**MAP – JAVA**  
Una mappa è una struttura dati che associa chiavi a valori univoci. In generale, una mappa fornisce un'implementazione di questa associazione, consentendo di accedere rapidamente a un valore dato una chiave.

Immagina una mappa geografica: associamo delle posizioni (le chiavi) a delle informazioni specifiche su quella posizione (i valori). Ad esempio, possiamo associare il nome di una città alla sua popolazione o alle sue coordinate geografiche. Quando desideriamo ottenere informazioni su una città, possiamo cercare la città nella mappa e ottenere rapidamente i dati associati ad essa.

Le mappe sono utili quando abbiamo bisogno di associare informazioni tra loro in modo efficiente e quando vogliamo accedere a queste informazioni tramite un identificatore univoco. Sono ampiamente utilizzate in programmazione per gestire dati strutturati in modo flessibile e efficiente.

**MAP<K, V> INTERFACE**  
L'interfaccia **Map** in Java definisce una collezione di coppie chiave-valore, dove ogni chiave è associata ad un unico valore. Questa interfaccia fornisce metodi (sena implementazione in quanto interfaccia) per inserire, rimuovere, ottenere e manipolare i dati all'interno della mappa. Ecco una descrizione dei principali metodi definiti nell'interfaccia **Map**:

1. **put(K key, V value)**: Aggiunge una coppia chiave-valore alla mappa. Se la chiave esiste già nella mappa, il valore associato viene sovrascritto con quello nuovo e il valore precedente viene restituito. Se la chiave è nuova, il valore viene aggiunto alla mappa e il metodo restituisce null.
2. **get(Object key)**: Restituisce il valore associato alla chiave specificata, o null se la chiave non è presente nella mappa.
3. **remove(Object key)**: Rimuove la coppia chiave-valore associata alla chiave specificata e restituisce il valore corrispondente, o null se la chiave non è presente nella mappa.
4. **containsKey(Object key)**: Verifica se la mappa contiene la chiave specificata. Restituisce true se la chiave è presente, altrimenti restituisce false.
5. **containsValue(Object value)**: Verifica se la mappa contiene almeno un'occorrenza del valore specificato. Restituisce true se il valore è presente, altrimenti restituisce false.
6. **size()**: Restituisce il numero di coppie chiave-valore presenti nella mappa.
7. **isEmpty()**: Verifica se la mappa è vuota. Restituisce true se la mappa non contiene alcuna coppia chiave-valore, altrimenti restituisce false.
8. **clear()**: Rimuove tutte le coppie chiave-valore dalla mappa, lasciandola vuota.
9. **keySet()**: Restituisce un set contenente tutte le chiavi presenti nella mappa.
10. **values()**: Restituisce una collezione contenente tutti i valori presenti nella mappa.
11. **entrySet()**: Restituisce un set contenente tutte le coppie chiave-valore presenti nella mappa sotto forma di oggetti **Map.Entry**.

L'interfaccia **Map** fornisce un modo flessibile e generico per gestire dati associativi in Java, consentendo l'utilizzo di diverse implementazioni specifiche come **HashMap**, **TreeMap**, **LinkedHashMap** e altre.

In Java, **Map<K, V>** è un'interfaccia che rappresenta una struttura dati che mappa chiavi univoche a valori corrispondenti. Qui, **K** rappresenta il tipo delle chiavi e **V** rappresenta il tipo dei valori.

Le implementazioni di **Map<K, V>** forniscono un modo efficiente per memorizzare coppie chiave-valore e per accedere rapidamente ai valori tramite le chiavi.

NOTA: Map<K,V> è solo un interfaccia, le classi sotto implementano tale interfaccia (ne scrivono i metodi)

Le implementazioni comuni di **Map<K, V>** includono:

1. **HashMap<K, V>**: Implementazione che utilizza una tabella hash per memorizzare le coppie chiave-valore. È molto efficiente per le operazioni di inserimento, ricerca e rimozione, ma non mantiene un ordine definito delle chiavi.
2. **TreeMap<K, V>**: Implementazione che memorizza le coppie chiave-valore in un albero rosso-nero ordinato in base alle chiavi. Le operazioni di inserimento, ricerca e rimozione hanno una complessità logaritmica rispetto al numero di elementi nella mappa.
3. **LinkedHashMap<K, V>**: Implementazione che mantiene l'ordine di inserimento delle coppie chiave-valore. È utile quando è necessario mantenere un ordine specifico dei valori.
4. **Hashtable<K, V>**: Implementazione simile a **HashMap**, ma sincronizzata, il che la rende thread-safe. Tuttavia, è meno efficiente di **HashMap**.

L'utilizzo di **Map<K, V>** è estremamente comune in Java per la memorizzazione di associazioni chiave-valore e viene utilizzato in una vasta gamma di applicazioni, dalla gestione di dati nei database alla memorizzazione di parametri di configurazione nell'applicazione.

**HASHMAP<K, V>**   
**HashMap<K, V>** è un'implementazione di base dell'interfaccia **Map<K, V>** in Java. Essa utilizza una tabella hash per memorizzare le coppie chiave-valore e offre un'efficienza notevole nelle operazioni di inserimento, ricerca e rimozione degli elementi.

Ecco alcuni punti chiave su **HashMap<K, V>**:

1. **Struttura interna**: **HashMap** utilizza una struttura di tabella hash interna per memorizzare i dati. Questa struttura consente un accesso rapido agli elementi sulla base delle loro chiavi. Quando si aggiunge un elemento, viene calcolato un codice hash della chiave e viene utilizzato per determinare la posizione nella tabella hash dove memorizzare l'elemento.
2. **Ordine degli elementi**: La classe **HashMap** non garantisce un ordine specifico degli elementi. Gli elementi possono essere restituiti in qualsiasi ordine durante l'iterazione sulla mappa. Se è necessario un ordine specifico, si dovrebbe considerare l'utilizzo di **LinkedHashMap**, che mantiene l'ordine di inserimento.
3. **Complessità temporale**: Le operazioni principali su **HashMap**, come **put()**, **get()** e **remove()**, hanno una complessità temporale media di O(1), assumendo che la funzione di hash distribuisca uniformemente gli elementi nella tabella hash. Tuttavia, in casi eccezionali, la complessità può diventare O(n), dove n è il numero di elementi nella mappa.
4. **Sicurezza thread**: **HashMap** non è thread-safe. Se si desidera utilizzare una mappa in un ambiente multithreading, è necessario sincronizzare manualmente l'accesso alla mappa o utilizzare una delle implementazioni sincronizzate come **Hashtable** o **ConcurrentHashMap**.

In generale, **HashMap<K, V>** è una scelta popolare per molte applicazioni grazie alla sua velocità ed efficienza. È ampiamente utilizzato in Java per la memorizzazione e il recupero di dati associati a chiavi univoche.

Ecco un'esempio di come dichiarare e utilizzare una mappa in Java:

import java.util.HashMap;

import java.util.Map;

public class Main {

    public static void main(String[] args) {

        // Dichiarazione e inizializzazione di una mappa

        Map<String, Integer> mappa = new HashMap<>();

        // Inserimento di elementi nella mappa

        mappa.put("Chiave1", 10);

        mappa.put("Chiave2", 20);

        mappa.put("Chiave3", 30);

        // Accesso ai valori tramite le chiavi

        int valore = mappa.get("Chiave2");

        System.out.println("Valore associato a Chiave2: " + valore);

        // Iterazione attraverso la mappa

        for (Map.Entry<String, Integer> entry : mappa.entrySet()) {

            String chiave = entry.getKey();

            int valoreCorrispondente = entry.getValue();

            System.out.println("Chiave: " + chiave + ", Valore: " + valoreCorrispondente);

        }

    }

}

In questo esempio, abbiamo utilizzato la classe HashMap, che è una delle implementazioni più comuni dell'interfaccia Map. Per utilizzare Map in un programma Java, devi importare il package java.util.Map e la classe specifica dell'implementazione della mappa che desideri utilizzare, ad esempio java.util.HashMap.

**HASHMAP<K, V> - METHODS**  
Ecco alcuni dei principali metodi della classe HashMap<K, V> in Java:

1. **put(K key, V value):** Aggiunge un'associazione chiave-valore alla mappa. Se la chiave esiste già nella mappa, il valore associato a quella chiave viene sovrascritto con il nuovo valore e il vecchio valore viene restituito.
2. **get(Object key):** Restituisce il valore associato alla chiave specificata, oppure null se la chiave non è presente nella mappa.
3. **remove(Object key):** Rimuove l'associazione chiave-valore corrispondente alla chiave specificata dalla mappa, se presente. Restituisce il valore associato a quella chiave, o null se la chiave non è presente nella mappa.
4. **containsKey(Object key):** Restituisce true se la mappa contiene la chiave specificata, altrimenti restituisce false.
5. **containsValue(Object value**): Restituisce true se la mappa contiene almeno un'associazione con il valore specificato, altrimenti restituisce false.
6. **isEmpty():** Restituisce true se la mappa è vuota, altrimenti restituisce false.
7. **size():** Restituisce il numero di coppie chiave-valore presenti nella mappa.
8. **clear():** Rimuove tutte le coppie chiave-valore dalla mappa.

Questi sono solo alcuni dei metodi forniti da HashMap. Ci sono altri metodi utili per iterare sulla mappa, ottenere le chiavi, ottenere i valori, etc. La documentazione ufficiale di Java fornisce una panoramica completa di tutti i metodi disponibili nella classe HashMap.

**MAP.ENTRY**  
In Java, **Map.Entry** è un'interfaccia annidata all'interno dell'interfaccia **Map**. Rappresenta una coppia chiave-valore all'interno di una mappa. Ogni oggetto **Map.Entry** contiene esattamente una mappatura chiave-valore.

Ecco una spiegazione breve:

* **Map**: Una **Map** in Java è un oggetto che mappa chiavi a valori. Ogni chiave può essere mappata al massimo con un solo valore. Alcune implementazioni comuni dell'interfaccia **Map** includono **HashMap**, **TreeMap** e **LinkedHashMap**.
* **Interfaccia Map.Entry**: Questa interfaccia rappresenta una singola mappatura chiave-valore all'interno di una mappa. Ha metodi per ottenere la chiave e il valore associati a quella mappatura, nonché per modificarne il valore.

L'interfaccia Map.Entry è una parte fondamentale delle mappe in Java perché rappresenta una singola coppia chiave-valore all'interno della mappa. In altre parole, ogni elemento all'interno di una mappa è un oggetto Map.Entry che contiene la coppia chiave-valore.

Ecco alcuni punti chiave:

**Rappresentazione di una Coppia Chiave-Valore:** Ogni oggetto Map.Entry contiene una chiave e il valore associato a quella chiave.

**Metodi di Accesso**: L'interfaccia Map.Entry fornisce metodi per accedere alla chiave e al valore associati alla coppia. Ad esempio, il metodo getKey() restituisce la chiave e il metodo getValue() restituisce il valore.

**Modifica dei Valori**: È possibile modificare il valore associato a una chiave utilizzando il metodo setValue(). Questo è utile per aggiornare i valori all'interno della mappa.

**Iterazione attraverso le Mappe**: Gli oggetti Map.Entry sono spesso utilizzati per iterare attraverso gli elementi di una mappa. Questo consente di accedere sia alla chiave che al valore di ciascuna coppia all'interno della mappa durante l'iterazione.

In sintesi, Map.Entry è un'interfaccia che fornisce un modo per rappresentare e manipolare coppie chiave-valore all'interno di una mappa in Java. È ampiamente utilizzata per lavorare con dati strutturati in cui è necessario associare una chiave a un valore

perchè devo usare map.entriy per accedere ad un valore della map invece di usare get di map ?

Usare **Map.Entry** per accedere ai valori di una mappa è utile quando si desidera iterare attraverso tutti gli elementi della mappa e ottenere sia la chiave che il valore associato ad ogni elemento. Questo è particolarmente utile quando si vuole eseguire operazioni complesse sui valori o si desidera lavorare con coppie chiave-valore.

Ecco alcune situazioni in cui l'utilizzo di **Map.Entry** può essere preferibile rispetto all'utilizzo del metodo **get()** della mappa:

1. **Iterazione Completa della Mappa**: Se si desidera iterare attraverso tutti gli elementi della mappa, utilizzare **Map.Entry** è più efficiente poiché si ottiene direttamente l'accesso sia alla chiave che al valore di ciascun elemento durante l'iterazione, senza dover fare una chiamata separata al metodo **get()** per ogni chiave.
2. **Migliore Performance**: In alcune implementazioni di mappe, l'accesso diretto tramite **Map.Entry** può essere più efficiente rispetto all'accesso tramite il metodo **get()**, specialmente quando si sta eseguendo un grande numero di operazioni.
3. **Operazioni Composte o Complesse**: Se si desidera eseguire operazioni complesse sui valori della mappa, come la modifica dei valori o la rimozione di alcuni elementi in base a determinate condizioni, l'utilizzo di **Map.Entry** può semplificare il processo, poiché si ha accesso diretto sia alla chiave che al valore.

Tuttavia, se si conosce già la chiave di cui si ha bisogno e non è necessario iterare attraverso tutti gli elementi della mappa, utilizzare il metodo **get()** può essere più diretto ed efficiente.

In breve, l'uso di **Map.Entry** è più adatto quando si desidera lavorare con coppie chiave-valore e si prevede di iterare attraverso tutti gli elementi della mappa.

**ENTRYSET()**  
**entrySet()** è un metodo della classe **Map** in Java che restituisce un insieme di oggetti **Map.Entry**. Questi oggetti rappresentano le coppie chiave-valore all'interno della mappa.

Ecco alcuni punti chiave relativi a **entrySet()**:

1. **Restituisce un insieme di Map.Entry**: Il metodo **entrySet()** restituisce un insieme di oggetti **Map.Entry**, ognuno dei quali rappresenta una coppia chiave-valore all'interno della mappa.
2. **Per iterare attraverso le voci della mappa**: Questo metodo è comunemente utilizzato per iterare attraverso tutte le coppie chiave-valore presenti nella mappa. Utilizzando l'insieme restituito da **entrySet()**, è possibile accedere a ogni coppia chiave-valore e quindi ottenere la chiave e il valore associato utilizzando i metodi **getKey()** e **getValue()** di **Map.Entry**.
3. **Interfaccia Set**: L'insieme restituito da **entrySet()** è di tipo **Set<Map.Entry<K, V>>**, dove **K** è il tipo della chiave e **V** è il tipo del valore nella mappa.

Ecco un esempio di come utilizzare **entrySet()** per iterare attraverso le voci di una mappa:

java

Map<String, Integer> map = new HashMap<>();

map.put("A", 1);

map.put("B", 2);

map.put("C", 3);

Set<Map.Entry<String, Integer>> entrySet = map.entrySet();

for (Map.Entry<String, Integer> entry : entrySet) {

    String key = entry.getKey();

    Integer value = entry.getValue();

    System.out.println("Chiave: " + key + ", Valore: " + value);

}

In questo esempio, **entrySet()** restituisce un insieme di tutte le voci nella mappa **map**. Iteriamo quindi attraverso questo insieme utilizzando un ciclo foreach, ottenendo ogni voce come un oggetto **Map.Entry**. Infine, otteniamo la chiave e il valore di ogni voce utilizzando i metodi **getKey()** e **getValue()** di **Map.Entry**.

**Set<Map.Entry<***type***,** *type***>>**   
è un tipo di dato generico in Java che rappresenta un insieme di oggetti **Map.Entry** con i tipi di chiavi  
e valori specificati fra <>

Vediamo cosa significa ogni parte di questa dichiarazione:

* **Set**: È il tipo principale della variabile. In questo caso, rappresenta un'interfaccia che rappresenta un insieme di elementi univoci, ovvero un insieme che non può contenere duplicati.
* **<Map.Entry<String, Integer>>**: Questo è il tipo generico all'interno delle parentesi angolari. Questo specifica il tipo degli elementi contenuti nell'insieme **Set**.
  + **Map.Entry<String, Integer>**: Rappresenta il tipo degli elementi dell'insieme, cioè oggetti **Map.Entry** dove la chiave è di tipo **String** e il valore è di tipo **Integer**.

Quindi, **Set<Map.Entry<String, Integer>>** indica un'istanza di un insieme che contiene oggetti **Map.Entry** dove la chiave è di tipo **String** e il valore è di tipo **Integer**.

Inizio modulo

**FILEINPUTSSTRRAM – aprire un file con FileInputStream**  
La classe **FileInputStream** in Java viene utilizzata per leggere dati da un file nel formato di byte.

FileInputStream fis = new FileInputStream(filePath);

In questo codice creo un oggetto **FileInputStream** per il file specificato nel percorso **filePath**.

Ecco cosa fa esattamente **FileInputStream**:

1. **Apre il file**: Quando crei un'istanza di **FileInputStream** specificando il percorso del file, il costruttore cerca di aprire il file specificato per la lettura dei dati. Se il file non esiste o non può essere aperto per qualsiasi motivo, verrà lanciata un'eccezione **FileNotFoundException**.
2. **Legge i dati**: Dopo aver aperto il file con successo, puoi utilizzare i metodi forniti dalla classe **FileInputStream** per leggere i dati dal file. Ad esempio, puoi leggere un byte alla volta, un array di byte o anche interi.
3. **Chiusura del file**: Dopo aver completato le operazioni di lettura, è buona pratica chiudere il file utilizzando il metodo **close()** della classe **FileInputStream**. Ciò rilascerà le risorse associate al file e consentirà al sistema di liberare eventuali lock sul file.

Vediamo un esempio di apertura file:

import java.io.FileInputStream;

import java.io.IOException;

public class FileOpener {

    public static void main(String[] args) {

        try {

            // Specifica il percorso del file da aprire

            String filePath = "path/to/your/file.txt";

            // Crea un oggetto FileInputStream per il file specificato

            FileInputStream fis = new FileInputStream(filePath);

            // Ora puoi leggere byte dal file utilizzando l'oggetto FileInputStream

            // Ricorda sempre di chiudere il flusso quando hai finito

            fis.close();

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}

alcuni dei **metodi** più comuni disponibili nella classe **FileInputStream**:

1. **int read()**: Legge il prossimo byte di dati dal flusso di input e restituisce il byte letto come un intero compreso tra 0 e 255. Restituisce -1 se ha raggiunto la fine del file.
2. **int read(byte[] b)**: Legge un numero massimo di byte dal flusso di input e li memorizza nell'array di byte specificato **b**. Restituisce il numero effettivo di byte letti o -1 se ha raggiunto la fine del file.
3. **int read(byte[] b, int off, int len)**: Legge fino a **len** byte di dati dal flusso di input nell'array di byte specificato **b**, a partire dall'offset specificato **off**. Restituisce il numero effettivo di byte letti o -1 se ha raggiunto la fine del file.
4. **long skip(long n)**: Salta **n** byte nel flusso di input. Restituisce il numero effettivo di byte saltati.
5. **int available()**: Restituisce il numero di byte che possono essere letti senza bloccare il flusso.
6. **void close()**: Chiude il flusso di input. Dovrebbe essere sempre chiamato quando hai finito di leggere il file per rilasciare le risorse associate.

Inizio modulo