Basandomi sui numerosi esercizi presenti nel documento (Fattoriali, Range, Iterator su Array, Filtri, Concatenazioni, ecc.), ho elaborato uno **schema mentale e operativo** per risolvere qualsiasi esercizio sugli Iterator in Java.
L'esercizio tipo richiede solitamente di implementare due interfacce:

1. **Iterable<T>** (Il contenitore/sequenza): definisce *cosa* iterare.
2. **Iterator<T>** (Il cursore): tiene traccia di *dove* siamo e *come* avanzare.

1. Fase di Analisi: Cosa sto iterando?

Prima di scrivere codice, identifica in quale di queste tre categorie rientra l'esercizio:

- **A. Iterazione su Dati Esistenti (Array/Liste):** Hai un array o una lista in memoria e devi scorrerlo (avanti, indietro, saltando elementi).
  - *Esempio dal PDF:* StringArrayIterator, ReverseRangeIterator, CharSeqIterator.
- **B. Generazione Generativa (Matematica):** I dati non esistono in memoria, vengono calcolati al volo.
  - *Esempio dal PDF:* FactIterator, FibIterator, PowIterator, RangeIterator.
- **C. Decorator/Wrapper (Composizione):** Hai già un altro iteratore (o più di uno) e devi modificarne il comportamento (filtrare, unire, trasformare).
  - *Esempio dal PDF:* FilteredIterator, AltIterator, Zipper, CatIterator.

2. Lo Schema della Classe Iterable (Il "Factory")

Questa classe è solitamente semplice. Il suo unico scopo è creare e restituire una nuova istanza dell'Iterator.

```
public class MiaSequenza implements Iterable<TipoDato> {
    // CAMPI: I parametri che definiscono la sequenza (es. start, end, array)
    private final int parametro1;

    // COSTRUTTORE: Inizializza i campi (spesso controlla null o validità)
    public MiaSequenza(int p1) { this.parametro1 = p1; }

    @Override
    public Iterator<TipoDato> iterator() {
        // Restituisce un NUOVO iteratore passando i parametri necessari
        return new MioIterator(parametro1);
    }
}
```

3. Lo Schema della Classe Iterator (Il Cuore)

Qui avviene la logica vera e propria. Devi gestire lo **STATO**.

A. Definizione dei Campi (Lo Stato)

Chiediti: *"Cosa devo ricordare tra una chiamata di next() e l'altra per sapere dove sono?"*

- **Caso Array:** Serve un int index.
- **Caso Generativo:** Serve il currentValue (valore attuale) e spesso un count o un limit per sapere quando fermarsi.
- **Caso Wrapper:** Serve un riferimento all'iteratore originale (private final Iterator<T> it;) e spesso un campo "cache" per il prossimo elemento (nextElement).

B. Il Metodo hasNext()

Deve restituire true se c'è ancora qualcosa, false altrimenti. **Non deve mai modificare lo stato (idempotente).**

- **Logica Array:** return index < array.length;
- **Logica Generativa:** return current < limit; (o condizione matematica).
- **Logica Wrapper (Complessa):** Spesso delega all'iteratore interno (it.hasNext()) o controlla se la "cache" è piena (vedi sezione *Pattern Avanzato* sotto).

C. Il Metodo next()

Esegue tre azioni fondamentali in quest'ordine preciso:

1. **Check:** Controlla se esiste l'elemento.
   ```
   if (!hasNext()) throw new NoSuchElementException();
   ```
2. **Capture & Advance:** Salva il valore da restituire e prepara lo stato per la *prossima* volta.
   - *Array:* var res = array[index]; index++;
   - *Generativo:* var res = current; current = calcolaProssimo(current);
3. **Return:** Restituisce il valore salvato.

4. I Tre Pattern Ricorrenti nel PDF

Ecco come applicare lo schema ai casi specifici trovati nel documento.

Pattern 1: Generatore Matematico (es. Fibonacci, Fattoriale)

- **Campi:** currentValue, nextValue (per Fibonacci), max/limit.
- **Costruttore:** Imposta il primo valore.
- **hasNext:** return currentValue <= max;
- **next:**
  ```
  var res = currentValue;
  // Calcola il prossimo stato
  currentValue = ... (es. current * index);
  return res;
  ```

Pattern 2: Iteratore su Array/Sequenza (es. ReverseRange, StringArray)

- **Campi:** index, riferimento all'array/dati.
- **Costruttore:** Imposta index a 0 (o length-1 per reverse).
- **hasNext:** return index < length; (o index >= 0).
- **next:** return array[index++]; (o index--).

Pattern 3: Il "Wrapper" con Lookahead (es. FilteredIterator, MergeIterator)

Questo è il più difficile (es. FilteredIterator, CondIterator). Poiché non sai se il *prossimo* elemento dell'iteratore sorgente è valido finché non lo controlli, devi usare una **Cache** (o Lookahead).

- **Campi:**
  - Iterator<T> source;
  - T nextItem; (La cache)
  - boolean hasNextItem; (Flag validità cache)
- **Logica:** Devi creare un metodo privato (es. findNext()) che cerca il prossimo elemento valido e lo

mette in nextItem.

```
// Esempio schema FilteredIterator
private void findNext() {
    hasNextItem = false;
    while (source.hasNext()) {
        T cand = source.next();
        if (predicato.test(cand)) { // Se soddisfa la condizione
            nextItem = cand;
            hasNextItem = true;
            break;
        }
    }
}

public boolean hasNext() {
    return hasNextItem; // Risponde basandosi sulla cache
}

public T next() {
    if (!hasNextItem) throw new NoSuchElementException();
    T res = nextItem;
    findNext(); // PREPARA GIÀ IL PROSSIMO per il futuro
    return res;
}
```

(Nota: Nel costruttore bisogna chiamare findNext() una volta per inizializzare).

5. Checklist per l'Esame

Quando scrivi il codice su carta/PDF, verifica questi punti (spesso valutati dai professori):

1. **Check Nullità:** Nel costruttore, se passi array o iteratori, usa Objects.requireNonNull().
2. **Eccezione:** next() lancia **sempre** NoSuchElementException se hasNext() è false.
3. **Immutabilità hasNext:** hasNext() non deve mai "consumare" l'elemento (non incrementare indici qui, a meno che non usi il pattern lookahead complesso).
4. **Complessità Costante:** Se richiesto (es. Fibonacci nel PDF), non ricalcolare tutta la sequenza ogni volta. Memorizza l'ultimo valore.
5. **Clean Code:** Usa @Override sulle firme dei metodi.

Esempio Pratico (Applicando lo schema a FactIterator del PDF)

1. **Analisi:** È di tipo Generativo (Matematico).
2. **Stato:** Mi serve sapere a che numero sono (n) e quanto vale il fattoriale corrente (fact).
3. **Codice:**
    - *Campi:* long currentFact = 1; int index = 1; int max;
    - *hasNext:* return index <= max;
    - *next:*
        - Check !hasNext -> Exception.
        - Salva res = currentFact.
        - Aggiorna: index++; currentFact *= index.
        - Return res.