Iterator Pattern

- Definition
- Real-world Usage
  - Code Example
- The Problem: Data exposure to the Client
- Class Diagram
- Structure of the Iterator Design Pattern
- Implementation(Example: Library)
  - Output

Resources

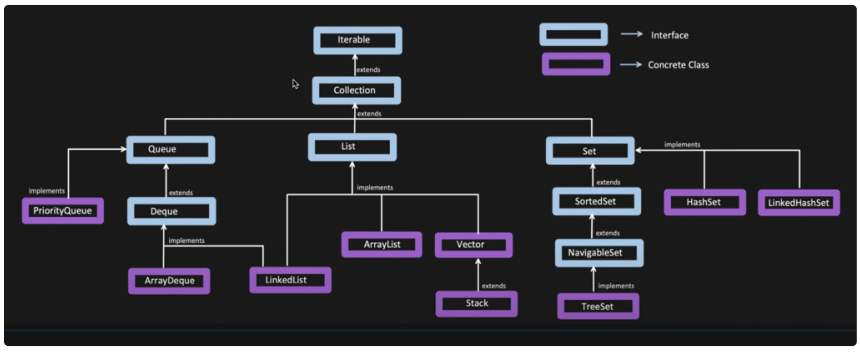
- 41. All Behavioral Design Patterns | Strategy, Observer, State, Template, Command, Visitor, Memento
- 33. Iterator Design Pattern Explained with Example | Low Level Design

Definition

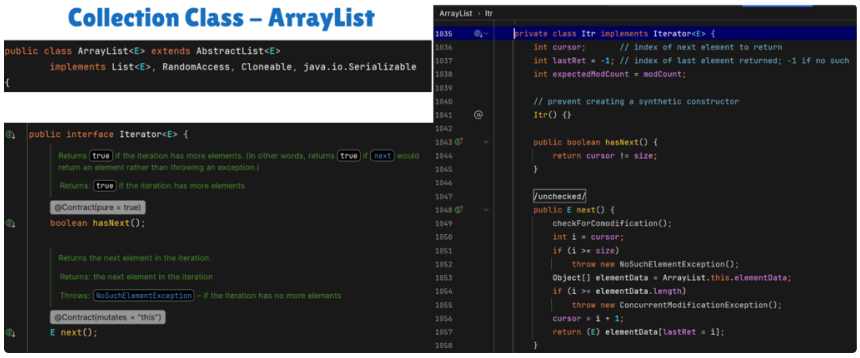
> **The Iterator design pattern is one of the behavioral design patterns that provides a way to access elements of a Collection sequentially without exposing the underlying representation of the collection.**

Real-world Usage

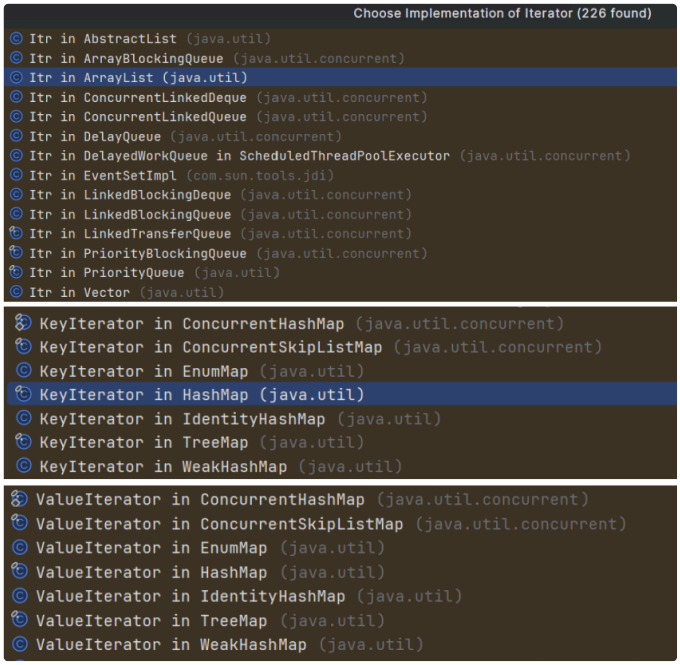
In the Java Collections Framework, the collection interface includes methods, such as `iterator()`, that allow a client to obtain an Iterator object from any collection that implements it. All collection classes have iterator implementation, e.g., (`ArrayList.iterator()`, `HashSet.iterator()`, `TreeSet.descendingIterator()`, etc).



Java Collections Framework



ArrayList implementation of Iterator



Iterator Interface Implementations

Code Example

```java
// Java Collections Usage Example
public class LinkedHashSetExample {
    public static void main(String[] args) {
        Set<Integer> intSet = new LinkedHashSet<>();
        intSet.add(2);
        intSet.add(77);
        intSet.add(82);
        intSet.add(63);
        intSet.add(5);

        // Common to all Collection Classes
        Iterator<Integer> iterable = intSet.iterator();
        while (iterable.hasNext()) {
            int value = iterable.next();
            System.out.println(value);
        }
    }
}
```

The Problem: Data exposure to the Client

```java
public class Book {
    private final String title;
    private final String author;
    private final String isbn;
```

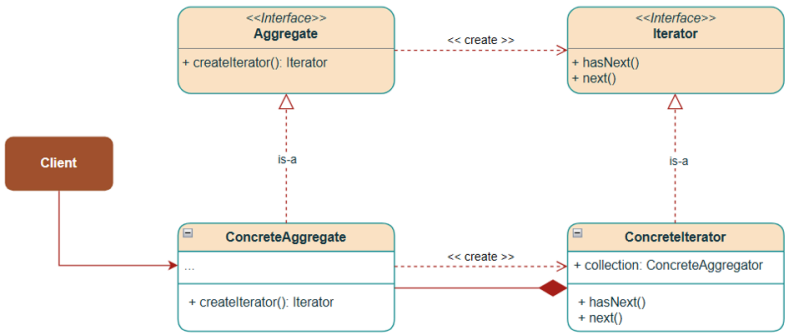
```
5
6       public Book(String tittle, String author, String isbn) {
7           this.title = tittle;
8           this.author = author;
9           this.isbn = isbn;
10      }
11
12      public static List<Book> getBooks() {
13          List<Book> books = List.of(
14                  new Book("To Kill a Mockingbird", "Harper Lee", "978-
    0-74-7356-5"),
15                  new Book("The Great Gatsby", "F. Scott Fitzgerald",
    "778-0-24-7156-5"),
16                  new Book("The Catcher in the Rye", "J.D. Salinger",
    "333-0-28-7446-8"),
17                  new Book("The Hobbit", "J.R.R. Tolkien", "783-0-14-
    1951-8"),
18                  new Book("Rich Dad Poor Dad", "Robert Kiyosaki", "183-
    0-12-1491-8"),
19                  new Book("Pride and Prejudice", "Jane Austen", "289-0-
    12-1678-8")
20          );
21          return books;
22      }
23
24      @Override
25      public String toString() {
26          return "Book [" + "Tittle='" + title + ", Author=" + author +
    ", age=" + "ISBN=" + isbn + "]";
27      }
28  }
```

```
1   public class Client {
2       public static void main(String[] args) {
3           System.out.println("\n###### Problem without Iterator Pattern
    Demo ######");
4           // Client has access to the entire book list in the library
5           List<Book> bookList = Book.getBooks();
6           for (Book book : bookList) {
7               System.out.println(book);
8           }
9       }
10  }
```

Problems with the above code are that:

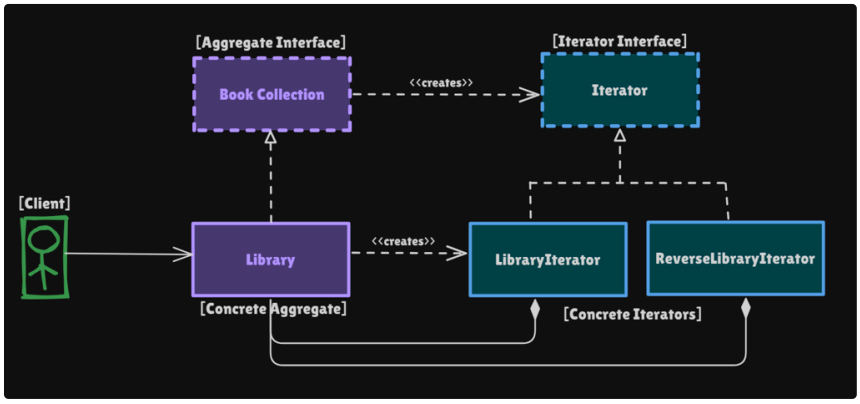
- No data encapsulation allows the client can modify data and access hidden information.
- Tight coupling of traversal logic to data structure and client.
- Violates SOLID principles if we try to implement new traversal logic.

Class Diagram



Structure of the Iterator Design Pattern

Let's understand the Structure of the Iterator Design Pattern using a Library example:



- **Iterator Interface(** `Iterator<T>` **)**: Defines the contract for traversing a collection with `hasNext()` and `next()` methods.
- **Concrete Iterator**: Implements the traversal logic for a specific collection type. It tracks the current position while traversing the collection.
  - `LibraryIterator` → traverses books from first to last.
  - `DescendingLibraryIterator` → traverses books from last to first by starting at `numberOfBooks - 1` and decrementing the position.
- **Aggregate Interface(** `BookCollection` **)**: Declares a method to create iterators (`createIterator()` and `createReverseIterator()`).
- **Concrete Aggregate(** `Library` **)**: The actual collection of books (`Library` class) that implements the aggregate interface(creates and returns **Concrete Iterators** `LibraryIterator` **and** `DescendingLibraryIterator`).

Implementation(Example: Library)

```java
// Book class representing individual books in a library
public class Book {
    private final String title;
    private final String author;
    private final String isbn;
    private int price;

    public Book(String title, String author, String isbn) {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public String getIsbn() {
        return isbn;
    }
```

```
25
26      public int getPrice() {
27          return price;
28      }
29
30      @Override
31      public String toString() {
32          return "Book [Title=" + title + ", Author=" + author + ",
 ISBN=" + isbn + "]";
33      }
34  }
```

```
1  // Aggregate interface
2  public interface BookCollection {
3      Iterator<Book> createIterator();
4
5      Iterator<Book> createReverseIterator();
6  }
```

```
1  // Iterator interface
2  public interface Iterator<T> {
3      boolean hasNext();
4
5      T next();
6  }
```

```
1   // Concrete Aggregate
2   public class Library implements BookCollection {
3
4       private final List<Book> books;
5
6       public Library(List<Book> books) {
7           this.books = books;
8       }
9
10      @Override
11      public Iterator<Book> createIterator() {
12          return new LibraryIterator(books);
13      }
14
15      @Override
16      public Iterator<Book> createReverseIterator() {
17          return new ReverseLibraryIterator(books);
18      }
19  }
```

```
1   // Concrete Iterator - for Library
2   public class LibraryIterator implements Iterator<Book> {
3       private final List<Book> books;
4       private int position = 0;
5
6       public LibraryIterator(List<Book> books) {
7           this.books = books;
8       }
9
10      @Override
11      public boolean hasNext() {
12          return position < books.size();
13      }
14
15      @Override
16      public Book next() {
17          return books.get(position++);
18      }
19  }
```

```
1   // Concrete Iterator - for Library
2   public class ReverseLibraryIterator implements Iterator<Book> {
3       private final List<Book> books;
4       private int position;
```

```
    public ReverseLibraryIterator(List<Book> books) {
        this.books = books;
        this.position = books.size() - 1;
    }

    @Override
    public boolean hasNext() {
        return position >= 0 && books.get(position) != null;
    }

    @Override
    public Book next() {
        if (!hasNext()) {
            return null;
        }
        return books.get(position--); // Return current book and move backward
    }
}
```

```
// Client
public class LibraryIteratorDemo {

    public static void main(String[] args) {
        System.out.println("\n###### Iterator Design Pattern ######");

        // Create Library
        Library library = getLibrary();

        // Forward iteration
        Iterator<Book> iterator = library.createIterator();
        System.out.println("\n==> Forward iteration:");
        displayLibrary(iterator);

        // Reverse iteration
        Iterator<Book> reverseIterator = library.createReverseIterator();
        System.out.println("\n==> Reverse iteration:");
        displayLibrary(reverseIterator);
    }

    private static Library getLibrary() {
        List<Book> books = List.of(
                new Book("To Kill a Mockingbird", "Harper Lee", "978-0-74-7356-5"),
                new Book("The Great Gatsby", "F. Scott Fitzgerald", "778-0-24-7156-5"),
                new Book("The Catcher in the Rye", "J.D. Salinger", "333-0-28-7446-8"),
                new Book("The Hobbit", "J.R.R. Tolkien", "783-0-14-1951-8"),
                new Book("Rich Dad Poor Dad", "Robert Kiyosaki", "183-0-12-1491-8"),
                new Book("Pride and Prejudice", "Jane Austen", "289-0-12-1678-8")
        );
        Library library = new Library(books);
        return library;
    }

    private static void displayLibrary(Iterator<Book> iterator) {
        while (iterator.hasNext()) {
            Book book = iterator.next();
            System.out.println(book);
        }
    }
}
```

Output

```
###### Iterator Design Pattern ######

==> Forward iteration:
Book [Title=To Kill a Mockingbird, Author=Harper Lee, ISBN=978-0-74-7356-5]
Book [Title=The Great Gatsby, Author=F. Scott Fitzgerald, ISBN=778-0-24-7156-5]
Book [Title=The Catcher in the Rye, Author=J.D. Salinger, ISBN=333-0-28-7446-8]
Book [Title=The Hobbit, Author=J.R.R. Tolkien, ISBN=783-0-14-1951-8]
Book [Title=Rich Dad Poor Dad, Author=Robert Kiyosaki, ISBN=183-0-12-1491-8]
Book [Title=Pride and Prejudice, Author=Jane Austen, ISBN=289-0-12-1678-8]

==> Reverse iteration:
Book [Title=Pride and Prejudice, Author=Jane Austen, ISBN=289-0-12-1678-8]
Book [Title=Rich Dad Poor Dad, Author=Robert Kiyosaki, ISBN=183-0-12-1491-8]
Book [Title=The Hobbit, Author=J.R.R. Tolkien, ISBN=783-0-14-1951-8]
Book [Title=The Catcher in the Rye, Author=J.D. Salinger, ISBN=333-0-28-7446-8]
Book [Title=The Great Gatsby, Author=F. Scott Fitzgerald, ISBN=778-0-24-7156-5]
Book [Title=To Kill a Mockingbird, Author=Harper Lee, ISBN=978-0-74-7356-5]

Process finished with exit code 0
```