Iterator design pattern

**1. Conceptual Understanding**

**What is the Iterator design pattern, and when is it used?**  
The **Iterator** pattern provides a way to sequentially access the elements of a collection without exposing its underlying structure. It is used when you want to traverse a collection in a uniform manner, regardless of its type.

**2. Implementation Questions**

**How would you implement the Iterator pattern in Python?**  
In Python, the Iterator pattern can be implemented using the \_\_iter\_\_ and \_\_next\_\_ methods:

python

Copy code

class CustomCollection:

def \_\_init\_\_(self, items):

self.items = items

def \_\_iter\_\_(self):

return CustomIterator(self.items)

class CustomIterator:

def \_\_init\_\_(self, items):

self.items = items

self.index = 0

def \_\_next\_\_(self):

if self.index < len(self.items):

value = self.items[self.index]

self.index += 1

return value

raise StopIteration

# Usage

collection = CustomCollection([1, 2, 3])

for item in collection:

print(item)

**What are the key components of the Iterator pattern?**

1. **Iterator Interface**: Defines methods for traversing elements (next, hasNext).
2. **Concrete Iterator**: Implements the traversal logic for a specific collection.
3. **Aggregate (Collection)**: Represents the collection and provides an interface to create iterators.
4. **Client**: Uses the iterator to access elements of the collection.

**How would you implement a reverse iterator?**  
Modify the iterator to traverse elements in reverse:

python

Copy code

class ReverseIterator:

def \_\_init\_\_(self, items):

self.items = items

self.index = len(items) - 1

def \_\_next\_\_(self):

if self.index >= 0:

value = self.items[self.index]

self.index -= 1

return value

raise StopIteration

**3. Real-World Scenarios**

**What are common use cases for the Iterator pattern?**

1. **Collections Framework**: Iterating over lists, sets, or maps in Java, Python, etc.
2. **File Systems**: Traversing files and directories.
3. **Database Result Sets**: Iterating over rows in a database query result.
4. **Custom Aggregates**: Iterating over non-standard data structures, like a tree or graph.

**How would you implement an iterator for a tree structure?**  
You can use **DFS** or **BFS** traversal:

python

Copy code

class TreeNode:

def \_\_init\_\_(self, value, children=[]):

self.value = value

self.children = children

class TreeIterator:

def \_\_init\_\_(self, root):

self.stack = [root]

def \_\_next\_\_(self):

if not self.stack:

raise StopIteration

node = self.stack.pop()

self.stack.extend(node.children[::-1]) # Add children in reverse for correct order

return node.value

**4. Behavioral Questions**

**What are the advantages and disadvantages of the Iterator pattern?**

* **Advantages**:
  1. Uniform interface for traversing different types of collections.
  2. Decouples traversal logic from collection implementation.
  3. Supports multiple traversal strategies (e.g., forward, reverse, filtered).
* **Disadvantages**:
  1. May introduce overhead for simple collections.
  2. Iterators can become complex for advanced data structures (e.g., graphs).

**How does the Iterator pattern follow the Single Responsibility Principle?**  
The Iterator pattern separates the responsibility of iterating over a collection from the collection itself, making both easier to maintain.

**How do you handle concurrent modification of a collection during iteration?**

1. **Fail-Fast Iterators**: Throw an exception if the collection is modified (e.g., Java's ConcurrentModificationException).
2. **Snapshot Iterators**: Work on a copy of the collection to avoid concurrency issues.
3. **Synchronization**: Use locks or thread-safe collections.

**5. Code Debugging Questions**

**Given a sample Iterator implementation, identify issues or suggest improvements.**

* **Problem**: StopIteration is not raised at the end of iteration.
  + **Solution**: Ensure \_\_next\_\_ raises StopIteration when traversal is complete.
* **Problem**: Iterator modifies the underlying collection during traversal.
  + **Solution**: Use a copy of the collection or ensure immutability during iteration.

**How would you test an Iterator implementation?**

1. Test iteration over an empty collection.
2. Test iteration over a single-element and multi-element collection.
3. Test edge cases like removing elements during iteration (if supported).
4. Verify behavior with custom traversal logic (e.g., reverse iteration).

**6. Advanced Topics**

**How would you implement a filter iterator?**  
A filter iterator applies a predicate to elements during iteration:

python

Copy code

class FilterIterator:

def \_\_init\_\_(self, items, predicate):

self.items = iter(items)

self.predicate = predicate

def \_\_next\_\_(self):

while True:

item = next(self.items)

if self.predicate(item):

return item

Usage:

python

Copy code

filtered = FilterIterator([1, 2, 3, 4], lambda x: x % 2 == 0)

for item in filtered:

print(item) # Outputs: 2, 4

**Can the Iterator pattern be used in a multithreaded environment? How?**  
Yes, but it requires thread safety:

1. Use locks to ensure mutual exclusion while iterating.
2. Use concurrent collections (e.g., ConcurrentLinkedQueue in Java).
3. Snapshot iterators can avoid modification issues by iterating over a copy.

**7. Comparison with Other Patterns**

**How does the Iterator pattern differ from the Composite pattern?**

* **Iterator**: Provides sequential access to elements in a collection.
* **Composite**: Composes objects into tree-like structures, allowing individual and group operations.

**How does the Iterator pattern compare to the Visitor pattern?**

* **Iterator**: Focuses on sequential traversal of a collection.
* **Visitor**: Focuses on performing operations on elements of a collection, often using double dispatch.