Eco-design Digitale di Base per i servizi ICT

Programmazione in Java

Massimo Giaccone, Luglio 2025

Obiettivi della lezione

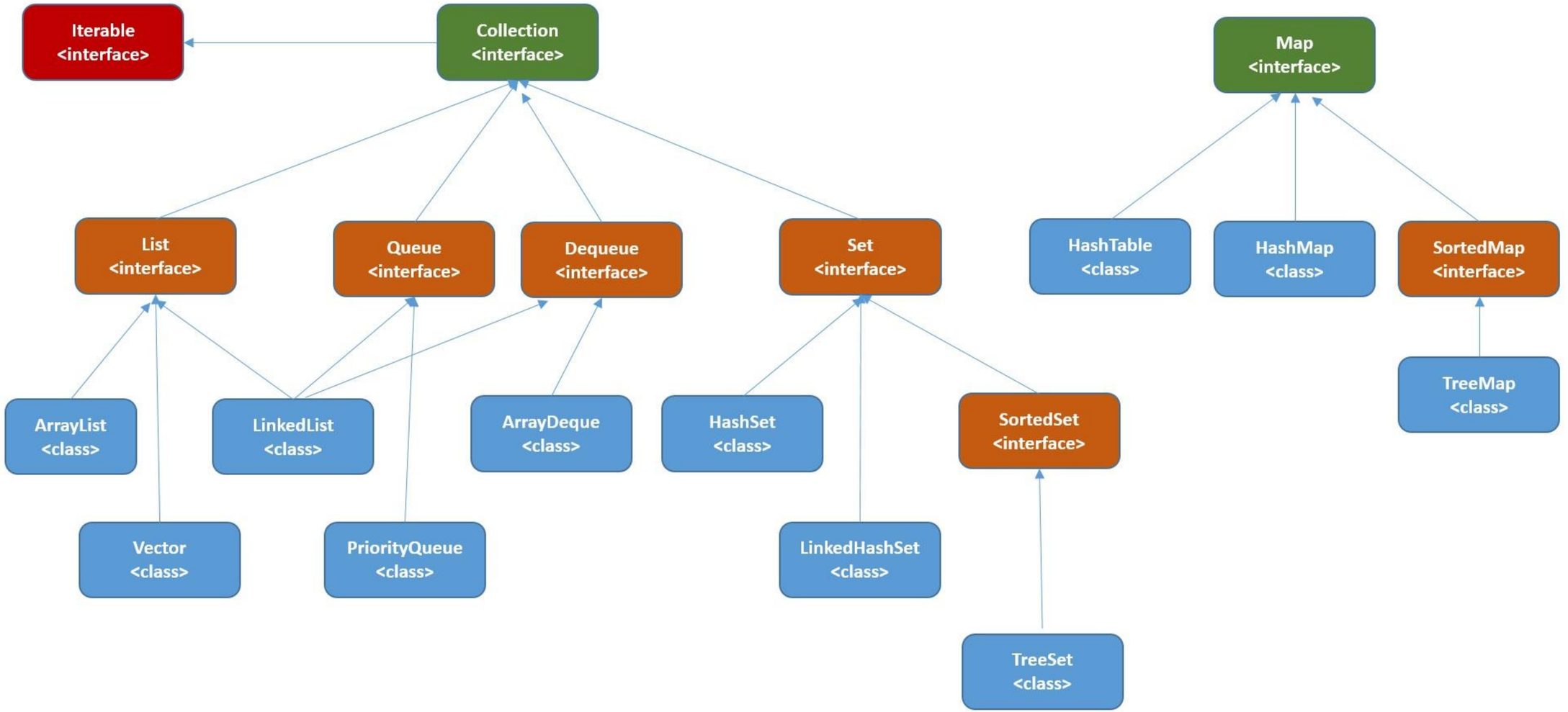
- Introdurre le Collection API di Java
- Approfondire l'utilizzo di Liste, Set e Mappe
- Comprendere l'uso di Iterator e ciclo foreach
- Usare Comparable e Comparator per ordinare oggetti

Cos'è la Java Collection API

- È un insieme di **interfacce** e **classi** che rappresentano strutture dati standard: liste, insiemi, code, mappe.

- Fornisce **algoritmi** comuni: ordinamento, ricerca, iterazione

Collection Framework Hierarchy



Tipi primitivi vs Wrapper

In Java, esistono due categorie principali di tipi di dato:

- i **tipi primitivi**, che sono più leggeri e utilizzati per operazioni semplici
- le **classi wrapper**, che li rappresentano come oggetti.
- I wrapper sono necessari per lavorare con le collezioni, poiché queste ultime possono contenere solo oggetti.

Tipi primitivi vs Wrapper

Tipo Primitivo	Classe Wrapper
int	Integer
double	Double
char	Character
boolean	Boolean

Quando usare i Wrapper

I wrapper:

- Sono necessari con collezioni (List<Integer>, non List<int>)
- Offrono metodi statici utili per la conversione e il parsing, e supportano l'autoboxing/unboxing per facilitare l'interoperabilità.
- Possono essere null, al contrario dei primitivi

ArrayList vs LinkedList

- **ArrayList**: lista basata su array, accesso veloce per indice

- **LinkedList**: lista doppiamente concatenata, efficiente in inserimenti/rimozioni

Esempio ArrayList

```java
import java.util.*;

class Main {
    Run main | Debug main
    public static void main(String[] args) {
        List<String> nomi = new ArrayList<>();
        nomi.add("Luca");
        nomi.add("Anna");
        System.out.println(nomi.get(0));
    }
}
```

Esempio LinkedList

```java
import java.util.*;

class Main {
    Run main | Debug main
    public static void main(String[] args) {
        LinkedList<String> lista = new LinkedList<>();
        lista.add("A");
        lista.addFirst("B");
        lista.addLast("C");
    }
}
```

Esercizio 1

Scrivere un programma che usa un ArrayList<String> per memorizzare i nomi degli studenti e stamparli con un ciclo for.

Set: HashSet e TreeSet

- **HashSet**: implementazione non ordinata, senza duplicati

- **TreeSet**: implementazione ordinata secondo ordine naturale o comparatore

Esempio HashSet

```java
import java.util.*;

class Main {
    Run main | Debug main
    public static void main(String[] args) {
        Set<String> colori = new HashSet<>();
        colori.add("rosso");
        colori.add("verde");
        colori.add("rosso");
        System.out.println(colori); // solo 2 elementi
    }
}
```

Esempio TreeSet

```java
import java.util.*;

class Main {
    Run main | Debug main
    public static void main(String[] args) {
        Set<Integer> numeri = new TreeSet<>();
        numeri.add(5);
        numeri.add(1);
        numeri.add(3);
        System.out.println(numeri); // [1, 3, 5]
    }
}
```

Le Mappe: HashMap e TreeMap

- **HashMap**: mappa chiave-valore, accesso rapido

- **TreeMap**: mappa ordinata per chiave

Le Mappe: HashMap

```java
import java.util.*;

class Main {
    Run main | Debug main
    public static void main(String[] args) {
        Map<String, Integer> eta = new HashMap<>();
        eta.put("Luca", 25);
        eta.put("Anna", 22);
        System.out.println(eta.get("Luca"));
    }
}
```

Esercizio 2

Crea un programma che memorizza nomi e numeri di telefono in una `HashMap` e consenta di cercarli.

TreeMap e ordinamento per chiave

Nel TreeMap gli elementi sono ordinati secondo la chiave. Questo è molto utile quando vogliamo stampare un dizionario ordinato alfabeticamente.

```java
import java.util.*;

class Main {
    Run main | Debug main
    public static void main(String[] args) {
        Map<String, Integer> voti = new TreeMap<>();
        voti.put("Matematica", 28);
        voti.put("Informatica", 30);
        voti.put("Fisica", 24);
        System.out.println(voti);
    }
}
```

Iteratori e foreach

Tutte le collezioni possono essere iterate con:

- Iterator<E>
- ciclo for-each (più leggibile)

Entrambi ci aiutano a percorrere ogni elemento senza preoccuparci della struttura interna.

Esempio foreach

```java
import java.util.*;

class Main {
    Run main | Debug main
    public static void main(String[] args) {
        List<String> nomi = List.of("Luca", "Anna", "Marco");
        for (String nome : nomi) {
            System.out.println(nome);
        }
    }
}
```

Esempio Iterator

```java
import java.util.*;

class Main {
    Run main | Debug main
    public static void main(String[] args) {
        List<String> nomi = List.of("Luca", "Anna", "Marco");
        Iterator<String> it = nomi.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

Comparable e Comparator

Per ordinare oggetti complessi, possiamo far implementare alla nostra classe l'interfaccia **Comparable**, oppure usare un **Comparator** esterno. La scelta dipende se vogliamo un ordine "di default" o se ci servono più criteri alternativi:

- **Comparable**: ordinamento naturale, tramite `compareTo()`
- **Comparator**: ordinamento personalizzato, esterno alla classe **Contesto orale:*

Esempio Comparable

```java
public class Studente implements Comparable<Studente> {
    protected String nome;
    protected int matricola;

    @Override
    public int compareTo(Studente s) {
        return this.matricola - s.matricola;
    }
}
```

Esempio Comparator

```java
class Main {
    Run main | Debug main
    public static void main(String[] args) {
        List<Studente> studenti = Arrays.asList(
            new Studente("Marco", 22, 28.5),
            new Studente("Anna", 20, 30.0),
            new Studente("Luca", 21, 25.0)
        );

        studenti.sort(new Comparator<Studente>() {
            @Override
            public int compare(Studente s1, Studente s2) {
                return s1.getNome().compareTo(s2.getNome());
            }
        });

        studenti.sort((Studente s1, Studente s2) -> s1.getNome().compareTo(s2.getNome()));

        System.out.println("Ordinati per nome:");
        for (Studente s : studenti) {
            System.out.println(s);
        }
    }
}
```

Differenze tra Comparator e Comparable

Caratteristica	Comparable	Comparator
Dove si definisce	Nella **classe stessa**	In una **classe esterna o anonima**
Quanti criteri supporta	**Uno solo** (ordinamento naturale)	**Più** (puoi crearne tanti diversi)
Metodo da implementare	compareTo(T o)	compare(T o1, T o2)
Quando si usa	Ordinamento predefinito	Ordinamento personalizzato temporaneo

Ordinamento con Collections.sort

La classe Collections fornisce numerosi metodi statici per operare sulle collezioni.

sort è tra i più utili e consente di ordinare una lista in-place, sfruttando l'ordinamento naturale (Comparable) o uno personalizzato (Comparator).

Collections.sort(listaStudenti); // se implementa Comparable

Collections.sort(listaStudenti, perNome); // con Comparator

Esercizio 3

Scrivi una classe Persona con nome e reddito. Crea una lista di persone e ordina prima per reddito (naturale), poi per nome (Comparator).

Metodi utili Collections

Collections è una classe utility piena di metodi statici comodi per manipolare liste: possiamo invertirle, mescolarle, trovare massimo e minimo. Queste operazioni rendono la manipolazione dei dati molto semplice.

- Collections.reverse(List)
- Collections.shuffle(List)
- Collections.max(List)
- Collections.min(List)

Autoboxing e Unboxing

Importante sapere che questa operazione non è gratuita in termini di performance.

```java
class Main {
    Run main | Debug main
    public static void main(String[] args) {
        List<Integer> numeri = new ArrayList<>();
        numeri.add(5); // autoboxing: int -> Integer
        int x = numeri.get(0); // unboxing: Integer -> int
    }
}
```

Esercizio 4

Inserisci in una List<Integer> una serie di voti. Calcola e stampa la media.

Conversioni da Array a List

A volte è necessario convertire tra array e liste*:*

- Da array a lista: Arrays.asList(array) → restituisce una lista immutabile
- Da array a lista: new ArrayList<>(Arrays.asList(...)) → restituisce una lista modificabile
- Da lista ad array: list.toArray(new String[0])

Esercizio 5

Data una lista di parole, converti in array e stampa solo quelle con lunghezza > 5.

Collezioni e null

L'utilizzo di null nelle collezioni può portare a problemi imprevedibili, specialmente se si usano comparatori. HashMap consente una chiave null, TreeMap no. È buona pratica evitare null ove possibile o gestirlo in modo esplicito.

Esercizio 6

Scrivi un programma che:

- Legge nomi da file
- Li salva in un Set per eliminare duplicati
- Li ordina con TreeSet
- Li stampa

Esercizio 7

Scrivi un programma che:

- Salva voti in una HashMap (chiave: materia)
- Ordina e stampa per materia
- Calcola media dei voti

Esercizio 8

Scrivi una classe Libro con titolo e autore. Salva libri in una List<Libro> e ordina per autore con un Comparator.

Best practices sulle collezioni

- Usare sempre le interfacce (List, Map, Set)
- Evitare accessi diretti con get(i) su liste grandi
- Scegliere la struttura in base all’uso: accesso, ordine, duplicati

Utilizzare le collezioni in modo efficace significa anche conoscere i compromessi e fare scelte consapevoli.