DIETI UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE
Scuola Politecnica e delle Scienze di Base
Università degli Studi di Napoli Federico II

Corso di
Ingegneria del Software

Le Classi Contenitore in Java (List, Set, Map)
Programmazione generica in Java

Sommario

- Classi contenitori in Java: Collection, List, Set e Map
- Funzioni di Utility
- Tipi generici
- Wildcard e vincoli

Thinking in Java, Capitolo «Containers in Depth»

«Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams», Capitolo 4

::.. Introduzione (1/2)

Il linguaggio Java fornisce al programmatore molte **classi predefinite**, che sono raggruppate in **packages** ("directory" organizzate gerarchicamente).

Se un programma usa una classe chiamata **<Class>** del package **<pack>** (diverso da java.lang) deve importarla esplicitamente con un comando import all'inizio del file:

```
import <pack>.<Class>;      // importa la classe: può usare tutto
                            // quanto è public nella classe
import <pack>.*;            // importa tutte le classi di <pack>
```

Eccezioni a questa regola:

•Ogni programma importa **automaticamente** tutte le classi del **proprio package** (directory), e ne ha visibilità di tutte le variabili, i metodi e i costruttori che non sono dichiarati private (quindi quelli public, protected, e quelli senza modificatore di visibilità).

•Il package **java.lang** è importato automaticamente.

::.. Introduzione (2/2)

In Java è possibile definire **array**, ovvero sequenze di elemento dello **stesso tipo** e con **lunghezza fissata** all'atto dell'istanziazione di una variabile array, e **non più modificabile** durante il ciclo di vita della variabile. Tale costrutto non riesce a soddisfare tutte le esigenze dei programmatori, pertanto Java dispone di apposite classi per astrarre casi in cui il tipo array non è adeguato.

::.. Stringhe (1/5)

Un esempio pratico è il caso di una sequenza di caratteri che prende il nome di **stringa**. Java dispone di un'apposita classe per la gestione di stringhe, chiamata **String**. Le istanze di String sono **stringhe non modificabili** e sono creabili con assegnazione di una sequenza di caratteri tra doppi apici ("") o con i costruttori della classe:

- **public String ()** - crea una stringa vuota;
- **public String (char[] value)** - crea una stringa che contiene i caratteri di value;
- **public String (StringBuffer buffer)** - trasforma una stringa modificabile in una non modificabile.

::.. Stringhe (2/5)

Alcuni metodi da conoscere:

- **public boolean equals (Object obj)** - restituisce true se le due stringhe sono uguali;
- **public boolean equalsIgnoreCase (String str)** - come equals, ignorando differenza tra maiuscole e minuscole;
- **public int compareTo (String str)** - confronta le stringhe rispetto all'ordinamento lessicografico
- **public char charAt (int i)** - restituisce l'i-esimo carattere contenuto nella stringa: i deve essere compreso tra 0 e la lunghezza della stringa (str.lenght() - 1);
- **public String substring (int i, int j)** - restituisce la sottostringa contenente i caratteri dalla posizione i alla posizione j-1;
- **public int indexOf (String str)** - restituisce l'indice da cui parte la prima sottostringa uguale a str, e restituisce -1 se non esiste.

::.. Stringhe (3/5)

Le stringhe implementate come istanze di String sono oggetti non modificabili né estendibili. Questo vuol dire che ogni qualvolta viene assegnato un nuovo valore ad una stringa (in un'operazione di concatenazione, rimozione o aggiunta caratteri) **in realtà vengono create nuove stringhe**.

```
// crea un oggetto String e lo assegna al riferimento
String s = new String("Ciao mondo");

// crea un nuovo oggetto String e lo assegna al riferimento;
// il precedente oggetto String rimane ancora nello heap
// (e sarà deallocato in futuro dal garbage collector)
s = "Hello world";
```

::.. Stringhe (3/5)

In Java è possibile realizzare **stringhe modificabili ed estendibili** con istanze della classe **StringBuffer**, creabili per mezzo dei costruttori della classe:

- **public StringBuffer ()** - crea uno StringBuffer vuoto, con lunghezza iniziale 16 (default);
- **public StringBuffer (int lenght)** – come prima ma con lunghezza iniziale length;
- **public StringBuffer (String str)** - contenuto iniziale: str. Consente di usare indirettamente i costruttori della classe String.

Alcuni metodi:

- **public String toString()** - restituisce la stringa corrispondente;
- **public StringBuffer append (Tipo value)** - concatena la rappresentazione testuale dell'argomento al proprio contenuto.

::.. Stringhe (3/5)

```
class Registro {
        private Persone [] listapersone = new Persone[10];

        ...

        public String toString() {

                StringBuffer registrocompleto = new StringBuffer();

                for(int i=0; i<listapersone.length(); i++) {
                        registrocompleto.append( listapersone[i] );
                }

                return registrocompleto.toString();
        }
}
```

::.. Stringhe (4/5)

La classe StringBuffer non è molto usata in modo esplicito dai programmatori, ma lo è dal compilatore Java. In particolare, i metodi **append** sono utilizzati dal compilatore per implementare l'operatore di concatenazione di stringhe'+'.

Esempio: consideriamo il comando

```
x = "a" + 4 + 'c';
```

viene compilato come se fosse l'espressione:

```
x = new StringBuffer().append("a").append(4).append('c').toString();
```

che crea un oggetto di classe StringBuffer vuoto, quindi ci concatena in sequenza "a", 4 e 'c', e infine restituisce la stringa corrispondente.

::.. Stringhe (5/5)

In genere, la quantità e i tipi esatti degli oggetti necessari in un programma sono noti solo in fase di esecuzione. Array impone invece tale conoscenza a tempo di compilazione. Con String e StringBuffer, possiamo costruire sequenze con **quantità non nota a tempo di compilazione**, ma solo per sequenze dello **stesso tipo**, ovvero di caratteri.

La libreria **java.util** offre un insieme completo di **classi contenitore** con cui poter definire insiemi di dati non dello stesso tipo, i cui **tipi e quantità** sono determinati a tempo di esecuzione.

::.. java.util (1/6)

La libreria **java.util** offre un insieme completo di **classi contenitore**, i cui tipi base sono **List**, **Set**, **Queue** e **Map**. A differenza degli array, le classi contenitore in Java si **ridimensionano automaticamente**, e permettono di inserire un numero di oggetti arbitrario senza definire la dimensione all'interno del programma.

Un contenitore (chiamato anche **container** o **collezione**) è un oggetto che raggruppa elementi multipli in una singola unità ed è utilizzato per memorizzare, recuperare e manipolare dati, per trasmetterli da un metodo ad un altro. Le varie classi contenitore sono state introdotte a partire dalla release 1.2, nel contesto del cosiddetto **collection framework**.

::.. java.util (2/6)

Tale framework per i contenitori è fatto da:

» **Interfacce**, ovvero i **tipi di dato astratti** che rappresentano le classi contenitori, permettono di manipolare i contenitori indipendentemente dai dettagli della rappresentazione e sono organizzate a formare una **gerarchia**;

» **Implementazioni**, ovvero **classi concrete** che implementano le interfacce e rappresentano le strutture dati riusabili;

» **Algoritmi**, ovvero i metodi che effettuano delle operazioni sui contenitori, come ordinamento o ricerca. Gli algoritmi sono polimorfici e riusabili, poiché gli stessi metodi possono essere applicati a differenti implementazioni.

::.. java.util (2/6)

L'uso di questo framework presenta i seguenti vantaggi:

- Riduce lo sforzo di programmazione e ne incrementa la velocità e qualità dello sviluppo;
- Riduce il tempo di apprendimento e sviluppo di nuove API;
- Aumenta il riuso di software;

ma anche i seguenti svantaggi:

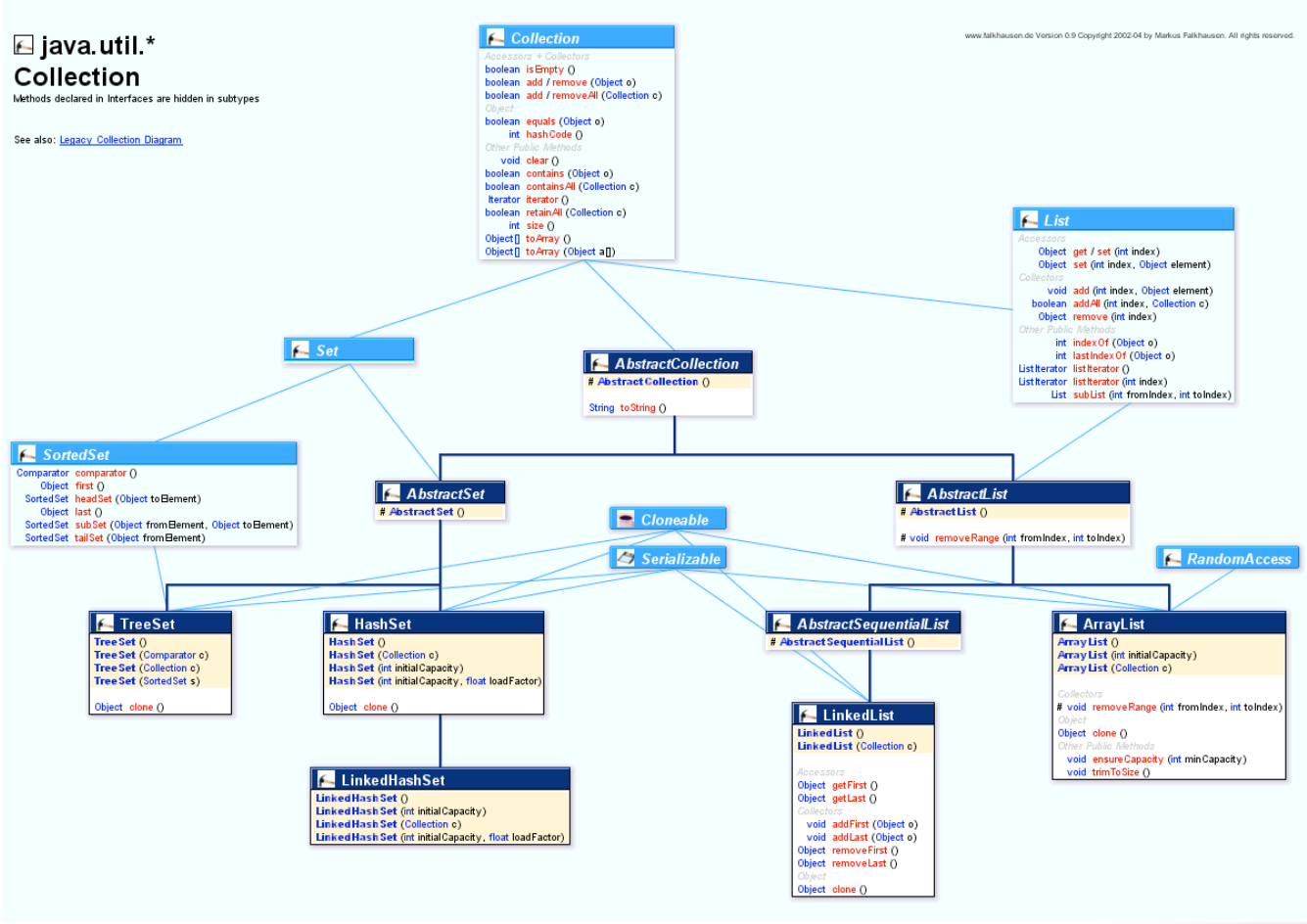
- Generalmente sono abbastanza complesse;
- Le classi contenitore di Java sono abbastanza semplici.

La libreria scinde la "gestione degli oggetti" in due concetti:

- **Collection** - una raccolta **sequenziale** di singoli elementi ai quali sono applicate una o più regole (**List**, **Set**, **Queue**);
- **Map** - un gruppo di **coppie chiave-valore** indicanti oggetti, per permettono di recuperare un valore mediante la chiave ad esso associata.

::.. java.util (3/6)

- Collection - una raccolta sequenziale di singoli elementi ai quali sono applicate una o più regole (List, Set, Queue);



::.. java.util (3/6)

- Collection - una raccolta sequenziale di singoli elementi ai quali sono applicate una o più regole (List, Set, Queue);

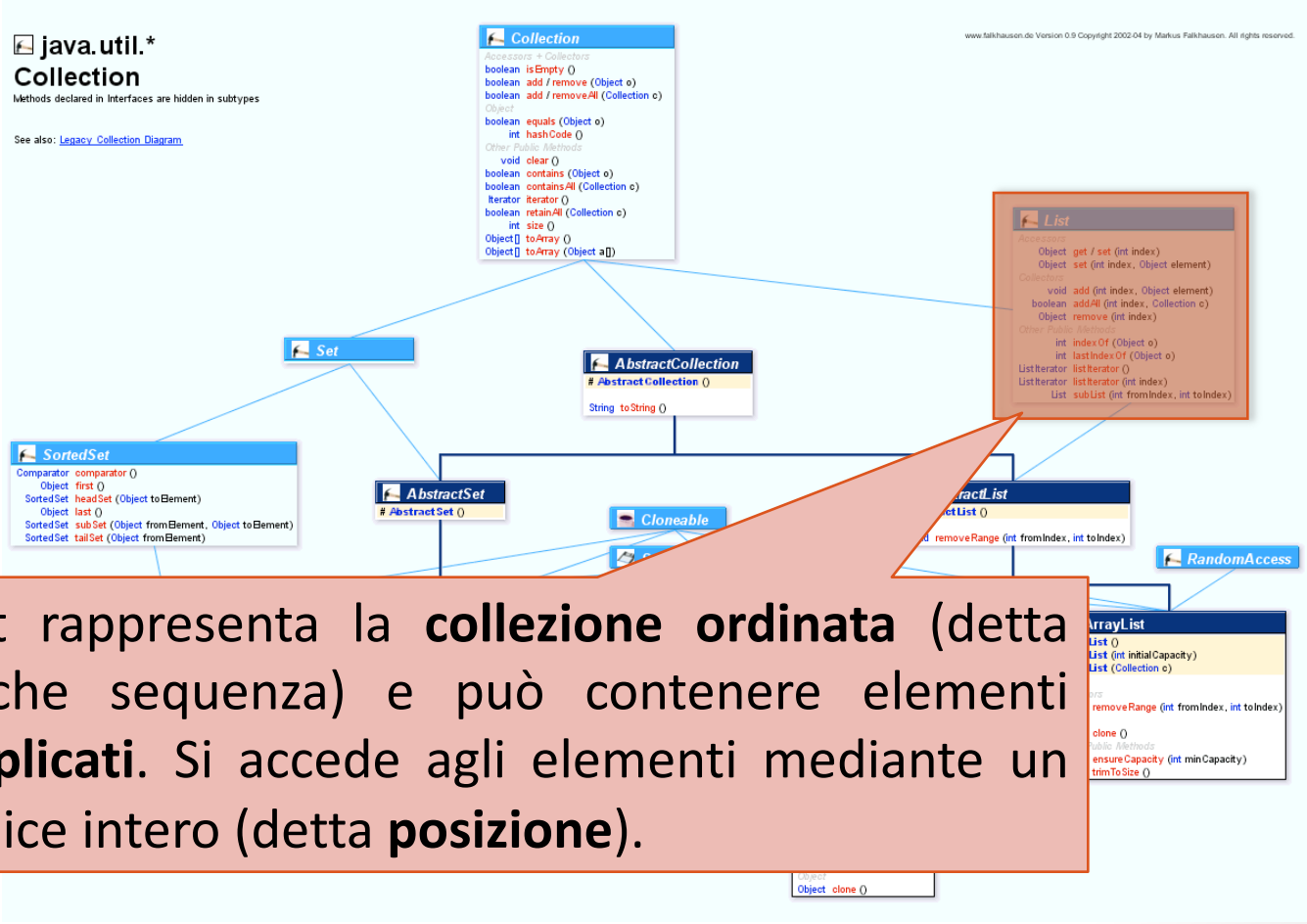
Collection è l'interfaccia radice di una gerarchia di classi per le sequenze di elementi e rappresenta il minimo comun denominatore che tutte le collezioni implementano. Alcune implementazioni ammettono **duplicati** altre no, oppure alcune offrono l'**ordinamento** degli elementi altre no. JDK non ha implementazioni di tale interfaccia, ma vengono fornite implementazioni delle sue sotto-interfacce come Set e List.

::.. java.util (3/6)

- Collection - una raccolta sequenziale di singoli elementi ai quali sono applicate una o più regole (List, Set, Queue);

::.. java.util (3/6)

- Collection - una raccolta sequenziale di singoli elementi ai quali sono applicate una o più regole (List, Set, Queue);

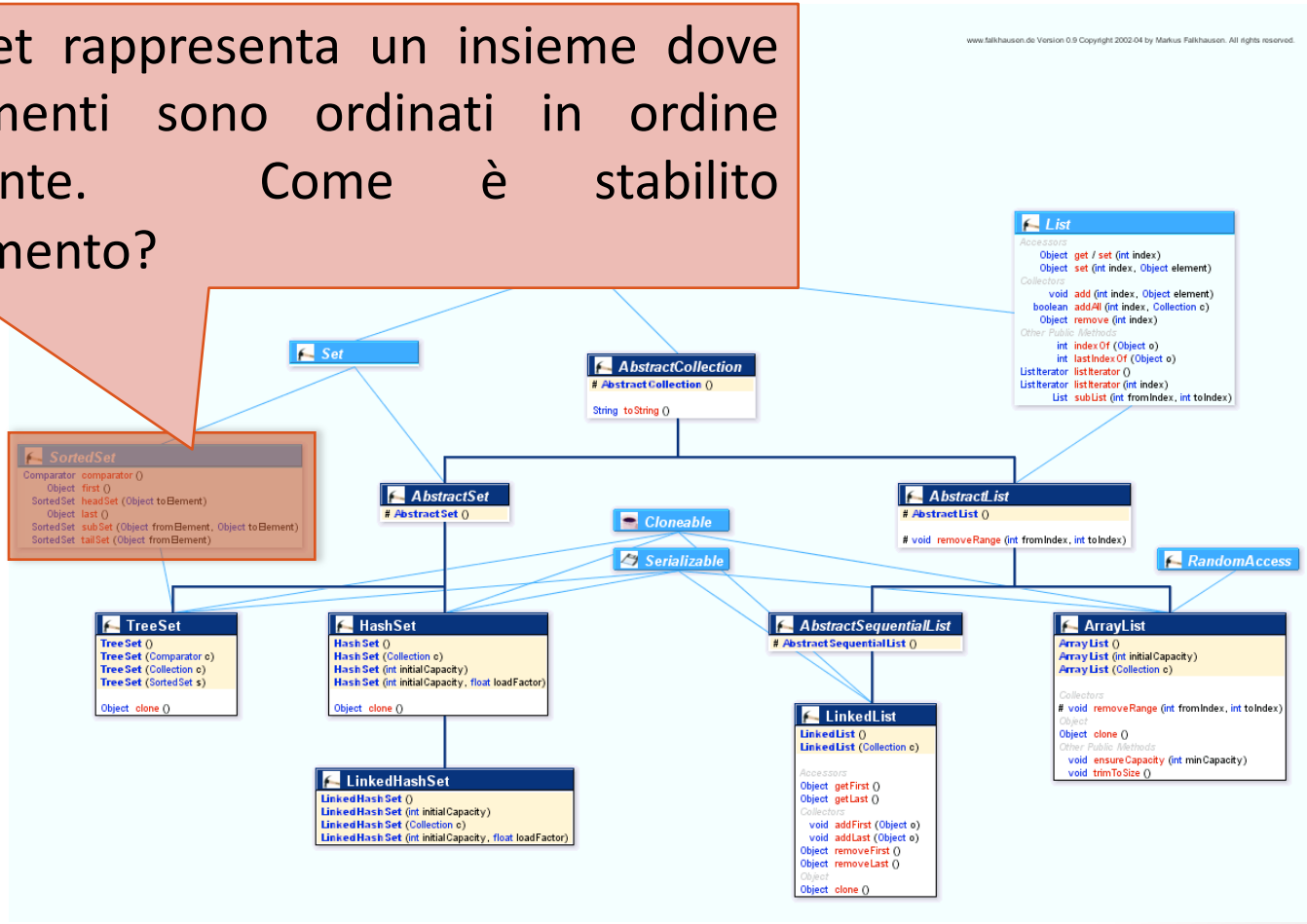


List rappresenta la **collezione ordinata** (detta anche sequenza) e può contenere elementi **duplicati**. Si accede agli elementi mediante un indice intero (detta **posizione**).

::.. java.util (3/6)

- Collection - una raccolta sequenziale di singoli elementi ai quali sono applicate una o più regole (List, Set, Queue);

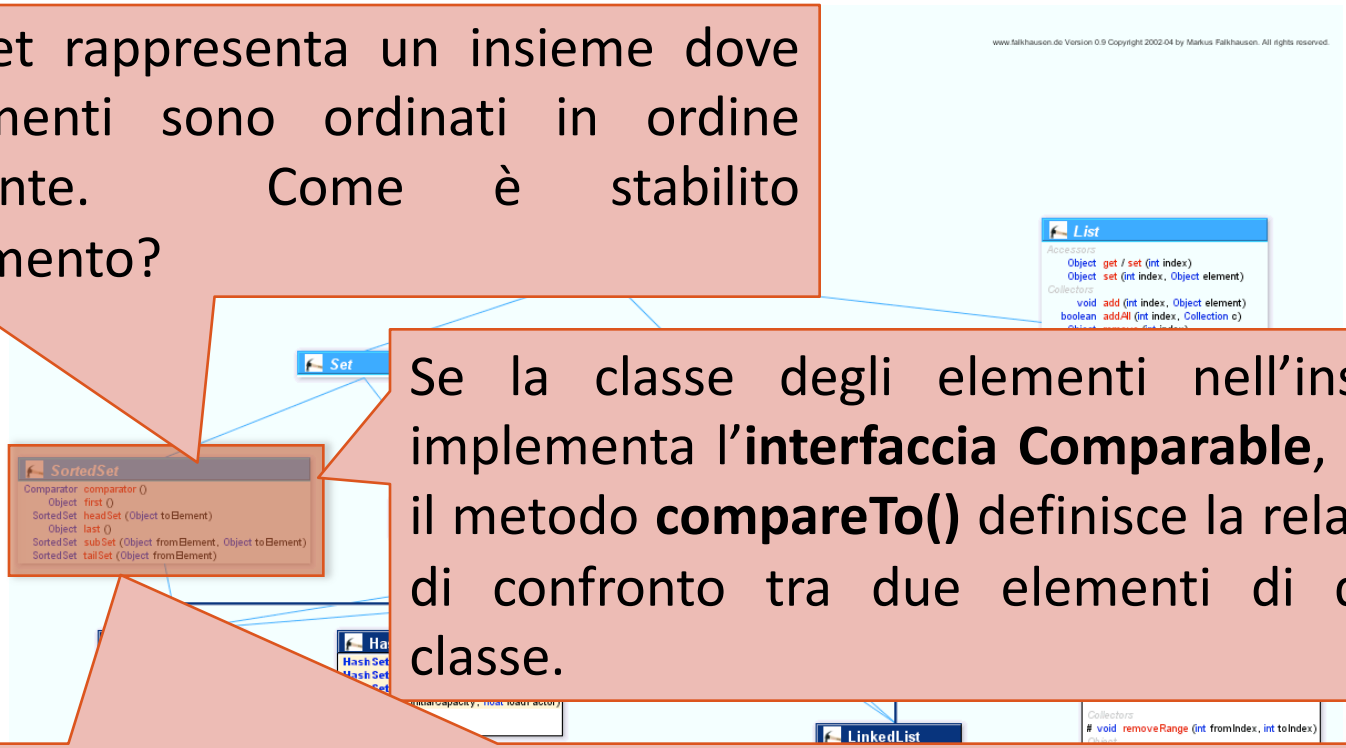
SortedSet rappresenta un insieme dove gli elementi sono ordinati in ordine ascendente. Come è stabilito l'ordinamento?



::.. java.util (3/6)

- Collection - una raccolta sequenziale di singoli elementi ai quali sono applicate una o più regole (List, Set, Queue);

SortedSet rappresenta un insieme dove gli elementi sono ordinati in ordine ascendente. Come è stabilito l'ordinamento?

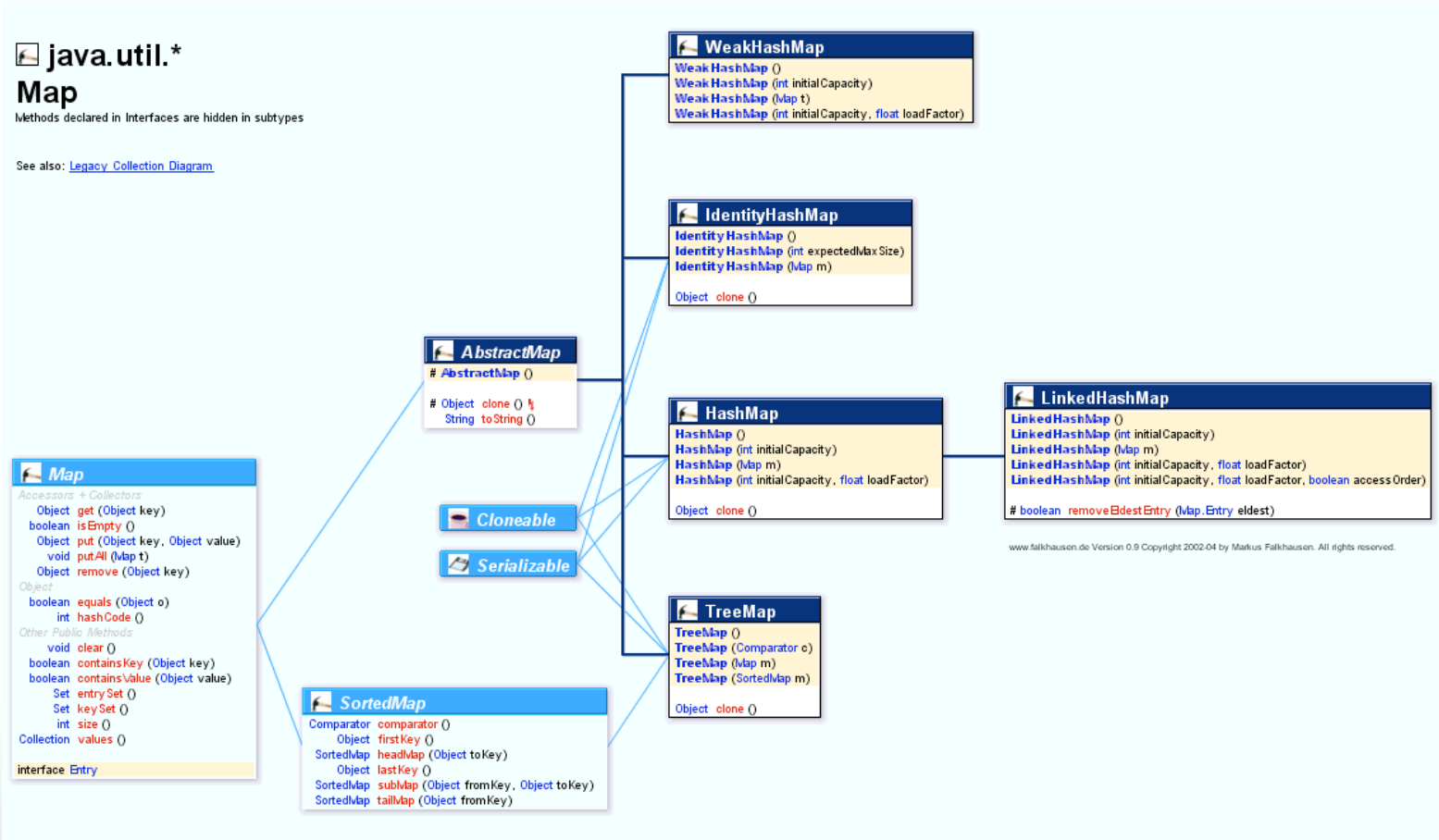


Se la classe degli elementi nell'insieme implementa l'**interfaccia Comparable**, allora il metodo **compareTo()** definisce la relazione di confronto tra due elementi di quella classe.

Oppure bisogna esplicitare la relazione di confronto fornendo all'insieme un oggetto che implementa l'**interfaccia Comparator**.

::.. java.util (4/6)

- Map - un gruppo di coppie chiave-valore indicanti oggetti, per permettono di recuperare un valore mediante la chiave ad esso associata.



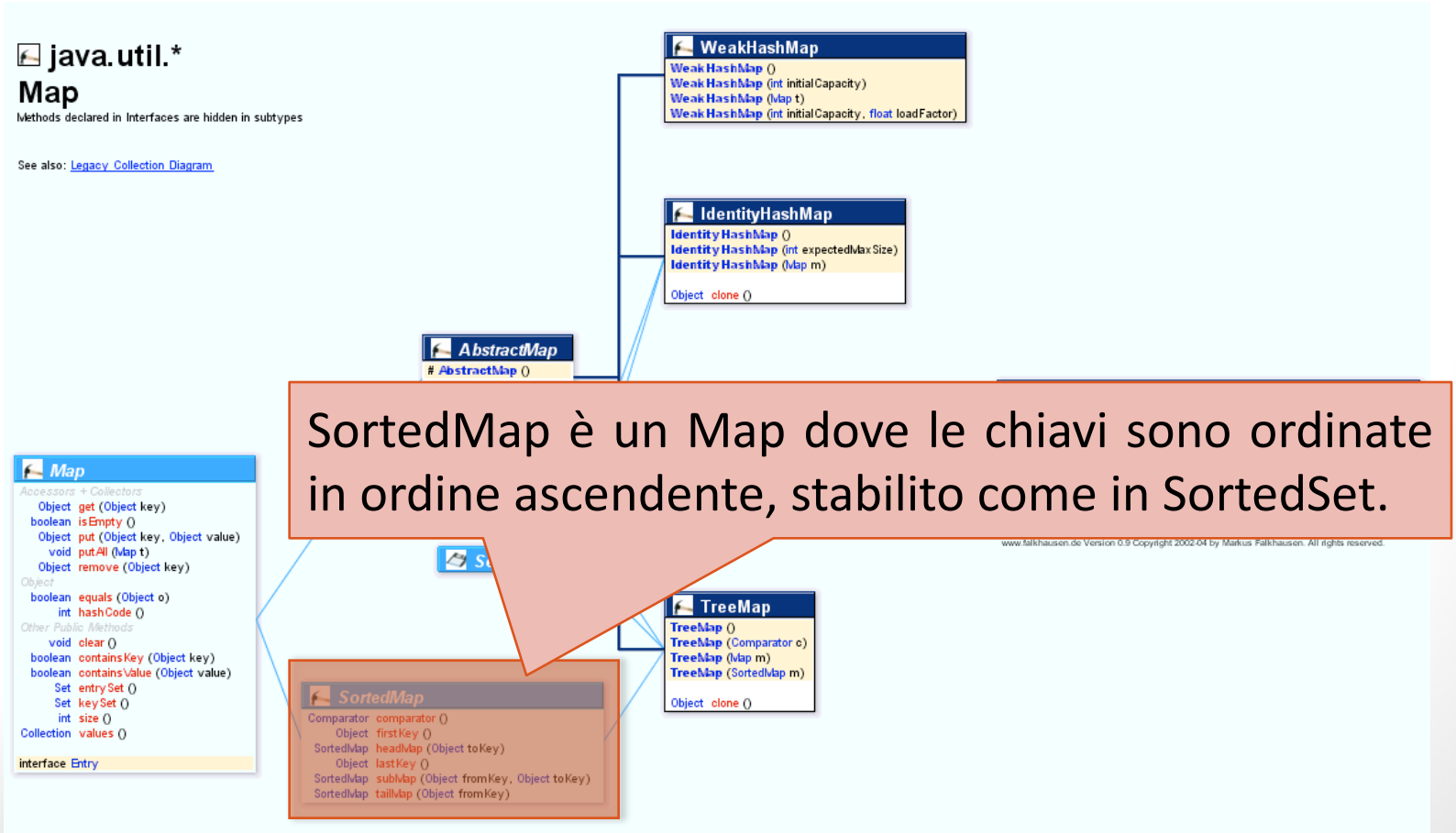
::.. java.util (4/6)

- Map - un gruppo di coppie chiave-valore indicanti oggetti, per permettono di recuperare un valore mediante la chiave ad esso associata.

Map è l'interfaccia radice delle classi che rappresentano una raccolta di oggetti che mappano una chiave ad un valore, e non possono contenere chiavi duplicate ovvero una chiave mappa un solo valore.

::.. java.util (4/6)

- Map - un gruppo di coppie chiave-valore indicanti oggetti, per permettono di recuperare un valore mediante la chiave ad esso associata.



::.. java.util (4/6)

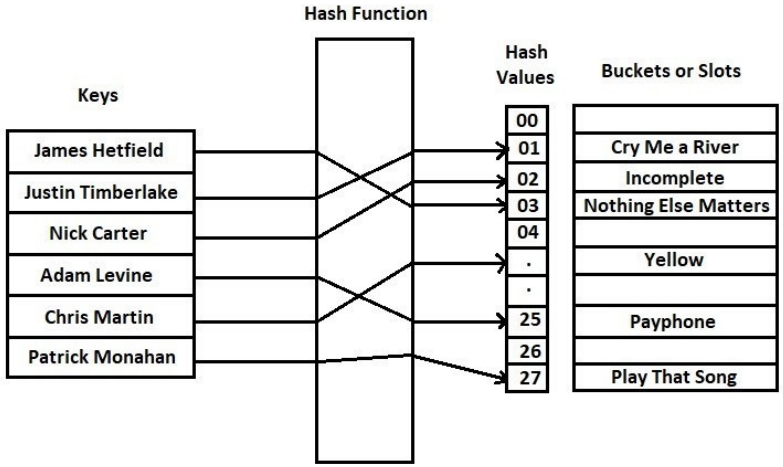
Di seguito sono riportate le principali implementazioni di Set, List e Map:

Interfacce	Implementazioni			
	Hash Table	Resizable Array	Balanced Tree	Linked List
Set	HashSet		TreeSet	
List		ArrayList		LinkedList
Map	HashMap		TreeMap	

Sono presenti almeno due implementazioni per ogni interfaccia. Le implementazioni primarie sono HashSet, ArrayList e HashMap.
TreeSet e TreeMap implementano rispettivamente SortedSet e SortedMap.

In aggiunta, sono presenti due classi, **Vector** e **Hashtable precedenti all'introduzione del framework**, che sono state mantenute e modificate per implementare le interfacce del framework.

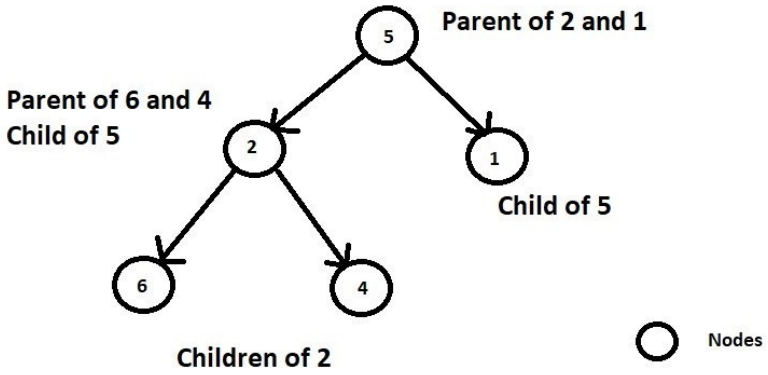
::.. java.util (4/6)



HashSet e **HashMap** sono basate su una struttura dati detta «hash»:

- In **HashMap**, quando si accede ad una chiave «K», la classe calcola una **funzione di hash «F(K)»** sulla rappresentazione **binaria** di «K»
- «F(K)» produce uno **scalare «i»**, che rappresenta l'indice di una tabella. L'elemento i-esimo è il valore «V» della coppia «K,V»
- È possibile che più chiavi «K1» e «K2» siano associati alla stesso slot della tabella (F(K1)==F(K2)). In questo caso («**collisione**»), i valori sono inseriti in una linked list (tante linked list per ogni slot).
- In **HashSet**, la funzione di hash «F(X)» viene calcolata per determinare se l'oggetto «X» è presente o no nel Set

::.. java.util (4/6)



TreeSet e **TreeMap** utilizzando una struttura dati ad albero

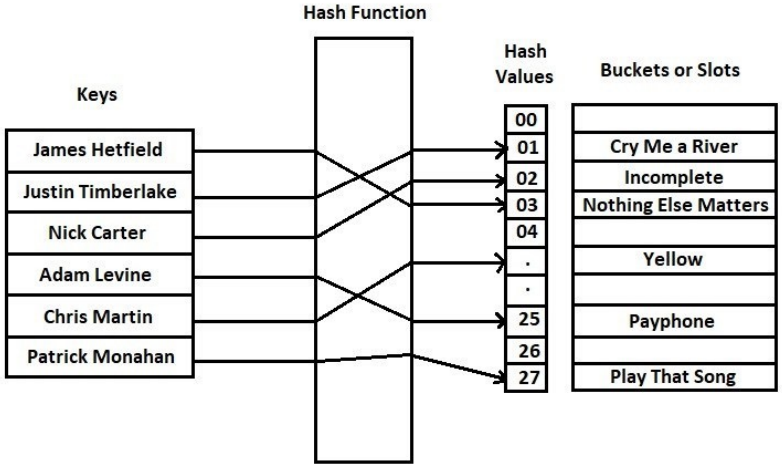
•Quando si aggiunge un elemento, gli algoritmi di inserimento fanno in modo che i nodi siano «**ordinati**» (gli elementi sulla sinistra precedono quelli sulla destra)

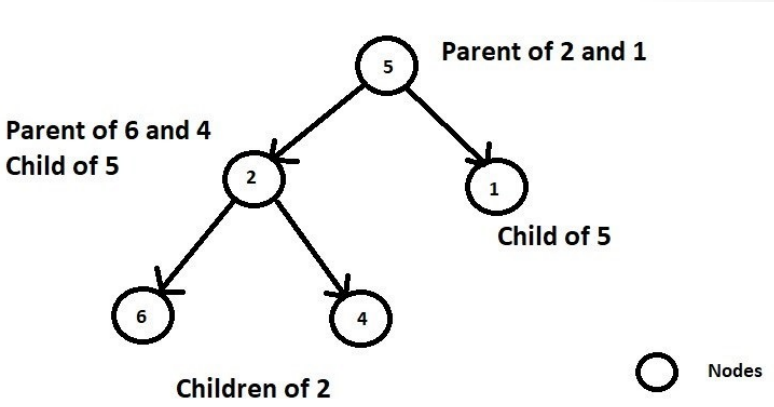
•In TreeMap, si ordina in base alla chiave della coppia «K,V», in TreeSet si ordina in base all'oggetto «X» inserito

•In virtù dell'ordinamento, la ricerca consiste nel percorrere un solo cammino **dalla radice fino al più a una delle foglie**

•Il tempo di accesso è proporzionale al numero di «livelli» dell'albero

::.. java.util (4/6)





HashMap: memorizzazione tabellare

- Tempo di accesso random **costante** (non aumenta con il numero di elementi)
- Visita non ordinata
- Lento nel ridimensionamento (richiede di ricopiare la tabella)

TreeMap: memorizzazione ad albero

- Tempo di accesso random **veloce, ma non costante** (aumenta con il numero di elementi)
- Visita ordinata
- Veloce nel ridimensionamento

CONTENITORI – COLLECTION

::.. Collection (1/2)

Metodi offerti:

- **int size()** – restituisce la dimensione della collezione;
- **boolean isEmpty()** – restituisce true se la collezione non ha elementi, false in caso contrario;
- **boolean contains(Object element)** – verifica che element è incluso nella collezione;
- **boolean add(Object element)** – aggiunge element alla collezione;
- **boolean remove(Object element)** – rimuove element dalla collezione;
- **boolean containsAll(Collection c)** – verifica se c è inclusa nella collezione
- **boolean addAll(Collection c)** – aggiunge tutti gli elementi di c nella collezione;
- **boolean removeAll(Collection c)**; - rimuove tutti gli elementi di c dalla collezione;
- **boolean retainAll(Collection c)** – mantiene nella collezione solo gli elementi che sono anche in c;

::.. Collection (2/2)

- **void clear ()** – elimina tutti gli elementi dalla collezione;
- **Object[] toArray()** - restituisce gli elementi della collezione come un array;
- **Iterator iterator()** – fornisce un iteratore alla collezione, definito per mezzo della seguente interfaccia:

```
interface Iterator  {
    boolean hasNext();      // Verificare se esiste un prossimo elemento
    Object next();          // Restituisce l'elemento corrente
    void remove();          // Elimina l'elemento corrente (opzionale)
}
```

Tale interfaccia è specializzata per la collection specifica di interesse:

```
interface ListIterator extends Iterator { … }
```

- Per convenzione tutte le implementazioni di Collection hanno un **costruttore** con argomento un'istanza di un'implementazione di Collection, per l'inizializzazione di una collezione a partire da un'altra:

```
List l = new ArrayList(c);     // Con c oggetto di una qualunque collezione
```

::.. Funzioni di utilità - Collections

Collections (da non confondere con l'interfaccia **Collection**) è una classe con **metodi statici** che offrono un ampio spettro di funzionalità impiegabili sulle collezioni:

- **boolean addAll(Collection c, T... elements)** – per aggiungere elementi ad una data collection c;
- **int binarySearch(List list, Object key[, Comparator c])** – per ricercare un elemento key in una lista, fornendo anche un comparatore;
- **void copy(List dest, List src)** – per copiare gli elementi della lista src in quella dest;
- **boolean disjoint(Collection c1, Collection c2)** – per verificare che due collection non abbiano elementi in comune;
- **void fill(List list, Object obj)** – per sostituire tutti gli elementi in list con obj;
- **int frequency(Collection c, Object o)** – per contare le occorrenze nella collection c del valore o;
- **Object max(Collection coll[, Comparator comp])** – per determinare il valore massimo in una collection; esiste anche il corrispettivo «min».

::.. Funzioni di utilità - Collections

- **boolean replaceAll(List list, Object oldVal, Object newVal)** – per sostituire un determinato elemento in list con un nuovo valore;
- **void reverse(List list)** – per invertire l'ordine degli elementi in list;
- **void sort(List list[, Comparator c])** – per ordinare gli elementi in una lista dato esplicitamente, o meno, un criterio di comparazione;
- **void swap(List list, int i, int j)** – per scambiare di posto gli elementi di posizione i e j in una lista list.

CONTENITORI – LIST

::.. List

List realizza una collezione di elementi in cui possono esserci elementi duplicati, e rispetto a Collection ci sono delle operazioni aggiuntive:

•**Accesso posizionale** (si parte da 0) ovvero manipolazione degli elementi in base alla posizione nella lista:

Object **get**(int index);

•**Ricerca di un determinato oggetto** e ritorno della posizione numerica:

int **indexOf**(Object o);

•Estrazione di sotto-liste:

List **subList**(int fromIndex, int toIndex);

Le possibili implementazioni di questa interfaccia sono **ArrayList**, **LinkedList** e **Vector**. LinkedList dispone di metodi di inserimento ed estrazione impiegabili per la realizzazione di code e pile. Esistono l'interfaccia Deque per la realizzazione di pile e code con classi ad hoc.

Due Liste sono uguali se gli elementi sono gli stessi nello stesso ordine.

::.. Iterare su List

```
public class ListTest {
   public static void main( String[] args ) {
            List l = new ArrayList();      // ArrayList «is-a» List
            l.add( "ten" );
            l.add( "eleven" );
            System.out.println( l.get(1) );          // stampa «eleven»

            for (int i = 0; i < l.size(); i++ ) {
                     String element = ( String ) l.get(i);
                     System.out.println( "Elemento :" + element );
            }
   }
}
```

::.. Iterare su List

```
public class ListTest {
   public static void main( String[] args ) {
            List l = new ArrayList();
            l.add( "ten" );
            l.add( "eleven" );
            System.out.println( l.get(1) );

            Object[] array  = l.toArray();
            for (int i = 0; i < array.length; i++ ) {
                     String element = ( String ) array[ i ];
                     System.out.println( "Elemento di array:" + element );
            }
   }
}
```

Nota: il codice

String[] str = (String[]) c.toArray();

farebbe sorgere un'eccezione di tipo **ClassCastException**, perchè in Java non è possibile convertire un Object[] in un T[]

::.. Iterare su List

```
public class ListTest {
   public static void main( String[] args ) {
            List l = new ArrayList();
            l.add( "ten" );
            l.add( "eleven" );
            System.out.println( l.get(1) );

            Object[] objectArray  = l.toArray();
            String[] stringArray = Arrays.copyOf(objectArray,
                                        objectArray.length, String[].class);

            for ( int i = 0; i < stringArray.length; i++ ) {
                        String element = stringArray[ i ];
                        System.out.println( "Elemento di str: " + element );
            }
   }
}
```

Copia gli elementi in un nuovo array dello stesso tipo indicato in ingresso

::.. Iterare su List

```
public class ListTest {
   public static void main( String[] args ) {
            List l = new ArrayList();
            l.add( "ten" );
            l.add( "eleven" );
            System.out.println( l.get(1) );

            for ( Iterator i = l.iterator(); i.hasNext(); /*vuoto*/) {
                        String element = ( String ) i.next();
                        System.out.println( "Elemento di array:" + element );
            }
   }
}
```

NOTA: Manca la terza parte del "for" (è il metodo "next()" ad avanzare l'iteratore)

::.. Iterare su Collection

```
public class CollectionsTest {
   public static void main( String[] args ) {
            Collection c = new ArrayList();  // ArrayList «is-a» Collection
            c.add( "ten" );
            c.add( "eleven" );
            // il metodo «c.get(1)» non è invocabile su Collection

            for ( Iterator i = c.iterator(); i.hasNext(); /*vuoto*/) {
                        String element = ( String ) i.next();
                        System.out.println( "Elemento di array:" + element );
            }
   }
}
```

La classe Iterator è utilizzabile su **qualunque Collection**

::.. Iterare su Collection

Le classi contenitori in Java contengono oggetti di tipo **Object** e sue sotto-classi:

- boolean **contains**(Object element);
- boolean **add**(Object element);
- boolean **remove**(Object element);

E' necessario effettuare un **casting** quando si recupera l'oggetto dalla collezione, ed è possibile mischiare tipi eterogenei nella collezione.

Non è possibile inserire valori di tipi primitivi, ma è necessario utilizzare **tipi wrapper** (Integer, Long e altri).

Java 5 risolve tali problemi utilizzando i **generics**, con le collezioni parametrizzate sul tipo dei dati da contenere:

```
List<Integer> l = new ArrayList<Integer>();
```

CONTENITORI - SET

::.. Set

Set dispone degli stessi metodi dell'interfaccia Collection, ma sono **proibiti elementi duplicati e nulli** nella sequenza.

Possibili implementazioni sono **HashSet**, **TreeSet** e **LinkedHashSet** dove si usa:

- **HashSet** per i Set in cui la velocità di consultazione è importante;
- **TreeSet** per mantenere un Set ordinato sostenuto da una struttura ad albero (è più efficiente estendere la struttura);
- **LinkedHashSet** possiede velocità di consultazione comparabile ad un HashSet, ma mantiene inoltre l'ordine nel quale gli elementi sono stati inseriti utilizzando internamente una lista concatenata.

::.. Set

```
import java.util.*;
public class FindDups  {
    public static void main(String args[]) {
        Set s = new HashSet();
        for (int i=0; i<args.length; i++)  {
            if (s.contains(args[i]))
                    System.out.println("Duplicate detected: "+args[i]);
            else
                    s.add(args[i]);
        }
        System.out.println( s.size() + " distinct words detected: " + s );
    }
}
```

C:> **java FindDups i came i saw i left**

OUTPUT

Duplicate detected: i

Duplicate detected: i

4 distinct words detected: **[came, left, saw, i]**

::.. Set

NOTA: Modificando HashSet in TreeSet, si ottiene l'ordinamento

```
import java.util.*;
public class FindDups  {
    public static void main(String args[]) {
        Set s = new TreeSet();
        for (int i=0; i<args.length; i++)  {
            if (s.contains(args[i]))
                    System.out.println("Duplicate detected: "+args[i]);
            else
                    s.add(args[i]);
        }
        System.out.println( s.size() + " distinct words detected: " + s );
    }
}
```

C:> **java FindDups i came i saw i left**

OUTPUT

Duplicate detected: i

Duplicate detected: i

4 distinct words detected: **[i, came, left, saw]**

::.. Set

```
import java.util.*;
public class FindDups  {
    public static void main(String args[]) {
        Set s = new TreeSet();
        for (int i=0; i<args.length; i++)  {
            if (s.contains(args[i]))
                    System.out.println("Duplicate detected: "+args[i]);
            else
                    s.add(args[i]);
        }

        for ( Iterator i = s.iterator(); s.hasNext(); /*vuoto*/) {
            String word = ( String ) i.next();
            System.out.println( "Word: " + word );
        }
    }
}
```

Anche TreeSet può essere ispezionata usando gli iteratori

CONTENITORI - MAP

::.. Map (1/5)

Metodi offerti:

- **Object put(Object key, Object value)** – per inserire una coppia chiave valore nella mappa ;
- **Object get(Object key)** – per ottenere un valore mappato dalla chiave key;
- **Object remove(Object key)** – per rimuovere la coppia identificata dalla chiave key;
- **boolean containsKey(Object key)** – per verificare se una chiave è presente nella mappa;
- **boolean containsValue(Object value)** – per verificare se un valore è presente nella mappa;
- **int size()** – per ritornare la dimensione della mappa;
- **boolean isEmpty()** – per verificare se la mappa ha elementi;
- **void putAll(Map t)** – per inserire in una mappa tutti gli elementi di un'altra mappa;
- **void clear()** – per eliminare tutti gli elementi;
- **Set keySet()** – per ottenere l'insieme di tutte le chiavi nella mappa;

::.. Map (2/5)

- **Collection values()** – per ottenere la lista dei valori nella mappa;
- **Set entrySet()** – per avere la sequenza di oggetti che rappresentano la coppia chiave valore modellata per mezzo di:

```
public interface Entry {
    Object getKey();
    Object getValue();
    Object setValue(Object value);
}
```

Non può contenere **duplicati delle chiavi**, e presenta tre diverse implementazioni: **HashMap**, **TreeMap**, e **Hashtable**. HashMap è da preferire rispetto ad Hashtable perché fornisce prestazioni costanti per l'inserimento e la ricerca di qualunque coppia. TreeMap è basata su una struttura ad albero e mantiene ordinati i dati.

Come le collezioni, anche le mappe hanno un costruttore con argomento una mappa per l'inizializzazione di una nuova mappa con gli elementi di quella specificata in ingresso.

::.. Map (3/5)

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

        Map map  = new HashMap();

        map.put( "key1", "value1" );
        map.put( "key2", "value2" );
        map.put( "key3", "value1" );

        Object value = map.get( "key1" );
        System.out.println( "Valore: " + value );
        System.out.println( "Valore: " + map.get( "key2" ) );

    }
}
```

::.. Map (4/5)

```
public class Freq {
    private static final Integer ONE = newInteger(1);
    public static void main(String args[]) {
        Map m = new HashMap();
        // Inizializza la tabella di frequenza dagli argomenti del main
        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? ONE : new Integer(freq.intValue() +
                    1)));
        }
        System.out.println( m.size() +" distinct words detected:");
        System.out.println(m);
} }
```

C:> java Freq if it is to be it is up to me to delegate

OUTPUT

8 distinct words detected:

{to=3, me=1, delegate=1, it=2, is=2, if=1, be=1, up=1}

::.. Map (5/5)

I valori in una mappa sono accessibili per mezzo delle chiavi, o anche per mezzo di appositi iteratori:

```
// itera su tutte le chiavi
for (Iterator i=m.keySet().iterator(); i.hasNext(); ) {
    System.out.println(i.next());
}

// itera su tutti i valori
for (Iterator i=m.entrySet().iterator(); i.hasNext(); /*vuoto*/) {
    Map.Entry e = (Map.Entry) i.next();
    System.out.println(e.getKey() + ": " + e.getValue());
}

//m2 sotto-mappa di m1
if (m1.entrySet().containsAll(m2.entrySet())) {
    ...
}
```

:: Foreach

Il costrutto «foreach» è una sintassi alternativa e semplificativa:

```
// itera sugli elementi di una lista
// dichiarata come «ArrayList<Elemento> mylist»
for ( Elemento e : mylist ) {
    System.out.println(e);
}

// itera su tutte le coppie di una map
// dichiarata come «HashMap<Chiave, Elemento> mydict»
for ( Map.Entry<Chiave,Elemento> e : dict.entrySet() ) {

    System.out.println(e.getKey() + ": " + e.getValue());
}
```