**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

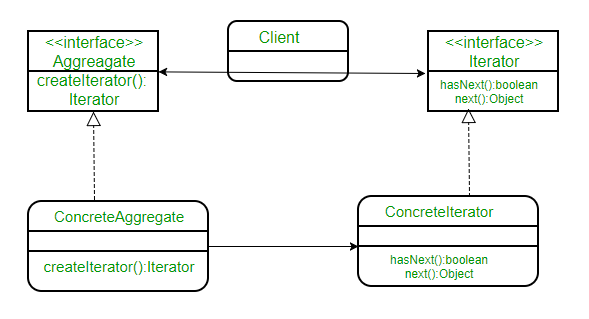
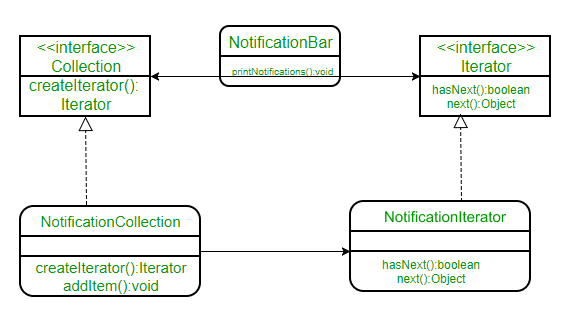
// anything else.

while (iterator.hasNext())

{

Notification notification = iterator.next());

}

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 methods for traversing or modifying the collection that is in addition to next()/hasNext() it can also provide functions for search, remove etc. Let’s understand this through an example. Suppose we are creating a notification 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()      {          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 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 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();      }  } |

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