add(count, new Integer(22)); // wrong
    count++;
}
```

Wrong

Add 2

```
List lst = new LinkedList();

lst.add(new Integer(10));
lst.add(new Integer(11));
lst.add(new Integer(13));
lst.add(new Integer(20));

int count = 0;
for (ListIterator itr = lst.iterator(); itr.hasNext(); ) {
    itr.next();
    if (count==2)
       itr.add(new Integer(22)); // ok
    count++;
}
```

Correct

For-each as alternative to iterator

- No need of modified elements
- No need of reversed access
- Example for(int v : collection) {….}

Algorithms

- Static methods of java.util.Collections class
  - Work on lists

- sort() – merge sort, n log(n)
- binarySearch() – requires ordered sequence
- shuffle() – unsort
- reverse()– requires ordered sequence
- rotate() – of given a distance
- min(), max() – in a Collection

Sort

```
List list;
…
Collections.sort(list);
```

- Sorts the list according to the “natural order”
  - java.lang.String objects are lexicographically ordered
  - java.util.Date objects are chronologically ordered
- TRICK?

Object Ordering

```
public interface Comparable {
    public int compareTo(Object o);
}
```

- java.lang
- Compares the receiving object with the specified object, and returns a negative integer, zero, or a positive integer as the receiving object is less than, equal to, or greater than the specified object
- Implemented by String, Integer, Double, Date, etc.

Custom Ordering

- How to sort a list of Student objects according to the "natural order" by means of the **`sort()`** method?

```
public class Student implements Comparable{
    private String first;
    private String last;

    public int compareTo(Object o){
        ...
    }
}
```

Custom Ordering

```
public int compareTo(Object o){

    if ( !(o instanceof Student) )
        throw new IllegalArgumentException();

    Student s = (Student)o;

    int cmp = lastName.compareTo(s.lastName);

    if(cmp!=0)
        return cmp;
    else
        return firstName.compareTo(s.firstName);
}
```

Custom ordering (alternative)

```
public interface Comparator {
    public int compare(Object o1, Object o2);
}
```

- java.util
- Compares its two arguments, returning a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second

Custom Ordering (alternative)

```
public class Student {
    private String first;
    private String last;
    private int ID;

    public int getID(){…}
}
```

```
List students = new LinkedList();

students.add(new Student("Mary",  "Smith",  34621));
students.add(new Student("Alice", "Knight", 13985));
students.add(new Student("Joe",   "Smith",  95635));

Collections.sort(students, new StudentIDComparator());
```

Custom Ordering (alternative)

```
class StudentIDComparator implements Comparator {

   public int compare(Object o1, Object o2){

      Student s1 = (Student)o1;
      Student s2 = (Student)o2;

      return s1.getID() < s2.getID();
   }
}
```

Note on sorted collections

- TreeSet()
  - Natural ordering (elements must be implementations of Comparable)
- TreeSet(Comparator c)
  - Ordering is according to the comparator rules, instead of natural ordering

- Same for TreeMap

Search

- `int binarySearch(List l, Object key)`
  - Searches the specified object
  - List must be sorted into ascending order according to natural ordering
- `int binarySearch(List l, Object key, Comparator c)`
  - Searches the specified object
  - List must be sorted into ascending order according to the specified comparator

Algorithms - Arrays

- Static methods of java.util.Arrays class
  - Work on object arrays

- sort()
- binarySearch()

Algorithms - Arrays

- `int binarySearch(Object[] a, Object key)`
  - Searches the specified object
  - Array must be sorted into ascending order according to natural ordering
- `int binarySearch(Object[] a, Object key, Comparator c)`
  - Searches the specified object
  - Array must be sorted into ascending order according to the specified comparator