

Sezione: Modificatori

Modificatori

Modificatori: visibilità

public

(non def.)

visibile da tutti

visibile da tutti nello stesso package

protected

visibile dalle sottoclassi

private

nascosta da tutti

Uso di metodi “di accesso”:

```
public class ACorrectClass {  
    private String aUsefulString;  
    public String getAUsefulString() {  
        // "get" the value  
        return aUsefulString;  
    }  
    private protected void setAUsefulString(String s)  
    {  
        // "set" the value  
        aUsefulString = s;  
    }  
}
```

Matrice degli accessi

Access Levels				
Specifier	Class	Package	Subclass	World
private	Y	N	N	N
no specifier	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

Vedi anche

<http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

I parametri del
main sono inclusi
in un vettore di
String

Parametri di ingresso

```
/* sum and average command lines */  
class SumAverage {  
    public static void main (String args[]) {  
        int sum = 0;  
        float avg = 0;  
        for (int i = 0; i < args.length; i++) {  
            sum += Integer.parseInt(args[i]);  
        }  
        System.out.println("Sum is: " + sum);  
        System.out.println("Average is: "  
            + (float)sum / args.length);  
    }  
}
```

Convenzioni

I nomi delle **Classi** iniziano con la MAIUSCOLA

I nomi degli **Oggetti** iniziano con la
MINUSCOLA

```
Pila p=new Pila();
```

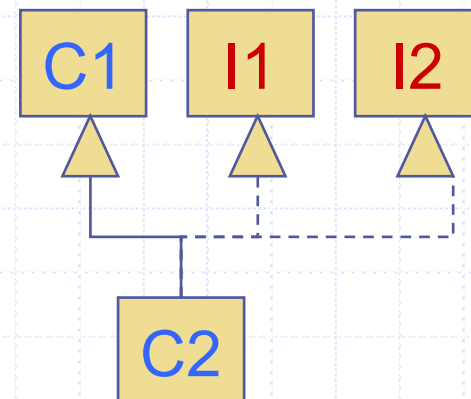


Interfacce

Interfacce

Un *interface* è una collezione di firme di metodi (senza implementazione).

Una interfaccia può dichiarare costanti.



Esempio di interface

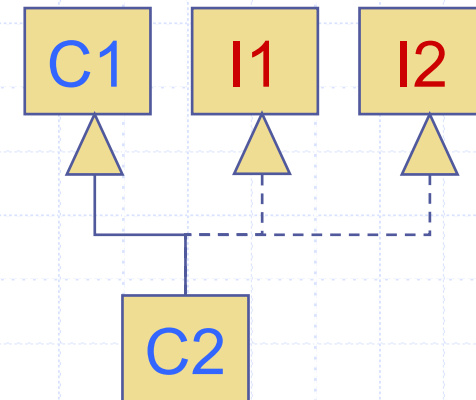
```
package strutture;  
public interface Stack{  
    public int estrai();  
    public void insert(int z);  
}
```

```
package strutture;  
public class Pila implements Stack{  
    ...  
}
```

```
package strutture;  
public class Coda extends Pila{  
    ...  
}
```


Interfacce

Le interfacce possono essere usate come
“tipi”



```
I1 x = new C2();
```

```
// I1 x = new I1(); NO!!
```



Collections

Vantaggi di usare una libreria

- Ottimizza il lavoro del programmatore che si può concentrare sulle parti di contesto specifico delle sue applicazioni
- Aumenta la velocità di scrittura e la qualità del codice e di nuove API

Vantaggi di usare una libreria

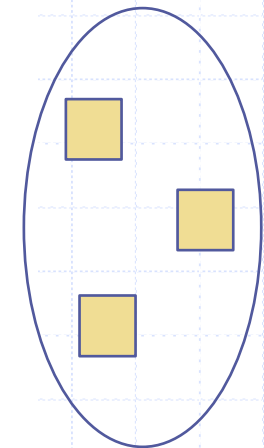
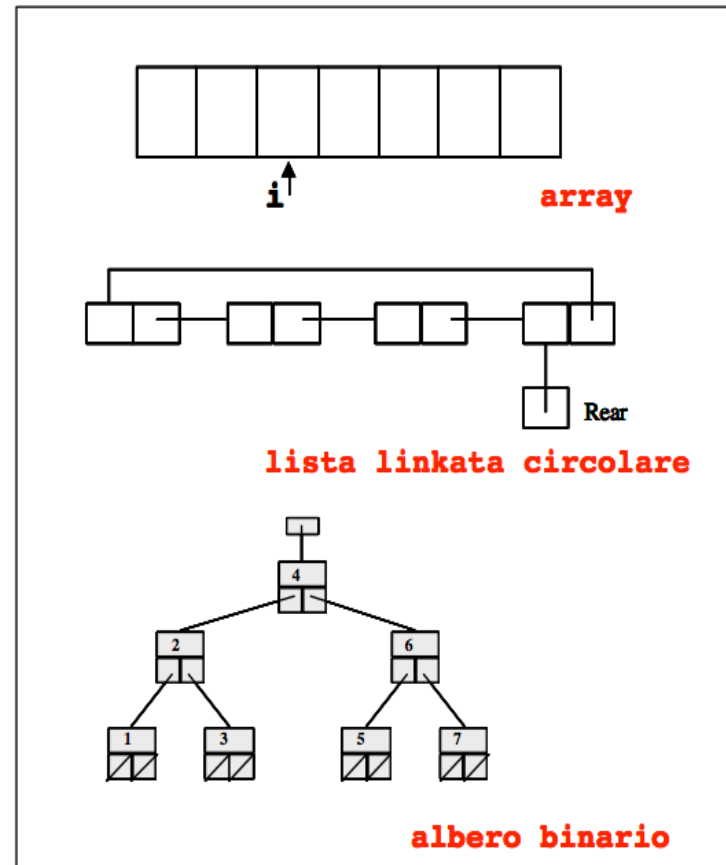
- Permette la interoperabilità tra API non correlate: la maggior parte delle API accetta collections come parametri in input ed in output
- Supporta il riuso di codice: strutture dati nuove che aderiscono al collection framework sono intrinsecamente riusabili

Riuso della conoscenza

Riuso del software

- Algoritmi
- Pattern

Esempi di strutture dati



bag

Collection: Basic operations

int size();

boolean isEmpty();

boolean contains(Object element);

boolean add(Object element);

boolean remove(Object element);

Iterator iterator();

Collections

Una collection è un oggetto che raggruppa elementi multipli (anche eterogenei) in una singola entità.

Collections sono usate per immagazzinare, recuperare e trattare dati, e per trasferire gruppi di dati da un metodo ad un altro.

Tipicamente rappresentano dati che formano gruppi “naturali”, come una mano di poker (una collection di carte), un mail folder (a collection di e-mail), o un elenco telefonico (una collection di mappe nome-numero).

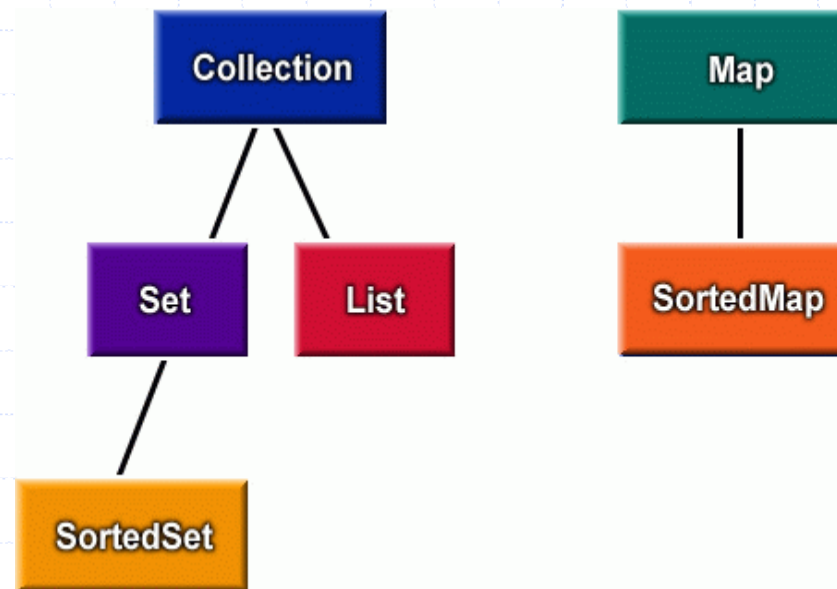
Collections Framework

Il Java Collection Framework contiene tre elementi:

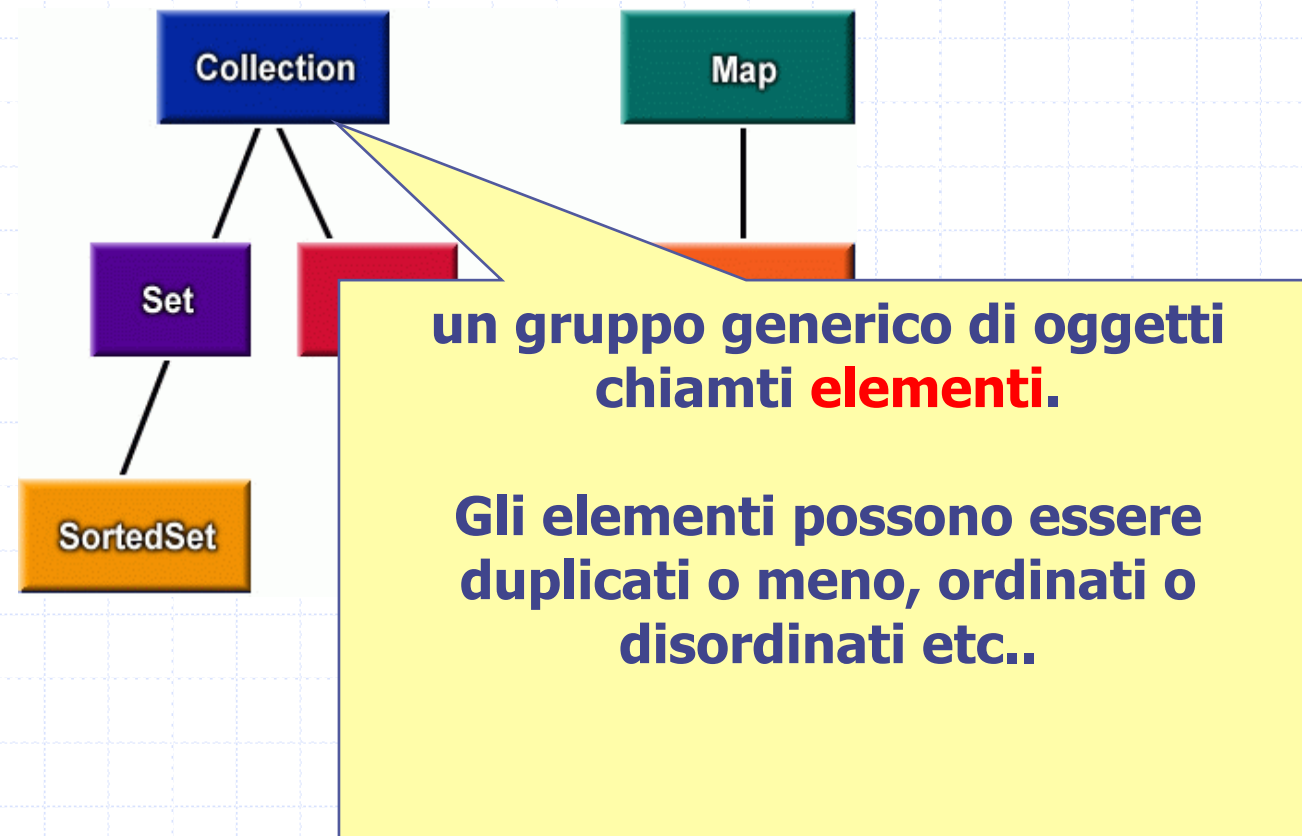
- **Interfacce**
- **Implementazioni** concrete delle interfacce precedenti;
- **Algoritmi**: metodi che implementano operazioni comuni a più strutture dati
 - Esempi: algoritmi di ricerca ed ordinamento: sort, shuffle, binarySearch, max, min...
 - Questi algoritmi sono polimorfi, nel senso che lo stesso metodo può essere usato per in diverse implementazioni concrete delle collections.

Core Collections Interfaces

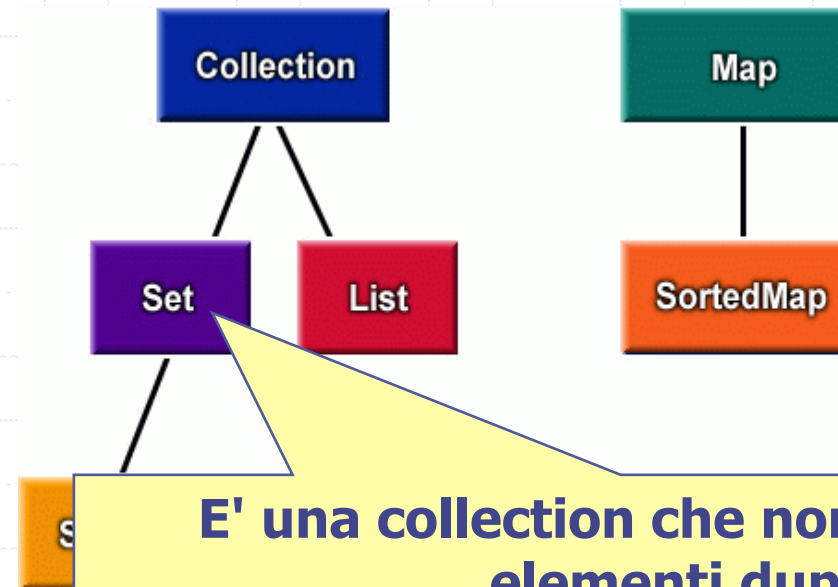
Occorre importare `java.util.*`



Core Collections Interfaces



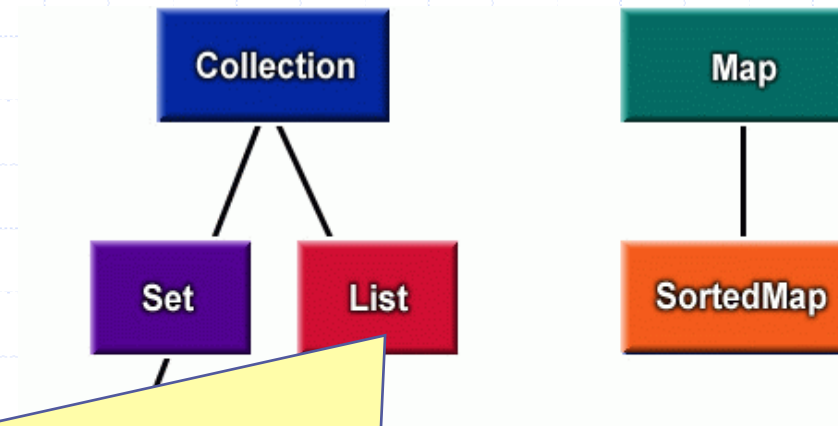
Core Collections Interfaces



E' una collection che non può contenere elementi duplicati

Esempi concreti: l'insieme dei processi che girano su un computer, una mazzo di carte da briscola...

Core Collections Interfaces



List è una collection **ordinata che può contenere **elementi duplicati**.**

**Normalmente fornisce all'utente controllo nell'inserimento.
L'utente accede agli elementi attraverso un **indice (posizione)**.**

Esempi di implementazione: vettori, liste linkate.

Esempi concreti: Mazzo di carte da Scala 40

Differenze tra Set e List

	Elementi duplicati	Elementi accessibili con un indice	Gli elementi mantengono l'ordine di inserimento
Set	NO	NO	NO
List	SI	SI	SI

Collection: Basic operations

int size();

boolean isEmpty();

boolean contains(Object element);

boolean add(Object element);

boolean remove(Object element);

Iterator iterator();

Collection: basic operations

The **add** method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't. It guarantees that the Collection will contain the specified element after the call completes, and returns true if the Collection changes as a result of the call.

The **remove** method is defined to remove *a single instance* of the specified element from the Collection, assuming the Collection contains the element, and to return true if the Collection was modified as a result.

Collection: bulk operations

// Bulk operations

boolean containsAll(Collection c);

boolean addAll(Collection c);

boolean removeAll(Collection c);

boolean retainAll(Collection c);


void clear();

// Array Operations

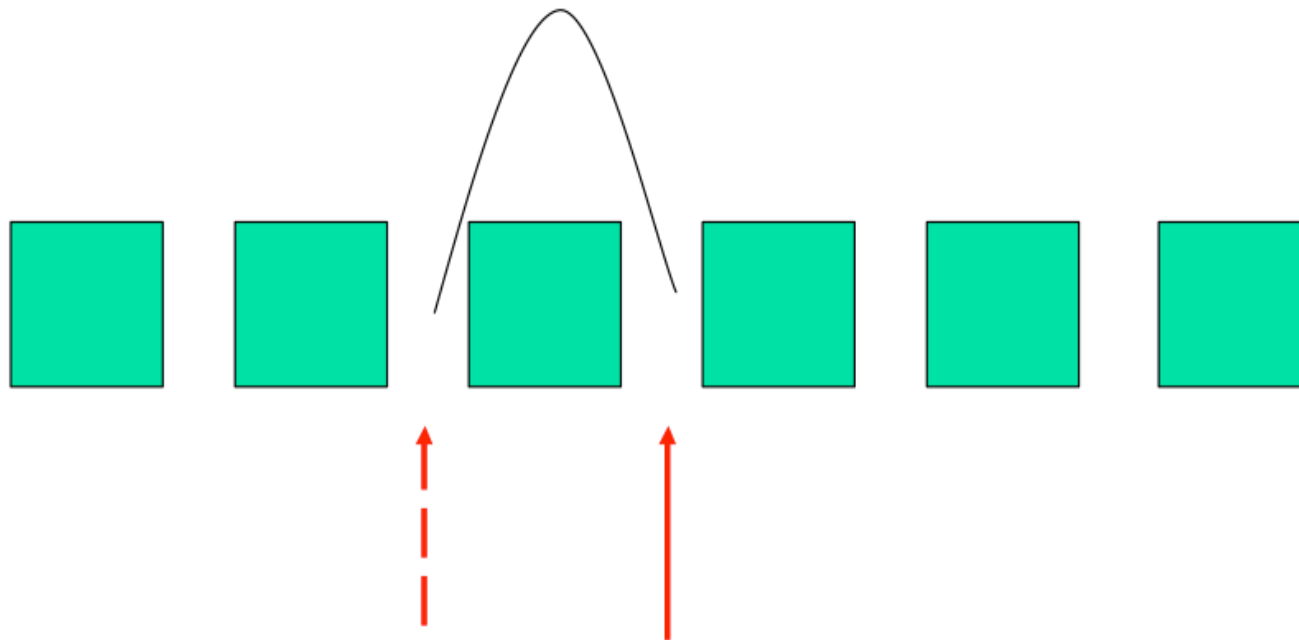
Object[] toArray();

Object[] toArray(Object a[]);

Core Collections Interfaces

		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interface s	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

Iterator



```
Iterator iter = c.iterator();  
obj = iter.next(); // muovi di un elemento e  
                  // restituisci un riferimento associato  
                  // all'elemento appena superato
```

The Iterator interface

```
public interface Iterator {  
    ■ boolean hasNext();  
    ■ Object next();  
    ■ void remove();  
}
```

hasNext returns true if there are more elements in the Collection

next() returns the next element in the Collection

remove() method removes from the underlying Collection the last element that was returned by next. The remove method may be called only once per call to next, and throws an exception if this condition is violated.

Using Iterators

```
void filter(Collection x) {  
    Iterator i=x.iterator();  
    while (i.hasNext()) {  
        if (!cond(i.next()))  
            i.remove();  
    }  
}
```

The code is *polymorphic*: it works for *any* Collection that supports element removal, regardless of implementation. That's how easy it is to write a polymorphic algorithm under the collections framework!

Fondamenti di Java

Implementazione di Pila e Coda
usando le Collection

Number

```
package structures;  
class Number {  
  
    private int n;  
  
    Number(int n) {  
        this.n = n;  
    }  
  
    int getInt() {  
        return n;  
    }  
  
    void setInt(int n) {  
        this.n = n;  
    }  
}
```

Stack

```
package structures;
import java.util.*;
public abstract class Stack extends LinkedList {

    public void inserisci(int x) {
        Number n = new Number(x);
        this.add(n);
    }

    abstract public int estrai();
}
```

Coda

```
class Coda extends Stack {  
  
    public int estrai() {  
        Number x = null;  
        Iterator iter = this.iterator();  
        if (iter.hasNext()) {  
            x = (Number) iter.next();  
            iter.remove();  
        } else {  
            System.out.println("Tentativo di  
                                estrarre da una Coda vuota");  
            System.exit(1);  
        }  
        return x.getInt();  
    }  
}
```


Coda

```
class Pila extends Stack {  
  
    public int estrai() {  
        Number x = null;  
        Iterator iter = this.iterator();  
        while (iter.hasNext()) {  
            x = (Number) iter.next();  
        }  
        if (x == null) {  
            System.out.println("Tentativo di  
                                estrarre da una Pila vuota");  
            System.exit(1);  
        }  
        iter.remove();  
        return x.getInt();  
    }  
}
```

main

```
public static void main(String[] args) {  
    Stack s=new Coda(); // Stack s=new Pila();  
    s.inserisci(1);  
    s.inserisci(2);  
    s.inserisci(3);  
    for (int k=0;k<=4;k++){  
        int v=s.estrai();  
        System.out.println(v);  
    }  
}
```

1
2
3

**Tentativo di estrarre da
una Coda vuota**

3
2
1

**Tentativo di estrarre da
una Pila vuota**

Stack

```
package structures;
import java.util.*;
public abstract class Stack extends ArrayList
{

    public void inserisci(int x) {
        Number n = new Number(x);
        this.add(n);
    }

    abstract public int estrai();
}
```

e se avessi esteso HashSet?

Esercizio

Create le classi Persona, Studente, Docente.
Creare una collezione e popolarla con varie istanze delle classi suddette.

Ispezionare la selezione per estrarre dati secondo un criterio di ricerca dato

Es.: stampare i numeri di matricola di tutti gli studenti il cui nome inizia per "M".