Provee un mecanismo estándar para acceder secuencialmente a los elementos de una colección ya que la interface `Iterator` declara métodos para acceder secuencialmente a los objetos de una colección. Una clase accede a una colección a través de dicha interface. Lo que se busca es acceder a los contenidos de los objetos incluidos sin exponer su estructura.

Este patrón nace para poder soportar diversas formas de recorrer objetos y para ofrecer una interfaz uniforme para recorrer distintos tipos de estructuras de agregación.

Se utiliza cuando:

- Una clase necesita acceder al contenido de una colección sin llegar a ser dependiente de la clase que es utilizada para implementar la colección, es decir sin tener que exponer su representación interna.
- Una clase necesita un modo uniforme de acceder al contenido de varias colecciones.
- Cuando se necesita soportar múltiples recorridos de una colección.

Este patrón debe ser utilizado cuando se requiera una forma estándar de recorrer una colección, es decir, cuando no es necesario que un "cliente" sepa la estructura interna de una clase (un cliente no siempre necesita saber si debe recorrer un `List` o un `Set`, ni tampoco qué clase concreta está recorriendo).

Las colecciones en el paquete `java.util` siguen el patrón `Iterator`. La causa de por qué Java utiliza este patrón es muy simple: existen alrededor de 50 colecciones y cada una de ellas es un caso de estudio: cada una funciona de manera distinta a la otra, con algoritmos muy distintos. Aprender cómo se debe recorrer una colección en Java es un proceso, por tanto, diverso y complejo. Pero si todas las colecciones se recorren de la misma forma estándar entonces se consigue un gran avance.

**EJEMPLO**

Vamos a realizar un ejemplo muy sencillo: una división o sucursal que contiene empleados. Para ello internamente implementaremos un `Array`.

```
public class Division {
    private Empleado[] empleados = new Empleado[100];
    private int numero = 0;
    private String nombreDivision;

    public Division(String n) {
        nombreDivision = n;
    }

    public String getName() {
        return nombreDivision;
    }

    public void add(String nombre) {
        Empleado e = new Empleado(nombre, nombreDivision);
        empleados[numero++] = e;
    }

    public DivisionIterator iterator() {
        return new DivisionIterator(empleados);
    }
}
```

```
public class Empleado {

    private String nombre;
    private String division;

    public Empleado(String n, String d) {
        nombre = n;
        division = d;
    }

    public String getName() {
        return nombre;
    }

    public void print() {
        System.out.println("Nombre: " + nombre + " Division: " + division);
    }
}
```

Y ahora crearemos un `Iterator` para recorrer los empleados de la división. El método `iterator` devuelve un objeto `DivisionIterator`.

```java
public class DivisionIterator implements Iterator<Empleado> {
    private Empleado[] empleado;
    private int location = 0;

    public DivisionIterator(Empleado[] e) {
        empleado = e;
    }

    public Empleado next() {
        return empleado[location++];
    }

    public boolean hasNext() {
        if (location < empleado.length && empleado[location] != null) {
            return true;
        } else {
            return false;
        }
    }

    public void remove() {
    }
}
```

En la aplicación usaremos los métodos `next` y `hasNext` para acceder a los nodos de la colección.

```java
    public static void main(String[] args) {
        Division d = new Division("Mi Sucursal");
        d.add("Empleado 1");
        d.add("Empleado 2");

        Iterator<Empleado> iter = d.iterator();
        while (iter.hasNext()) {
            Empleado e = (Empleado) iter.next();
            e.print();
        }
    }

}
```

Ventajas:

- Es posible acceder a una colección de objetos sin conocer el código de los objetos.
- Utilizando varios objetos `Iterator` es simple tener y manejar varios recorridos al mismo tiempo (incluso de diferentes modos).
- Las clases `Iterator` simplifican el código de las colecciones, ya que el código de los recorridos se encuentra en ellas y no en las colecciones.

**INTERFACES `Iterator` y `ListIterator`**

```
public interface Iterator<T> {
	boolean hasNext();
	T next();
	void remove();
}
```

La interfaz `Iterator` dispone de los siguientes métodos:

- `public abstract boolean hasNext();`
  Devuelve true si aún quedan elementos por recorrer y `false` en caso contrario.

- `public abstract java.lang.Object next();`
  La primera vez que se le llama devuelve el primer elemento. En cada llamada posterior devuelve el siguiente elemento del recorrido. Este método tiene como precondición que `hasNext()` devuelva `true`. Si no se cumple la precondición, entonces elevará la excepción `NoSuchElementException`.

- `public abstract void remove();`
  Elimina del agregado el último elemento devuelto por el método `next()`. Este método es la única forma segura de eliminar un elemento mientras se recorre una colección.

La interface `ListIterator` permite recorrer una lista en ambas direcciones y hacer algunas modificaciones. Sus métodos son los siguientes:

- `public abstract void add(java.lang.Object);`
- `public abstract boolean hasNext();`
- `public abstract boolean hasPrevious();`
- `public abstract java.lang.Object next();`
- `public abstract int nextIndex();`
- `public abstract java.lang.Object previous();`
- `public abstract int previousIndex();`
- `public abstract void remove();`
- `public abstract void set(java.lang.Object);`

Los elementos se numeran de `0` a `n-1`. Los valores posibles del índice son de `0` a `n`.

Se puede considerar que el índice `i` apunta entre los elementos `i-1` e `i` (`previous()` devolvería el elemento `i-1` y `next()` devolvería el elemento `i`).

Si el índice es `0`, `previousIndex()` devuelve `-1` y si es `n` `nextIndex()` devuelve `size()`.