<https://www.geeksforgeeks.org/iterator-pattern/>

<https://www.journaldev.com/1716/iterator-design-pattern-java>

Iterator Pattern

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);

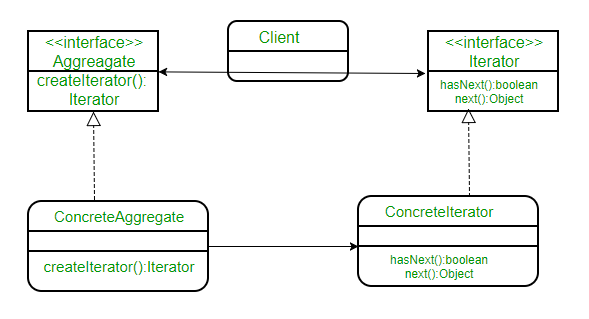
And if it were some other collection like set, tree etc. way of iterating would change slightly. Now, what if we build an iterator that provides a generic way of iterating over a collection independent of its type.

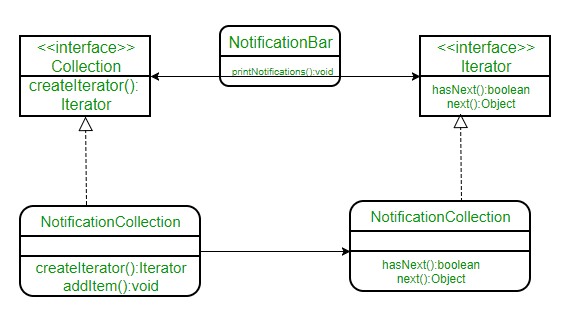
// Create an iterator

Iterator iterator = notificationList.createIterator(); // It wouldn’t matter if list is Array or ArrayList or // anything else. while (iterator.hasNext())

{ Notification notification = iterator.next()); }

***The iterator pattern provides a way to access the elements of an aggregate object without exposing its underlying representation.***





// 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()

{

return notification;

}

}

// 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

{

// indicates whether there are more elements to

// iterate over

boolean hasNext();

// 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 notifiactionList 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 NotificationBar(NotificationCollection notifications)

{

this.notifications = 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();

}

}

**package** com.example.designpattern.behavioural.iterator;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** IteratorPatternTest2 {

**public** **static** **void** main(String[] args) {

ChannelCollection channels = *populateChannels*();

ChannelIterator baseIterator = channels.iterator(ChannelTypeEnum.***ALL***);

**while** (baseIterator.hasNext()) {

Channel c = baseIterator.next();

System.***out***.println(c.toString());

}

System.***out***.println("\*\*\*\*\*\*");

// Channel Type Iterator

ChannelIterator englishIterator = channels.iterator(ChannelTypeEnum.***ENGLISH***);

**while** (englishIterator.hasNext()) {

Channel c = englishIterator.next();

System.***out***.println(c.toString());

}

}

**private** **static** ChannelCollection populateChannels() {

ChannelCollection channels = **new** ChannelCollectionImpl();

channels.addChannel(**new** Channel(98.5, ChannelTypeEnum.***ENGLISH***));

channels.addChannel(**new** Channel(99.5, ChannelTypeEnum.***HINDI***));

channels.addChannel(**new** Channel(100.5, ChannelTypeEnum.***FRENCH***));

channels.addChannel(**new** Channel(101.5, ChannelTypeEnum.***ENGLISH***));

channels.addChannel(**new** Channel(102.5, ChannelTypeEnum.***HINDI***));

channels.addChannel(**new** Channel(103.5, ChannelTypeEnum.***FRENCH***));

channels.addChannel(**new** Channel(104.5, ChannelTypeEnum.***ENGLISH***));

channels.addChannel(**new** Channel(105.5, ChannelTypeEnum.***HINDI***));

channels.addChannel(**new** Channel(106.5, ChannelTypeEnum.***FRENCH***));

**return** channels;

}

}

**interface** ChannelCollection {

**public** **void** addChannel(Channel c);

**public** **void** removeChannel(Channel c);

**public** ChannelIterator iterator(ChannelTypeEnum type);

}

**interface** ChannelIterator {

**public** **boolean** hasNext();

**public** Channel next();

}

**class** ChannelCollectionImpl **implements** ChannelCollection {

**private** List<Channel> channelsList;

**public** ChannelCollectionImpl() {

channelsList = **new** ArrayList<>();

}

**public** **void** addChannel(Channel c) {

**this**.channelsList.add(c);

}

**public** **void** removeChannel(Channel c) {

**this**.channelsList.remove(c);

}

@Override

**public** ChannelIterator iterator(ChannelTypeEnum type) {

**return** **new** ChannelIteratorImpl(type, **this**.channelsList);

}

**private** **class** ChannelIteratorImpl **implements** ChannelIterator {

**private** ChannelTypeEnum type;

**private** List<Channel> channels;

**private** **int** position;

**public** ChannelIteratorImpl(ChannelTypeEnum ty, List<Channel> channelsList) {

**this**.type = ty;

**this**.channels = channelsList;

}

@Override

**public** **boolean** hasNext() {

**while** (position < channels.size()) {

Channel c = channels.get(position);

**if** (c.getTYPE().equals(type) || type.equals(ChannelTypeEnum.***ALL***)) {

**return** **true**;

} **else**

position++;

}

**return** **false**;

}

@Override

**public** Channel next() {

Channel c = channels.get(position);

position++;

**return** c;

}

}

}

**enum** ChannelTypeEnum {

***ENGLISH***, ***HINDI***, ***FRENCH***, ***ALL***;

}

**class** Channel {

**private** **double** frequency;

**private** ChannelTypeEnum TYPE;

**public** Channel(**double** freq, ChannelTypeEnum type) {

**this**.frequency = freq;

**this**.TYPE = type;

}

**public** **double** getFrequency() {

**return** frequency;

}

**public** ChannelTypeEnum getTYPE() {

**return** TYPE;

}

@Override

**public** String toString() {

**return** "Frequency=" + **this**.frequency + ", Type=" + **this**.TYPE;

}

}