Iterator Pattern

Iterator pattern formalizes how we move through a collection of data in a particular class.

Iterator in the Real World

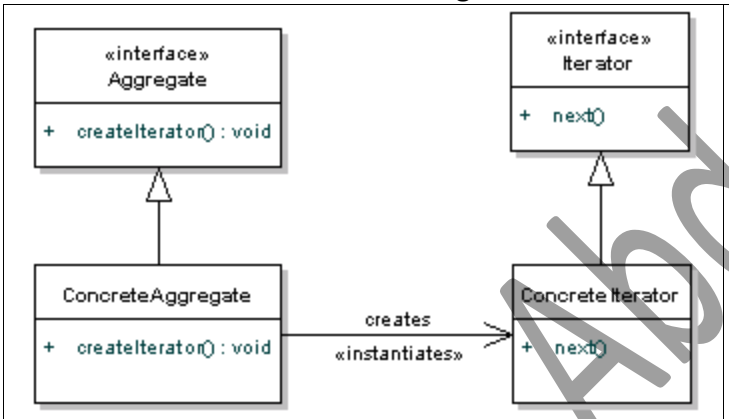
MP3 player control is a good example of an iterator. The user doesn't mind how to view their list of songs, once they get to see them somehow. In older mp3 players, this was done using simple forward and back buttons. With the iPod this changed to the wheel navigation concept. The iPhone moves this on further to use swipe movements. Nevertheless, the same idea is provided by all interfaces - a way to iterate through your music collection.

The Iterator Pattern

The Iterator pattern is known as a behavioral pattern, as it's used to manage algorithms, relationships and responsibilities between objects. The definition of Iterator as provided in the original Gang of Four book on Design Patterns states:

"Provides a way to access the elements of an aggregate object without exposing its underlying representation."

Let's take a look at the diagram definition before we go into more detail.



The Aggregate defines an interface for the creation of the Iterator object. The ConcreteAggregate implements this interface, and returns an instance of the ConcreteIterator. The Iterator defines the interface for access and traversal of the elements, and the ConcreteIterator implements this interface while keeping track of the current position in the traversal of the Aggregate.

Using this pattern, you can build on the standard concept of iteration to define special iterators that only return specific elements in the data set.

Would I Use This Pattern?

This pattern is useful when you need access to elements in a set without access to the entire representation. When you need a uniform traversal interface, and multiple traversals may happen across elements, iterator is a good choice. It also makes you code much more reasonable, getting rid of the typical for loop syntax across sections of your codebase.

Example 1: Let's look at an example of this. We have an Item class, which represents an item on a menu. An item has a name and a price.

Item.java

```
public class Item {
	String name;
	float price;
	public Item(String name, float price) {
		this.name = name;
```

```
                this.price = price;
        }
        public String toString() {
                return name + ": $" + price;
        }
}
```

Here is the Menu class. It has a list of menu items of type Item. Items can be added via the addItem() method. The iterator() method returns an iterator of menu items. The MenuIterator class is an inner class of Menu that implements the Iterator interface for Item objects. It contains basic implementations of the hasNext(), next(), and remove() methods.

Menu.java

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Menu {
        List<Item> menuItems;
        public Menu() {
                menuItems = new ArrayList<Item>();
        }
        public void addItem(Item item) {
                menuItems.add(item);
        }
        public Iterator<Item> iterator() {
                return new MenuIterator();
        }
        class MenuIterator implements Iterator<Item> {
                int currentIndex = 0;
                @Override
                public boolean hasNext() {
                        if (currentIndex >= menuItems.size()) {
                                return false;
                        } else {
                                return true;
                        }
                }
                @Override
                public Item next() {
                        return menuItems.get(currentIndex++);
                }
                @Override
                public void remove() {
                        menuItems.remove(--currentIndex);
                }

        }

}
```

The Demo class demonstrates the iterator pattern. It creates three items and adds them to the menu object. Next, it gets an Item iterator from the menu object and iterates over the items in the menu. After this, it calls remove() to remove the last item obtained by the iterator. Following this, it gets a new iterator object from the menu and once again iterates over the menu items.

Demo.java

```
import java.util.Iterator;
public class Demo {
	public static void main(String[] args) {
		Item i1 = new Item("spaghetti", 7.50f);
		Item i2 = new Item("hamburger", 6.00f);
		Item i3 = new Item("chicken sandwich", 6.50f);
		Menu menu = new Menu();
		menu.addItem(i1);
		menu.addItem(i2);
		menu.addItem(i3);
		System.out.println("Displaying Menu:");
		Iterator<Item> iterator = menu.iterator();
		while (iterator.hasNext()) {
			Item item = iterator.next();
			System.out.println(item);
		}
		System.out.println("\nRemoving last item returned");
		iterator.remove();
		System.out.println("\nDisplaying Menu:");
		iterator = menu.iterator();
		while (iterator.hasNext()) {
			Item item = iterator.next();
			System.out.println(item);
		}
	}
}
```

The console output is shown here.

Console Output

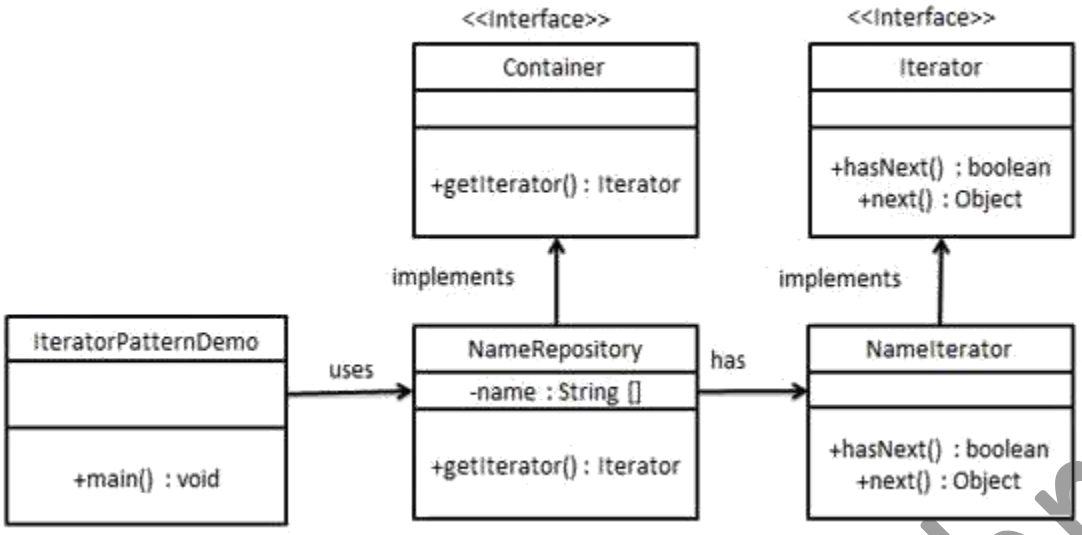
```
Displaying Menu:
spaghetti: $7.5
hamburger: $6.0
chicken sandwich: $6.5

Removing last item returned

Displaying Menu:
spaghetti: $7.5
hamburger: $6.0
```

Note that since the menu utilizes a Java collection, we could have used an iterator obtained for the menu list rather than write our own iterator as an inner class.

Example 2:

We're going to create a Iterator interface which narrates navigation method and a Container interface which retruns the iterator . Concrete classes implementing the Container interface will be responsible to implement Iterator interface and use it

IteratorPatternDemo, our demo class will use NamesRepository, a concrete class implementation to print a Names stored as a collection in NamesRepository.



Step 1

Create interfaces.

Iterator.java

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

Container.java

```
public interface Container {
   public Iterator getIterator();
}
```

Step 2

Create concrete class implementing the *Container* interface. This class has inner class *NameIterator* implementing the *Iterator* interface.

NameRepository.java

```
public class NameRepository implements Container {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};
   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }
   private class NameIterator implements Iterator {
      int index;
      @Override
      public boolean hasNext() {
         if(index < names.length){
            return true;
         }
         return false;
      }
      @Override
      public Object next() {
         if(this.hasNext()){
            return names[index++];
         }
         return null;
```

```
      }
   }
}
```

Step 3
Use the *NameRepository* to get iterator and print names.
IteratorPatternDemo.java

```
public class IteratorPatternDemo {
   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();
      for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
         String name = (String)iter.next();
         System.out.println("Name : " + name);
      }
   }
}
```

Step 4
Verify the output.

```
Name : Robert
Name : John
Name : Julie
Name : Lora
```