**Iterator Design Pattern:** The Iterator design pattern is a [behavioral design pattern](https://www.geeksforgeeks.org/behavioral-design-patterns/" \t "_blank) that provides a way to access the elements of an aggregate object (like a list) sequentially without exposing its underlying representation. It defines a separate object, called an iterator, which encapsulates the details of traversing the elements of the aggregate, allowing the aggregate to change its internal structure without affecting the way its elements are accessed.

* Iterator Pattern is a relatively simple and frequently used design pattern. There are a lot of data structures/collections available in every language.
* Each collection must provide an iterator that lets it iterate through its objects. However, while doing so it should make sure that it does not expose its implementation.

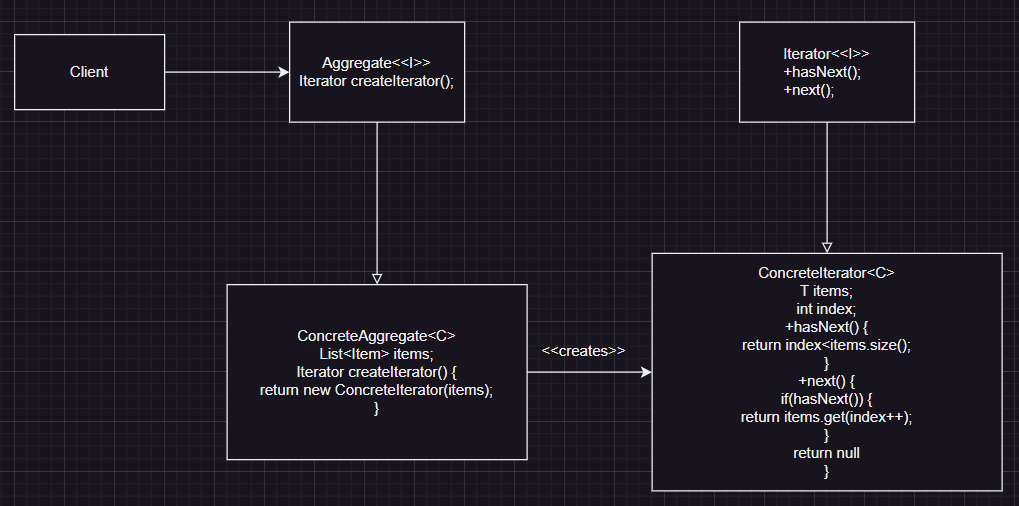
Example: Let’s say we have a collection of employees in a company, and we want to iterate over the employees to calculate their total salary. However, the employees are stored in different types of collections (arrays, lists, etc.), and we want to iterate over them without exposing the underlying collection types.

**When to use Iterator Design Pattern**

* **Need for sequential access:** Use the Iterator pattern when you need to access elements of a collection sequentially without exposing its underlying representation. This pattern provides a uniform way to iterate over different types of collections.
* **Decoupling iteration logic:** Use the Iterator pattern when you want to decouple the iteration logic from the collection. This allows the collection to change its internal structure without affecting the way its elements are accessed.
* **Support for multiple iterators:** Use the Iterator pattern when you need to support multiple iterators over the same collection. Each iterator maintains its own iteration state, allowing multiple iterations to occur concurrently.
* **Simplifying client code:** Use the Iterator pattern to simplify client code that iterates over a collection. Clients only need to interact with the iterator interface, abstracting away the complexity of the collection’s internal structure.

**When to not use Iterator Design Pattern**

* **When the collection is not accessed sequentially:** If the collection is not accessed sequentially, using the Iterator pattern may add unnecessary complexity. Consider other patterns or direct access methods based on the specific access patterns required.
* **When the collection structure is fixed:** If the structure of the collection is fixed and unlikely to change, using the Iterator pattern may be overkill. Direct access methods may be more appropriate and simpler to implement.
* **When performance is critical:** In performance-critical applications, the overhead of using iterators may be significant, especially if the collection is large. In such cases, consider direct access methods for better performance.
* **When the language provides better alternatives:** Some languages provide built-in constructs or libraries that offer more efficient ways to iterate over collections. In such cases, using these alternatives may be more appropriate than implementing the Iterator pattern from scratch.



In the above diagram we showed that Aggregate interface can have different implementations, & each implementation is responsible for having respective iterator based on its underlying data structure. Client no need to worry about internal implementation.

Iterator provides an abstraction for implementation so client can simply use hasNext() & next() methods, aggregate will take care which iterator need to use to iterate its underlying data structure.