





**🧩 What is the Iterator Pattern?**

The **Iterator Pattern** provides a way to access the elements of a collection **sequentially** without exposing its internal structure.

It encapsulates the traversal logic and separates it from the collection itself.

**✅ Use Cases of Iterator Pattern**

| **Use Case** | **Example** |
| --- | --- |
| ✅ Custom collection traversal | You have a custom data structure and want to traverse it uniformly |
| ✅ Multiple traversal strategies | Forward, reverse, filtering, skipping etc. |
| ✅ Unified access to different containers | You want to treat arrays, lists, sets, stacks in the same way |
| ✅ External iteration control | Client decides how/when to traverse |
| ✅ Hiding complex internal structures | Complex tree, graph, or composite pattern |

**🧰 Structure of Iterator Pattern**

**📌 Participants**

| **Component** | **Description** |
| --- | --- |
| **Iterator** | Interface for traversing elements (hasNext(), next()) |
| **ConcreteIterator** | Actual iterator implementation |
| **Aggregate** | Interface or class representing a collection |
| **ConcreteAggregate** | Concrete collection that provides an iterator |
| **✅ Advantages**   | **Advantage** | **Description** | | --- | --- | | ✅ **Encapsulation** | Hides the internal structure of the collection | | ✅ **Flexible traversal** | Can implement multiple iterator types (e.g., reverse, filtered) | | ✅ **Uniform interface** | Same interface to iterate any collection | | ✅ **Separation of concerns** | Keeps iteration logic out of the collection class | | ✅ **Supports multiple iterations** | Can create multiple independent iterators |   **❌ Disadvantages**   | **Disadvantage** | **Description** | | --- | --- | | ❌ **Overhead for simple collections** | Java's built-in for-each or stream may be easier | | ❌ **Doesn’t support parallelism** | Simple iterators aren’t concurrent or thread-safe | | ❌ **Extra classes** | Can lead to many small classes in custom implementations | | ❌ **Read-only by default** | Modification during iteration often needs special handling |   **🆚 Built-in Java Iterators**  Java already supports this pattern natively via:  Iterator<String> it = list.iterator();  while (it.hasNext()) {  System.out.println(it.next());  }  Or more commonly:  java  CopyEdit  for (String name : list) {  System.out.println(name);  }  **✅ Summary Table**   | **Aspect** | **Iterator Pattern** | | --- | --- | | Type | Behavioral | | Core Idea | Unified traversal of a collection | | Key Methods | hasNext(), next() | | Java Built-in | java.util.Iterator, Iterable | | Real-world Uses | Collections, Tree structures, Custom data stores |   **✅ Real Java Example: Custom Collection**  **1. Iterator<T> Interface**  public interface Iterator<T> {  boolean hasNext();  T next();  }  **2. Concrete Iterator**  public class NameIterator implements Iterator<String> {  private String[] names;  private int index = 0;  public NameIterator(String[] names) {  this.names = names;  }  @Override  public boolean hasNext() {  return index < names.length;  }  @Override  public String next() {  return names[index++];  }  }  **3. Aggregate Interface (optional)**  java  CopyEdit  public interface IterableCollection {  Iterator<String> createIterator();  }  **4. Concrete Collection**  java  CopyEdit  public class NameRepository implements IterableCollection {  private String[] names = {"Alice", "Bob", "Charlie"};  @Override  public Iterator<String> createIterator() {  return new NameIterator(names);  }  }  **5. Client Code**  java  CopyEdit  public class IteratorPatternDemo {  public static void main(String[] args) {  NameRepository repo = new NameRepository();  Iterator<String> iterator = repo.createIterator();  while (iterator.hasNext()) {  System.out.println(iterator.next());  }  }  } |  |