DALLE INTERFACCE ALLE IMPLEMENTAZIONI: JAVA

Java

General purpose Implementations

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

- Implementazioni fondamentali (modificabili):
  - per **`Set`**: **`HashSet`**, **`TreeSet`**, **`LinkedHashSet`**, **`EnumSet`**
  - per **`List`**: **`ArrayList`**, **`LinkedList`**
  - per **`Map`**: **`HashMap`**, **`TreeMap`**, **`LinkedHashMap`**, **`EnumMap`**
  - per **`Queue`**: **`ArrayBlockingQueue`**, **`PriorityQueue`**, **`LinkedList`**
  - per **`Deque`**: **`ArrayDeque`**, **`LinkedList`**
- Le implementazioni *immodificabili* non corrispondono a classi pubbliche: sono prodotte solo dai factory methods **`List.of`**, **`Set.of`**, etc.

COLLEZIONI: LINEE GUIDA

Quali usare?

- Regole generali per **`Set`** e **`Map`**
  - se serve *l'ordinamento*, necessariamente **`TreeMap`** e **`TreeSet`** che implementano le sotto-interfacce **`SortedMap`** e **`SortedSet`**
  - altrimenti, **`HashMap`** e **`HashSet`** sono *nettamente più efficienti* (tempo di esecuzione costante anziché log(N))
- Regole generali per **`List`**
  - di norma meglio **`ArrayList`**, che ha *tempo di accesso costante* (anziché lineare con la posizione) perché è realizzata su array
  - preferire invece **`LinkedList`** se l'operazione più frequente è l'aggiunta in testa o l'eliminazione di elementi in mezzo alla lista.

COLLEZIONI: LINEE GUIDA

Quali usare? *(continua)*

- Implementazioni specifiche per **`Set`** e **`Map`**:
  - se serve un *ordine precidibile di iterazione*, **`LinkedHashMap`** e **`LinkedHashSet`**

    This implementation maintains a doubly-linked list through all of its entries. The *iteration ordering* is the *insertion-order*, even if an element is re-inserted.

  - caso particolare: se gli elementi del set o le chiavi della mappa sono *enumerativi*, utile scegliere **`EnumMap`** ed **`EnumSet`**

    The internal implementation is *extremely compact and efficient.* Iteration reflects *the natural ordering* of enum constants.

COLLEZIONI: COSTRUZIONE

- Le collezioni concrete si costruiscono
  - *vuote*, con un costruttore
  - *già inizializzate*, con appositi inizializzatori o metodi factory statici (in alcuni linguaggi, ciò si applica solo a collezioni *immodificabili*)
- Premesso che ogni linguaggio al riguardo fa le sue scelte, tipicamente:
  - i costruttori producono una collection inizialmente vuota
  - ulteriori *costruttori per copia* producono una collection a partire da un'altra (anche di diverso tipo) fornita come argomento
  - collezioni pre-inizializzate sono tipicamente ottenute tramite *metodi factory statici* (in alcuni linguaggi, solo per collezioni *immodificabili*)

COLLEZIONI: COSTRUZIONE

- In Java

Java

  - tutte le classi-collection definiscono un costruttore a zero argomenti che produce una collezione vuota (modificabile)
  - tutte le classi-collection definiscono un costruttore per copia che accetta come argomento *un'altra* **`Collection`**
  - apposite factory internalizzate producono collezioni pre-popolate *immodificabili* per i casi standard: **`List.of`**, **`Set.of`**, **`Map.of`**

- In Kotlin

Kotlin

  - costruzione base come sopra
  - collezioni pre-popolate sono prodotte da factory method, sia per le versioni *modificabili* - **`listOf(…)`**, **`setOf(…)`**, **`mapOf(…)`** - che *immodificabili* - **`mutableListOf(…)`**, **`mutableSetOf(…)`**, etc.
  - per uniformità, tale approccio vale anche per gli array: **`arrayOf(…)`**