Object Oriented Programming

Java

Part 9: Utility classes

Introduction (1)

- Java provides a set of utility classes:
  - with important functionality to the programmer.
  - distributed in the development environment in different packages (inside **src.zip** file)
    - src/java/lang # classes of the language (`Integer`,...)
      - » Automatically imported
    - src/java/util # diverse utilities (`Vector`,...)
    - src/java/math # `Math` class
    - src/java/io # I/O classes

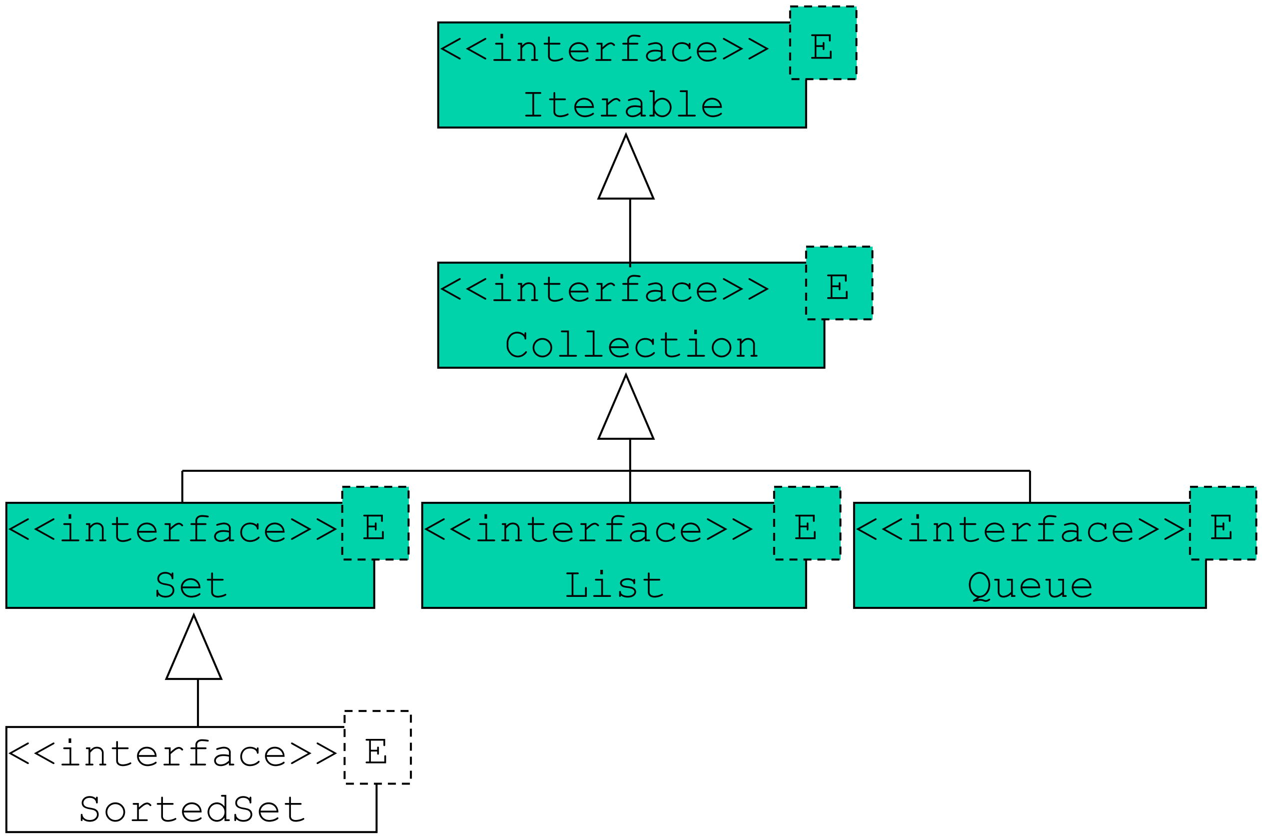
Introduction (2)

- The J2SE provides several groups of interfaces. In these slides we focus 4 of them:
  1. `Comparator` and `Comparable` – describe comparison between objects (for instance, for sorting).
  2. `Collection` – describe collections of objects.
  3. `Map` – describe functions between objects.
  4. `Iterator` – describe iterations over collections of objects, without knowing the way objects are organized inside the collection.
- The code of the classes is available in:

http://www.docjar.com

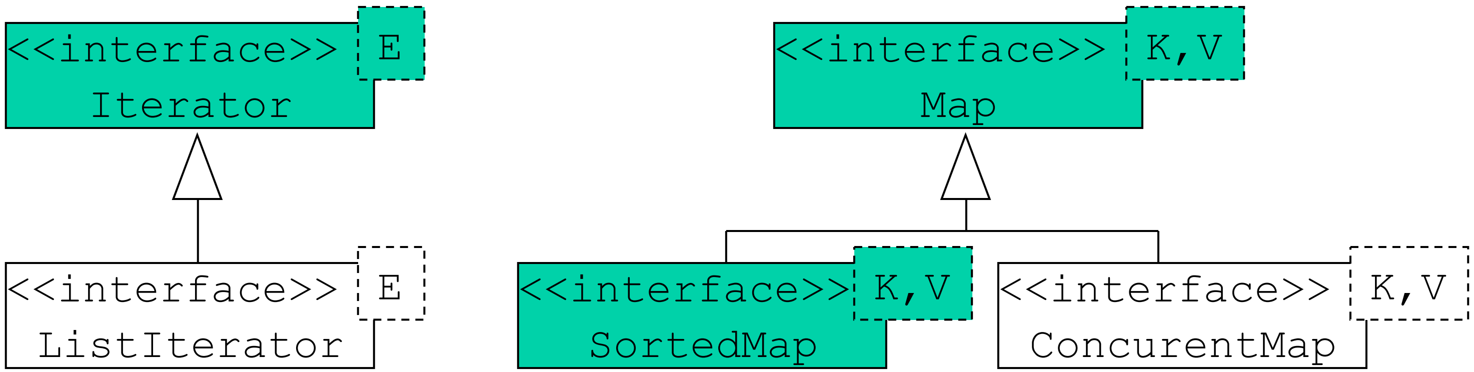
Introduction (3)

General hierarchy of interfaces for ADTs in J2SE 5



Introduction (4)

General hierarchy of interfaces for ADTs in J2SE 5



Sorting (1)

- Classes that require sorting implement one of two interfaces:
  - **`Comparable`**
  - **`Comparator`**

`Comparable` interface (1)

- Used when there is a **natural order** (e.g.: `Character`, `Integer`, `Date`).
- Implemented inside the by the method **`compareTo`**, which implies **total order** inside the class.
- **Simpler to implement, but less flexible** than the `Comparator` interface.

`Comparable` interface (2)

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

- The value returned by the `compareTo` should be:
  - **< 0** if this object is less than the object passed by parameter
  - **= 0** if this object is equal (with `equals`) to the object received as parameter
  - **> 0** otherwise

`Comparable` interface (3)

```
public class Account implements Comparable<Account> {
     private static long nbNextAccount = 0;
     protected long nbAccount; // account number
     protected float balance;  // current balance
     //...
     public boolean equals(Object obj) {
          return nbAccount==((Conta)obj).nbAccount;
     }
     public int compareTo(Account other) {
          if (nbAccount > other.nbAccount) return 1;
          else if (nbAccount == other.nbAccount) return 0;
          else return -1;
     }
     //...
}
```

`Comparable` interface (4)

```
Account mc = new Account("Manuel Silva",1000);
Account outra = new Account("Luís Silva",200);
System.out.println(mc.compareTo(mc));
System.out.println(mc.compareTo(outra));
System.out.println(outra.compareTo(mc));
```

In the terminal is printed

```
0
1
-1
```

`Comparable` interface (5)

- Interfaces define types, so we can have:

```
Comparable<Account> cc;
```

- It is possible to define, for instance, a method to sort an array of `Comparable` objects (without knowing to which class these objects belong):

```
class Sort {
    static Comparable<?>[] sort(Comparable<?>[] objs) {
        // sort details ...
        return objs;
    }
}
```

`Comparable` interface (6)

- The class **`java.util.Arrays`** provides a method that allows to sort objects in an `Object` array according to the natural ordering of its elements:

```
public static void sort(Object[] a,
        int fromIndex, int toIndex)
```

  – Sort the objects in array `a` from index `fromIndex` (inclusive) to index `toIndex` (exclusive).
  – All elements in [`fromIndex`, `toIndex`] must implement the `Comparable` interface.
  – All elements in that range must be mutually comparable (that is, `obj1.compareTo(obj2)` must not throw an exception `ClassCastException`).

`Comparable` interface (7)

```
public class Sort {
    static Comparable<?>[] sort(Comparable<?>[] objs) {
        Comparable<?>[] res = new Comparable<?>[objs.length];
        System.arraycopy(objs, 0, res, 0, objs.length);
        java.util.Arrays.sort(res, 0, res.length);
        return res;
    }
}
```

```
Account mc = new Account(“Manuel Silva”,1000);
Account outra = new Account(“Luís Silva”,200);
Comparable<?>[] accounts = new Comparable<?>[2];
accounts[0]=outra;
accounts[1]=mc;
Accounts = Sort.sort(accounts);
```

`Comparator` interface (1)

- Used when there is an **application-dependent order** (e.g.: sorting a list of students of a certain course may be performed according to their number, name, or mark).
- Implemented outside the class (but it can use the `compareTo` over the class fields), realizing the `Comparator` interface.
- **More complex implementation but more powerful** than the one offered by the `Comparable` interface.

`Comparator` interface (2)

```
public interface Comparator<T> {
   public int compare(T o1, T o2);
}
```

- Value returned by `compare` must be:
  - `<` 0 if the object `o1` is less than the object `o2`
  - `=` 0 if the object `o1` is equal to the object `o2`
  - `>` 0 otherwise

Comparator interface (3)

- Despite not making sense to define a natural ordering of accounts by balance, you may need to sort accounts by balance somewhere in an application...

```
import java.util.Comparator;
public class ComparatorByBalance implements Comparator<Account> {
    public int compare(Account o1, Acount o2) {
        if (o1.balance > o2.balance) return 1;
        else if (o1.balance == o2.balance) return 0;
        else return -1;
    }
}
```

`Comparator` interface (4)

- The class **`java.util.Arrays`** provides a generic method that allows to order the objects in an array according to an order induced by a `Comparator`:

```
public static <T> void sort(
        T[] a, int fromIndex, int toIndex,
        Comparator<? super T> c)
```

  – Sorts the objects in array `a` from index `fromIndex` (inclusive) to index `toIndex` (exclusive).
  – All elements in this range must be mutually comparable with the specified `Comparator` (that is, `obj1.compare(obj2)` must not throw an exception `ClassCastException`).

`Comparator` interface (5)

- An array of accounts could be ordered by balance in this way:

```
Account[] accounts = new Account[2];
accounts[0] = new Account("Manuel Silva",1000);
accounts[1] = new Account("Luís Silva",200);
java.util.Arrays.sort(
        accounts, 0, accounts.length,
        new ComparatorByBalance());
```

`Comparator` interface (6)

- Given a list of students, the natural criterion would be to sort students by number. To impose an ordering by name:

```
import java.util.Comparator;
public class StudentComparatorByName
              implements Comparator<Student> {
     public int compare(Student o1, Student o2) {
          String name1 = o1.firstName();
          String name2 = o2.lastName();
          if (name1.equals(name2)) {
               name1 = o1.lastName();
               name2 = o2.lastName();
               return name1.compareTo(name2);
          } else return name1.compareTo(name2);
     }
}
```

`Comparator` interface (7)

```
public static void main(String[] args) {
     Student[] students = new Student[args.length];
     for(int i=args.length-1; i>=0; i--)
          students[i] = new Student(args[i]);
     System.out.println(students);
     System.out.println("*** Ordered by name ***");
     java.util.Arrays.sort(students,
          new StudentComparatorByname());
     System.out.println(students);
}
```

`Comparator` interface (8)

- If one wants to sort the student's array according to other criteria, for instance, according to their marks, one should simply develop another implementation of `Comparator` and invoke the method `sort` from `java.util.Arrays`.

```
System.out.println("*** Ordered by mark ***");
java.util.Arrays.sort(students,
        new StudentsComparatorByMark());
System.out.println(students);
```

`Iterator` interface

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

- **The `Iterator` interface must be implemented by those classes that want to iterate over its elements, one by one.**

`Iterable` interface

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

- **A class that implements the `Iterable` interface offers an `Iterator` which can then be used in for-each loops.**

`Collection` interface (1)

- A **collection**, or **container**, is an object that contains diverse objects (eventually repeated) in a single unit.
- Prototypes of methods are grouped in:
  - Basic operations.
  - Bulk operations which perform an operation on the entire collection.
  - Operations that convert the collection into an array.

Collection interface (2)

```
public interface Collection<E> extends Iterable<E> {
     // Basic operations
     int             size();
     boolean         isEmpty();
     boolean         contains(Object elem);
     boolean         add(E elem);
     boolean         remove(Object elem);
     Iterator<E>     iterator();
     // Bulk operations
     boolean         containsAll(Collection<?> coll);
     boolean         addAll(Collection<? extends E> coll);
     boolean         removeAll(Collection<?> coll);
     boolean         retainAll(Collection<?> coll);
     void            clear();
     // Array operations
     Object[]        toArray();
     <T> T[]         toArray(T dest[]);
}
```

`Collection` interface (3)

- **All methods that need the notion of equivalence between objects use the `equals` method** (`contains`, `add`, `remove`, `containsAll`, `addAll`, `removeAll` e `retainAll`).
- The `Collection` interface does not make any restriction about adding `null` elements to the collection.

`Collection` interface (4)

- It is possible to use a for loop and the `Iterator` methods to step through the contents of a collection:
  - It is possible to `add`/`remove` objects to/from the collection during the iteration.
  - It possible to update the objects during the iteration.
  - It is possible to iterate over multiple collections.

```
public class RemoveShortStrings {
     public static void remove(Collection<String> c){
          // remove strings with length 0 or 1
          for (Iterator<String> i=c.iterator(); i.hasNext(); )
               if (i.next().length()<2)
                    i.remove();
     }
}
```

`Collection` interface (5)

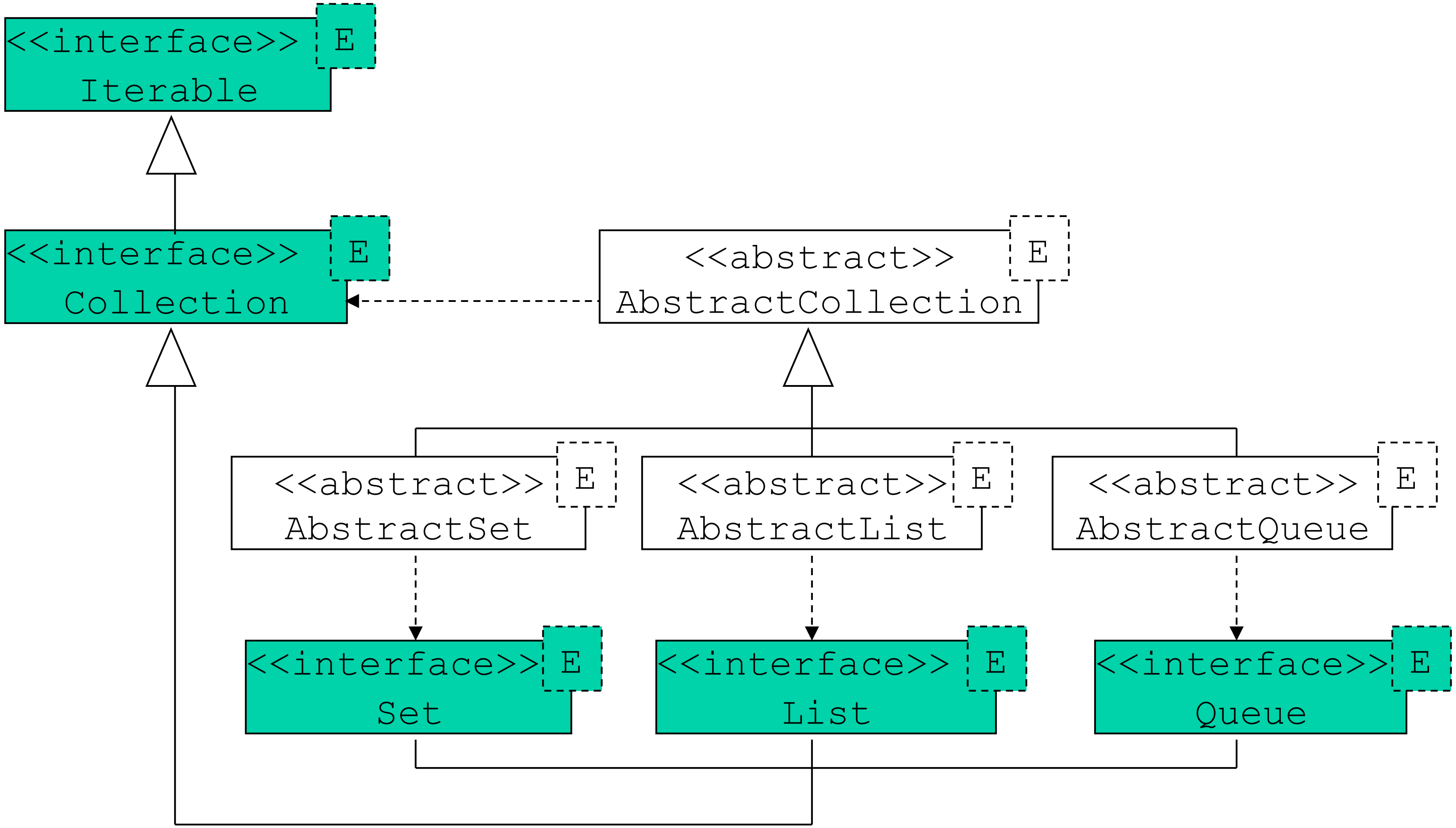
- Stepping through the elements of a collection can also be performed with a for-each loop:
  - The advantage of the for-each loop is purely syntactic.
  - It is not possible to `add`/`remove` objects to/from the collection during the iteration.
  - It possible to update the objects during the iteration.
  - It is not possible to iterate over multiple collections.

```
public class PrintShortStrings {
     public static void print(Collection<String> c){
          for(String s:c)
               if (s.length()<2)
                    System.out.println(s);
     }
}
```

`Collection` interface (6)

- From the **`Collection`** interface several interfaces are derived:
  - **`Set`**: collection without duplicate elements
  - **`List`**: list of elements
  - **`Queue`**: queue of elements

`Collection` interface (7)



Interfaces' implementation (1)

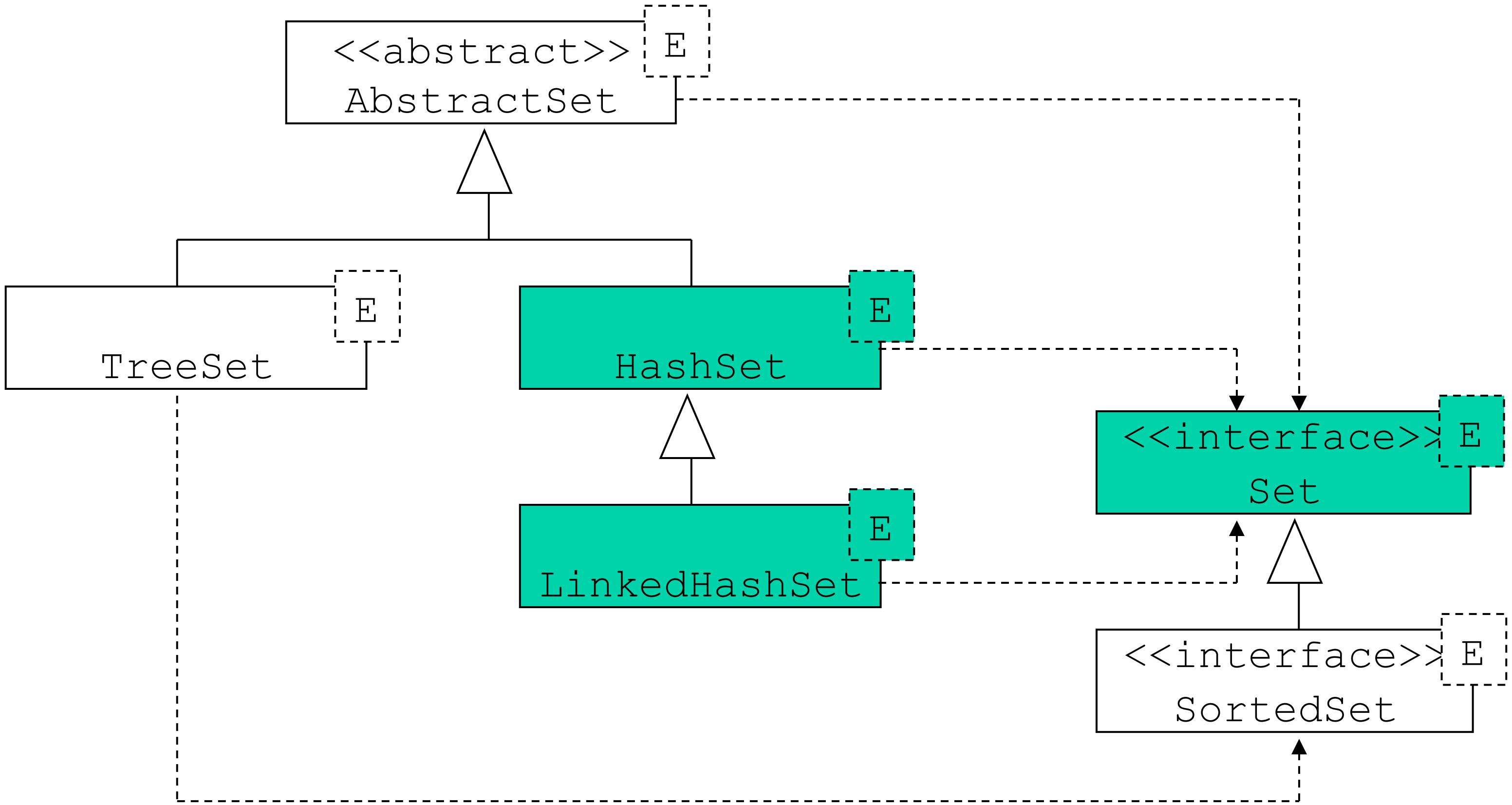
- There are diverse structures underlying data to implement interfaces:
  1. **Linear**: the objects are ordered in positions, each object has only a predecessor (except the first) and a successor (except the last).
  2. **Hierarchical**: each object has only a predecessor (except the root) and it might has a fixed number of successors.
  3. **Unsorted**: there are no relation between two objects.

Interfaces' implementation (2)

- The J2SE 5 implements the `Set/List/Map` interfaces through four data structures:
  - **Hash tables**.
  - **Variable length arrays**.
  - **Balanced tress**.
  - **Linked lists**.

	Underlying data structure				
Interfaces	Hash tab.	Var. len. arrays	Bal. trees	Linked lists	Hash + linked list
`Set`	`HashSet`	---	`TreeSet`	---	`LinkedHashSet`
`List`	---	`ArrayList`	---	`LinkedList`	---
`Map`	`HashMap`	---	`TreeMap`	---	`LinkedHashMap`

`Set` interface (1)



`Set` interface (2)

- The `Set` interface adds no new method of its own:
  - It only provides the methods from `Collection`.
  - Extra restrictions are imposed to the `add` method in order to avoid duplicate elements in this collection.

`Set` interface (3)

- Some implementations of the `Set` interface:
  - **`HashSet`**: the best for most of the uses.
  - **`LinkedHashSet`**: imposed order in the `Iterator` (insertion order).
  - **`TreeSet`**: imposed order in the `Iterator` (natural order or an order defined by a `Comparator`).
- Exposure of the implementation should be avoided:

```
Set<Integer> s = new HashSet<Integer>(); // preferable
```

```
HashSet<Integer> s = new HashSet<Integer>(); // to avoid!
```

`Set` interface (4)

- Typical operations over sets:
  - Union:

```
s1.addAll(s2)
```

  - Intersection:

```
s1.retainAll(s2);
```

  - Relative complement:

```
s1.removeAll(s2);
```

  - Is subset:

```
s1.containsAll(s2);
```

`Set` interface (5)

- In order to delete duplicate elements from a collection:

```
Collection<Integer> withDuplicates;
//...
```

```
Collection<Integer> withoutDuplicates = new HashSet<Integer>();
withoutDuplicates.addAll(withDuplicates);
```

```
Collection<Integer> withoutDuplicates
        = new HashSet<Integer>(withDuplicates);
```

`Set` interface (6)

```
Set<String> s1 = new HashSet<String>();
s1.add("Ana"); s1.add("Joao");
System.out.print("s1 = ",s1);

Set<String> s2 = new HashSet<String>();
s2.add("Joao"); s2.add("Luis");
System.out.print("s2 = ",s2);

Set<String> s3;
s3 = new HashSet<String>(s1); s3.addAll(s2);
System.out.print("Union(s1,s2) = ",s3);
s3 = new HashSet<String>(s1); s3.retainAll(s2);
System.out.print("Intersection(s1,s2) = ",s3);
s3 = new HashSet<String>(s1); s3.removeAll(s2);
System.out.print("RelComplement(s1,s2) = ",s3);
```

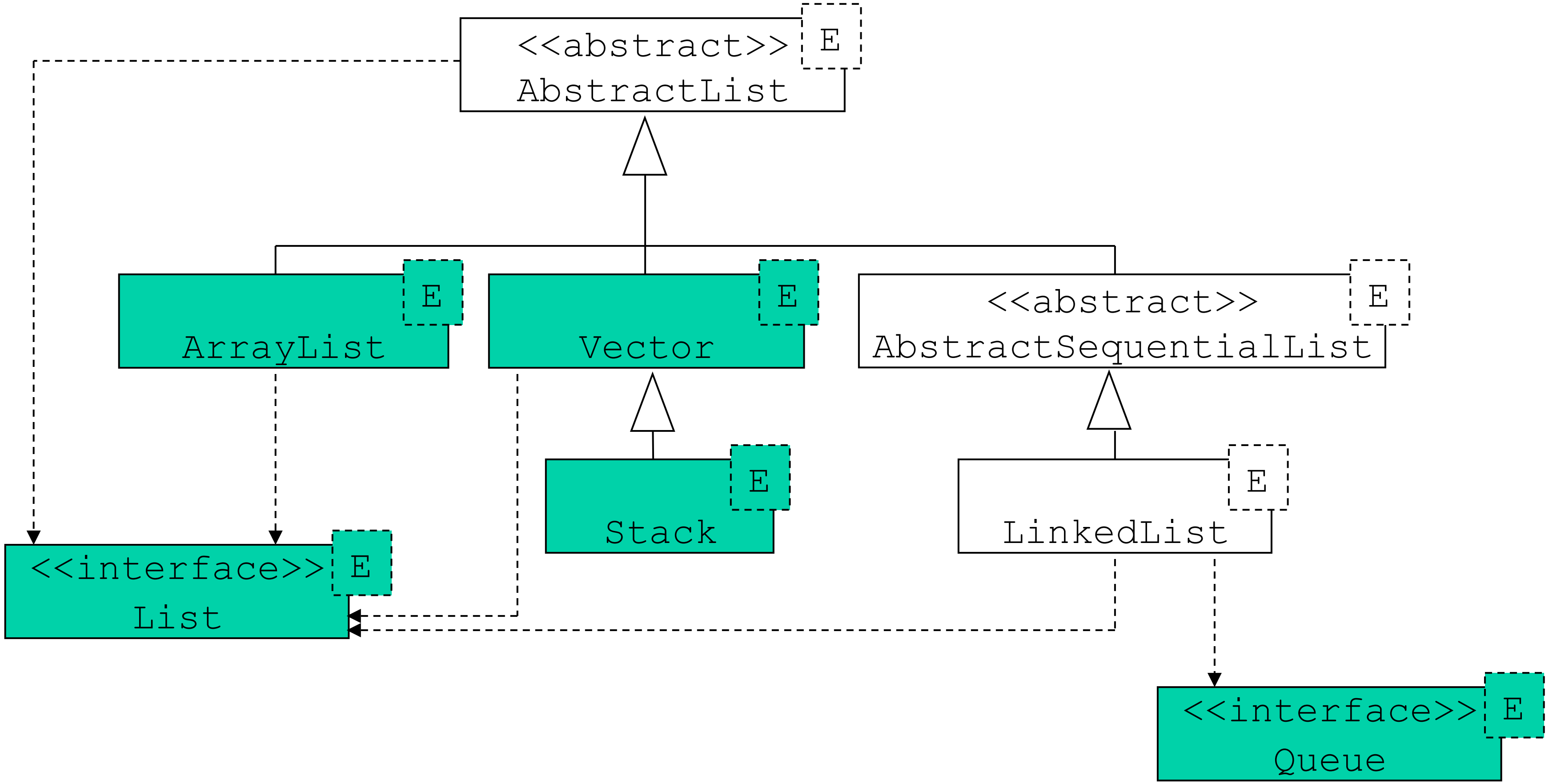
`Set` interface (7)

Relatively to the previous example:

- In the terminal is printed:

```
s1 = [Joao, Ana]
s2 = [Joao, Luis]
Union(s1,s2) = [Joao, Luis, Ana]
Intersection(s1,s2) = [Joao]
RelComplement(s1,s2) = [Ana]
```

`List` interface (1)



`List` interface (2)

```
public interface List<E> extends Collection<E> {
     E get(int index);
     E set(int index, E elem);
     void add(int index, E elem);
     E remove(int index);
     int indexOf(Object elem);
     int lastIndexOf(Object elem);
     List<E> subList(int min, int max);
     ListIterator<E> listIterator(int index);
     ListIterator<E> listIterator();
}
```

`ArrayList` class (1)

- It is an implementation of `List` that store its elements in an array:
  - The array has an initial capacity.
  - When the initial capacity is exceeded it is built a new array and its content is copied.
  - A correct value for the initial capacity of the `ArrayList` improves its performance.
- **Complexity:**
  - **Adding (in position i) and removing (in position i): O(n-i) where n is the length of the list and i<n.**
    - **Add (in the end) and remove (from the end): O(1)**
  - **Accessing an element (in any position): O(1)**

`ArrayList` class (2)

```
public class ArrayList<E>
     extends AbstractList<E>
     implements List<E>, ...
{
     ArrayList() {...}
     ArrayList(int initialCapacity) {...}
     ArrayList(Collection<? extends E> coll) {...}
     void trimToSize() {...}
     void ensureCapacity(int minCapacity) {...}
}
```

- By default, the initial capacity of an `ArrayList` is 10.

`LinkedList` class (1)

- It is an implementation of `List` with a doubly linked list.
- The `LinkedList` class also implements the interface `Queue`.
- **Complexity:**
  - **Add (in position i) and remove (from position i): O(min{i,n-i}), where n is the length of the list.**
    - **Add (in the beginning or in the end) and remove (from the beginning or the end): O(1).**
  - **Accessing an element in position i: O(min{i,n-i}), where n is the length of the list.**

`LinkedList` class (2)

- **From the complexity analysis one can conclude:**
  - **A `LinkedList` should be used wherever:**
    - **The length of the list varies.**
    - **It is important to add or remove elements in arbitrary positions of the list.**
  - **It is preferable to use an `ArrayList` whenever:**
    - **New elements are added/removed to/from the end of the list.**
    - **It is important to access its elements very efficiently.**

`LinkedList` class (3)

```
public class LinkedList<E>
     extends AbstractSequentialList<E>
     implements List<E>, Queue<E>, ...
{
     LinkedList() {...}
     LinkedList(Collection<? extends E> coll) {...}
     E getFirst() {...}
     E getLast {...}
     E removeFirst() {...}
     E removeLast() {...}
     void addFirst(E elem) {...}
     void addLast(E elem) {...}
}
```

Legacy collections types

- The interfaces/classes presented so far are new in the `java.util` package (version 5 or later).
- The `java.util` package has always contained some other collections, which are not deprecated because they are in widespread use in existing code (previous to version 5). The most proeminent are:
  - `Vector`
  - `Stack`
- Although `Vector` and `Stack` are legacy types, they implement the `List` interface, so they work just like any other collection.

`Vector` class (1)

```
public class Vector<E>
     extends AbstractList<E>
     implements List<E>, ...
{
     //New methods, as in ArrayList
     Vector() {...}//ArrayList()
     Vector(int initialCapacity) {...}//ArrayList(initialCapacity)
     Vector(Collection<? Extends E> coll) {...}//ArrayList(coll)
     void trimToSize() {...}
     void ensureCapacity(int minCapacity) {...}
     //... Difference to ArrayList (next slide)
```

- By default, the initial capacity of a `Vector` is 10.

Vector class (2)

```
    //Difference to ArrayList
    Vector(int initialCapacity, int capacityIncrement) {...}
    void copyInto(Object[] anArray) {...}
    int indexOf(Object elem, int index) {...}
    int lastIndexOf(Object elem, int index) {...}
    void setSize(int newSize) {...}
    int capacity() {...}
    //Legacy methods with equivalence in ArrayList
    void addElement(E elem) {...}//add(elem)
    void insertElement(E elem, int index) {...}//add(index,elem)
    void setElement(E elem, int index) {...}//set(index,elem)
    void removeElement(int index) {...}//remove(index)
    boolean removeElement(Object elem) {...}//remove(elem)
    void removeAllElements() {...}//clear()
    E elementAt(int index) {...}//get(index)
    E firstElement() {...} //get(0)
    E lastElement() {...} //get(size()-1)
}
```

`Vector` class (3)

- Using `Vector` as legacy code:

```
Vector vector = new Vector(20); // to avoid
```

- Using `Vector` as a generic collection:

```
Vector<?> vector = new Vector<String>(20);
```

```
Vector<String> vector = new Vector<String>(20);
```

`Vector` class (4)

```
import java.util.Vector;
public class Main {
   private final int SIZE = 10;
   private Vector<Integer> vector = new Vector<Integer>(SIZE);
   //...
   public static void main(String[] args) {
      Integer iobj;
      for(int index=0;index<SIZE;index++)
         vector.addElement(new Integer(index));
      for(int index=2;index<SIZE;index+=2){
         iobj=(Integer)vector.elementAt(index-1);
         vector.setElementAt(new Integer(2*iobj.intValue()),index);
      }
      for(int index=0;index<SIZE;index++)
         System.out.println(
            "Índice = "+index+" "+vec.elementAt(index));
   }
}
```

Vector class (5)

```
import java.util.Vector;
public class Main {
     private final int SIZE = 10;
     private Vector<Integer> vector = new Vector<Integer>(SIZE);
     //...
     public static void main(String[] args) {
          Integer iobj;
          for(int index=0;index<SIZE;index++)
               vector.add(new Integer(index));
          for(int index=2;index<SIZE;index+=2){
               iobj=(Integer)vector.get(index-1);
               vector.set(new Integer(2*iobj.intValue()),index);
          }
          for(int index=0;index<SIZE;index++)
               System.out.println(
                    "Índice = "+index+" "+vec.get(index));
     }
}
```

`Vector` class (6)

- The `Vector` class (as any generic type), jointly with the appropriate hierarchy, allows to invoke a method in all its elements without knowing the actual type of the object.
- Example:
  - Consider `CurrentAccount` and `SavingsAccount` both deriving from the abstract class `Account`.
  - In the `Account` is defined an abstract method `interest()`, which is then implemented in the concrete classes.

`Vector` class (7)

```
Vector<Account> v = new Vector<Account>();
//...
v.add(new CurrentAccount());
v.add(new SavingsAccount());
//...
for(i=0;i<v.size();i++)
   (v.get(i)).interest();
```

- The `interest()` method runs the implementation of the respective actual type (dynamic binding).
- Further extending the `Account` hierarchy does not require changes in the existing code for computing the interest, easing the development of applications.

`Stack` class

- The `Stack` class extends `Vector` and adds new methods to obtain a data structure with a LIFO structure.

```
public class Stack<E> extends Vector<E> {
     Stack();
     E push(E item);
     E pop();
     E peek();
     boolean empty();
     int search(Object o);
}
```

`List` implementations

<u>Advantages</u>

- Solve the drawback of the arrays' constant length.

<u>Disadvantages</u>

- It can only store objects (data of primitive type must be stored within wrapper classes).
- Access to arrays is more efficient.

`Arrays` class (1)

- The **static `Arrays` class** is provided by the J2SE with methods to manipulate arrays.
- The great majority of the methods has several overloads:
  - One for arrays for each primitive type.
  - One for `Object` arrays.
- There are also two variants of some methods:
  - One acting in the entire array.
  - One acting on a subarray specified by two supplied indexes.

`Arrays` class (2)

- The methods of the `Arrays` utility class are:
  - **`static void sort`**: sorts in ascending order, with parameters:
    1. Array to sort (mandatory)
    2. Two indexes that define the subarray (by default, coincides with the entire array)
    3. A `Comparator` object that induces an order in the array elements (by default, natural order defined by the `Comparable`)
  - **`static int binarySearch`**: binary search (the array must be sorted in ascending order), with parameters:
    1. Array where to search (mandatory)
    2. Value to search for (mandatory)
    3. A `Comparator` object that induces an order in the array elements (by default, natural order defined by the `Comparable`)

`Arrays` class (3)

```
Integer[] ints = new Integer[2];
ints[0]=1;
ints[1]=2;
System.out.println(Arrays.binarySearch(ints,1));
ints[0]=2;
ints[1]=1;
System.out.println(Arrays.binarySearch(ints,1));
```

In the terminal is printed

```
0
-1
```

`Arrays` class (4)

- **`static void fill`**: Fill the array entries, with parameters:
  1. Array to fill (mandatory)
  2. Two indexes that define the subarray (by default, coincides with the entire array)
  3. Value to insert (mandatory)
- **`static boolean equals`**: Test for equivalence between arrays (use `equals` on each non-`null` element of the array), with parameters:
  1. Two arrays of the same type (mandatory)
- **`static boolean deepEquals`**: Test for equivalence between multidimensional arrays (based on contents), with parameters:
  1. Two arrays of type `Object` (mandatory)

`Arrays` class (5)

- **`static int hashCode`**: returns the array hash code (use the `hashCode` of each non-`null` element).
- **`static int deepHashCode`**: returns the hash code of the `Object[]` array (based on contents, taking into account nested arrays).
- **`static String toString`**: returns a string that represents the textual content of the array received as parameters.
- **`static String deepToString`**: returns a string that represents the textual content, taking into account nested arrays, of the `Object[]` array received as parameter.
- **`static <T> List<T> asList(T[] t):`** returns a `List` with the elements of the array received as parameter.
    - This method acts as bridge between arrays and collections (complements the `toArray` method in collections).

`Arrays` class (6)

```
Integer[][] ints = new Integer[2][5];
Arrays.fill(ints[0],0); Arrays.fill(ints[1],1);
System.out.println("ints="+Arrays.deepToString(ints));
Integer other[][] = new Integer[2][5];
Arrays.fill(other[0],0); Arrays.fill(other[1],1);
System.out.println("other="+Arrays.deepToString(other));
System.out.println(ints.hashCode()+"\t"+
   Arrays.hashCode(ints)+"\t"+
   Arrays.hashCode(ints[0])+"\t"+Arrays.hashCode(ints[1])+"\t"+
   Arrays.deepHashCode(ints));
System.out.println(other.hashCode()+"\t"+
   Arrays.hashCode(other)+"\t"+
   Arrays.hashCode(other[0])+"\t"+Arrays.hashCode(other[1])+"\t"+
   Arrays.deepHashCode(other));
```

In the terminal is printed

```
ints=[[0, 0, 0, 0, 0], [1, 1, 1, 1, 1]]
other=[[0, 0, 0, 0, 0], [1, 1, 1, 1, 1]]
16795905 922240875   28629151 29583456 917088098
12115735 676418749   28629151 29583456 917088098
```

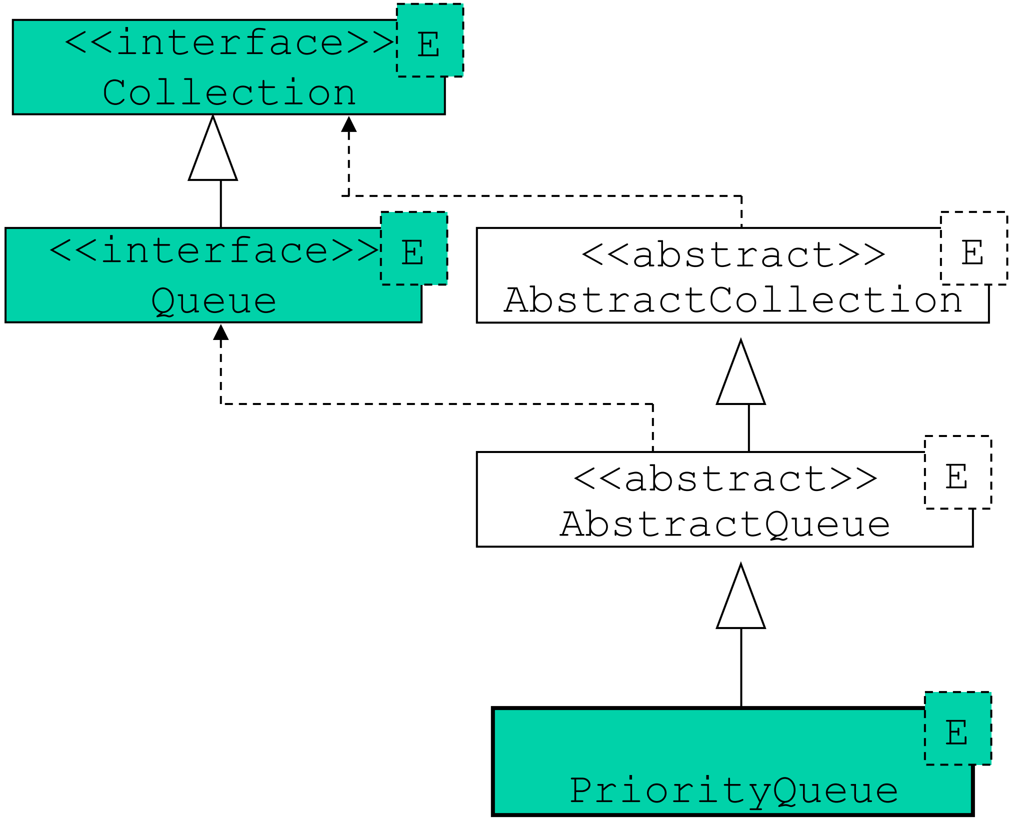
`Arrays` class (7)

```
List<?> list = new ArrayList<Object>();
list.add(1);
list.add("Hello");
Object[] objects = new Object[2];
objects = list.toArray();
System.out.println(list.equals(Arrays.asList(objects)));
System.out.println(objects.equals(list.toArray()));
System.out.println(Arrays.equals(objects,list.toArray()));
```

In the terminal is printed

```
true
false
true
```

Queue interface (1)



`Queue` interface (2)

```
public interface Queue<E> extends Collection<E> {
     E element();
     E peek();
     E remove();
     E pool();
     boolean offer(E elem);
}
```

`Queue` interface (3)

- Although collections allow for `null` elements, a `Queue` must not contain `null` elements, as `null` is used in the return of the `peek` and `poll` methods to indicate that the `Queue` is empty.
- The `LinkedList` class is the simpler implementation of the `Queue` interface.
  - For historic reasons `null` elements are allowed in a `LinkedList`.
  - Inserting `null` elements in a `LinkedList` must be avoided whenever it is being used as a `Queue`.

`PriorityQueue` class (1)

- The `PriorityQueue` is other implementation of `Queue`.
- It is based on a priority heap.
- The head of the priority queue is the smallest element in it.
  - The smallest element is determined either by the elements' natural order or by a supplied comparator.
  - Whether the smallest element represents the element with the highest or lowest priority depends on how the natural order or the comparator is defined.

`PriorityQueue` class (2)

- The `PriorityQueue` iterator is not guaranteed to traverse the elements in priority order.
- But it guarantees that removing elements from the queue occurs in a given order.

`PriorityQueue` class (3)

- `PriorityQueue` constructors:

```
public PriorityQueue()
public PriorityQueue(int initialCapacity)
public PriorityQueue(int initialCapacity,
                     Comparator<? super E> comp)
public PriorityQueue(Collection<? extends E> coll)
public PriorityQueue(SortedSet<? extends E> coll)
public PriorityQueue(PriorityQueue<? extends E> coll)
```

- The capacity is unlimited, but the adjustment is computationally expensive.

`PriorityQueue` class (4)

```
public class Task {
     String name; // identifier
     int level;   // priority

     public int level() {
          return level;
     }
     public void newLevel(int value) {
          level = value;
     }
     public Task(String name, int l) {
          this.name=name;
          level = l;
     }
}
```

`PriorityQueue` class (5)

```
private static class TaskComparator implements Comparator<Task> {
     public int compare(Task l, Task r) {
          return l.level() - r.level();
     }
}
```

PriorityQueue class (6)

```
PriorityQueue<Task> pq =
     new PriorityQueue<Task>(10,new TaskComparator());
Task t;
for (char letter='A';letter<='G';letter++)
     pq.add(new Task("Task "+letter,((letter-'A')%4));
while (!pq.isEmpty()){
     t=pq.poll();
     System.out.println(t.toString()+ " priority="+t.level());
}
```

In the terminal is printed:

```
Task A priority=0
Task E priority=0
Task B priority=1
Task F priority=1
Task C priority=2
Task G priority=2
Task D priority=3
```

Map interface (1)

- The Map<K,V> interface does not extend the Collection interface.
- Main characteristics of a Map<K,V>:
  - One does not add an element to a map, one adds a key/value pair.
  - A map allows to look up the value stored under a key.
  - A given key maps to one value or no values.
  - A value can be mapped to by many keys.
- A **map** establishes a partial function from keys to values.

`Map` interface (2)

- Basic methods of the `Map<K,V>` interface:

```
int size();
boolean isEmpty();
boolean containsKey(Object key);
boolean containsValue(Object value);
V get(Object key);
V put(K key, V value);
V remove(Object key);
void putAll(Map<? extends K, ? extends V> otherMap)
void clear();
```

`Map` interface (3)

- Some methods to see a `Map<K,V>` as a `Collection`:

```
Set<K> keySet();
Collection<V> values();
```

- From the `Map` interface are derived other interfaces:
  - **`SortedMap`**: keys are ordered
  - **`ConcurrentMap`**

HashMap class (1)

- The HashMap class is an implementation of the Map interface by an hash table.
- It is very efficient.
  - With a well-written hashCode method, adding, removing or finding a key/value pair is O(1).
- Constructors of the HashMap class:

```
public HashMap(int initialCapacity, float loadFactor)
public HashMap(int initialCapacity)
public HashMap()
public HashMap(Map<? extends K, ? extends V> map)
```

`HashMap` class (2)

```
import java.util.*;

String str;
Long l;
Map store = new HashMap(); // name is used as key

str = “Miguel”; l = new Long(1327);
store.put(str,l);
l = (Long) store.get(str);
if (l!=null)
    System.out.println(“Codigo de ”+str+“=”+l.longValue());
str = “Luisa”; l = new Long(9261);
store.put(str,l);
l = (Long) store.get(str);
if (l!=null)
    System.out.println(“Codigo de ”+str+“=”+l.longValue());
```

`SortedMap` interface

```
interface SortedMap<K,V> extends Map<K,V> {
    Comparator<? super K> comparator();
    K firstKey();
    K lastKey();
    SortedMap<K,V> subMap(K minKey, K maxKey);
    SortedMap<K,V> headMap(K maxKey);
    SortedMap<K,V> tailMap(K minKey);
}
```

`TreeMap` class

- The `TreeMap` class is an implementation of the `SortedMap` interface by a binary balanced tree.
  - The access is not so efficient as with `HashMap`.
    - Adding, removing or finding a key/value pair is O(log n).
  - But the elements are always ordered.
- In the `HashMap` example just replace the declaration:

```
Map store = new TreeMap();
```