PARADIGMI DI PROGRAMMAZIONE

A.A. 2021/2022

Laurea triennale in Informatica

5: Libreria Standard

LIBRERIA STANDARD

Per poter affrontare efficacemente alcuni esempi, è necessario avere una idea delle caratteristiche della libreria standard che il linguaggio mette a disposizione.

La documentazione, ovvero i JavaDoc, della libreria standard sono di ottima qualità: consultateli spesso.

https://docs.oracle.com/en/java/javase/17/docs/api/jav
summary.html

MODULI

I JavaDoc sono organizzati per "moduli".

I moduli corrispondono alla suddivisione introdotta in Java 9 con Project Jigsaw.

Un modulo è un insieme di package e tipi, di cui può controllare l'accesso dall'esterno.

Si tratta di una unità di organizzazione del codice Java superiore al package.

Il principale caso d'uso dei moduli è la modularizzazione della piattaforma Java.

I moduli consentono di separare il JDK in parti più piccole e creare delle distribuzioni che contengono solo i moduli necessari, allo scopo di rendere più agevole la pubblicazioni di applicazioni complete.

Il progetto è riuscito solo in parte: ad oggi, pochi software fanno effettivo uso dei moduli, e l'evoluzione delle tecniche di distribuzione ha superato quella che era l'intenzione iniziale.

Per gli scopi del corso non è rilevante approfondire l'argomento.

INPUT/OUTPUT

Il modello di I/O di Java non è dissimile dal modello POSIX comune a molte altre librerie standard, ed è contenuto nel package java.io.

Le principali astrazioni sono il File, l'InputStream e l'OutputStream, da cui derivano le varie implementazioni.

Gli usi più comuni sono attraverso le implementazioni delle classi Reader e Writer che forniscono metodi semplici per la lettura e scrittura di file testuali, come per esempio BufferedReader e PrintWriter.

```
BufferedReader rd = new BufferedReader(
    new FileReader(".hgignore"));
String line = rd.readLine();
while (line != null) {
  System.out.println(line);
}
rd.close();
```

La libreria standard è organizzata per gerarchia di capacità (le sottoclassi implementano particolari funzionalità) e promuove l'uso della composizione per costruire le catene di elaborazione necessarie.

Questa versatilità a volte produce una API prolissa e ingombrante, per dare spazio a tutti i punti di accesso per i vari casi d'uso.

JAVA.LANG.SYSTEM

L'oggetto System fornisce, insieme ad altri servizi relativi all'interazione con il sistema, gli oggetti che rappresentano gli stream classici: System.in, System.out, System.err.

JAVA.NIO

Nella release 1.4, viene aggiunto a Java il package `java.nio` che aggiunge nuove astrazioni, gestione asincrona delle operazioni di I/O, e miglioramenti nelle performance di casi specifici.

COLLECTIONS API

La parte più importante della libreria standard di Java è l'ampia libreria di Collezioni che sono usate diffusamente in tutte le classi di sistema, e che ha ricevuto un importante aggiornamento nella versione 8.

COLLECTION

L'interfaccia `Collection` è la radice della libreria. Contiene i metodi più generali (dimensioni, test di contenitore vuoto, aggiunta, rimozione) che tutte le interfacce più specifiche includono.

Non ci sono implementazioni dirette di questa interfaccia, ma solo interfacce più specializzate.

Diversi metodi sono marcati come "opzionali", quindi in realtà le singole implementazioni sono libere di lanciare `UnsupportedOperationException` se non implementano l'operazione: il caso tipico sono le viste non modificabili di altre collezioni, che non permettono la modifica del loro contenuto.

La maggior parte delle collezioni distingue i contenuti nel senso del metodo `java.lang.Object#equals()`, che quindi è necessario implementare correttamente in questi casi.

L'operatore di confronto == non è utilizzabile fra oggetti, in quanto confronta solo l'identità: due istanze di una classe sono sempre diverse anche se rappresentano lo stesso valore.

```
class Point {
  public int x, y;

  Point(int x, int y) {
    this.x = x; this.y = y;
  }

  Point twoTimes() {
    x *= 2; y *= 2;
    return this;
  }
}
```

```
Point a = new Point(2, 1), b = new Point(3, 4),
  c = new Point(2, 1);
a == b // false
a == a // true
a == c // false
a == a.twoTimes() // true
```

```java
@Override
public boolean equals(Object other) {
  if (other instanceof Point) {
    Point o = (Point) other;
    return this.x == o.x && this.y == o.y;
  } else
    return false;
}
@Override
public int hashCode() {
  return (x & 0xffff) << 16 + (y & 0xffff);
}
```

```
Point a = new Point(2, 1), b = new Point(3, 4),
  c = new Point(2, 1);
a == b // false
a == a // true
a == c // false
a.equals(c) // true
a.equals(a.twoTimes()) // ?
```

La classe `Collections` raccoglie diversi metodi di utilità per applicare algoritmi alle collezioni, o per aggiungere particolari funzioni ad una collezione esistente.

Metodo	Risultato
`checkedTTT`	Controllo al runtime del tipo
`emptyTTT`	Collezione vuota
`syncronizedTTT`	Collezione sincronizzata
`unmodifiableTTT`	Vista non modificabile

Metodo	Risultato
`binarySearch`	Ricerca in una lista
`disjoint`	Verifica se disgiunte
`fill`	Riempie una collezione
`min, max`	Trova massimo e minimo
`reverse`	Inverte l'ordine
`shuffle`	Ordina in modo casuale
`sort`	Ordina la collezione

La classe `Arrays` raccoglie altri metodi di utilità, concentrati invece sul trattamento degli array. Ci sono metodi che declinano quelli di `Collections` su vari tipi di array primitivi, ed alcuni relativi specificamente agli array.

ITERATOR/ABLE

Un `Iterator` consente di elencare una collezione un elemento alla volta, individuando quando la si è attraversata completamente.

Una classe `Iterable` può fornire un `Iterator` per essere attraversata.

Iterator	Significato
`next`	Prossimo elemento
`hasNext`	Verso se ci sono altri elementi
`remove`	Rimuove l'elemento attuale
`forEachRemaining`	Consuma il resto della collezione

`Iterable`	**Significato**
`forEach`	Applica ad ogni elemento
`iterator`	Fornisci un `Iterator`
`spliterator`	Fornisci uno `Spliterator`

LIST

L'interfaccia `List` rappresenta un elenco ordinato di elementi, indirizzabili per posizione. Sono permessi elementi duplicati.

Fornisce uno specifico iteratore, `ListIterator` capace di movimento bidirezionale e modifiche sulla lista attraversata.

Implementazione	Caratteristiche
`ArrayList`	Ridimensionabile, basata su array
`LinkedList`	Basata su nodi concatenati
`Vector`	Legacy, basato su array, sincrono

L'interfaccia `List` fornisce un comodo metodo `of` per creare rapidamente una lista a partire da un elenco di oggetti.

```
var list = List.of(1, 2, 3);
```

SET

L'interfaccia `Set` definisce un insieme, cioè un contenitore di oggetti senza ripetizioni (nel senso di `equals`) non ordinato.

E' una pessima idea mutare un elemento in un Set in modi che cambiano il suo significato riguardo a `equals`.

Implementazione	Caratteristiche
`AbstractSet`	Scheletro di implementazione
`HashSet`	Basato su HashMap
`LinkedHashSet`	Ordinato in inserimento
`TreeSet`	Dotato di ordine interno
`EnumSet`	Specializzato per le `enum`

`SortedSet` è un insieme su cui è definito un ordine totale: è possibile enumerarlo secondo tale ordine, ed individuare inizio e fine dell'insieme.

`NavigableSet` è un insieme ordinato su cui è possibile muoversi sfruttando l'ordine, cercando direttamente (per es.) l'elemento minore o maggiore di un elemento dato.

DEQUEUE

L'interfaccia Dequeue rappresenta una **Double Ended Queue**, cioè una struttura dati da cui è possibile aggiungere e togliere elementi da uno dei due capi: l'inizio, o la fine.

Può essere usata come coda FIFO o come stack LIFO.

Implementazione	Caratteristiche
`ArrayDequeue`	Basata su array
`LinkedList`	Altro uso della stessa classe

Caratteristica delle **Dequeue** è avere due set di metodi differenti a seconda del comportamento in caso di impossibilità dell'azione richiesta:

Operazioni	**Conseguenze**
`add`,`remove`,`get`	Eccezione
`offer`,`poll`,`peek`	Valore speciale

MAP

Una interfaccia molto usata è `Map`, che rappresenta una mappa chiave-valore.

Gli oggetti usati come chiavi devono avere la coppia `equals`/`hashCode` correttamente definita. Valgono le stesse cautele già dette sul mutare lo stato di una chiave.

L'interfaccia mette a disposizione tre diverse viste sui suoi contenuti:

- un elenco di `Entry`, cioè le coppie chiave-valore
- l'insieme delle chiavi
- l'elenco dei valori

Ovviamente, in generale non sono permesse chiavi non uniche. Le implementazioni variano invece riguardo a permettere o meno `null` come valore, o come conservare l'ordinamento delle chiavi.

Classe	Implementazione
`HashMap`	Base, chiavi distinte per `hashCode`
`TreeMap`	Chiavi ordinate
`Hashtable`	Implementazione storica, sincrona

Classe	Implementazione
`EnumMap`	Specifica per chiavi `enum`
`WeakHashMap`	Chiavi "deboli", non impediscono la GC
`IdentityHashMap`	Specifica basata sull'identità

Anche `Map` mette a disposizione un metodo `of` per costruire rapidamente una mappa (immutabile) a partire da un elenco di coppie.

```
var map = Map.of("A", 1, "B", 2, "C", 3);
```

Come per `Set`, esistono le corrispettive `SortedMap` e `NavigableMap` a partire da un ordine totale sulle chiavi.

STREAM

Una parte importante dell'aggiornamento di Java 8 è stata l'introduzione del concetto di Stream in modo pervasivo nella libreria delle collezioni.

A molto interfacce è stato aggiunto il metodo `stream()` che permette di trattare le collezioni con questa metafora.

Uno `Stream` è una sequenza di elementi, non necessariamente finita.

L'obiettivo dell'astrazione dello stream è la descrizione dei passi di elaborazione che verranno effettuati sugli elementi, e l'ottimizzazione della loro esecuzione.

Le operazioni sugli Stream vengono *composte* in sequenza, in una *pipeline*, fino ad arrivare ad una operazione detta *terminale* che produce il risultato.

Nessuna operazione viene eseguita finché non viene richiamata l'operazione terminale.

Il codice che implementa la pipeline ha ampie libertà su come riordinare e disporre l'esecuzione delle operazioni intermedie. Queste ultime devono:

- non interferire, cioè non modificare gli elementi dello stream
- (nella maggior parte dei casi) non avere uno stato interno

Gli stream possono essere costruiti sia da collezioni di partenza, sia da altri tipi di astrazioni, come file, canali di comunicazione, generatori casuali.

Le operazioni intermedie sugli stream di dividono in *stateful* e *stateless*.

Il loro uso influenza la costruzione e l'efficienza della *pipeline* che le contiene.

Stateless	Significato
`filter`	Solo gli elementi che soddisfano un predicato
`drop/takeWhile`	Escludi/mantieni elementi finché vale un predicato
`map`	Trasforma ogni elemento
`peek`	Esegue un'operazione senza consumare l'elemento

Stateful	Significato
`distinct`	Elementi distinti
`concat`	Concatena due stream
`limit`	Tronca lo stream
`skip`	Salta l'inizio dello stream
`sorted`	Ritorna uno stream ordinato

Terminale	Significato
`all/any/noneMatch`	Vero se uno/tutti/nessuno gli elementi soddisfano il predicato
`collect`	Riduce lo stream ad un risultato
`findAny/First`	Ritorna un o il primo elemento

Terminale	Significato
`flatMap`	Trasforma ogni elemento in nuovi elementi
`forEach/Ordered`	Esegue un'operazione per ogni elemento
`min/max`	Minimo o massimo
`reduce`	Riduce lo stream con una operazione associativa

Generatore	Significato
`generate`	Produce uno stream a partire da un `Supplier`
`iterate`	Produce uno stream applicando una funzione a partire da un seme

Gli stream sono una astrazione estremamente utile in quanto consentono di descrivere il significato dell'elaborazione, invece del metodo.

La singola implementazione ha così più informazioni per ottimizzare l'esecuzione.

OPTIONAL

Usare *null* come valore di una variabile o del risultato di un metodo è ambiguo. Cosa rappresenta *null*?

- valore assente?
- operazione impossibile?
- risultato non trovato?
- errore di programmazione?

Questa ambiguità porta a rumorosi idiomi di *programmazione difensiva* e controlli sul valore delle variabili che non hanno però un reale legame con l'algoritmo implementato, e quindi nascondono l'intenzione dell'autore invece che chiarirla.

La classe `Optional` importa in Java un concetto che in altri linguaggi è chiamato `Option` o `MayBe`. Si tratta di una classe che rende esplicita la rappresentazione di un valore che potrebbe esserci, oppure no. Si risolve così l'ambiguità descritta in precedenza.

```java
var opt = Optional.empty();
var vl = Optional.of("Value");

vl.ifPresent(v -> System.out.println(v.length()));
var l = vl.map(s -> s.length()).orElse(0);
```

L'uso di questa classe è estremamente consigliato, visti gli evidenti vantaggi nella chiarezza del codice.

TIME API

La gestione del tempo è un problema difficile da gestire elgantemente, per tutti i dettagli, le eccezioni e le irregolarità che lo caratterizzano.

La prima API temporale di Java, che ruota attorno a `java.util.Date`, è stata sostituita in Java 8 (JSR-310) dal package `java.time`, più regolare e preciso.

`Instant` è un singolo, astratto, instante nel tempo.

`LocalDate`, `LocalTime`, `LocalDateTime` rappresentano una data, un'ora o un istante in uno specifico calendario. `ZonedDateTime` trasporta anche l'informazione del fuso orario.

Ci sono inoltre classi specifiche per singole unità temporali (ora, mese, anno, ecc.), e per intervalli di tempo.

Il package `java.time.format` contiene classi molto efficaci per leggere e formattare dati temporali.