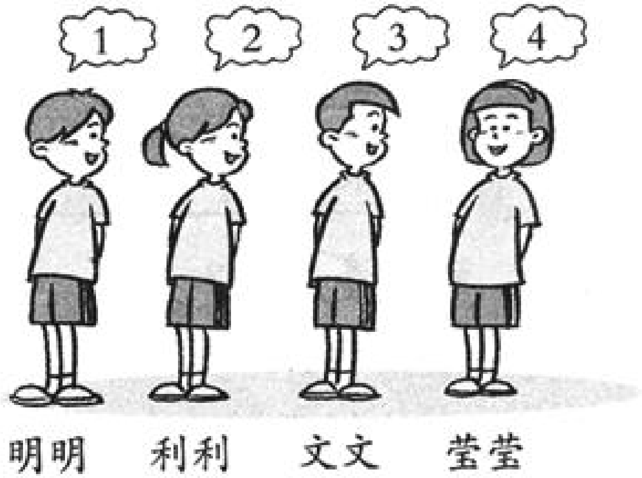
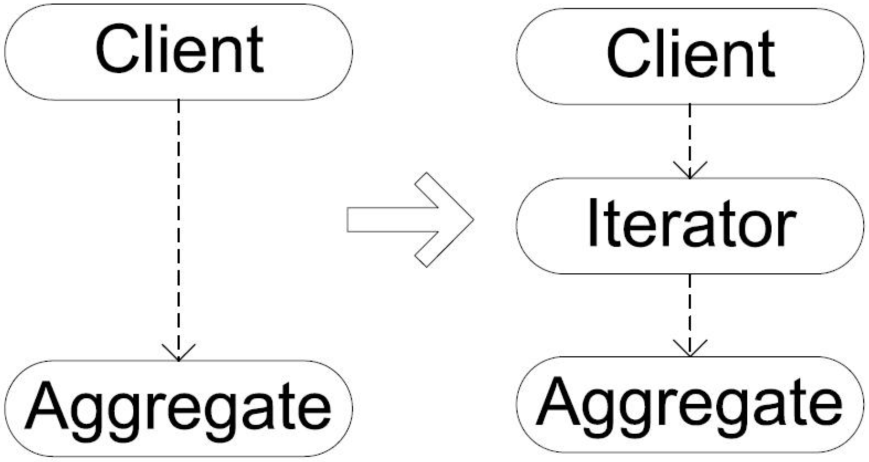
Iterator (迭代器, Behavioral Pattern)

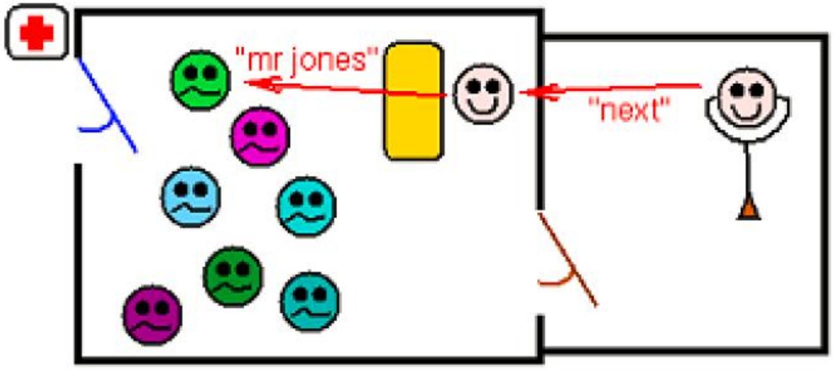


Kai SHI

Traversal Issue (OCP)

- Traversal mechanism is unchanged, but the traversed aggregate is changed. The code in client side should be modified because different aggregates have different traversal interface.
- Aggregate is unchanged, but traversal mechanism is changed. For example, add filtering algorithm. The interface of aggregate should be modified to introduced the new traversal approaches.





Iterator Pattern

- Intent
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Also Known As
  - Cursor

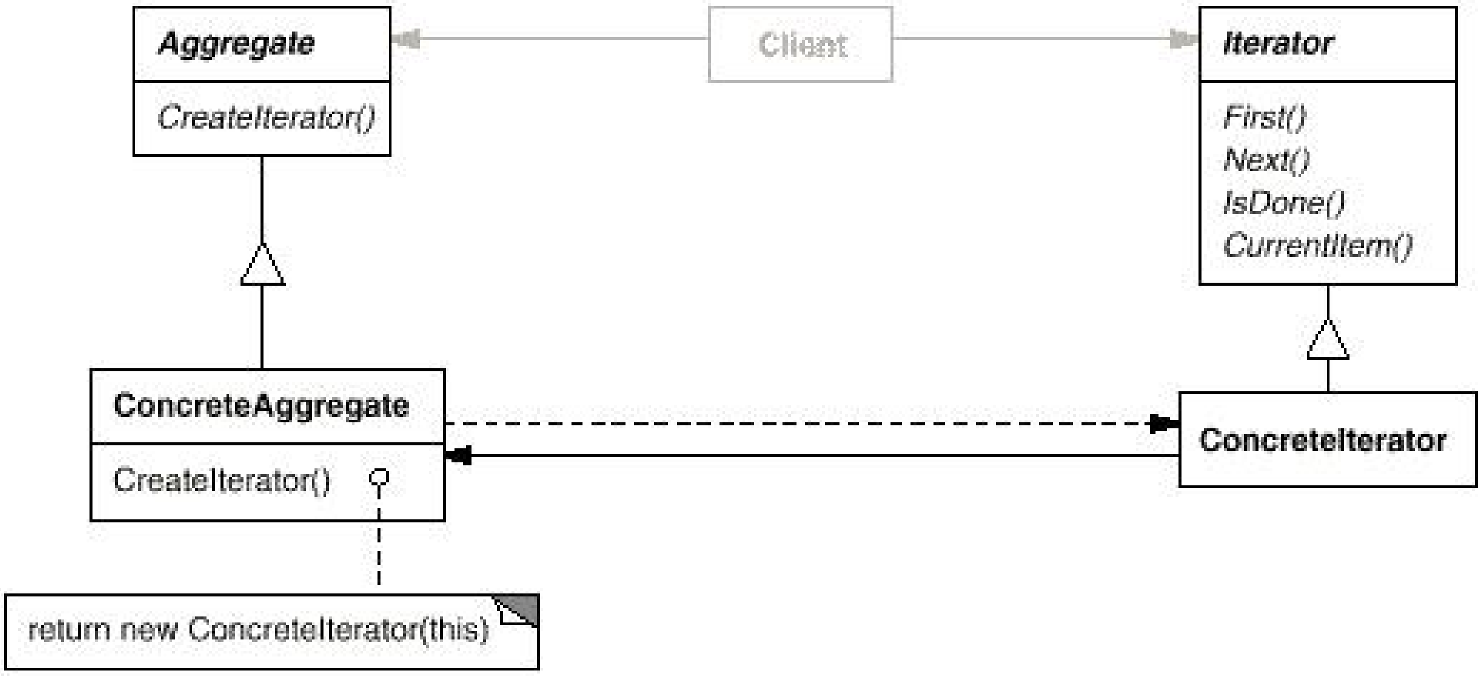
Motivation

- An aggregate object such as a list should give you a way to access its elements without exposing its internal structure.
- The key idea in iterator pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object.
- Separating the traversal mechanism from the List object lets us define iterators for different traversal policies without enumerating them in the List interface.
  - FilteringListIterator might provide access only to those elements that satisfy specific filtering constraints.

Applicability:
Use the Iterator pattern

- To access an aggregate object's contents without exposing its internal representation.
- To support multiple traversals of aggregate objects.
- To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Structure


Aggregate
CreateIterator()
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

Participants

- Iterator
  - defines an interface for accessing and traversing elements.
- ConcreteIterator
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- Aggregate
  - defines an interface for creating an Iterator object.
  - CreateIterator() is an example of a factory method.
- ConcreteAggregate
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Collaborations

- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences

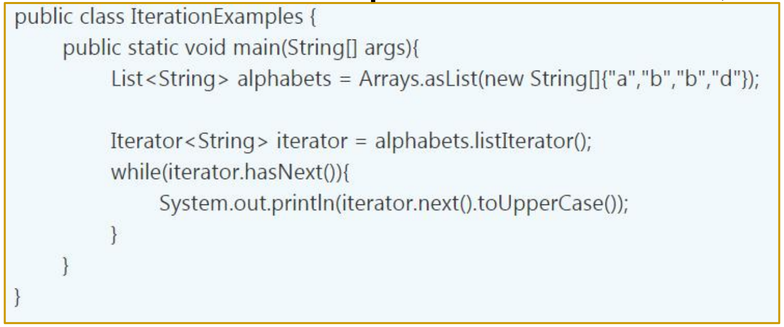
- It supports variations in the traversal of an aggregate.
  - Complex aggregates may be traversed in many ways.
- Iterators simplify the Aggregate interface.
- More than one traversal can be pending on an aggregate.
  - An iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.

Implementation 1: **Where** the **concrete iterator** is defined?

- Public Iterator: Concrete iterator is defined as a class independent from aggregate.
  - More straightforward;
  - Polymorphic iteration;
  - Can storing multiple cursor for different clients.
  - Need the aggregate expose the details, thus break the encapsulation.
- Private Iterator: Concrete iterator is defined as a inner class in the aggregate.
  - Less straightforward;
  - Protect the encapsulation of aggregate;
  - Suggested in most cases. See sourse of java.util.ArrayList

Implementation 2: Who controls the iteration?

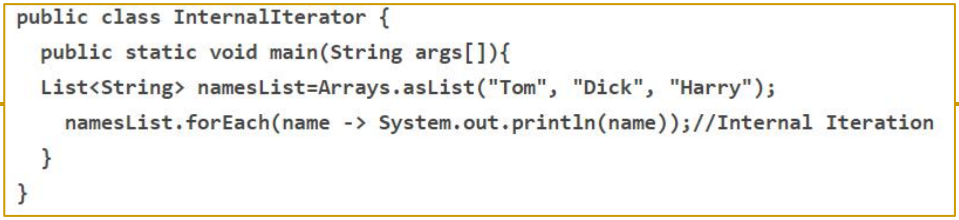
- **Active** Iterator (**External** Iterator): The **client controls the iteration**;
  - Clients that use an active iterator must advance the traversal and request the next element explicitly from the iterator.
  - more flexible than passive iterators;



```
public class IterationExamples {
        public static void main(String[] args){
                List<String> alphabets = Arrays.asList(new String[]{"a","b","b","d"});

                Iterator<String> iterator = alphabets.listIterator();
                while(iterator.hasNext()){
                        System.out.println(iterator.next().toUpperCase());
                }
        }
}
```

- **Passive** Iterator (**Internal** Iterator): The iterator controls the iteration;
  - The client hands an passive iterator an operation to perform, and the iterator applies that operation to every element in the aggregate.
  - Easier to use, because it define the iteration logic for you.



```
public class InternalIterator {
  public static void main(String args[]){
  List<String> namesList=Arrays.asList("Tom", "Dick", "Harry");
    namesList.forEach(name -> System.out.println(name));//Internal Iteration
  }
}
```

Java 8 way to create a stream and then iterate internally
"->": Lambda Expressions

Implementation 3: Who defines the traversal algorithm?

- The aggregate might define the traversal algorithm and use the iterator to store just the state of the iteration (cursor), it points to the current position in the aggregate.
- The iterator is responsible for the traversal algorithm, then it's easy to use different iteration algorithms on the same aggregate, and it can also be easier to reuse the same algorithm on different aggregates.
  - Defining the iterator in aggregate’s the inner class if traversal algorithm might need to access the private variables of the aggregate.

Implementation 4 (1/3): How robust is the iterator?

- It can be dangerous to modify an aggregate while you're traversing it.
  - Copied Iterator:
    - A simple solution is to copy the aggregate and traverse the copy, but that's too expensive to do in general.
  - Robust iterator:
    - Ensures that insertions and removals won't affect traversal, and it does it without copying the aggregate. On insertion or removal, the aggregate either adjusts the internal state of iterators it has produced, or it maintains information internally to ensure proper traversal.

Implementation 4 (2/3): Static Iterator and Dynamic Iterator

- Static Iterator: A copied iterator which contains a snapshot of the aggregate when iterator is created. New changes are invisible to the traversal approach.
- Dynamic Iterator: Dynamic Iterator is opposited to the static one. Any changes to the aggregate are allowed and available when traversing the aggregate.
  - Completely Dynamic Iterator is not easy to be implemented.

Implementation 4 (3/3): Fail-Fast (快速失败) in Java

- Fail-fast is a property of a system or module with respect to its response to failures.
- A fail-fast system is designed to immediately report at its interface any failure or condition that is likely to lead to failure.
- Fail-fast systems are usually designed to stop normal operation rather than attempt to continue a possibly-flawed process.
- Fail-fast Iterator throws an exception when the aggregate is changed during iteration.

- Code: iterator.FailFastTest

Implementation 5: Additional Iterator operations.

- The minimal interface to Iterator consists of the operations First, Next, IsDone, and CurrentItem.
- Some additional operations might prove useful.
  - Last, Previous, SkipTo
- Filter Iterator
  - A common iterator traverse each element of an aggregate.
  - A filter iterator compute the element of the aggregate and return the elements which match a certain condition.

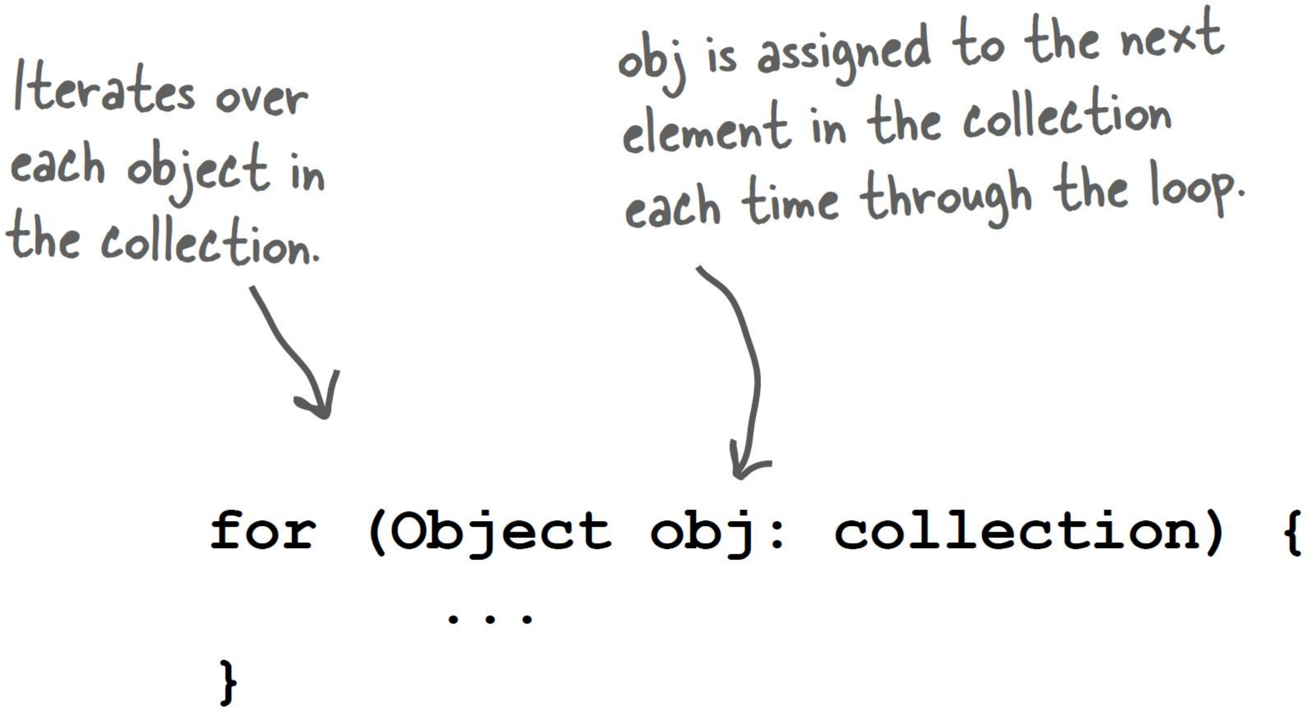
Implementation 6: Iterators for Composites

- **External** iterators can be **difficult** to implement over **recursive** aggregate structures (e.g, Composite pattern), because a position in the structure may span many levels of nested aggregates.
  - An active (external) iterator has to store a path through the Composite to keep track of the current object.
  - It's easier just to use an passive (internal) iterator. It can record the current position simply by calling itself recursively, thereby storing the path implicitly in the call stack.
- Composites often need to be traversed in more than one way.
  - Pre-order, post-order, in-order, and breadth-first traversals are common.

Implementation 7: Null iterators

- A NullIterator is a degenerate (退化) iterator that's helpful for handling boundary conditions.
- By definition, a NullIterator is always done with traversal; that is, its IsDone operation always evaluates to true.
- Null Iterator can make traversing tree-structured aggregates (like Composites) easier.
    - At each point in the traversal, we ask the current element for an iterator for its children. Aggregate elements return a concrete iterator as usual. But **leaf** elements **return** an instance of **NullIterator**.

Iterators and Collections in Java 5



Iterates over each object in the collection.

obj is assigned to the next element in the collection each time through the loop.

```
for (Object obj: collection) {
         ...
}
```