Playfy

Sezioni

La traccia

Scopo della prova è progettare e implementare una gerarchia di oggetti utili a rappresentare delle *playlist* contenenti una selezione di *brani* di diversi *album* musicali.

Album, brani e durate

Un **album** è un elenco (ordinato, non vuoto e senza ripetizioni) di *brani* con un *titolo* (non vuoto) e una *durata complessiva*. Un **brano** è caratterizzato da un *titolo* (non vuoto) e da una *durata* (positiva); album e brano sono evidentemente immutabili.

Dato un album, è necessario essere in grado di *individuare*:

- un brano, dato il suo titolo;
- un brano, data la sua posizione nell'album;
- la posizione nell'album, dato un brano.

Per rendere più complete le funzionalità dell'album è possibile renderlo in grado di *enumerare* (con un iteratore) i brani che lo costituiscono.

Le **durate** sono misurate in secondi (e non possono evidentemente essere negative). Una possibile rappresentazione di una durata è data da una stringa suddivisa in ore, minuti e secondi, ad esempio, la durata di 7295 secondi corrisponde alla stringa `2:01:35` e viceversa; per tale ragione potrebbe essere sensato avere modo di costruire una durata da una stringa del genere, o viceversa di ottenere la stringa così formata a partire da una durata. Per finire, può tornare utile essere in grado di effettuare *somme* e *sottrazioni* (purché il risultato non sia

Skip to main content

Un esempio di (rappresentazione testuale di) un album è:

```
Titolo album: In the Court of the Crimson King
1 - "21st Century Schizoid Man" (07:24)
2 - "I Talk to the Wind" (06:04)
3 - "Epitaph" (08:49)
4 - "Moonchild" (12:13)
5 - "The Court of the Crimson King" (09:26)
Durata totale: 43:56
```

Suggerimento implementativo

Riflettete sul fatto che un brano dipende strettamente dall'album in cui è contenuto, nel senso ad esempio che brani pur con lo stesso titolo, ma appartenenti ad album diversi, sono da considerare distinti; per questa ragione può aver senso implementare il brano come classe interna (*inner class*) dell'album. In tal caso, occorre però prestare attenzione al costruttore dell'album: esso non può ricevere un elenco di brani (che possono essere istanziati solo all'interno dell'album stesso), ma potrebbe piuttosto ricevere, ad esempio, un elenco di titoli e uno di corrispondenti durate.

Le Playlist

Una **playlist** è un elenco (ordinato) di *brani* (possibilmente di album diversi) con un *nome* e una *durata complessiva* (che può essere 0 se la playlist è vuota). Una playlist è evidentemente mutabile ed è necessario che sia possibile *aggiungere* e *rimuovere* brani, così come *individuarne* la presenza e la posizione nella playlist.

Un esempio di (rappresentazione testuale di) una playlist è:

```
Nome playlist: Mescolotto
1 - "Another Brick in the Wall, Part 1" (03:11), (da "The wall")
2 - "21st Century Schizoid Man" (07:24), (da "In the Court of the Crimson King")
3 - "Another Brick in the Wall, Part 2" (03:59), (da "The wall")
4 - "Hey You" (04:40), (da "The wall")
5 - "Is There Anybody Out There?" (02:44), (da "The wall")
6 - "The Court of the Crimson King" (09:26), (da "In the Court of the Crimson King"
7 - "Mother" (05:32), (da "The wall")
Durata totale: 45:23
```

Si noti che di ciascun brano è riportato, oltre a titolo e durata, anche l'album di provenienza.

Skip to main content

Per rendere più complete le funzionalità della playlist è possibile renderla in grado di *enumerare*:

- tutti i suoi brani (con l'indicazione dell'album da cui provengono),
- i suoi brani che appartengono a un dato album,
- gli album dei suoi brani (senza ripetizione).

Dato l'esempio precedente, la seconda enumerazione, riguardo all'album `In the Court of the Crimson King` deve restituire i brani

```
"21st Century Schizoid Man" (07:24)
"The Court of the Crimson King" (09:26)
```

mentre la terza enumerazione deve restituire gli album

```
The wall
In the Court of the Crimson King
```

Inoltre, una playlist deve essere in grado di *fondersi* con un'altra playlist dando origine a una nuova playlist (con un titolo da specificare) che contenga tutti i suoi brani (nell'ordine in cui compaiono in essa), seguiti dai brani che compaiono nell'altra playlist (nell'ordine in cui compaiono nell'altra playlist, a meno che non siano già comparsi in precedenza nella fusione).

La playlist dell'esempio precedente è data dalla fusione della playlist

```
Nome playlist: La mia gioventù
1 - "Another Brick in the Wall, Part 1" (03:11), (da "The wall")
2 - "21st Century Schizoid Man" (07:24), (da "In the Court of the Crimson King")
3 - "Another Brick in the Wall, Part 2" (03:59), (da "The wall")
4 - "Hey You" (04:40), (da "The wall")
5 - "Is There Anybody Out There?" (02:44), (da "The wall")
6 - "The Court of the Crimson King" (09:26), (da "In the Court of the Crimson King"
Durata totale: 31:24
```

con la playlist

```
Nome playlist: I Pink Floyd
1 - "Mother" (05:32), (da "The wall")
2 - "Hey You" (04:40), (da "The wall")
Durata totale: 10:12
```

Skip to main content

si osservi, infatti, che `Mescolotto` contiene il brano dal titolo `Hey You` una sola volta (nella posizione in cui compare nelle prima playlist).

La classe di test

Potete implementare dei test in una o più classi; gli esempi di esecuzione provvisti assumono che alla fine la soluzione (ossia l'*unica classe che contiene un metodo* `main`) sia in grado di leggere una sequenza (possibilmente alternata) di album e playlist ed emettere alcune informazioni di conseguenza come descritto di seguito.

La classe deve essere in grado di leggere degli album rappresentati come segue:

```
ALBUM In the Court of the Crimson King
7:24 - 21st Century Schizoid Man
6:04 - I Talk to the Wind
8:49 - Epitaph
12:13 - Moonchild
9:26 - The Court of the Crimson King
.
```

si noti che la prima riga inizia con `ALBUM` (seguito dal titolo) e l'ultima riga è costituita dal solo carattere `.`; di ciascun brano è specificata la durata ed il titolo (separati dalla prima occorrenza del carattere `-` sulla linea).

Assumendo che gli album vengano letti e memorizzati in sequenza, la classe deve essere poi in grado di leggere delle playlist rappresentate come segue:

```
PLAYLIST La mia gioventù
1 3
2 1
1 5
1 14
1 15
2 5
.
```

si noti che la prima riga inizia con `PLAYLIST` (seguito dal nome) e l'ultima riga è costituita dal solo carattere `.`; di ciascun brano è specificato l'album (il primo numero) e la posizione del brano (il secondo numero); album e brani sono numerati a partire da 1.

La classe di test deve:

- emettere nel flusso d'uscita standard ogni album (non appena ne termina la lettura)

Skip to main content

- fondere tutte le playlist in una di nome `Fusa` e alla fine dell'input emetterla nel flusso d'uscita standard;

al termine dell'esecuzione deve inoltre emettere nel flusso d'uscita standard tutti gli album della playlist ottenuta per fusione e, per ciascun album, tutti i brani della playlist che provengono da esso.

Un esempio di output da emettere al termine dell'esecuzione è dato da:

```
Nome playlist: Fusa
1 - "Another Brick in the Wall, Part 1" (03:11), (da "The wall")
2 - "21st Century Schizoid Man" (07:24), (da "In the Court of the Crimson King")
3 - "Another Brick in the Wall, Part 2" (03:59), (da "The wall")
4 - "Hey You" (04:40), (da "The wall")
5 - "Is There Anybody Out There?" (02:44), (da "The wall")
6 - "The Court of the Crimson King" (09:26), (da "In the Court of the Crimson King"
7 - "Mother" (05:32), (da "The wall")
8 - "Batman" (01:58), (da "Naked City")
9 - "The Sicilian Clan" (03:27), (da "Naked City")
10 - "The James Bond Theme" (03:02), (da "Naked City")
Durata totale: 45:23

The wall
  "Another Brick in the Wall, Part 1" (03:11)
  "Another Brick in the Wall, Part 2" (03:59)
  "Hey You" (04:40)
  "Is There Anybody Out There?" (02:44)
  "Mother" (05:32)

In the Court of the Crimson King
  "21st Century Schizoid Man" (07:24)
  "The Court of the Crimson King" (09:26)

Naked City
  "Batman" (01:58)
  "The Sicilian Clan" (03:27)
  "The James Bond Theme" (03:02)
```

La soluzione

Le durate

Ogni oggetto del tema avrà a che fare con una durata (sia esso un brano, un album o una playlist) e su di esse sarà necessario fare un po' di conti (ad esempio sommarle per ottenere le durate totali, o farne la differenza se un brano verrà rimosso da una playlist). Per tale

Skip to main content

all'interno delle altre classi) saranno presumibilmente passate come parametri e restituite da diversi metodi, è bene che il tipo sia *immutabile* (per evitare la necessità di proteggerne ogni volta le istanze con delle copie).

Volendo adoperare una delle feature più recenti di Java, il tipo può essere definito tramite un `record`; chi non li avesse studiati, può definire il tipo in modo sostanzialmente equivalente tramite una classe concreta con un solo attributo dichiarato come `public final int durata`.

L'invariante di tale rappresentazione è che l'intero sia non negativo, lo zero non va escluso: servirà per rappresentare la durata delle playlist vuote; dato che il tipo è immutabile, sarà sufficiente accertarsi che questo sia vero in costruzione. Osservate che sebbene sarebbe possibile usare anche una stringa di formato `HH:MM:SS` come rappresentazione, essa sarebbe scomoda per effettuare le operazioni di somma e differenza.

Oltre al costruttore (che avrà per parametro il numero di secondi della durata), è bene avere un metodo statico di *fabbricazione* che costruisca una data a partire da una stringa nel formato `HH:MM:SS`, `MM:SS` o `SS`; è preferibile avere un metodo di fabbricazione, in modo che la classe abbia un unico costruttore (che diventa di fatto l'unico luogo dove controllare l'invariante di rappresentazione).

Il metodo di fabbricazione è tedioso, ma banale da implementare: basta dividere la stringa in parti col metodo `split` e sommare gli interi corrispondenti alle varie parti (dopo averli moltiplicati per l'opportuna potenza di 60)

```
static Durata valueOf(final String durata) {
  if (Objects.requireNonNull(durata, "La durata non può essere null.").isEmpty())
    throw new IllegalArgumentException("La durata non può essere vuota.");
  final String[] parti = durata.split(":");
  final int numParti = parti.length;
  if (numParti > 3)
    throw new IllegalArgumentException("L'orario non può comprendere \"💀" più di
  try {
    int ore = numParti < 3 ? 0 : toHMS(parti[numParti - 3], false);
    int minuti = numParti < 2 ? 0 : toHMS(parti[numParti - 2], true);
    int secondi = toHMS(parti[numParti - 1], true);
    return new Durata(3600 * ore + 60 * minuti + secondi);
  } catch (IllegalArgumentException e) {
    throw new IllegalArgumentException("Formato della durata invalito. " + e.getMes
  }
}
```

[sorgente]

Skip to main content

- sia non vuota,
- possa essere "convertita" in un intero (con `Integer.parseInt`),
- abbia un valore corretto: sia compresa tra 0 (incluso) e 60 (escluso) per minuti e secondi e sia positiva nel caso delle ore.

Può essere pertanto comodo un metodo statico di utilità che si accerti di queste condizioni e segnali, con una opportuna eccezione, il caso in cui siano violate

```
private static int toHMS(final String componente, final boolean bounded) {
  if (Objects.requireNonNull(componente, "La componente non può essere null.").isEm
    throw new IllegalArgumentException("La componente non può essere vuota.");
  int hms;
  try {
    hms = Integer.parseInt(componente);
  } catch (NumberFormatException e) {
    throw new IllegalArgumentException("La componente \"" + componente + "\" non è
  }
  if (hms < 0)
    throw new IllegalArgumentException(
        "Il valore della componente \"" + componente + "\" non può essere negativo.
  if (bounded && hms > 59)
    throw new IllegalArgumentException(
        "Il valore della componente \"" + componente + "\" deve essere minore di 60
  return hms;
}
```

[sorgente]

Il chiamante (ossia il metodo di fabbricazione), può far tesoro del messaggio dell'eccezione sollevata dal metodo di utilità avvolgendone tutte le chiamate in un unico `try-catch` e risollevando l'eccezione, riusandone il messaggio

```
try {
  // chiamate a hoMSS
} catch (IllegalArgumentException e) {
  throw new IllegalArgumentException("Formato della durata invalido. " + e.getMessa
}
```

I metodi di *produzione* che effettuano le operazioni sono banali, la somma

```
public Durata somma(final Durata altra) {
  return new Durata(
      this.secondi + Objects.requireNonNull(altra, "La durata non può essere null."
}
```

Skip to main content

deve solo controllare il caso di parametro nullo, mentre la differenza

```
public Durata sottrai(final Durata altra) {
  return new Durata(
      this.secondi - Objects.requireNonNull(altra, "La durata non può essere null."
}
```

[sorgente]

può delegare al costruttore (pur documentandolo) il controllo del fatto che dalla differenza non risulti una durata negativa.

Album e brani

Implementare i brani (e gli album che li contengono) non è banale, come anticipato nel suggerimento implementativo della traccia.

Dato un brano è necessario poter determinare l'album a cui appartiene. Non solo per poter distinguere brani di album diversi che abbiano accidentalmente il medesimo titolo, o per poter aggiungere il titolo dell'album a quello del brano emettendo il contenuto delle playlist (come risulta dagli esempi); ma sopratutto perché ha poco senso, in generale, parlare di un brano se non nel contesto dell'album a cui appartiene.

Per rappresentare questo legame sono possibili due scelte:

- implementare album e brani in classi indipendenti, memorizzando nel brano un riferimento all'album a cui appartiene,
- implementare il brano come una classe interna (non statica) dell'album.

Entrambe le scelte richiedono che il legame stabilito tra brano e album sia documentato nell'invariante di rappresentazione, costruito e preservato per tutta la durata di vita delle due entità. Osserviamo, peraltro, che non è viceversa accettabile che questo legame sia stabilito solo nella playlist.

Le due sezioni seguenti discutono molto approfonditamente le caratteristiche delle due scelte di cui sopra, chi è meno interessato ai dettagli può proseguire la lettura direttamente con la sezione sull'implementazione.

Skip to main content

Classi indipendenti

La prima soluzione può apparire più semplice, nel senso che non richiede dimestichezza con le classi interne, ma potrebbe nascondere alcuni problemi nel caso in cui i tipi fossero, come sarebbe opportuno, immutabili.

Il primo problema è che il costruttore di brano necessita di un album (per poter definire il valore del riferimento a esso) e il brano potrebbe avere solo un costruttore che richieda un elenco di brani; questo renderebbe le due classi non istanziabili: non ci sarebbe verso di creare un brano prima di un album, o un album prima di un brano!

A tal problema potrebbe essere posto rimedio facendo in modo che l'album abbia un costruttore che (come suggerito) invece di un elenco di brani riceva un elenco di titoli e di durate, provvedendo a costruire i brani al suo interno, dove sarà possibile usare il riferimento `this` come valore da passare al costruttore di album.

Un esempio di bozza del codice potrebbe essere il seguente

```
public class Brano {
  private final Album album;
  ...
  public Brano(final Album album, final String titolo, final Durata durata) {
    this.album = album;
    ...
  }
  ...
  public Album album() {
    return album;
  }
  ...
}

public class Album {
  private final Brano[] brani;
  ...
  public Album(List<String> titoli, List<Durate> durate, ...) {
    ...
    brani = new Brano[titoli.size()];
    for (int i = 0; i < titoli.size(); i++)
      brani[i] = new Brano(this, titoli[i], durate[i]);
    ...
  }
}
```

Il secondo problema è che il brano, per poter essere istanziato, deve avere (almeno) un

Skip to main content

che siano liberamente creati altri brani che si riferiscono a esso (oltre a quelli che contiene). Non è ovvio cioè come impedire che le classi vengano impiegate come segue

```
Album album = new Album(List.of("Primo", "Secondo"), List.of(new Durata(10), new Du
Brano terzo = new Brano(album, "Terzo", new Durata(30));
```

determinando la creazione di un terzo brano che non rappresenterebbe nulla di sensato.

Mentre è banale per l'invariante di rappresentazione dell'album controllare che in `brani` ci siano solo quelli il cui attributo `album` sia esso stesso

```
private boolean repOk() { // in Album
  for (final Brano brano : brani)
    if (brano.album() != this) return false;
  ...
}
```

non è però possibile adottare un atteggiamento simile nel brano; se `contiene` fosse un metodo dell'album che consentisse di determinare se un dato brano gli appartiene (ossia se è un elemento di `brani`), si potrebbe essere tentati di scrivere il seguente

```
private boolean repOk() { // in Brano
  if (!album.contains(this)) return false;
  ...
}
```

Questo di certo impedirebbe la creazione impropria del "Terzo" brano nell'esempio precedente, ma si finirebbe di nuovo in una condizione di non istanziabilità: talvolta è necessario creare un brano prima di aggiungerlo a un album! Nel costruttore stesso dell'album, l'istruzione

```
brani[i] = new Brano(this, titoli[i], durate[i]);
```

causa l'invocazione del `repOk` di brano (per via dell'istruzione `new`) che restituisce `false`, dato che al momento della costruzione l'assegnamento all'elemento dell'array non è ancora avvenuto!

Skip to main content

Il brano interno all'album

Le classi interne (*inner class*) sono lo strumento linguistico offerto da Java per modellare esattamente la circostanza in cui ci troviamo: quella di un oggetto (il brano) che ha senso solo se legato all'istanza di un altro (l'album).

Un esempio di bozza del codice con il brano interno all'album è

```
public class Album {

  public class Brano {
    private Brano(final String titolo, final Durata durata) {
      ...
    }
    ...
    public Album album() {
      return Album.this;
    }
  }
  ...
  private final Brano[] brani;
  ...
  public Album(List<String> titoli, List<Durate> durate, ...) {
    ...
    brani = new Brano[titoli.size()];
    for (int i = 0; i < titoli.size(); i++)
      brani[i] = new Brano(this, titoli[i], durate[i]);
    ...
  }
}
```

La necessità di realizzare il legame tra le istanze di brani e album è risolta in modo "automatico" dal linguaggio; in un brano è possibile ottenere il riferimento all'istanza di album che lo racchiude semplicemente con l'espressione `Album.this`.

Resta sempre il problema che non è possibile costruire un album se il suo costruttore richiede che ne siano indicati i brani, che a loro volta non possono essere costruiti prima di avere una istanza dell'album; per questa ragione, il costruttore dell'album riceve, come nella scelta precedente, una lista di titoli e durate.

Con la classe interna è però possibile risolvere il problema della creazione di ulteriori brani oltre a quelli contenuti nell'album. È sufficiente rendere il costruttore del brano `private` per far sì che esso possa essere invocato soltanto nel contesto del codice del'album, che può garantire di farlo solo nel modo adatto a garantire che, una volta creato un brano, esso gli

Skip to main content

L'implementazione del brano

Assumendo quindi di seguire il suggerimento implementativo del tema d'esame, procediamo con la descrizione della soluzione basata sulla classe interna.

La rappresentazione di un brano è data semplicemente da una stringa (che ne memorizzi il titolo) e da una durata che, essendo immutabili, possono essere lasciate pubbliche; l'invariante si limita a richiedere che non siano `null`, il titolo non sia vuoto e la durata non sia zero (codice evidenziato)

```
  public final String titolo;

  public final Durata durata;


  private Brano(final String titolo, final Durata durata) {
    if (Objects.requireNonNull(titolo, "Il titolo non può essere null.").isEmpty())
      throw new IllegalArgumentException("Il titolo non può essere vuoto.");
    if (Objects.requireNonNull(durata, "La durata non può essere null.").secondi() ==
      throw new IllegalArgumentException("La durata non può essere pari a zero.");
    this.titolo = titolo;
    this.durata = durata;
  }
```

[sorgente]

Avendo reso pubblici gli attributi non è necessario scrivere metodi "osservazionali" che li riguardino, può però avere senso aggiungerne alcuni come: un metodo che consenta di risalire da un brano all'album che lo contiene

```
  public Album album() {
    return Album.this;
  }
```

[sorgente]

e uno per sapere se il brano appartiene ad un dato album

```
  public boolean appartiene(final Album album) {
    return Album.this.equals(Objects.requireNonNull(album, "L'album non può essere nu
  }
```

Skip to main content

Infine può essere comodo avere un metodo che consenta di ottenere una rappresentazione come stinga che contenga, facoltativamente, anche l'indicazione del titolo dell'album (da usare nel `toString` di questa classe e quindi di quella delle playlist)

```
public String asString(final boolean conAlbum) {
  return String.format(
      "\"%s\" (%s)%s", titolo, durata, conAlbum ? ", (da \"" + album().titolo + "\"
}
```

[sorgente]

Non c'è alcun bisogno di definire i metodi `equals` e `hashCode` per i brani: è del tutto plausibile ritenere diverse anche due istanze con la stesso titolo e durata; si pensi ad esempio a un album corrispondente alla registrazione di un podcast in cui i brani siano una serie di interviste di titolo e durata differente alternate a uno "stacchetto" musicale che abbia sempre la stessa durata e "Intermezzo" per titolo.

L'implementazione dell'album

La rappresentazione dell'album, oltre al titolo, deve contenere un elenco ordinato di brani; dato che il loro numero è fissato ha senso usare un array (non c'è bisogno di scomodare le liste). L'array non deve contenere duplicati, ossia due riferimenti identici (la coincidenza dei titoli, come osservato in precedenza, non è viceversa proibita).

Può essere conveniente precomputare la durata complessiva (che resterà immutata, dato che l'elenco dei brani è fissato) e immagazzinarla in un attributo; in tal caso è però necessario specificare nell'invariante di rappresentazione la coincidenza tra il valore di tale attributo e la somma delle durate degli elementi dell'array.

Titolo e durata possono essere pubblici (sono infatti immutabili), ma certamente non può esserlo l'array (per quanto dichiarato `final`): renderlo pubblico consentirebbe di alterarne il contenuto!

Rappresentazione e costruttore sono quindi dati dal seguente codice:

```
public final String titolo;

public final Durata durata;

private final Brano[] brani;
```

Skip to main content

```
    this.titolo = Objects.requireNonNull(titolo, "Il titolo non può essere null.");
    if (titolo.isEmpty()) throw new IllegalArgumentException("Il titolo non può esser
    Objects.requireNonNull(titoli, "L'elenco dei titoli non può essere null.");
    Objects.requireNonNull(durate, "L'elenco delle durate non può essere null.");
    if (titoli.size() != durate.size())
      throw new IllegalArgumentException("Titoli e durate devono essere nello stesso
    if (titoli.isEmpty())
      throw new IllegalArgumentException("Il numero dei brani non può essere 0.");
    brani = new Brano[titoli.size()];
    Durata durata = new Durata(0);
    for (int i = 0; i < titoli.size(); i++) {
      final Durata di = durate.get(i);
      try {
        brani[i] = new Brano(titoli.get(i), di);
      } catch (IllegalArgumentException e) {
        throw new IllegalArgumentException(
            "Le specifiche del brano " + (i + 1) + " sono errate. " + e.getMessage())
      }
      durata = durata.somma(di);
    }
    this.durata = durata;
  }
```

[sorgente]

Si osservi che, per le ragioni illustrate in precedenza, il costruttore non riceve un elenco di brani, bensì due liste “parallele” di stringhe (i titoli) e durate; è compito del costruttore controllare che le liste abbiano la stessa dimensione, non siano vuote e che, una volta che i valori corrispondenti siano usati per costruire un brano, non venga sollevata una eccezione (che, nel caso, verrà rilanciata come eccezione del costruttore dell’album, con l’aggiunta dell’indicazione circa il numero di brano che l’ha causata).

Se i parametri sono accettati, il costruttore prosegue inizializzando gli attributi in modo che l’invariante descritto sia verificato; dato che gli attributi sono immutabili o, nel caso dell’array, non viene mai assegnato altro valore ad alcuno dei suoi elementi fuori dal costruttore, è ovvio constatare che l’invariante è sempre preservato.

Due dei metodi osservazionali richiesti dalla traccia sono relativi alla posizione dei brani

```
  public Brano brano(final int pos) {
    try {
      return brani[pos - 1];
    } catch (IndexOutOfBoundsException e) {
      throw new IndexOutOfBoundsException(
          "Il numero di brano non è compreso tra 1 e " + brani.length);
    }
  }

  public int posizione(final Brano brano) {
```

Skip to main content

```
    return 1 + Arrays.asList(brani).indexOf(brano);
  }
```

[sorgente]

Il metodo che consente di determinare la posizione di un brano nell'album adopera il metodo `indexOf` della lista ottenuta avvolgendo l'array col metodo `Arrays.asList`, ma potrebbe essere parimenti implementato con un ciclo `for`; l'uso di `indexOf` si basa sull'identità (non avendo ridefinito i metodi `equals` e `hashCode` in brano). Si osservi il dettaglio dato dal fatto che le posizioni sono corrette aggiungendo o togliendo 1 (a seconda dei casi); questo è dovuto al fatto che le posizioni nell'album corrispondono a interi positivi (mentre in generale negli array a interi non negativi).

Il metodo che consente di rintracciare un brano dato il titolo

```
  public Brano brano(final String titolo) {
    Objects.requireNonNull(titolo, "Il titolo non può essere null.");
    for (final Brano b : brani) if (b.titolo.equals(titolo)) return b;
    return null;
  }
```

[sorgente]

è più delicato da specificare: dal momento che sono possibili più brani col medesimo titolo, occorre specificare cosa accade nel caso di ripetizioni; qui è stato scelto di restituire il primo (nell'ordine in cui compaiono nell'album), ma, in alternativa sarebbe stato possibile scegliere di "sottospecificare".

Osservate che in nessun caso i metodi che cercano (dato un brano o un titolo) sollevano eccezione in caso di fallimento nella ricerca, ma piuttosto restituiscono un valore convenzionale (0 o `null`); questo è dovuto al fatto che in genere non costituisce condizione eccezionale cercare un elemento che non c'è (come si nota da molti metodi analoghi nelle API di Java). In questo modo, peraltro, le ricerche possono essere convenientemente usate anche per determinare l'esistenza (di un dato brano, o di un brano di dato titolo).

Diverso il discorso per il metodo che restituisce un brano data la sua posizione: dal momento che è possibile sapere a priori quali sono i valori corretti con cui invocarlo, grazie al metodo osservazionale

```
  public int numeroBrani() {
    return brani.length;
  }
```

Skip to main content

[sorgente]

la richiesta di un brano di posizione inesistente solleva una `IndexOutOfBoundsException` (conformemente a quel che farebbe un array o `List`) nel caso la posizione sia un indice che eccede i limiti legittimi.

Come richiesto, la classe permette l'iterazione sui suoi brani

```
@Override
public Iterator<Album.Brano> iterator() {
  return Arrays.asList(brani).iterator();
}
```

[sorgente]

l'uso di `Arrays.asList` consente di adoperare il metodo `iterator` delle liste invece di scrivere l'iteartore "a mano" (cosa comunque banale); osservate che l'array non è strutturalmente modificabile, ragione per cui l'iteratore non espone la rappresentazione (al rischio di modifiche dall'esterno) perché in caso di invocazione di `remove` solleverà una `UnsupportedOperationException`.

Le playlist

Le playlist hanno parecchie somiglianze con gli album, sono entrambi elenchi di brani (con una durata complessiva). Non bisogna però farsi trarre in inganno:

- gli album sono immutabili, le playlist no;
- gli album hanno un titolo, le playlist un nome (che potrebbero dover soddisfare vincoli diversi);
- gli album contengono sempre almeno un brano, le playlist possono essere vuote (o diventarlo);
- tutti i brani di un album sono relativi a quell'album, le playlist viceversa in genere contengono brani di album diversi;
- effettuare ricerche per titolo ha senz'altro senso in un album (anche in presenza di titoli ripetuti, caso comunque raro), in una playlist invece (dove è molto probabile che ci siano titoli ripetuti) la ricerca per titolo ha meno senso e dovrebbe quanto meno essere affiancata da quella per titolo e album;
- nell'emettere nel flusso d'uscita un album non ha senso riportarne il titolo per ogni

Skip to main content

Non appare quindi molto semplice raccogliere (senza forzature) caratteristiche così dissimili in un supertipo (ad esempio una classe astratta) che possa essere fruttuosamente utilizzato per definire album e playlist per estensione; il codice di cui consentirebbe di evitare la ripetizione molto verosimilmente si limiterebbe a quello di alcuni metodi osservazionali (che sono in ogni caso di banale implementazione).

A prescindere dalla difficoltà e dall'efficacia (in termini di risparmio di codice) derivante dalla definizione di un supertipo, ne sarebbe ancor più discutibile l'utilità. Nella traccia del progetto non c'è alcuna indicazione del fatto che potrebbe essere necessario sfruttare il polimorfismo per gestire in modo omogeneo playlist e album. Una classe astratta, o interfaccia, non sarebbe praticamente mai usata come tipo per nessuna delle variabili del progetto!

La rappresentazione di una playlist

```
private String nome;
private Durata durata = new Durata(0);
private final List<Album.Brano> brani = new ArrayList<>();
```

[sorgente]

prevede un `nome` (che potrebbe essere cambiato) e un elenco di `brani`, il cui numero può aumentare o diminuire (pertanto è più pratico mantenerlo in una lista). Come nel caso dell'album, può aver senso memorizzare la `durata` complessiva in un attributo, il cui valore dovrà essere aggiornato in caso di aggiunte o rimozioni in modo che risulti equivalente alla somma delle durate dei brani dell'elenco. Per queste ragioni, ad eccezione della lista che può essere allocata una volta per tutte, `nome` e `durata` non possono essere dichiarati `final` (dato che i loro tipi sono immutabili).

L'invariante di rappresentazione (oltre alle banali richieste circa i `null`, non ammessi per gli attributi e gli elementi della lista e il nome che non deve essere vuoto) deve semplicemente garantire che la durata corrisponda alla somma delle durate. Dato che la classe è mutabile, per ogni metodo mutazionale sarà però necessario riflettere sulla preservazione di tale invariante.

Per quanto riguarda il nome è utile avere la coppia di metodi

```
public String nome() {
  return nome;
```

Skip to main content

```
public void nome(final String nome) {
  if (Objects.requireNonNull(nome, "Il nome non può essere null.").isEmpty())
    throw new IllegalArgumentException("Il nome non può essere null o vuoto.");
  this.nome = nome;
}
```

[sorgente]

che consentano di conoscerlo o modificarlo (prestando attenzione a preservare l'invariante di rappresentazione); il costruttore, peraltro, consiste di fatto in una invocazione del secondo metodo sopra riportato.

Ci sono poi i metodi relativi ai brani in relazione al loro numero e posizione; essi sono molti simili a quelli per gli album (e costituiscono l'unica ripetizione effettiva di codice tra le due classi)

```
public int numeroBrani() {
  return brani.size();
}

public Album.Brano brano(final int pos) {
  try {
    return brani.get(pos - 1);
  } catch (IndexOutOfBoundsException e) {
    throw new IndexOutOfBoundsException(
        "Il numero di brano non è compreso tra 1 e " + brani.size());
  }
}

public int posizione(final Album.Brano brano) {
  return 1 + brani.indexOf(Objects.requireNonNull(brano, "Il brano non può essere n
}
```

[sorgente]

Dato che la playlist è mutabile occorrono almeno due metodi in grado di accodare o rimuovere un brano dato alla playlist

```
public void accoda(final Album.Brano brano) {
  brani.add(Objects.requireNonNull(brano, "Il brano non può essere null."));
  durata = durata.somma(brano.durata);
}

public void rimuovi(final Album.Brano brano) {
  if (brani.remove(Objects.requireNonNull(brano, "Il brano non può essere null."))
    durata = durata.sottrai(brano.durata);
}
```

Skip to main content

L'unica cosa degna di nota nelle implementazioni dei due metodi è l'osservazione che gli aggiornamenti della durata complessiva sono collocati in posizioni del codice che possono essere raggiunte se e solo se l'aggiunta o la rimozione avvengono effettivamente; questo consente di preservare l'invariante di rappresentazione. Ovviamente è possibile immaginare una messe di metodi analoghi, che funzionino anche tenendo conto della posizione, dell'album, o di combinazioni varie di tali parametri.

Osserviamo però che i due metodi scelti sono sufficienti a popolare completamente le playlist, nonché a sviluppare le funzionalità richieste dal resto del progetto e questo basta.

Un metodo di produzione consente di ottenere la fusione tra playlist

```
public Playlist fondi(final String nome, final Playlist altra) {
  final Playlist fusa = new Playlist(nome);
  for (final Album.Brano brano : this) fusa.accoda(brano);
  for (final Album.Brano brano :
      Objects.requireNonNull(altra, "La playlist non può essere null."))
    if (posizione(brano) == 0) fusa.accoda(brano);
  return fusa;
}
```

[sorgente]

Anche in questo caso, l'unica parte degna di nota è quel che accade accodando i brani della seconda lista, che non devono essere aggiunti se già presenti nella lista corrente (codice evidenziato); vale la pena osservare l'uso del metodo `posizione` per determinare se un brano è contenuto nella playlist.

Ora restano da implementare i tre iteratori richiesti. Il primo, che enumera tutti i brani,

```
public Playlist fondi(final String nome, final Playlist altra) {
  final Playlist fusa = new Playlist(nome);
  for (final Album.Brano brano : this) fusa.accoda(brano);
  for (final Album.Brano brano :
      Objects.requireNonNull(altra, "La playlist non può essere null."))
    if (posizione(brano) == 0) fusa.accoda(brano);
  return fusa;
}
```

[sorgente]

è banalmente ottenibile tramite l'iteratore della lista, a patto di avvolgendolo con `Collections.unmodifiableCollection` (per proteggere la rappresentazione).

Skip to main content

Gli altri due richiedono uno sforzo in più e la loro implementazione sarà descritta nelle sezioni seguenti.

Qui osserviamo solo che si potrebbe essere tentati dall'implementarli riempiendo dapprima una lista con gli elementi da iterare, restituendone quindi l'iteratore; tale soluzione è però non accettabile dal punto di vista dell'efficienza e vanifica i benefici derivanti dall'astrazione iterazione che consente di trattare gli elementi di una collezione uno alla volta (senza preallocare una copia di tutti gli elementi su cui iterare)!

Enumerare i brani di un dato album

Per enumerare i brani di un dato album è possibile costruire un iteratore come classe anonima; lo stato di tale iteratore

```
private final Iterator<Album.Brano> it = iterator();

private Album.Brano next = null;
```

[sorgente]

è dato da un iteratore su tutti i brani della playlist e dal prossimo brano da restituire (nonché dall'album dato come parametro, che è visibile in quanto parametro formale del metodo che crea l'iteratore).

Come avviene spesso, è opportuno implementare la logica di avanzamento nel metodo `hasNext`:

```
@Override
public boolean hasNext() {
  if (next != null) return true;
  while (it.hasNext()) {
    next = it.next();
    if (next.appartiene(album)) return true;
  }
  next = null;
  return false;
}
```

[sorgente]

Se `next` è diverso da `null` quello sarà il valore da restituire alla prossima invocazione dell'omonimo metodo. Viceversa, per sapere se sarà possibile restituire un nuovo brano, è

Skip to main content

dato; se ciò non accadesse, all'esaurimento dell'iteratore su tutti i brani, verrà segnalato che non ci sono più altri brani dell'album dato.

Il metodo `next` a questo punto è di banale implementazione:

```
@Override
public Album.Brano next() {
  if (!hasNext()) throw new NoSuchElementException();
  final Album.Brano ret = next;
  next = null;
  return ret;
}
```

[sorgente]

Dopo aver invocato il metodo `hasNext` per far eventualmente avanzare l'iteratore, è sufficiente restituire (se possibile) quanto riferito dall'attributo `next` e "invalidarlo" attribuendogli il valore `null`.

Enumerare gli album senza ripetizione

Anche in questo caso useremo una classe anonima, il cui stato

```
private final Iterator<Album.Brano> it = iterator();

private Album next = null;

private final Set<Album> restituiti = new HashSet<>();
```

[sorgente]

coincide con quello dell'iteratore precedente, a cui si aggiunge però un insieme utile a tener traccia degli album che sono già stati restituiti dal metodo `next`.

La logica di avanzamento, posta nel metodo `hasNext`,

```
@Override
public boolean hasNext() {
  if (next != null) return true;
  while (it.hasNext()) {
    next = it.next().album();
    if (!restituiti.contains(next)) {
      restituiti.add(next);
      return true;
```

Skip to main content

```
    next = null;
    return false;
  }
```

[sorgente]

è molto simile al caso precedente; si cercano album non ancora restituiti (annotando man mano nell'insieme quelli trovati). Il metodo `next` è di nuovo banale da implementare

```
  @Override
  public Album next() {
    if (!hasNext()) throw new NoSuchElementException();
    final Album ret = next;
    next = null;
    return ret;
  }
```

[sorgente]

risultando sostanzialmente identico a quello dell'iteratore precedente.

La classe di test

La classe di test è di implementazione tediosa, ma del tutto banale.

Il suo metodo `main` dovrà conservare un elenco di album (utili a popolare le playlist) e la playlist di nome "Fusa" (inizialmente vuota, a cui andranno fuse le playlist man mano che saranno lette)

```
  final List<Album> album = new ArrayList<>();
  Playlist fusa = new Playlist("Fusa");
```

[sorgente]

Il codice è diviso in una parte di lettura (di album e playlist) e una finale in cui viene stampata la playlist fusa (in vari modi, grazie anche agli iteratori su album e brani degli album). La lettura segue l'usuale schema bastato su uno `Scanner`

```
try (final Scanner s = new Scanner(System.in)) {
  while (s.hasNextLine()) {
    final String line = s.nextLine();
    if (line.startsWith("ALBUM")) {
```

Skip to main content

```
    } else if (line.startsWith("PLAYLIST")) {

      // lettura di una playlist

    }
  }
}
```

La lettura di un album, dopo averne individuato il titolo dalla linea corrente, inizia un nuovo ciclo di lettura in ciascun iterato del quale effettua la suddivisione della linea letta nelle parti relative a titolo e durata che usa per popolare le liste che saranno richieste dal costruttore dell'album

```
final String titolo = line.substring(6);
final List<String> titoli = new ArrayList<>();
final List<Durata> durate = new ArrayList<>();
while (s.hasNextLine()) {
  final String aLine = s.nextLine();
  if (aLine.equals(".")) {
    final Album a = new Album(titolo, titoli, durate);
    album.add(a);
    System.out.println(a);
    break;
  }
  final String[] p = aLine.split("-", 2);
  durate.add(Durata.valueOf(p[0].strip()));
  titoli.add(p[1].strip());
}
```

[sorgente]

quando incontra una linea costituita da un solo punto (codice evidenziato) interrompe il ciclo interno di lettura, non prima di aver costruito l'album, averlo aggiunto alla lista e averlo emesso nel flusso d'uscita.

La lettura di una playlist può essere implementata in maniera simile: dopo averne individuato il nome dalla linea corrente, si inizia un nuovo ciclo di lettura. In ciascun iterato è effettuata la suddivisione della linea letta nelle parti relative al numero di album (che va corretto di 1, dato che le posizioni della lista in cui sono contenuti gli album partono da 0) e del brano da aggiungere

```
final Playlist pl = new Playlist(line.substring(9));
while (s.hasNextLine()) {
  final String plLine = s.nextLine();
  if (plLine.equals(".")) {
    fusa = fusa.fondi("Fusa", pl);
    System.out.println(pl);
```

Skip to main content

```
    final String[] p = plLine.split(" ", 2);
    final int aIdx = Integer.parseInt(p[0].strip()) - 1;
    final int bIdx = Integer.parseInt(p[1].strip());
    pl.accoda(album.get(aIdx).brano(bIdx));
  }
```

[sorgente]

anche in questo caso, quando si incontra una linea costituita da un solo punto (codice evidenziato) interrompe il ciclo interno di lettura, non prima di aver costruito la playlist, averla fusa a quella di nome “Fusa” e averla emessa nel flusso d’uscita.

A questo punto resta solo da emettere la playlist di nome “Fusa” e, con due cicli innestati, stamparne gli album e per ciascuno di essi i brani (ottenuti grazie ai due iteratori appositi della classe `Playlist`)

```
  System.out.println(fusa);
  final Iterator<Album> ait = fusa.album();
  while (ait.hasNext()) {
    final Album a = ait.next();
    System.out.println(a.titolo);
    final Iterator<Album.Brano> bit = fusa.brani(a);
    while (bit.hasNext()) System.out.println("\t" + bit.next());
  }
```

[sorgente]