ITERATOR DESIGN PATTERN

- The **Iterator Design Pattern** is **a behavioral pattern** that provides a standard way to **access elements of a collection sequentially** without exposing its internal structure.
- You need to **traverse a collection** (like a list, tree, or graph) in a consistent and flexible way.
- You want to support **multiple ways to iterate** (e.g., forward, backward, filtering, or skipping elements).
- You want to **decouple traversal logic from collection structure**, so the client doesn't depend on the internal representation.
- When faced with this need, **developers often write custom for loops or expose the underlying data structures** (like ArrayList or LinkedList) directly
- The **Iterator Pattern solves** this by abstracting the iteration logic into a dedicated object — the iterator. Collections provide an iterator via a method like createIterator(), and the client uses it to access elements one by one.

THE PROBLEM: TRAVERSING A PLAYLIST

- Imagine you're building a **music streaming app** that allows users to create and manage playlists. Each playlist stores a list of songs and provides features like:
    - Playing songs one by one
    - Skipping to the next or previous song
    - Shuffling songs
    - Displaying the current song queue

```
class Playlist {
   private List<String> songs = new ArrayList<>();

   public void addSong(String song) {
      songs.add(song);
   }

   public List<String> getSongs() {
      return songs;
   }
}
```

- Client Code

```
class MusicPlayer {
   public void playAll(Playlist playlist) {
      for (String song : playlist.getSongs()) {
         System.out.println("Playing: " + song);
```

```
    }
  }
}
```

WHY THIS IS A PROBLEM

- **Breaks Encapsulation**
  By exposing the internal list of songs (getSongs()), you allow clients to directly modify the collection. This can lead to unintended side effects, like removing songs from the list, reordering them, or injecting nulls
- **Tightly Couples Client to Collection Type**
  The client assumes that the playlist uses a List<String>. If you ever decide to change the internal storage (e.g., to a LinkedList, array, or even a streaming buffer), the client code breaks.
- **Limited Traversal Options**
  The client is stuck with the default iteration order. Supporting multiple traversal styles (e.g., reverse, shuffled, filtered) requires rewriting the loop logic every time — violating the Single Responsibility and Open/Closed principles.
- **Duplication and Rigidity**
  As more features are added (e.g., previewing songs, playing only favorites), the logic for traversing songs gets duplicated across multiple classes.

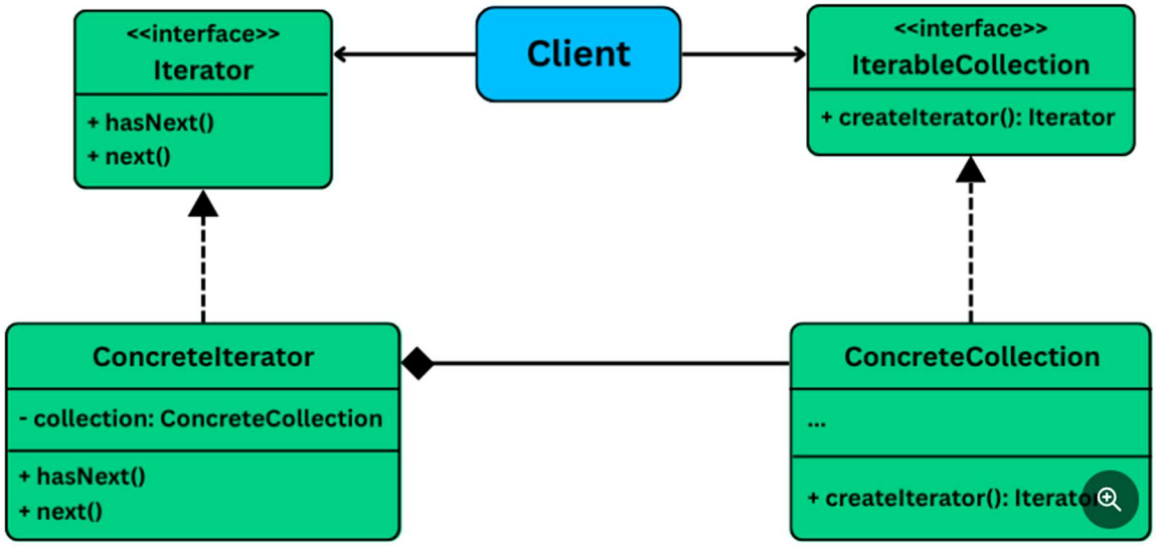
WHAT WE REALLY NEED

- Traverse songs in a **playlist without exposing the internal collection**
- Allow different traversal styles (forward, reverse, shuffle)
- **Abstract the iteration logic** from the data structure itself
- Preserve **encapsulation** and allow the playlist to change internally without affecting client code

WHAT IS THE ITERATOR PATTERN

- The **Iterator Pattern** provides a standard way **to traverse elements in a collection** without exposing its internal structure.
- Instead of letting clients access the underlying data (like an ArrayList), the collection provides an **iterator object** that offers sequential access to its elements through a common interface (e.g., hasNext(), next()).

Class Diagram



1. **Iterator (interface)**
    - Defines the contract for traversing a collection. Declares methods for traversing elements like:
    - hasNext()— checks if there are more elements
    - next()— returns the next element in the sequence
2. **ConcreteIterator**
    - Implements the Iterator interface for a specific collection.
    - Maintains the current position in the collection and iterates over it one item at a time**.**
3. **IterableCollection (interface)**
    - Defines a method for creating an iterator.
    - Separates the collection's structure from traversal logic
4. **ConcreteCollection**
    - Stores the actual data and implements the IterableCollection interface.
    - It delegates traversal logic to the iterator.
    - Returns iterator to traverse collection items

IMPLEMENTING ITERATOR

- Let’s now implement **the Iterator Pattern** to decouple the way we traverse songs in a playlist from how the playlist is internally structured.
- We’ll build:

- A Playlist that stores songs
- A PlaylistIterator that can move through the playlist

- A client that uses the iterator without needing to know how the playlist stores its songs

1. Define the Iterator Interface

```
interface Iterator<T> {
   boolean hasNext();
   T next();
}
```

2. Define the IterableCollection Interface

```
    interface IterableCollection<T> {
   Iterator<T> createIterator();
}
```

3. **Implement the Concrete Collection – Playlist**

```
class Playlist implements IterableCollection<String> {
   private final List<String> songs = new ArrayList<>();

   public void addSong(String song) {
      songs.add(song);
   }

   public String getSongAt(int index) {
      return songs.get(index);
   }

   public int getSize() {
      return songs.size();
   }

   @Override
   public Iterator<String> createIterator() {
      return new PlaylistIterator(this);
   }
}
```

4. Implement the Concrete Iterator – PlaylistIterator

```
class PlaylistIterator implements Iterator<String> {
   private final Playlist playlist;
   private int index = 0;

   public PlaylistIterator(Playlist playlist) {
      this.playlist = playlist;
```

```
    }

    @Override
    public boolean hasNext() {
        return index < playlist.getSize();
    }

    @Override
    public String next() {
        return playlist.getSongAt(index++);
    }
}
```

5. Client Code – Using the Iterator

```
public class MusicPlayer {
    public static void main(String[] args) {
        Playlist playlist = new Playlist();
        playlist.addSong("Shape of You");
        playlist.addSong("Bohemian Rhapsody");
        playlist.addSong("Blinding Lights");

        Iterator<String> iterator = playlist.createIterator();

        System.out.println("Now Playing:");
        while (iterator.hasNext()) {
            System.out.println(" ♪  " + iterator.next());
        }
    }
}
```

WHAT WE ACHIEVED

- **Encapsulation**: The client never accesses the internal list directly
- **Consistent traversal interface**: All iterators follow the same hasNext()/next() pattern
- **Open/Closed Principle**: We can add new iterators (reverse, shuffled) without changing the playlist or player
- **Flexible architecture**: Works with different collection types, not just lists
- Reusable logic: The iterator can be used in any context that needs to traverse the playlist

OTHER REAL-TIME EXAMPLES

- **E-commerce Cart** → Iterating products in shopping cart.
- **Library System** → Iterating books in a catalog.

- **Social Media Feed** → Iterating over posts without knowing DB structure.
- **File Explorer** → Iterating files inside a folder.