Trending Now DSA Data Structures Algorithms Interview Preparation Data Science T

Read Discuss

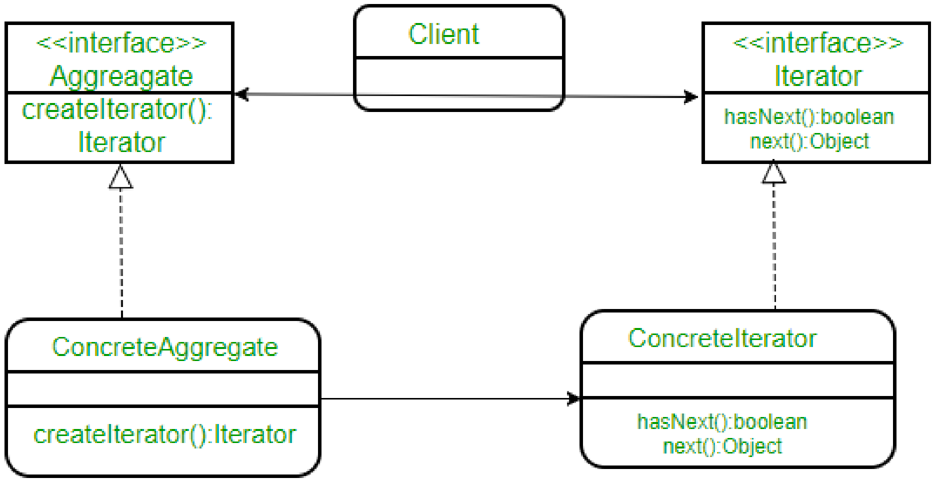
Iterator Pattern is a relatively simple and frequently used design pattern. There are a lot of data structures/collections available in every language. Each collection must provide an iterator that lets it iterate through its objects. However while doing so it should make sure that it does not expose its implementation. Suppose we are building an application that requires us to maintain a list of notifications. Eventually, some part of your code will require to iterate over all notifications. If we implemented your collection of notifications as array you would iterate over them as:

```
// If a simple array is used to store notifications
for (int i = 0; i < notificationList.length; i++)
     Notification notification = notificationList[i]);


// If ArrayList is Java is used, then we would iterate
// over them as:
for (int i = 0; i < notificationList.size(); i++)
    Notification notification =
(Notification)notificationList.get(i);
```

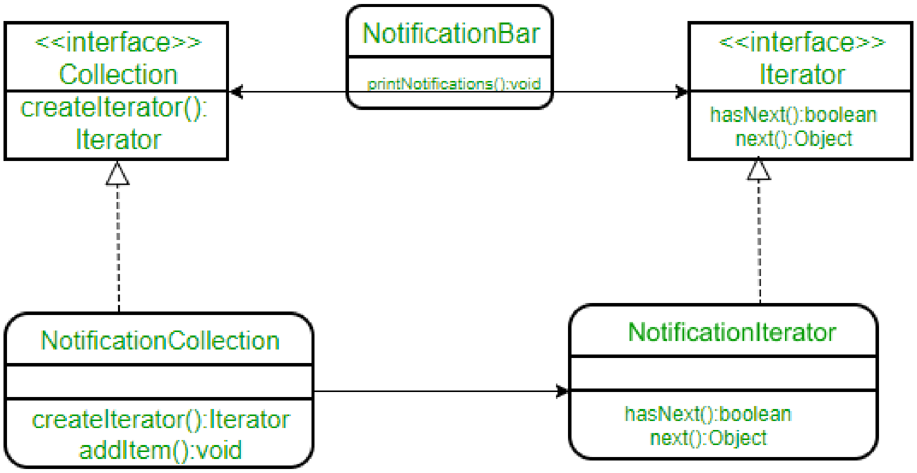
And if it were some other collection like set, tree etc. way of iterating would change slightly. Now, what if we build an iterator that provides a generic way

```
// Create an iterator
Iterator iterator = notificationList.createIterator();

// It wouldn’t matter if list is Array or ArrayList or
// anything else.
while (iterator.hasNext())
{
    Notification notification = iterator.next());
}
```

Iterator pattern lets us do just that. Formally it is defined as below: ***The iterator pattern provides a way to access the elements of an aggregate object without exposing its underlying representation. Class Diagram:***



Here we have a common interface Aggregate for client as it decouples it from the implementation of your collection of objects. The ConcreteAggregate implements createIterator() that returns iterator for its collection. Each ConcreteAggregate’s responsibility is to instantiate a ConcreteIterator that can iterate over its collection of objects. The iterator interface provides a set of bar in our application that displays all the notifications which are held in a notification collection. NotificationCollection provides an iterator to iterate over its elements without exposing how it has implemented the collection (array in this case) to the Client (NotificationBar). The class diagram would be:



Below is the Java implementation of the same:

Java

```
// A Java program to demonstrate implementation
// of iterator pattern with the example of
// notifications

// A simple Notification class
class Notification
{
    // To store notification message
    String notification;

    public Notification(String notification)
    {
        this.notification = notification;
    }
    public String getNotification()
```

```
// Collection interface
interface Collection
{
    public Iterator createIterator();
}

// Collection of notifications
class NotificationCollection implements Collection
{
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    Notification[] notificationList;

    public NotificationCollection()
    {
        notificationList = new Notification[MAX_ITEMS];

        // Let us add some dummy notifications
        addItem("Notification 1");
        addItem("Notification 2");
        addItem("Notification 3");
    }

    public void addItem(String str)
    {
        Notification notification = new Notification(str);
        if (numberOfItems >= MAX_ITEMS)
            System.err.println("Full");
        else
        {
            notificationList[numberOfItems] = notification;
            numberOfItems = numberOfItems + 1;
        }
    }

    public Iterator createIterator()
    {
        return new NotificationIterator(notificationList);
    }
}

// We could also use Java.Util.Iterator
interface Iterator
```

```
    // returns the next element
    Object next();
}

// Notification iterator
class NotificationIterator implements Iterator
{
    Notification[] notificationList;

    // maintains curr pos of iterator over the array
    int pos = 0;

    // Constructor takes the array of notificationList are
    // going to iterate over.
    public  NotificationIterator (Notification[] notificationList)
    {
        this.notificationList = notificationList;
    }

    public Object next()
    {
        // return next element in the array and increment pos
        Notification notification =  notificationList[pos];
        pos += 1;
        return notification;
    }

    public boolean hasNext()
    {
        if (pos >= notificationList.length ||
            notificationList[pos] == null)
            return false;
        else
            return true;
    }
}

// Contains collection of notifications as an object of
// NotificationCollection
class NotificationBar
{
    NotificationCollection notifications;
    }

    public void printNotifications()
    {
        Iterator iterator = notifications.createIterator();
        System.out.println("-------NOTIFICATION BAR------------");
        while (iterator.hasNext())
        {
            Notification n = (Notification)iterator.next();
            System.out.println(n.getNotification());
        }
    }
}

// Driver class
class Main
{
    public static void main(String args[])
    {
        NotificationCollection nc = new NotificationCollection();
        NotificationBar nb = new NotificationBar(nc);
        nb.printNotifications();
    }
}
```

Output:

```
-------NOTIFICATION BAR------------
Notification 1
Notification 2
Notification 3
```

Notice that if we would have used ArrayList instead of Array there will not be any change in the client (notification bar) code due to the decoupling achieved by the use of iterator interface. **Further Read** – Iterator Method in Python

References: Head First Design Patterns

This article is contributed by **Sulabh Kumar.**If you like GeeksforGeeks and

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Last Updated : 08 Apr, 2022

12

Similar Reads

1. Design Patterns in Java - Iterator Pattern
2. Implementing Iterator pattern of a single Linked List
3. Command Pattern
4. Observer Pattern | Set 1 (Introduction)
5. Observer Pattern | Set 2 (Implementation)
6. Singleton Design Pattern | Implementation
7. Decorator Pattern | Set 1 (Background)
8. The Decorator Pattern | Set 2 (Introduction and Design)
9. Decorator Pattern | Set 3 (Coding the Design)
10. Strategy Pattern | Set 2 (Implementation)

Previous Next

Vote for difficulty

Current difficulty : Medium

Easy Normal Medium Hard Expert

Improved By : Shraddhesh Bhandari, creatsiv, simmytarika5

Article Tags : Design Pattern

Improve Article Report Issue

GeeksforGeeks

A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org

Company

About Us

Careers

In Media

Contact Us

Terms and Conditions

Privacy Policy

Copyright Policy

Explore

Job Fair For Students

POTD: Revamped

Python Backend LIVE

Android App Development

DevOps LIVE

DSA in JavaScript

Languages

Python

Java

C++

GoLang

SQL

R Language

Android Tutorial

Data Structures

Array

String

Linked List

Stack

Queue

Tree

Graph

Algorithms

Sorting

Searching

Greedy

Dynamic Programming

Pattern Searching

Recursion

Backtracking

Web Development

HTML

CSS

JavaScript

Bootstrap

ReactJS

AngularJS

NodeJS

Computer Science

GATE CS Notes

Operating Systems

Computer Network

Database Management System

Software Engineering

Digital Logic Design

Engineering Maths

Python

Python Programming Examples

Django Tutorial

Python Projects

Python Tkinter

OpenCV Python Tutorial

Python Interview Question

Data Science & ML

DevOps

Maths For Machine Learning

Pandas Tutorial

NumPy Tutorial

NLP Tutorial

Deep Learning Tutorial

Kubernetes

Azure

GCP

Competitive Programming

Top DSA for CP

Top 50 Tree Problems

Top 50 Graph Problems

Top 50 Array Problems

Top 50 String Problems

Top 50 DP Problems

Top 15 Websites for CP

System Design

What is System Design

Monolithic and Distributed SD

Scalability in SD

Databases in SD

High Level Design or HLD

Low Level Design or LLD

Top SD Interview Questions

Interview Corner

Company Preparation

Preparation for SDE

Company Interview Corner

Experienced Interview

Internship Interview

Competitive Programming

Aptitude

GfG School

CBSE Notes for Class 8

CBSE Notes for Class 9

CBSE Notes for Class 10

CBSE Notes for Class 11

CBSE Notes for Class 12

English Grammar

Commerce

Accountancy

Business Studies

Microeconomics

UPSC

Polity Notes

Geography Notes

History Notes

UPSC Previous Year Papers

SSC/ BANKING

SSC CGL Syllabus

SBI PO Syllabus

SBI Clerk Syllabus

IBPS PO Syllabus

IBPS Clerk Syllabus

Aptitude Questions

SSC CGL Practice Papers

Write & Earn

Write an Article

Improve an Article

Pick Topics to Write

Write Interview Experience

Internships

Video Internship