# Design Patterns, Iterators, & Iterable

## Design Patterns

- Design Pattern: A classic approach for solving a common problem that arises when writing code.
- Creational Patterns:
    - Abstract Factory
    - Builder
    - Factory Method
    - Prototype
    - Singleton
- Structural Patterns:
    - Adapter
    - Bridge
    - Composite
    - Decorator
    - Facade
    - Flyweight
    - Proxy
- Behavioral Patterns:
    - Chain of responsibility
    - Command
    - Interpreter
    - Iterator

    - Mediator
    - Memento
    - Observer
    - State
    - Strategy
    - Template Method
    - Visitor

## Iterator Interface

- **The Iterator Design Pattern:**
    - Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
    - Common Operation: Loops through the items in a collection, one at a time.
- **Java's Support for the Iterator Pattern:**
    - The Iterator pattern is so well-known that Java (and most other programming languages) offer built-in language support.
    - Two built-in interfaces:
        - `Iterator<T>`
        - `Iterable<T>`
    - Iterator supports all built-in collections:
        - `List`
        - `Set`
        - `Map`
- **Java's `Iterator<T>` Interface:**
    - `T` is called a generic type, and it represents the type of the items stored in the collection.

    - `hasNext()` method: (Required and default) Answers the question: "Are there still remaining items to check?"
        - Returns a `boolean`.
    - `next()` method: (Required and default) Returns the next item in the collection.
        - Returns an object of type `T`.
        - Throws a `NoSuchElementException` after all elements in the collection are seen.
- **The iterator pattern usually assumes that the collection *will not be modified while the iterator is active.***
    - Collection is read-only when iterator is active.
- **Designing an Iterator Class:**
    - The iterator object tracks progress through the collection.
    - Knows which items have been seen, and which are coming up next.
    - Manages the order of items, but DOES NOT modify the underlying collection.

    ⇒ At the bare minimum, the iterator object must have:

    1. Access to the collection.
    2. A way to track which items have been seen.
- **Iterator Design Strategy #1:**
    - Encapsulate the raw collection.
    - Add a 'cursor' field to track progress through the collection (like an index).
    - Each time `next()` is called, update the cursor field.
    - **Advantages**:
        - Memory-efficient (does not require cloning the collection)
    - **Disadvantages**:
        - Hard to change the order of the items.

- Undefined behavior if the collection is modified externally.

- Example:

```
public class Alphabetizer implements Iterator<String> {
    private String[] collection;
    private int cursor;

    Alphabetizer(String[] collection) {
        this.collection = collection;
        this.cursor = 0;
    }

    @Override
    public boolean hasNext() {
      // Use the cursor and the collection length
      // to figure out if there are still items left to visit
        return cursor < collection.length;
    }

    @Override
    public String next() {
        // Retrieve the item that the cursor points to,
        // increment the pointer, and return the item
        if (hasNext()) {
            String item = collection[cursor];
            cursor++;
            return item;
        } else {
            throw new NoSuchElementException();
        }
    }
}
```

- **Iterator Design Strategy #2:**
  - Encapsulates a *clone* of the raw collection.

- Sort or manipulate the cloned collection to make iteration easier.
- Add a “cursor” field to track progress through the cloned collection.
- Each time `next()` is called, update the cursor field.
- **Advantages**:
  - Changing the order of items in the cloned collection does not affect the original (external) collection.
  - Changing the order (or number) of items in the original collection does not affect the iterator.
  - Convenient for iterators that sort the collection
- **Disadvantages**:
  - Memory-inefficient (requires a full copy of the collection)
  - Cannot work for infinite collections

```
public class Alphabetizer implements Iterator<String> {
    private String[] collection;
    private int cursor;

    Alphabetizer(String[] collection) {
      // Make a private, sorted copy of the collection
        this.collection = collection.clone();
        Arrays.sort(this.collection);
        this.cursor = 0;
    }

    @Override
    public boolean hasNext() {
      // Use the cursor and the collection length
      // to figure out if there are still items left to visit
        return cursor < collection.length;
    }

    @Override
```

```
    public String next() {
        // Retrieve the item that the cursor points to,
        // increment the pointer, and return the item
        if (hasNext()) {
            String item = collection[cursor];
            cursor++;
            return item;
        } else {
            throw new NoSuchElementException();
        }
    }
}
```

- **Iterator Design Strategy #3:**
  - Encapsulate another iterator for the raw collection.
  - Each time `next()` or `hasNext()` is called, use the other iterator's `next()` and `hasNext()` methods.
  - **Advantages**:
    - Relies on the other iterator object to do the "hard work".
  - **Disadvantages**:
    - Can be tricky to implement.
    - Requires an iterator to already exist for the collection.

```
public class Alphabetizer implements Iterator<String> {
    private Iterator<String> iterator;

    Alphabetizer(String[] collection) {
        // Sort the collection and wrap it in an iterator
        String[] sortedCollection = collection.clone();
        sortedCollection = Arrays.sort(sortedCollection);
        this.iterator = Arrays.asList(sortedCollection).iterato
    }
```

```
    @Override
    public boolean hasNext() {
        // Delegate to the encapsulated iterator's hasNext meth
        return iterator.hasNext();
    }

    @Override
    public String next() {
        // Delegate to the encapsulated iterator's next method
        if (hasNext()) {
            return iterator.next();
        } else {
            throw new NoSuchElementException();
        }
    }
}

// => This approach leverages the raw Iterator<String> to
// handle the iteration logic, which simplifies the code by
// avoiding manual index management (cursor).
```

## Iterable Interface

- **Getting an `Iterator<>` Object From Java's Built-In Collections:**
    - Use `<collection_name>.iterator()` to get the iterator object.
    - In Java, most built-in collections (such as List, Set, and Map) are "iterable," meaning they implement the `Iterable<T>` interface.
    - This interface allows these collections to be used with iterators, which makes it possible to loop through elements one by one.
    - Example:

```
List<String> myList = new ArrayList<>();
Iterator<String> myListIterator = myList.iterator();
```

- **What is an `Iterable<T>` Object?**
    - An `Iterable<T>` object is a collection that can be iterated over using an **iterator**.
    - The `Iterable<T>` interface requires the implementing class to provide an `iterator()` method, which returns an `Iterator<T>` object.
- **Java's Iterable Interface:**

```
public interface Iterable<T> {
  // Creates and returns a new iterator for the collection
    Iterator<T> iterator(); // Required method to implement

    default void forEach(Consumer<? super T> action) {
        // Default (optional) method to perform an action on
    }

    default Spliterator<T> spliterator() {
        // Default (optional) method to split the collection
    }
}
```

    - The iterator() method is the key method here. It creates and returns a new Iterator<T> object for the collection.
- **Basic Example of Using an `Iterator<>`:**

```
List<Integer> ages = new ArrayList<>();
ages.add(20);
ages.add(19);
ages.add(21);

// Create an iterator for the list
Iterator<Integer> iterator = ages.iterator();

// Loop through the elements using the iterator*
while (iterator.hasNext()) {
```

```
    Integer age = iterator.next();
    System.out.println(age);
}
```

## For-Each Loops

- **For-Each Loop (Enhanced For Loop)**
    - The **for-each loop** is syntactic sugar for iterators. It simplifies the process of looping through each element in an iterable collection. This is how a for-each loop can be used:

    ```
    List<Integer> ages = new ArrayList<>();
    ages.add(20);
    ages.add(19);
    ages.add(21);

    // Use a for-each loop to iterate through the collection*
    for (Integer age : ages) {
        System.out.println(age);
    }
    ```

    - The **for-each loop** automatically creates an iterator behind the scenes and uses it to traverse the collection.
    - **Advantage**:
        - It's easy and concise.
        - Gives you direct access to each element in the collection.
        - It simplifies iteration and eliminates the need for manual iterator management.
    - **Disadvantage**:
        - You cannot directly access the index of each element.
        - *Read-Only:* Avoid modifying a collection inside this loop.
- **Example Comparison: While Loop vs For-Each Loop**

- **Using Iterator and While Loop:**

```
List<Integer> ages = new ArrayList<>();
ages.add(20);
ages.add(19);
ages.add(21);

Iterator<Integer> iterator = ages.iterator();

while (iterator.hasNext()) {
    Integer age = iterator.next();
    System.out.println(age);
}
```

- **Using For-Each Loop:**

```
List<Integer> ages = new ArrayList<>();
ages.add(20);
ages.add(19);
ages.add(21);

for (Integer age : ages) {
    System.out.println(age);
}
```

⇒ Both loops achieve the same result, but the for-each loop is more concise.

## Summary

- **Iterable**: A collection that provides an `iterator()` method to retrieve an iterator.
- **Iterator**: An object that provides methods (`hasNext()` and `next()`) to traverse a collection.
- **For-Each Loop**: A shorthand for iterating over `Iterable<T>` objects.