**¿Cómo se recorre una colección?**
Varias formas:

- con los métodos propios de la clase concreta (típicamente con el método get())
- con un objeto iterador.
- con for mejorado
- con programación funcional (lo veremos en próximos boletines)

**Porque puede ser útil la alternativa de usar un iterador para recorrer una colección**

- Los Iteradores ofrecen una forma de recorrer una colección standard común a muchas colecciones sin necesidad de conocer los métodos específicos de una colección.

- Indicamos en boletines anteriores que si mientras recorremos una colección hacemos inserciones y/o borrados pueden ocurrir efectos inesperados.

**¿De donde saco el objeto iterador?**
Las colecciones del API tiene un método iterator() resultado de implementar la interface Iterable. Comprueba esto en el api.

Podemos observar por ejemplo que las colecciones TreeSet y LinkedList tienen este método con el siguiente código.

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.TreeSet;

public class App {
    public static void main(String[] args) {
        LinkedList<Integer> ll= new LinkedList<>();
        TreeSet<Integer> ts= new TreeSet<>();
        Iterator<Integer> iteradorLista=ll.iterator();
        Iterator<Integer> iteradorSet=ts.iterator();
    }
}
```

OJO, JALEO DE NOMBRES, NO CONFUNDIR: método *iterator()* con interface *Iterator* ni con con interface *Iterable*
El método iterator() devuelve un objeto Iterator
public Iterator<E> iterator()

Iterator no es una clase, es una interface que especifica los siguientes métodos

***Method Summary***

| All Methods | Instance Methods | Abstract Methods | Default Methods |
|---|---|---|---|

| Modifier and Type | Method and Description |
|---|---|
| default void | **forEachRemaining**(Consumer<? super E> action)<br>Performs the given action for each remaining element until all elements have been processed or the action throws an exception. |
| boolean | **hasNext**()<br>Returns true if the iteration has more elements. |
| E | **next**()<br>Returns the next element in the iteration. |
| default void | **remove**()<br>Removes from the underlying collection the last element returned by this iterator (optional operation). |

## Ejemplo de uso de iterador para borrar en bucle

Con for normal puede haber problemas porque al ir borrando se va variando el size() de la lista

Observa que el siguiente código no borra 4 y 8 porque se sale del bucle antes de lo esperado

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

class App {
    public static void main(String[] args) {
        List<Integer> lista = new ArrayList<>(List.of(2,4,6,8,1));

        for (int i = 0; i < lista.size(); i++) {
            if (lista.get(i) % 2 == 0) {
                lista.remove(i);
            }
        }
        System.out.println(lista);

    }
}
```

SALIDA
[4, 8, 1]

Para solucionar esto tendríamos que decrementar el índice cuando se borra para compensar el borrado.

```
import java.util.ArrayList;
import java.util.List;
class App {
    public static void main(String[] args) {
        List<Integer> lista = new ArrayList<>(List.of(2,4,6,8,1));

        for (int i = 0; i < lista.size(); i++) {
            if (lista.get(i) % 2 == 0) {
                lista.remove(i);
                i--;
            }
        }
        System.out.println(lista);
    }
}
```

salida:
[1]

Pero esta cuestión y otras más ya están solucionadas automáticamente si recorremos la colección con un objeto iterador

```
import java.util.ArrayList;
```

```
import java.util.Iterator;
import java.util.List;
class App {
    public static void main(String[] args) {
        List<Integer> lista = new ArrayList<>(List.of(2,4,6,8,1));
        Iterator<Integer> it = lista.iterator();
        while(it.hasNext()){
            Integer i = it.next();
            if(i%2==0){
                it.remove();
            }
        }
        System.out.println(lista);
    }
}
```

**El interface Iterable y el bucle for mejorado**

Iterator e Iterable son ambos interfaces y están relacionados, pero son distintos. Una clase que implemente Iterable tendrá un método iterator() que automáticamente devolvera un objeto iterador.

Para recorrer una colección con un for mejorado tiene que implementar Iterable

Entenderemos mejor los conceptos de Iterator e Iterable a través del siguiente ejemplo.

**Ejemplo: conseguir que *MiListaEnlazada* se pueda recorrer con un iterador y con for mejorado**

Recuerda nuestra lista enlazada casera. Tal y como la escribimos inicialmente no se puede recorrer con for mejorado o con iterador.

```
class Persona{
    String nombre;
    int edad;

    Persona(String nombre,int edad){
        this.nombre=nombre;
        this.edad=edad;
    }
    @Override
    public String toString(){
        return "("+nombre+", "+edad+")";
    }
}


class Nodo<E>{
  private Nodo<E> sig;
  E dato;

  Nodo(E dato, Nodo<E> sig){
    this.dato = dato;
    this.sig = sig;
  }
  void setSiguiente(Nodo<E> sig) {
    this.sig = sig;
  }
  Nodo<E> getSiguiente() {
    return sig;
  }
  E getDato() {
    return dato;
  }
}

class MiListaEnlazada<E> {
    private Nodo<E> primero=null;
```

```
    void insertar(E dato){
        primero= new Nodo<>(dato,primero);
    }

    int tamano(){
        int i=0;
        Nodo<E> temp=primero;
        while(temp!=null){
            i++;
            temp=temp.getSiguiente();
        }
        return i;
    }
    E obtener(int indice){
        Nodo<E> temp=primero;
        int i=0;
        while(i<indice){
            temp=temp.getSiguiente();
            i++;
        }

        return   temp.getDato();
    }


}

class App{
    public static void main(String[] args){
        MiListaEnlazada<Persona> mle1=new MiListaEnlazada<>();
        MiListaEnlazada<Integer> mle2=new MiListaEnlazada<>();
        mle1.insertar(new Persona("yo",10));mle1.insertar(new Persona("tu",11));mle1.insertar(new Persona("el",12));
        mle2.insertar(91);mle2.insertar(92);mle2.insertar(93);

        System.out.println("Recorrer MiListaEnlazada<Integer> con métodos especificos de clase ");
        for(int i=0;i<mle2.tamano();i++)
            System.out.print(mle2.obtener(i)+"  ");

        System.out.println("\nRecorrer MiListaEnlazada<Persona> con métodos especificos de clase ");
        for(int i=0;i<mle1.tamano();i++)
            System.out.print(mle1.obtener(i)+"  ");

        //¡NO ES POSIBLE USAR FOREACH NI ITERATOR! descomenta y observa el error
        /*  System.out.println("\nCon foreach ...");
            for(Persona p : mle1){
               System.out.print(p);
            }

        */

    }
}
```

¡comprueba que al descomentar el código comentado da error!

¿Qué tenemos que hacer para que funcione sobre MiListaEnlazada iterator() y for mejorado?

1. Que MiListaEnlazada implemente Iterable
2. Por tanto tenemos que escribir un método iterator()
3. Lo que a su vez implica devolver un objeto Iterator
4. Lo que implica crear una clase que Implementa Iterator

vamos a conseguir esto pero en dos fases para entender todo mejor:

- Fase 1. Implementar Iterator: Conseguir recorrer la lista con un iterador
- Fase 2: Implementar Iterable: Escribir un método iterator() y recorrer la lista con for mejorado

Vamos a suponer además el siguiente escenario de programadores. Por un lado está el programador de MiListaEnlazada que tiene que ser capaz de permitir a otros programadores que usen su clase para recorrerla con el standard Iterator, por otro lado tenemos al programador de App que justamente es un usuario de MiListaEnlazada

**MiIterator: Una clase que implementa el interface Iterator**

Si queremos recorrer nuestra lista enlazada de la forma standard que indica el interface Iterator hay que añadir al código anterior 2 cosas:

1. Una clase que implementa Iterator. La llamaremos MiIterator
2. Probar la clase anterior para recorrer la lista creando un objeto de esa clase y utilizando los métodos de Iterator hasNext() y next()

Partiendo del código anterior y siguiendo al profesor conseguimos el siguiente código(es mejor que lo vayas haciendo con el profesor que que lo copies directamente)

```
import java.util.Iterator;

class Persona{
    String nombre;
    int edad;

    Persona(String nombre,int edad){
        this.nombre=nombre;
        this.edad=edad;
    }
    @Override
    public String toString(){
        return "("+nombre+", "+edad+")";
    }
}


class Nodo<E>{
  private Nodo<E> sig;
  E dato;

  Nodo(E dato, Nodo<E> sig){
    this.dato = dato;
    this.sig = sig;
  }
  void setSiguiente(Nodo<E> sig) {
    this.sig = sig;
  }
  Nodo<E> getSiguiente() {
    return sig;
  }
  E getDato() {
    return dato;
  }
}
class MiIterator<E> implements Iterator<E>{
    int posicion;//posicion actual

    MiListaEnlazada<E> mle;//lista a recorrer


    MiIterator(MiListaEnlazada<E> mle){//le paso la lista a recorrer

        posicion=0;
        this.mle=mle;
    }


     @Override
    public boolean hasNext() {
        if(mle==null||posicion>=mle.tamano())
            return false;
        else
            return true;

    }
```

```
    @Override
    public E next() {
        return mle.obtener(posicion++);
    }

}
class MiListaEnlazada<E> {
    private Nodo<E> primero=null;
    void insertar(E dato){
        primero= new Nodo<>(dato,primero);
    }

    int tamano(){
        int i=0;
        Nodo<E> temp=primero;
        while(temp!=null){
            i++;
            temp=temp.getSiguiente();
        }
        return i;
    }
    E obtener(int indice){
        Nodo<E> temp=primero;
        int i=0;
        while(i<indice){
            temp=temp.getSiguiente();
            i++;
        }

        return   temp.getDato();
    }


}

class App{
    public static void main(String[] args){
        MiListaEnlazada<Persona> mle1=new MiListaEnlazada<>();
        mle1.insertar(new Persona("yo",10));mle1.insertar(new Persona("tu",11));mle1.insertar(new Persona("el",12));
        //el programador de App tiene que ser consciente de que existe una clase que implementa Iterator
        MiIterator<Persona> mi= new MiIterator<>(mle1);
        while(mi.hasNext()){
            System.out.print(mi.next()+" ");
        }
       // pero for mejorado no funciona con MiListaEnlazada

    }
}
```

En la segunda fase vamos a mejorar este código implementando Iterable, lo que permitirá al programador de App ignorar que existe una clase MiIterator y además poder usar el for mejorado

**Implementar Iterable: Escribir un método iterator() y recorrer la lista con for mejorado**

```
import java.util.Iterator;

class Persona{
    String nombre;
    int edad;

    Persona(String nombre,int edad){
        this.nombre=nombre;
        this.edad=edad;
    }
    @Override
    public String toString(){
        return "("+nombre+", "+edad+")";
    }
}
```

```
class Nodo<E>{
  private Nodo<E> sig;
  E dato;

  Nodo(E dato, Nodo<E> sig){
    this.dato = dato;
    this.sig = sig;
  }
  void setSiguiente(Nodo<E> sig) {
    this.sig = sig;
  }
  Nodo<E> getSiguiente() {
    return sig;
  }
  E getDato() {
    return dato;
  }
}
class MiIterator<E> implements Iterator<E>{
    int posicion;
    MiListaEnlazada<E> mle;

    MiIterator(MiListaEnlazada<E> mle){
        posicion=0;
        this.mle=mle;
    }


    @Override
    public boolean hasNext() {
        return posicion<mle.tamano();
    }

    @Override
    public E next() {
        return mle.obtener(posicion++);
    }

}
class MiListaEnlazada<E> implements Iterable<E>{
    private Nodo<E> primero=null;
    void insertar(E dato){
        primero= new Nodo<>(dato,primero);
    }

    int tamano(){
        int i=0;
        Nodo<E> temp=primero;
        while(temp!=null){
            i++;
            temp=temp.getSiguiente();
        }
        return i;
    }
    E obtener(int indice){
        Nodo<E> temp=primero;
        int i=0;
        while(i<indice){
            temp=temp.getSiguiente();
            i++;
        }

        return   temp.getDato();
    }

    @Override
    public Iterator<E> iterator() {
        return new MiIterator<E>(this);
    }


}

class App{
```

```
    public static void main(String[] args){
        MiListaEnlazada<Persona> mle1=new MiListaEnlazada<>();
        mle1.insertar(new Persona("yo",10));mle1.insertar(new Persona("tu",11));mle1.insertar(new Persona("el",12));

        //ahora el programador de App trabaja a nivel Interface
        //no tiene que saber que existe, ni le importa la clase MiIterator
        Iterator<Persona> mi= mle1.iterator();
        while(mi.hasNext()){
            System.out.print(mi.next()+" ");
        }

        //el for mejorado hace automático el proceso anterior
        for(Persona p:mle1){
            System.out.print(p+" ");
        }
    }
}
```

**for mejorado VERSUS ITERATOR**

El bucle for mejorado es elegante, muy legible y sobre todo ¡es más fácil!. Pero...en ciertas situaciones necesitamos iterator.

El ejemplo del principio del boletín con el for mejorado genera un error de concurrencia

```
class App {
    public static void main(String[] args) {
        List<Integer> lista = new ArrayList<>(List.of(2,4,6,8,1));

        for(Integer i:lista){
            if(i%2==0){
                lista.remove(i);
            }
        }

        System.out.println(lista);
    }
}
```

Esto es debido a que el for mejorado usa internamente un iterador que bloquea la colección para que mientras se recorra dicha colección con el iterador no se puede modificar desde otro lado.

Pues bien, fíjate mucho que el remove() que usamos en el ejemplo de arriba no es el del iterador si no el método remove() de List con lo que es otro código que intenta acceder a la lista para modificarla y por eso se genera error de concurrencia. con la sintaxis del for mejorado el manejo del iterador es algo interno y no hay manera de indicarle que queremos usar el remove() del iterador.

**CONCLUSIÓN:** si hay borrados/insercciones durante el recorrido la solución más genérica y segura es usar Iterator.

**Ejercicio U7_B9_E1:** Resuelve este ejercicio con iterator.

Ya que no hay modificaciones no es realmente necesario usar iteradores pero lo hacemos para practicar el concepto.

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
public class App {

    public static void main(String[] args) {
        Map<String,Double> hm= new HashMap<>();
```

```
        // repasando autoboxing. La sentencia comentada tiene problemas con autoboxing
        //El autoboxing  permite pasar double a Double pero no int a Double. 1500 es int
        //hm.put("Elías", 1500);
        hm.put("Elías", 1500.0);
        hm.put("Román", 1900.0);
        hm.put("Telma", 2400.0);

        Set<String> conjuntoDeLlaves=hm.keySet();
        System.out.println("recorrer con for mejorado");
        for(String s:conjuntoDeLlaves)
            System.out.println(s+ "  "+hm.get(s));


    }
}
```

**Ejercicio U7_B9_E2:** Añade código iterator para que *a* pase a contener sólo números impares

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
class App {
    public static void main(String[] args) {
        List<Integer> a= new LinkedList<>();
        a.add(1);a.add(2);a.add(3);a.add(4);
        a.add(5);a.add(6);a.add(7);a.add(8);a.add(9);
        System.out.println(a);
        //añadir aquí código

        System.out.println(a);
    }
}
```

**Ejercicio U7_B9_E3:**

Modifica el siguiente ejemplo de forma que usando un iterador realice los borrados correctamente, es decir, que elimine de la colección aquellos elementos con la coordenada x o y  menor de 15.

```
import java.awt.Point;
import java.util.ArrayList;

class App{
    public static void main(String[] args) {
        ArrayList<Point> al= new ArrayList<>();

        al.add(new Point(2,56));
        al.add(new Point(22,56));
        al.add(new Point(32,5));
        al.add(new Point(99,99));
        System.out.println(al);
        for(Point p: al){
            if(p.x < 15 || p.y<15){
                al.remove(p);
            }
        }
        System.out.println(al);
    }
}
```

**PATRÓN ITERATOR**

Anteriormente realmente estuvimos aplicando, sin saberlo, el patrón Iterator. Vamos a ver el patrón Iterator de manera más formal.

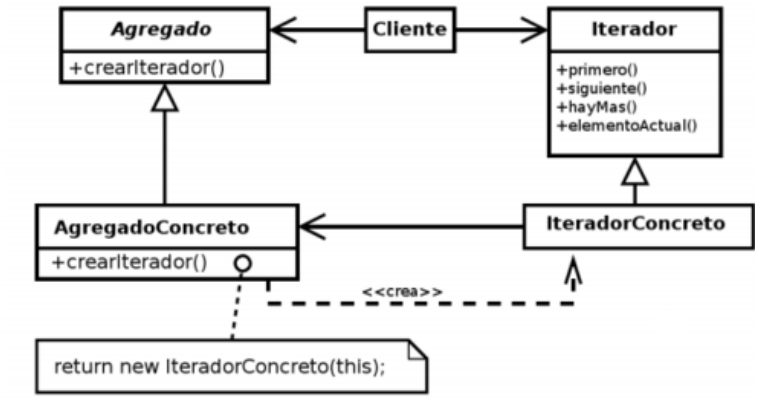
Crear iteradores para una colección necesita una serie de pasos repetitivos y una estructura que está descrita en el patrón Iterator. Como ya sabes un patrón es una solución "genérica" a un problema.
**Problema**: Piensa en una clase que internamente almacena un conjunto de datos, por ejemplo:

- Una lista de enteros que incluye un conjunto de enteros
- una Agenda que incluye un conjunto de objetos Contacto.
- un Equipo que incluye un conjunto de objetos Jugador.
- una lista genérica que incluye un conjunto de elementos de tipo T
- etc.

Las listas, conjuntos etc. ya están "listos para iterar" por el trabajo de los programadores del JDK. Piensa ahora entonces más en clases como Agenda o Equipo. Tu objetivo va a ser por ejemplo recorrer una Agenda sin enterarte si la Agenda se base en un array, arraylist, mapa o lo que sea.

**Solución:** crear un diseño que me permita acceder a las colecciones de forma independiente a su representación interna

y esto es lo que hace el patrón iterator utilizando el siguiente modelo genérico




esto está extraído de
https://es.wikipedia.org/wiki/Iterador_(patr%C3%B3n_de_dise%C3%B1o)

vamos a establecer correspondencias entre el diagrama anterior y el trabajo que hicimos con "MiListaEnlazada".

*Agregado:* sería una superclase abstracta o interface(como List o Set) pero no hay en nuestro ejemplo.
*AgregadoConcreto:* MiListaEnlazada que en este caso también se comunica con el cliente directamente ya que no hay superclase
*cliente:* App
*Iterador:* interface Iterator del JDK
*IteradorConcreto:* MiIterator
*crearIterator():* iterator()
*hayMas():*hasNext()
*siguiente():*next()

**PATRÓN DE DISEÑO ITERATOR EN EL JDK**
El JDK aplica este patrón en el frame collections en muchas situaciones. Pero es un framework muy grande y hay muchas variantes y extensiones de este patrón.

Una de las cosas que hay que tener en cuenta es que, respecto al diagrama UML anterior, la clase de *IteradorConcreto* no es "visible", por ejemplo, si obtenemos un iterador de un *ArrayList* usamos el método iterator() que devuelve algo de tipo Iterator<E>, que es un objeto que lo único que sé es que implementa Iterator. ¿Y para qué quiero saber más?

Otra cuestión es que además entra en juego el interface Iterable

**el interface Iterable del jdk**

No es un patrón, es simplemente un interface que obliga a las clases que lo implementan a devolver un iterador. Es una forma de marcar las clases que valen para trabajar con el for mejorado. Para recorrer una clase con for mejorado tiene que ser Iterable, es decir, tiene que implementar Iterable.

**Scanner implementa Iterator, no Iterable. List implementa Iterable**

Scanner es una clase que hace un uso del patrón anterior también con retoques

Comprueba en el api como la clase Scanner implementa Iterator<String>. Esto quiere decir que ella mismo incorpora en su código los métodos next(), hasNext(), ...Si implementara Iterable tendría un método iterator() que devolvería el iterador con dichos métodos

**EL PATRÓN DE DISEÑO ITERATOR OCULTA IMPLEMENTACIONES**

Ocultar implementaciones es un objetivo de muchos patrones.Suele ser interesante no exponer(ocultar) los detalles internos de una clase a sus clientes para conseguir:

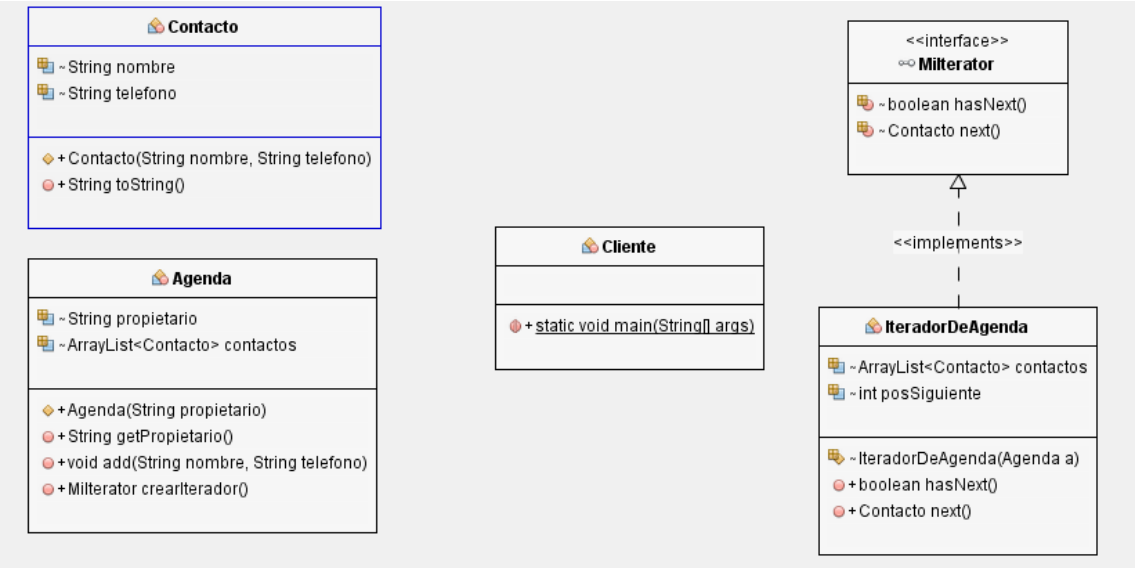
1. Simplificar el uso de la clase. El programador de una clase cliente de dicha clase quiere un uso de ella sencillo e inmediato. No quiere estudiar detalles de implementación.
2. Favorecer la posibilidad de cambiar el interior sin afectar a los clientes
3. Evitar un mal uso de la clase

El patrón Iterator ayuda a esto cuando uno de los atributos de la clase es una colección y deseamos que el cliente recorra esa colección pero sin que tenga que ser consciente de la implementación de la misma.

**EJEMPLO:**

Tengo una clase Agenda, uno de sus atributos es un ArrayList, el cliente puede recorrer los contactos con next() pero no sabe si realmente hay un ArrayList, un mapa u otra colección para guardar los datos. Si en el futuro la clase Agenda cambia el ArrayList por una LinkedList o un mapa al cliente le da igual porque la sigue recorriendo con next()

Vamos a aplicar el patrón Iterator a la Agenda pero para hacer más pequeño el ejemplo no usamos la clase Abstracta "Agregado" del patrón de forma que no tenemos AgendaAbstracta y AgendasConcretas, simplemente tenemos una clase Agenda que además no va a ser genérica, sólo puede ser una agenda de objetos Contacto.




.

Observa que *el interface MiIterator* no necesita ser genérico pues siempre itera sobre objetos Contacto.

```
 import java.util.ArrayList;

class Contacto {
    String nombre;
    String telefono;

    public Contacto(String nombre, String telefono) {
        this.nombre = nombre;
        this.telefono = telefono;
    }

    @Override
    public String toString() {
        return "nombre=" + nombre + ", teléfono="+telefono ;
    }

}

interface  MiIterator {
    boolean hasNext();
    Contacto next();
}
 class IteradorDeAgenda  implements MiIterator{
    ArrayList<Contacto> contactos;
    int posSiguiente;
    IteradorDeAgenda(Agenda a){
        this.contactos=a.contactos;
        posSiguiente=0;
    }
    @Override
    public boolean hasNext() {
        return posSiguiente<contactos.size();//tambien cubre caso array vacio pues
    }

    @Override
    public Contacto next() {
       return contactos.get(posSiguiente++);
    }

}

 class Agenda {
    String propietario;
    ArrayList<Contacto> contactos= new ArrayList<>();

    public Agenda(String propietario){
        this.propietario=propietario;
```

```
    }
    public String getPropietario(){
        return propietario;
    }
    public void add(String nombre, String telefono){
      contactos.add(new Contacto(nombre,telefono));
    }

    public  MiIterator crearIterador(){
         return new IteradorDeAgenda(this);
     }

}


class Cliente{
    public static void main(String[] args){
      Agenda agendaChuchi= new Agenda("Chuchi");
      agendaChuchi.add("eluno", "11111");
      agendaChuchi.add("dosi", "22222");
      agendaChuchi.add("tresi", "333333");
      MiIterator  it= agendaChuchi.crearIterador();
      System.out.println("Agenda de "+ agendaChuchi.getPropietario());
      while(it.hasNext()){
          System.out.println(it.next());
      }
    }
}
```

La Agenda, está implementada con un ArrayList pero observa cómo gracias al uso del iterador el cliente ignora cómo está implementada la agenda, simplemente itera sobre ella.

**Ejercicio U7_B9_E4:** Cambia el ArrayList por un Array y comprueba que el código de la clase Cliente sigue funcionando exactamente igual. Evidentemente habrá que modificar la clase Agenda y su iterador. Suponemos que como mucho tendremos 100 contactos.

**Ejercicio U7_B9_E5:** Ahora, devuelta con la versión con ArrayList nos empeñamos en poder recorrer la agenda con un for mejorado. Para esto tenemos que "comunicarnos" con los programadores del JDK y tenemos que usar Iterator e Iterable de JDK. Como estos interfaces son genéricos los usaremos con Objetos Contacto de la forma Iterable<Contacto> e Iterator<Contacto>

Consigue pues que funcione el siguiente Cliente haciendo los cambios oportunos en el código del ejemplo con ArrayList

```
class Cliente{
    public static void main(String[] args){
      Agenda agendaChuchi= new Agenda("Chuchi");
      agendaChuchi.add("eluno", "11111");
      agendaChuchi.add("dosi", "22222");
      agendaChuchi.add("tresi", "333333");

      System.out.println("Agenda de "+ agendaChuchi.getPropietario());
      for (Contacto c : agendaChuchi) {
          System.out.println(c);
      }
    }
}
```