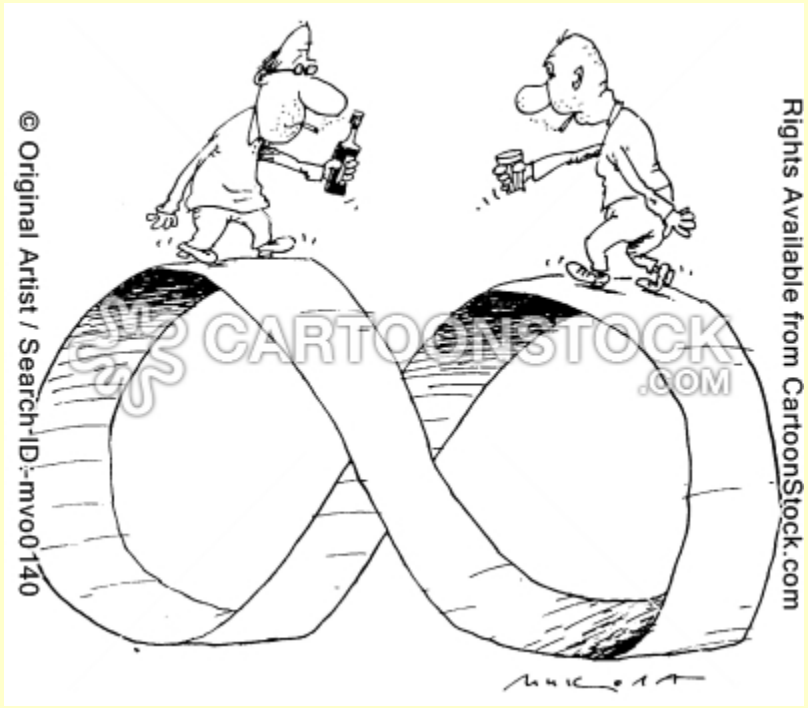
Patron: Iterator (Itérateur)



/~gibson/Teaching/DesignPatterns/CSC4522-DesignPatterns-Iterator.pdf

https://fr.wikipedia.org/wiki/Itérateur

Un itérateur est un objet qui permet de parcourir tous les éléments contenus dans un autre objet, le plus souvent un conteneur (liste, arbre, etc).

Un synonyme d'itérateur est curseur, notamment dans le contexte des bases de données.

Un itérateur ressemble à un pointeur disposant essentiellement de trois primitives :

- *accéder* à l'élément pointé en cours (dans le conteneur), et
- *se déplacer* pour pointer vers l'élément suivant, et
- déterminer si l'itérateur a *épuisé* la totalité des éléments du conteneur.

Le but d'un itérateur est de permettre à son utilisateur de *parcourir* le conteneur, tout en isolant l'utilisateur de la structure interne du conteneur, potentiellement complexe.

Le plus souvent l'itérateur est conçu en même temps que la classe-conteneur qu'il devra parcourir, et ce sera le conteneur lui-même qui créera et distribuera les itérateurs pour accéder à ses éléments.

Différences avec l'indexation

On utilise souvent un index dans une simple boucle, pour accéder séquentiellement à tous les éléments d'un tableau

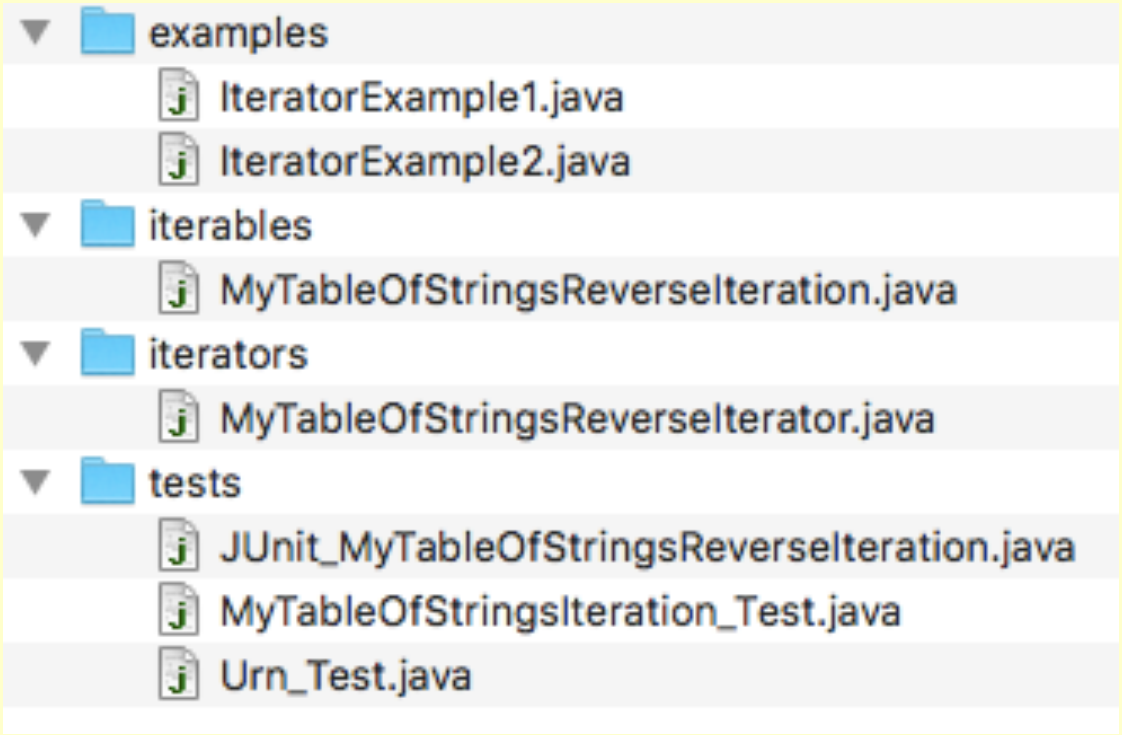
Cette approche reste possible en programmation objet pour certains conteneurs, mais l'utilisation des **itérateurs** a certains avantages:

- Un simple compteur dans une boucle n'est pas adapté à toutes les structures de données
- Avec les **itérateurs**, le code est plus lisible, réutilisable, et robuste
- Un itérateur peut implanter des restrictions additionnelles sur l'accès aux éléments
- Un itérateur peut *dans certains cas* permettre que le conteneur soit modifié, sans être invalidé pour autant.

See also- http://sourcemaking.com/design_patterns/iterator

Itérateur: Java Example using enumerations (legacy code)

~gibson/Teaching/CSC4522/SourceCode/Iterator.zip



Itérateur: Java Example using enumerations (legacy code)

~gibson/Teaching/CSC4522/SourceCode/Iterator.zip

```
import java.util.*;
public class IteratorExample1 {


private static void traverse(Enumeration e) {
while (e.hasMoreElements()) {
        System.out.println(e.nextElement());
        }

}

// main()

    // Create a Vector and add some items to it.
    // Traverse the vector using an Enumeration.
    // Now create a hash table and add some items to it.
    // Traverse the hash table keys using an Enumeration.
    // Traverse the hash table values using an Enumeration.
```

Itérateur: Java Example using enumerations (legacy code)

```
public static void main(String args[]) {
// Create a Vector and add some items to it.
Vector v = new Vector();
v.addElement(new Integer(5));
v.addElement(new Integer(9));
v.addElement(new String("Hi, There!"));

// Traverse the vector using an Enumeration.
Enumeration ev = v.elements();
System.out.println("\nVector values are:");
traverse(ev);

// Now create a hash table and add some items to it.
Hashtable h = new Hashtable();
h.put("Bob", new Double(6.0));
h.put("Joe", new Double(18.5));
h.put("Fred", new Double(32.0));

// Traverse the hash table keys using an Enumeration.
Enumeration ekeys = h.keys();
System.out.println("\nHash keys are:");
traverse(ekeys);

// Traverse the hash table values using an Enumeration.
Enumeration evalues = h.elements();
System.out.println("\nHash values are:");
traverse(evalues);
}
```

Itérateur: Java Example using enumerations (legacy code)

TO DO: Execute the code and understand how it works

```
Vector values are:
5
9
Hi, There!

Hash keys are:
Joe
Bob
Fred

Hash values are:
18.5
6.0
32.0
```

Types d'itérateurs

Les bibliothèques de certains langages poussent très loin le concept d'itérateurs comme la STL (Standart Template Library) de C++.

Les environnements de dernière génération comme Java ou .NET ne sont pas en reste, ils proposent en standard de nombreuses ***collections*** et différents types d'itérateurs pour les parcourir.

On peut classifier les itérateurs selon le sens possible de déplacement :

- Itérateurs unidirectionnels :
  - Déplacement en avant, c'est le plus standard.
  - Déplacement en arrière.
- Itérateurs Bi-directionnel :
  - Déplacement avant-arrière

Mais aussi selon le mode d'accès aux données :

- Itérateurs en lecture seule
- Itérateurs en lecture/ecriture

Les collections (et Itérateurs sur collection) en Java

Java dispose de base d'un riche ensemble de collections –les Java Collections Framework- définis par de nombreuses classes et interfaces.

Cet ensemble a été remis à neuf depuis la version 1.2 avec l'introduction de plusieurs interfaces pour « abstraire » les conteneurs comme la liste.

Les collections ont été unifiées et simplifiées autour de quelques classes et interfaces essentielles.

Java met à disposition un itérateur standard basique (classe Iterator) pour parcourir les éléments d'une collection (que dans un seul sens).

Java propose aussi un itérateur plus élaboré (classe ListIterator) qui offre un parcours bi-directionnel du conteneur.

Exemple de parcours avec itérateur

```
// création d'une liste d'éléments
ArrayList persons = new ArrayList();
 // remplir la liste d'éléments
Person p= new Person("john", "smith");
persons.add(p);
...
// demande un itérateur au conteneur
Iterator i = persons.iterator();
// tant qu'il un élément suivant dans la collection
while(i.hasNext())
{
    // extrait (une référence sur) l'élément suivant
    Person p= (Person)i.next();

    // traite l'élément extrait: l'affiche
    p.print();
}
```

Exemple de parcours avec itérateur

Version simplifiée plus concise pour le parcours des éléments, possible depuis java 1.5 :

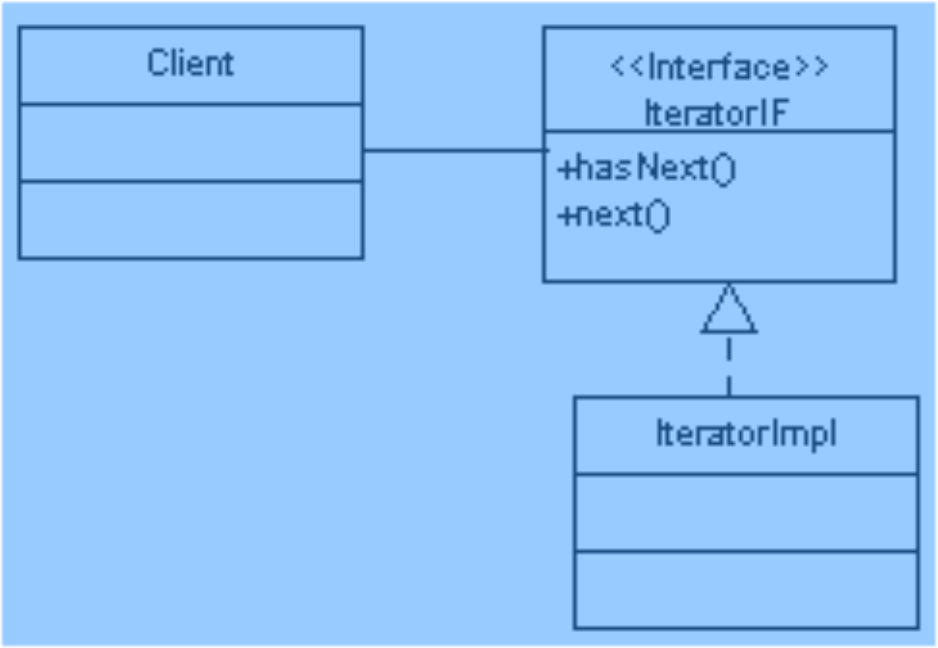
```
for(Person p : persons)
{
   p.print();
}
```

Iterable et iterator (interfaces) en Java

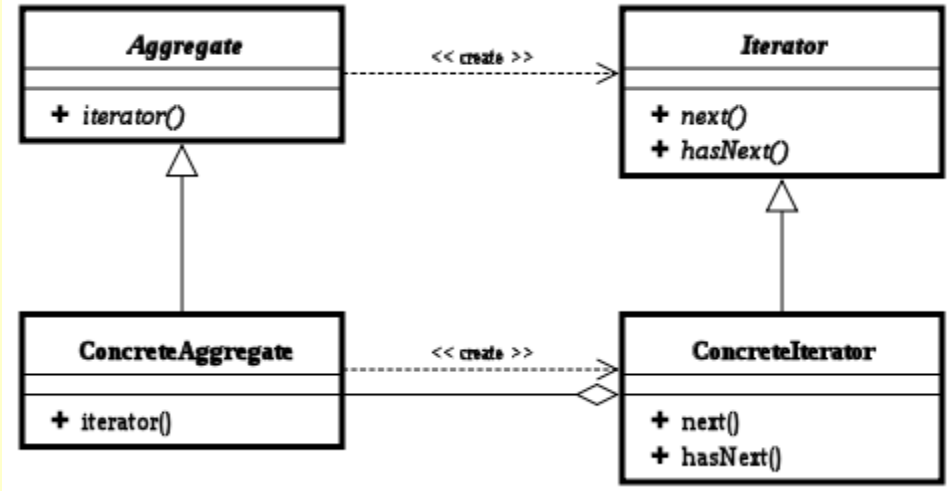
En Java il y a 2 interfaces pour gérer les itérateurs :

`Iterable<Type>` : indique que l'objet est une collection qui peut être parcourue par un itérateur => création de l' itérateur

`Iterator<Type>` : indique que l'objet est un itérateur => réalise le parcours



UML – concrete iterator



UML – abstract/ concrete iterator

```
public class MyTableOfStrings implements Iterable<String> {

protected String[] data;

public MyTableOfStrings(String [] data) {
this.data = data;
}

public int length(){return data.length;}

public Iterator<String> iterator() {
return new MyTableOfStrings_Iterator(this);
}

}
```

```
public class MyTableOfStrings_Iterator implements Iterator<String> {

private int index;
private MyTableOfStrings table;

public MyTableOfStrings_Iterator(MyTableOfStrings tab) {
index = tab.length()-1;
table = tab;
}

public String next() {
index--;
return table.data[index +1];
}

public boolean hasNext() {
return index >= 0;
}

public void remove() {
throw new UnsupportedOperationException();
}

}
```

```
public class MyTableOfStrings_Test {

public static void main(String[] s) {

String [] data = {"one", "two", "three"};
MyTableOfStrings t = new MyTableOfStrings(data);

System.out.println("Iterate over original data array");
for (String value : data) {
System.out.println(" "+value);
}
System.out.println("\nIterate over same data in MyTableOfStrings");
for (String value : t) {
System.out.println(" "+value);
}
}

}
```

p_Iterator.MyTableOfStrings_Test

Version:

1 A simple test for the `MyTableOfStrings`

EXPECTED OUTPUT :

```
Iterate over original data array
 one
 two
 three

Iterate over same data in MyTableOfStrings
 three
 two
 one
```

Author:

J Paul Gibson

TO DO : Compile and execute the test class

Random Iteration: Reservoir Sampling

In the previous example we saw how the `Iterator` code decides the order in which to visit the elements.

By default Java iterates through arrays from the 1st to the last elements. In the example we iterate through `MyTableOfStrings` **in reverse order.**

TO DO:

Change the iterator code so that the elements are visited **in random order**. Do not do this by shuffling the elements.

A more complex data structure: an urn/ballot box of bulletins/votes

p_Iterator.Urn

Version:

1 An urn of votes, where each vote is a table of strings, eg:

```
[["gibson", "smith", "hughes"],
["jones", "bell"],
["raffy", "lallet"]]
```

represents three preferential votes with the first vote being -
first preference for gibson, second preference for smith and third preference for hughes

Author:

J Paul Gibson

Your task is to iterate through the Strings in the Urn ***(in any order)***

Look at the `Urn_Test` Code and write the `Urn` and `Urn_Iterator` classes appropriately. (You may choose to change the Urn data structure if you wish)

Check that the test, executed on your code, produces the expected results

```
public class Urn_Test {

public static void main(String[] s) {

String [] preferences1 = {"gibson", "smyth", "hughes"};

String [] preferences2 = {"jones", "bell"};

String [] preferences3 = {"raffy", "lallet", "muller", "bac"};

String [][] votes = { preferences1, preferences2, preferences3};

Urn urn = new Urn (votes);

// Iterate over strings in any order
System.out.println("\nIterate over strings on bulletins in Urn");
for (String value : urn) {System.out.println(" "+value);}
}
}
```

Simple Validation Test

```
Iterate over strings on bulletins in Urn
 bac
 lallet
 raffy
 muller
 bell
 jones
 smyth
 hughes
 gibson
```