Programación de sistemas

Listas enlazadas

Departamento de Ingeniería Telemática

Contenidos

- *** Introducción a los genéricos
- Estructuras de datos
- Listas enlazadas
  - La clase Node
  - La clase LinkedList
  - Ventajas y desventajas de las listas enlazadas

Introducción a los genéricos (*generics*)

- Servicio proporcionado por Java (J2SE 5.0 y superiores)
- Permiten comprobar los tipos a los que pertenecen las instancias durante la compilación
  - Los fallos se producen **en tiempo de compilación** (se garantiza que siempre hay un tipo compatible durante la ejecución)

Introducción a los genéricos (*generics*)

Sin Genéricos

```
List list = new ArrayList();
list.add("test");
Integer value = (Integer) list.get(0);
```

Error en tiempo de ejecución
"ClassCastException".
Se detiene el programa

Con Genéricos

```
List<Integer> list = new ArrayList<Integer>();
list.add("test");
Integer value = list.get(0);
```

Error en tiempo de compilación
"The method add(Integer) in the type List<Integer> is not applicable for the arguments (String)"
Se detecta a tiempo antes de ejecutar el programa

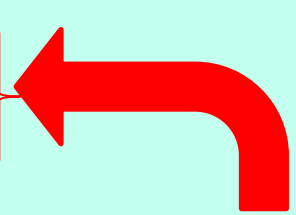
Introducción a los genéricos (*generics*)

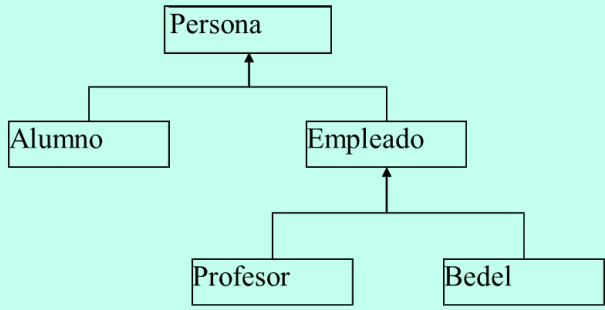
Creación de una clase que utiliza genéricos

```
public class ArrayList<E>{
    public boolean add(E e){…}
    public E get(int index){…}
…
}
```

Uso de una clase que utiliza genéricos

```
ArrayList<Empleado> employees = new ArrayList<Empleado>();
employees.add(new Empleado());
Empleado myEmployee = employees.get(0);
employees.add(new Bedel());
employees.add(new Persona());
employees.add(new Object());
```





Errores de compilación

Estructuras de datos

- Abstracción que representa un conjunto de datos en un programa con el objetivo de facilitar su manipulación
- Diferentes estructuras de datos presentan ventajas o desventajas dependiendo de la naturaleza de los datos y los tipos de manipulaciones que se necesite hacer sobre los mismos

Estructuras de datos lineales

- Organizan los datos en secuencia, donde cada dato se relaciona con un dato anterior (excepto el primero) y un dato posterior (excepto el último)
- Ejemplos de estructuras de datos lineales:
  - ✓ Arrays
  - ✓ Listas enlazadas
  - ✓ Pilas
  - ✓ Colas
  - ✓ Colas dobles

Arrays

- **Ventajas** para el almacenamiento de colecciones lineales de datos:
  - ✓ Acceso aleatorio: se puede acceder a cualquier posición del array en tiempo constante.
  - ✓ Uso eficiente de memoria cuando todas las posiciones están ocupadas: por guardarse en posiciones consecutivas de memoria.

Arrays

- **Desventajas**
  - ✓ Tamaño estático: debe asignarse un tamaño al crear el array, y no se puede cambiar. Da lugar a problemas:
    - Uso no eficiente de memoria por tener que reservar espacio para el caso peor
    - Posibilidad de sobrepasar el tamaño reservado en tiempo de ejecución
  - ✓ Necesidad de memoria contigua:
    - Puede ocurrir que, pese a haber suficiente memoria libre, no haya un bloque contiguo suficientemente grande

Arrays

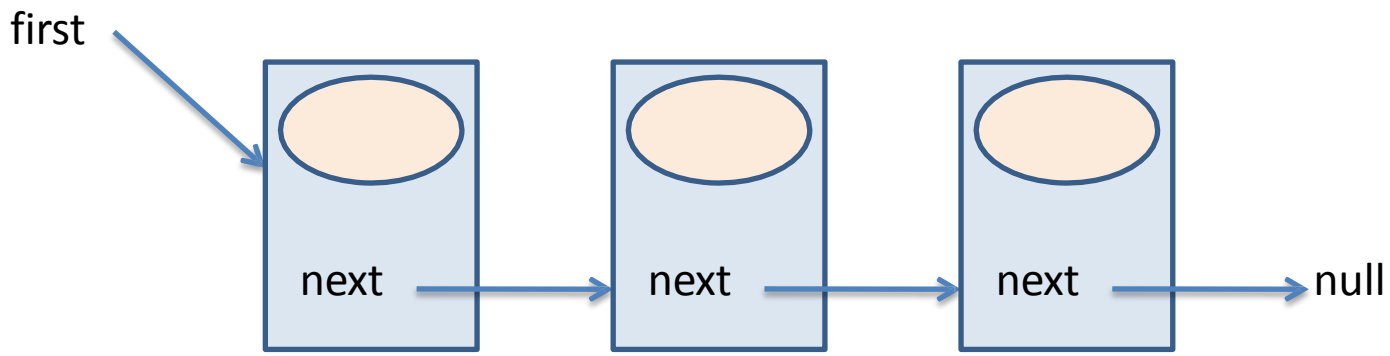
- **Desventajas**
  - ✓ Ciertas operaciones tienen un coste no óptimo:
    - Inserciones y eliminaciones de datos en la primera posición o posiciones intermedias: necesidad de desplazar datos entre posiciones consecutivas.
    - Concatenación de dos o más arrays: necesidad de copiar los datos a un nuevo array.
    - Partición de un array en varios fragmentos: necesidad de copiar datos a nuevos arrays.

Ejercicio 1

- Crea un array de diez elementos de tipo entero e inicialízalos todos con el valor 0. Después inserta un elemento adicional con valor 1 en la cuarta posición del array

Listas enlazadas

- Secuencia ordenada de **nodos** donde cada nodo almacena:
  - ✓ Un dato (información)
  - ✓ Una referencia al siguiente nodo
- Los nodos no tienen por qué estar contiguos en memoria



La clase Node

```
public class Node<E> {
    private E info;
    private Node<E> next;

    public Node(E info)   {…}

    public Node<E> getNext() {…}
    public void setNext(Node<E> next) {…}
    public E getInfo() {…}
    public void setInfo(E info) {…}

}
```

Uso de genéricos para almacenar información de diferentes tipos

Dos atributos: dato y referencia al siguiente nodo

Constructor para inicializar el dato

Métodos get y set de los atributos

Ejercicio 2

- Completa el código de la clase Node. Incluye tres constructores: uno que no recibe información para inicializar ningún atributo; otro que permite inicializar el atributo `info`, y otro que permite inicializar los dos atributos

La clase MyLinkedList

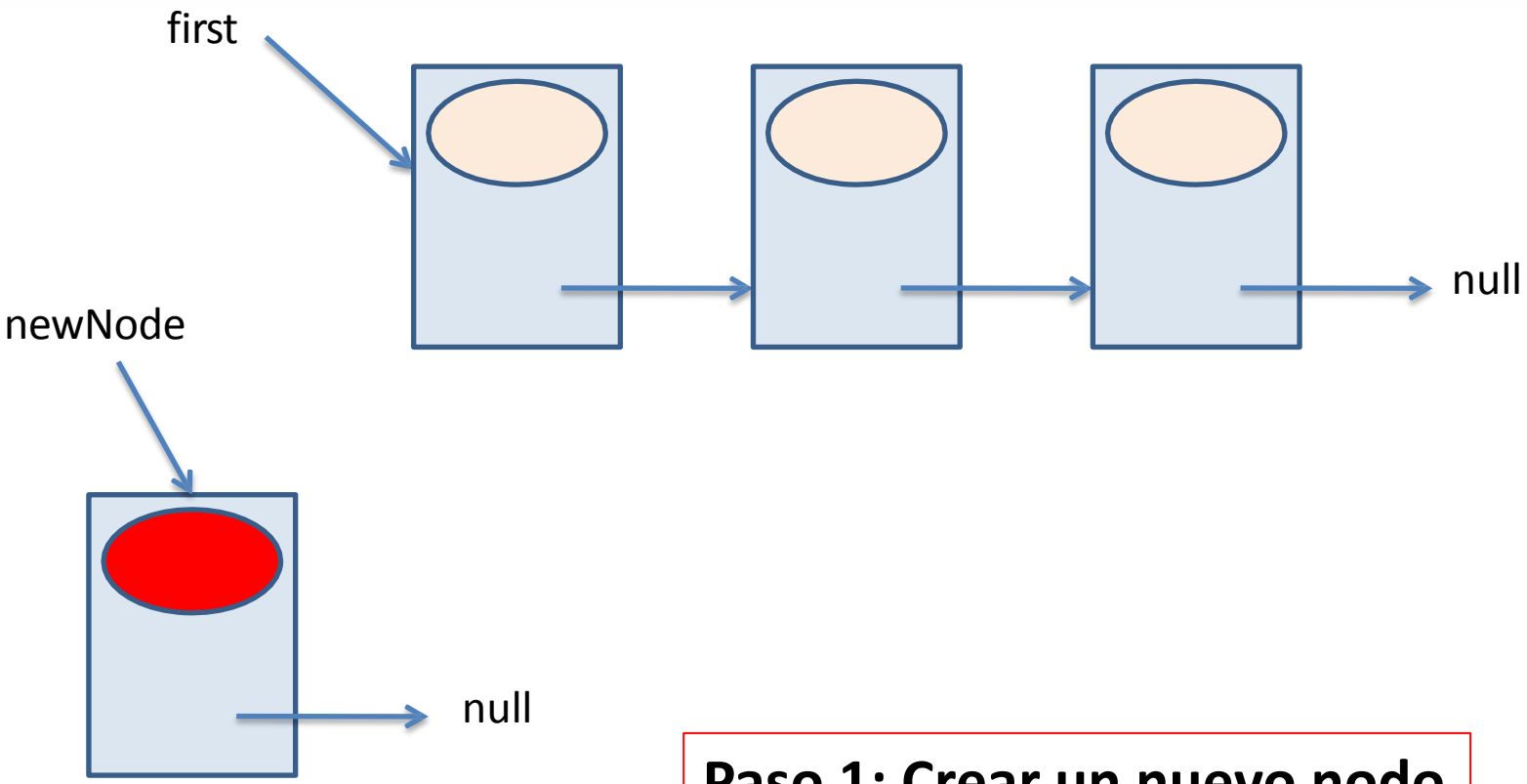
```
public class MyLinkedList<E> {
    private Node<E> first;

    public MyLinkedList() {…}

    public void insert(E info) {…}
    public E extract() {…}
    public void insert(E info, Node<E> previous) {…}
    public E extract(Node<E> previous) {…}
    public void print() {…}
    public Node<E> searchLastNode() {…}
    public int search(E info) {…}
…
}
```

Inserción al principio

```
public void insert(E info)
```

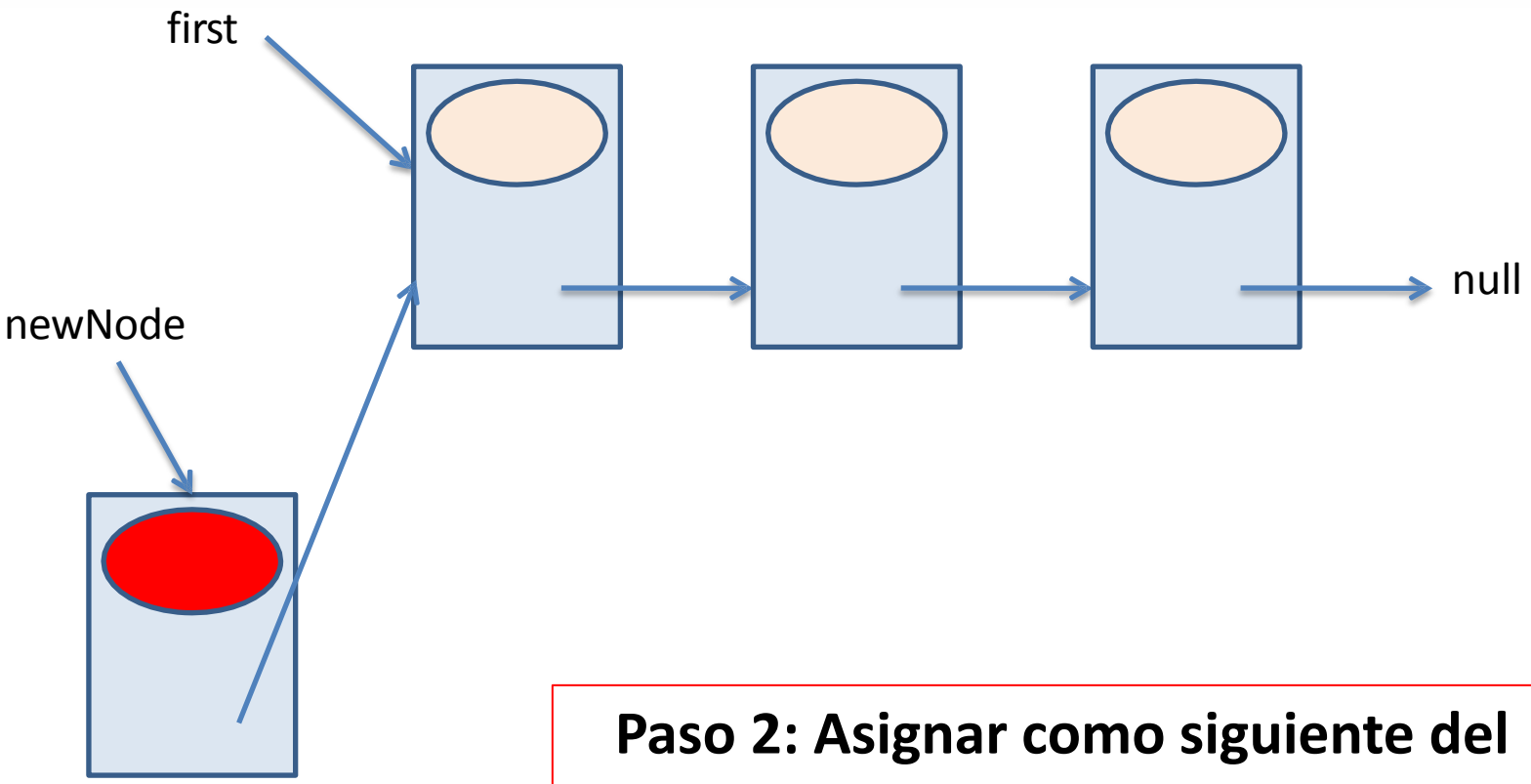


Paso 1: Crear un nuevo nodo

Node<E> newNode = new Node<E>(info);

Inserción al principio

```
public void insert(E info)
```

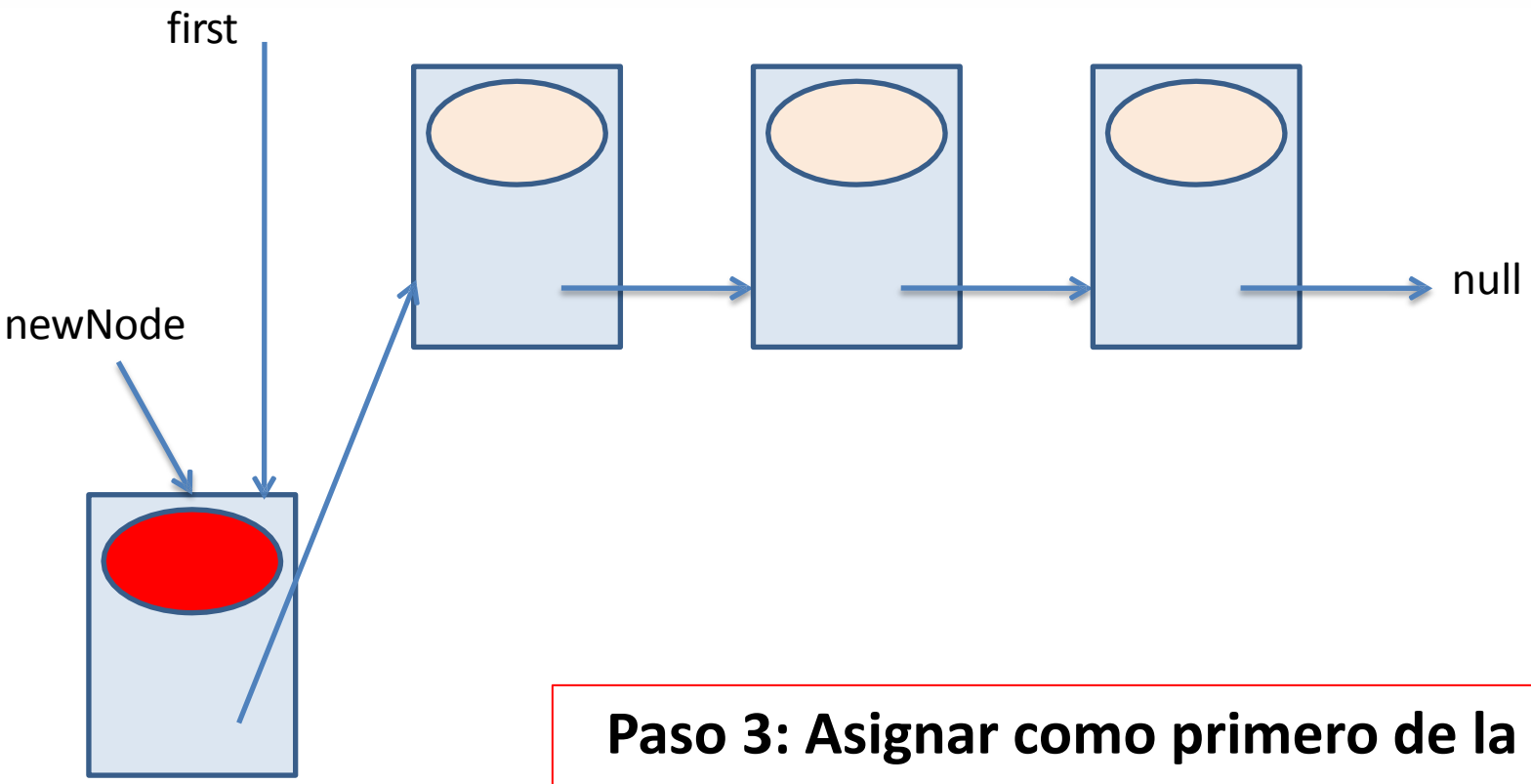


Paso 2: Asignar como siguiente del nodo creado al primero de la lista

newNode .setNext(first);

Inserción al principio

```
public void insert(E info)
```

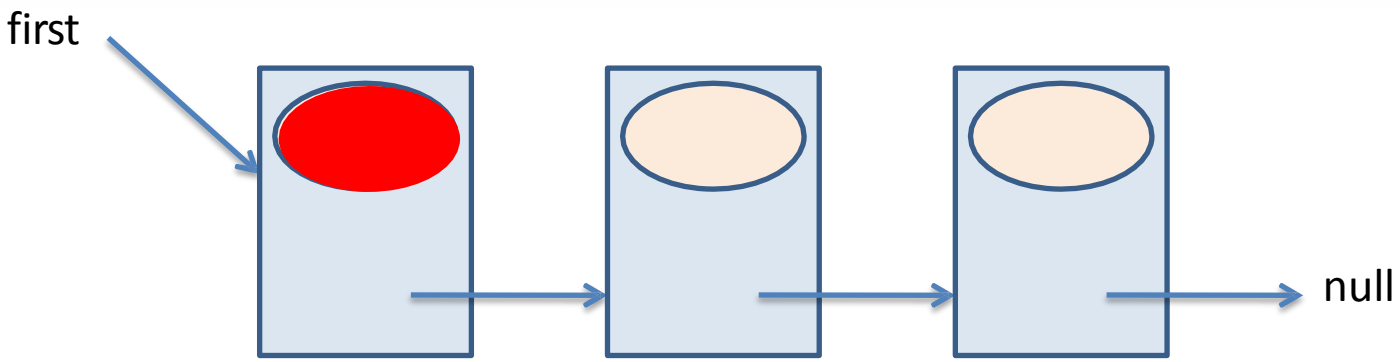


Paso 3: Asignar como primero de la lista al nodo creado

first = newNode;

Extracción del primer nodo

```
public E extract()
```

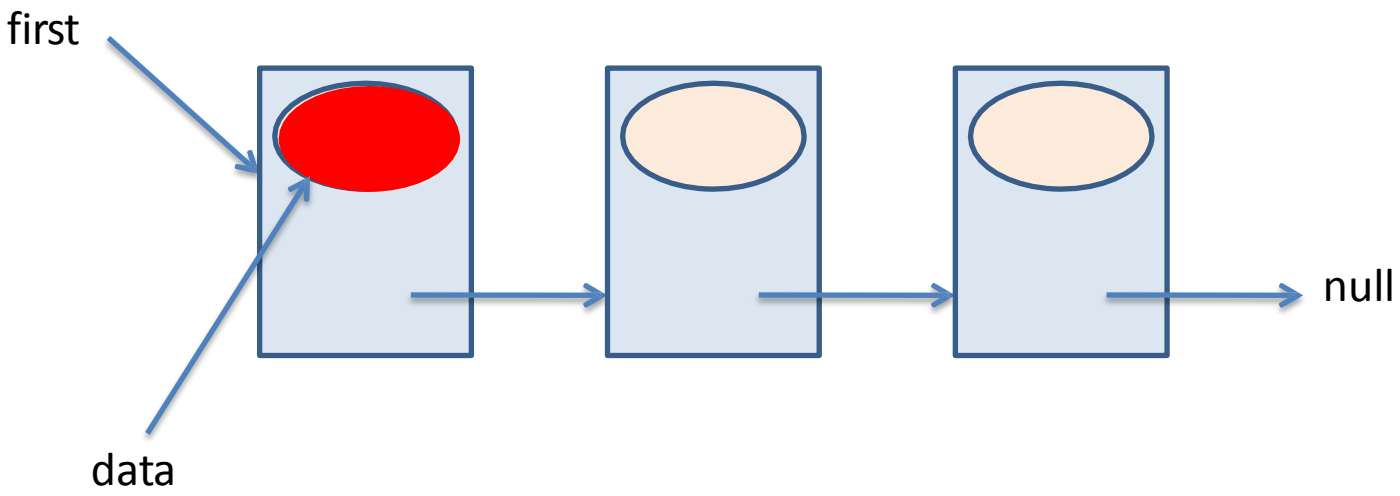


Paso 1: Comprobar que la lista no está vacía

```
if (first != null)
```

Extracción del primer nodo

```
public E extract()
```

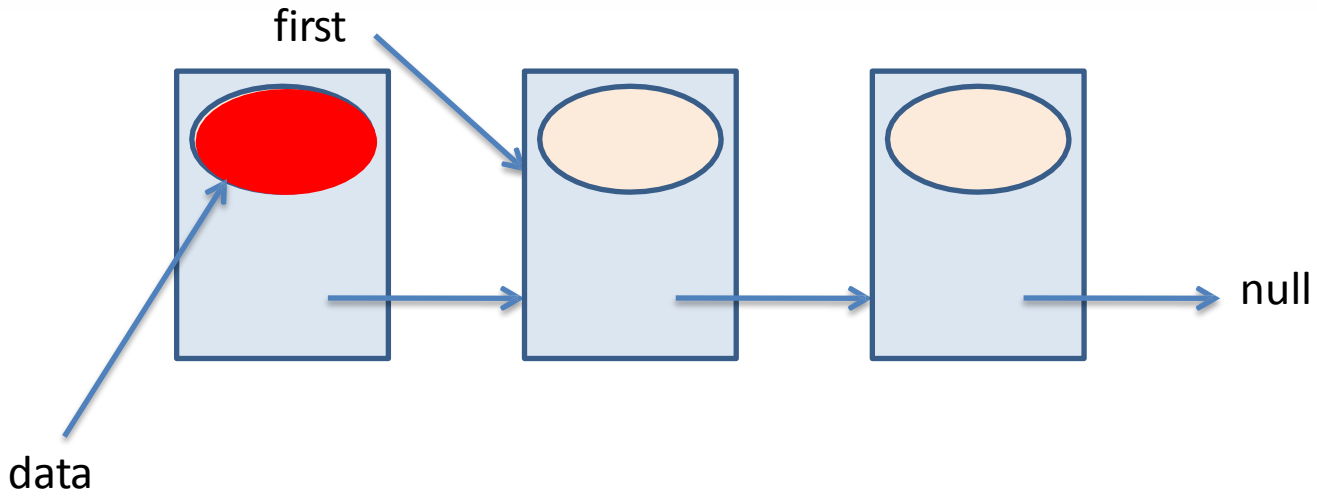


Paso 2: Recuperar la información del nodo a extraer

E data = first.getInfo();

Extracción del primer nodo

```
public E extract()
```

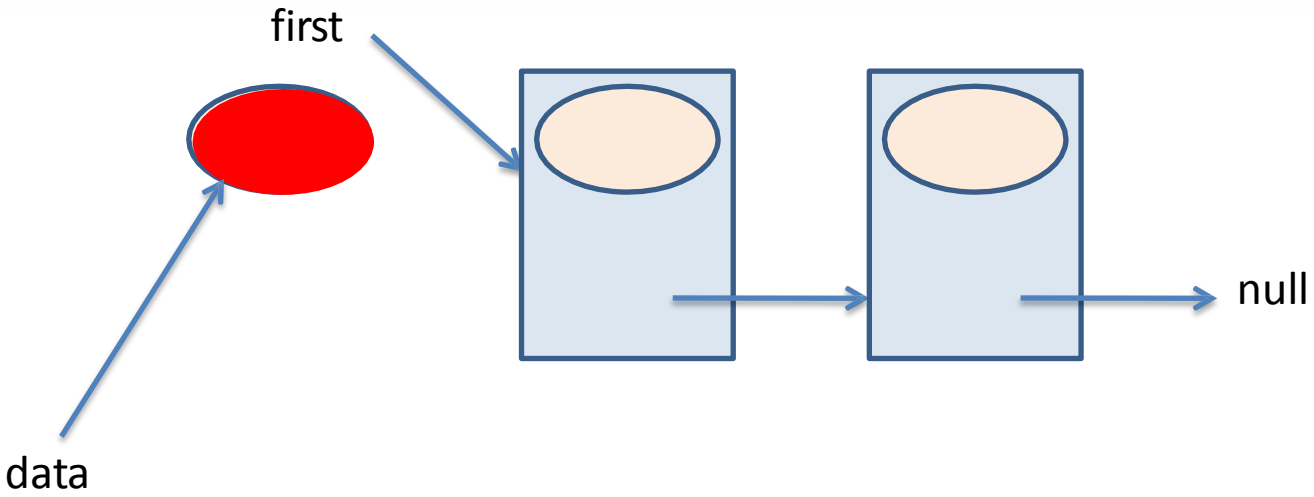


Paso 3: Asignar como primer nodo de la lista al segundo

first = first.getNext();

Extracción del primer nodo

```
public E extract()
```



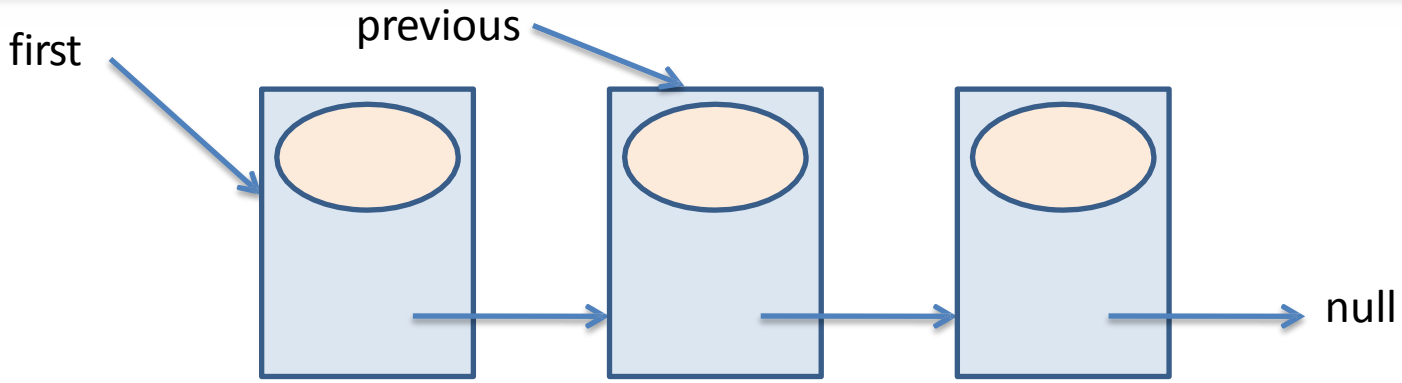
El nodo queda aislado y ya no pertenece a la lista

Paso 4: Devolver el dato

return data;

Inserción en un punto intermedio

```
public void insert(E info, Node<E> previous)
```

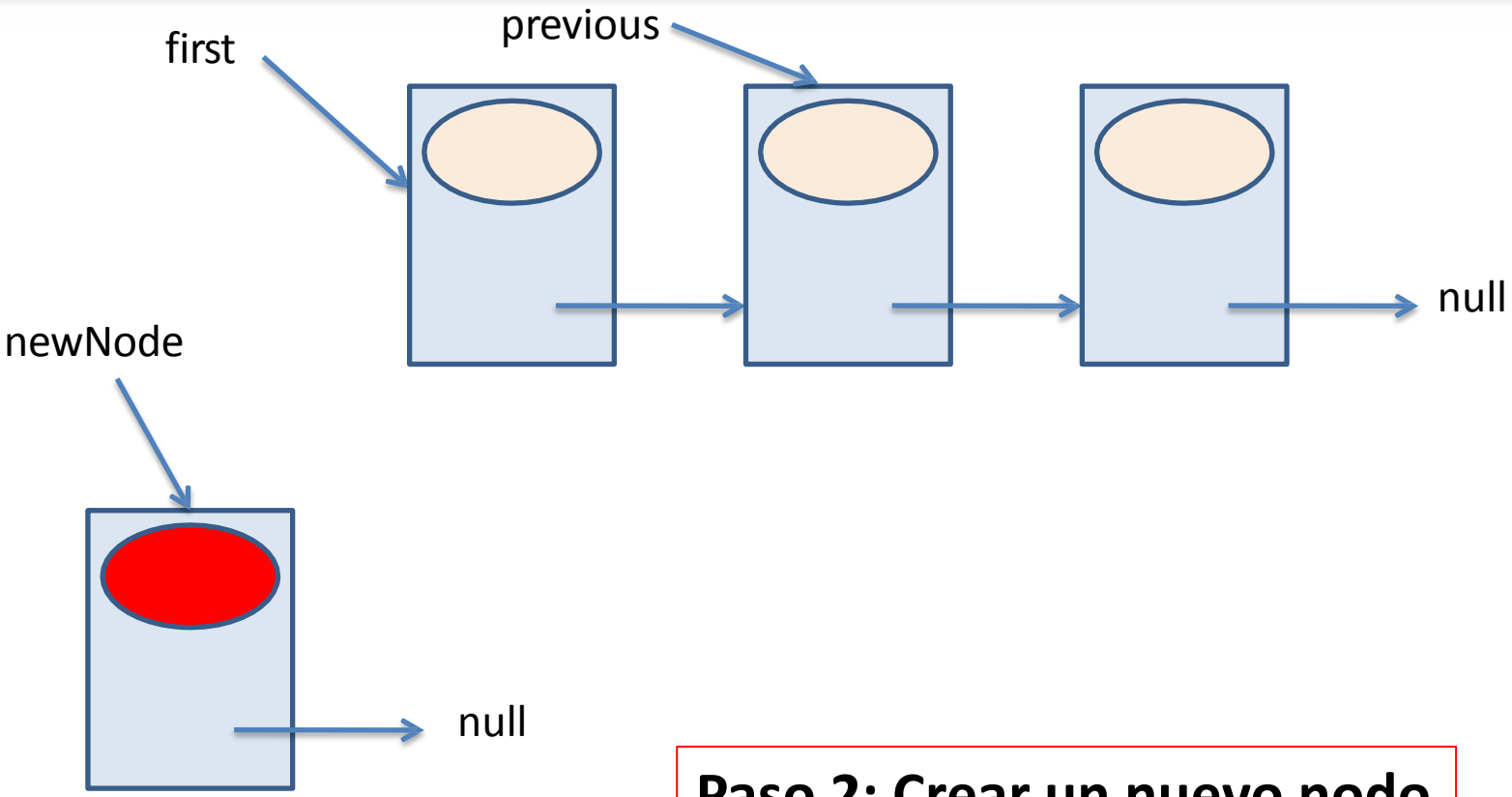


Paso 1: Comprobar que previous no es null

```
if (previous != null)
```

Inserción en un punto intermedio

```
public void insert(E info, Node<E> previous)
```

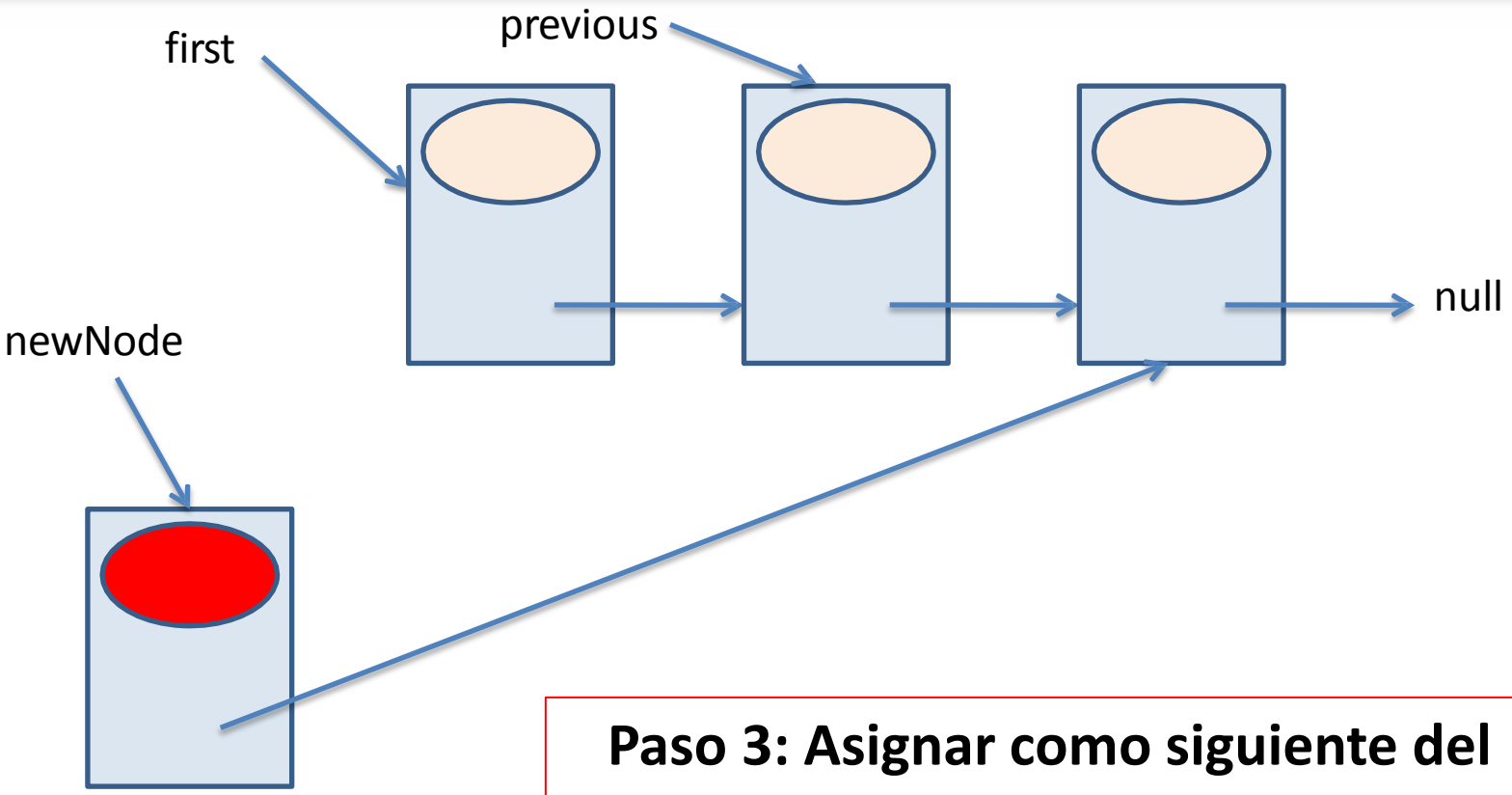


Paso 2: Crear un nuevo nodo

```
Node<E> newNode = new Node<E>(info);
```

Inserción en un punto intermedio

```
public void insert(E info, Node<E> previous)
```

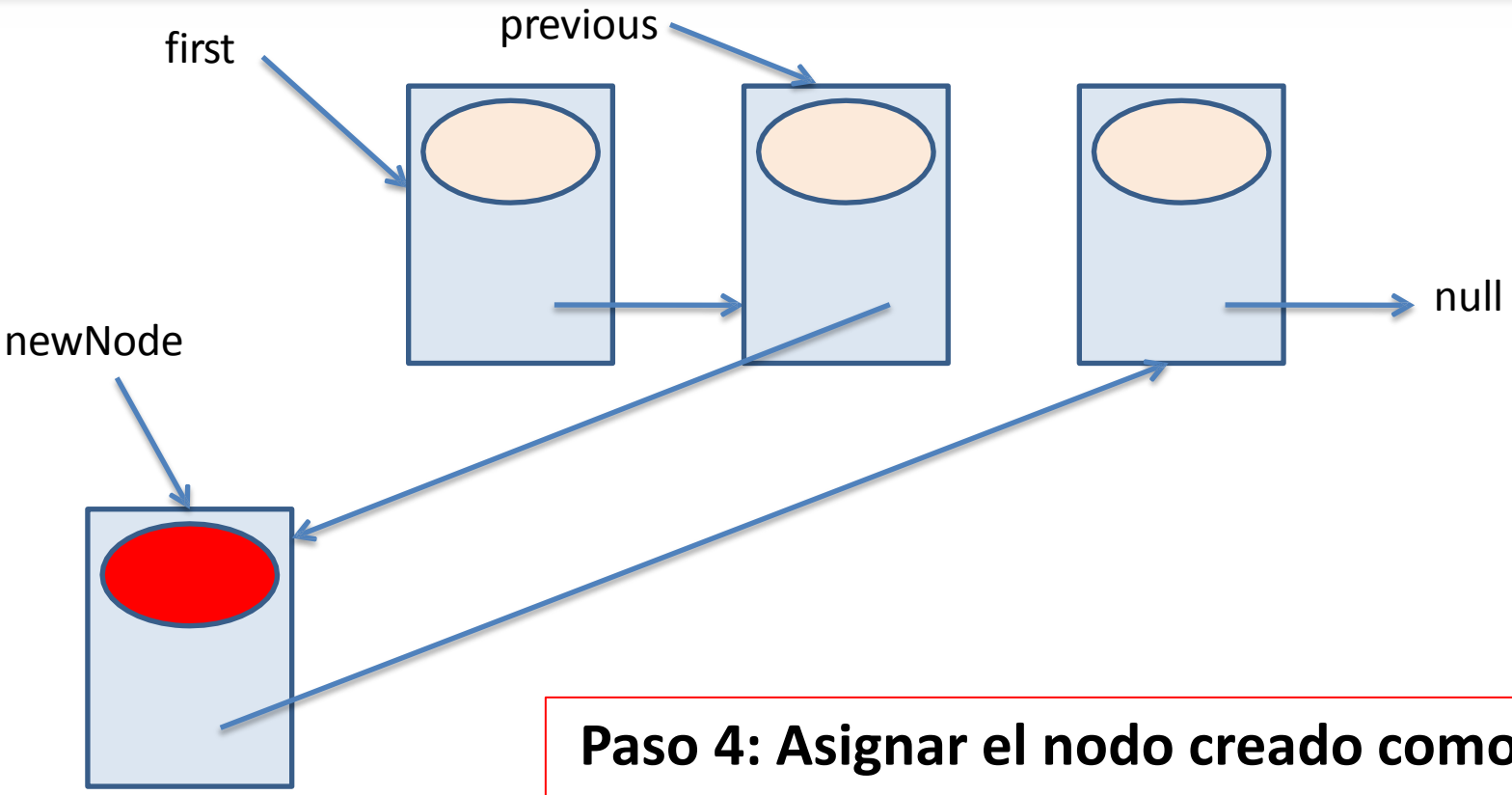


Paso 3: Asignar como siguiente del nodo creado al siguiente a *previous*

```
newNode.setNext(previous.getNext());
```

Inserción en un punto intermedio

```
public void insert(E info, Node<E> previous)
```

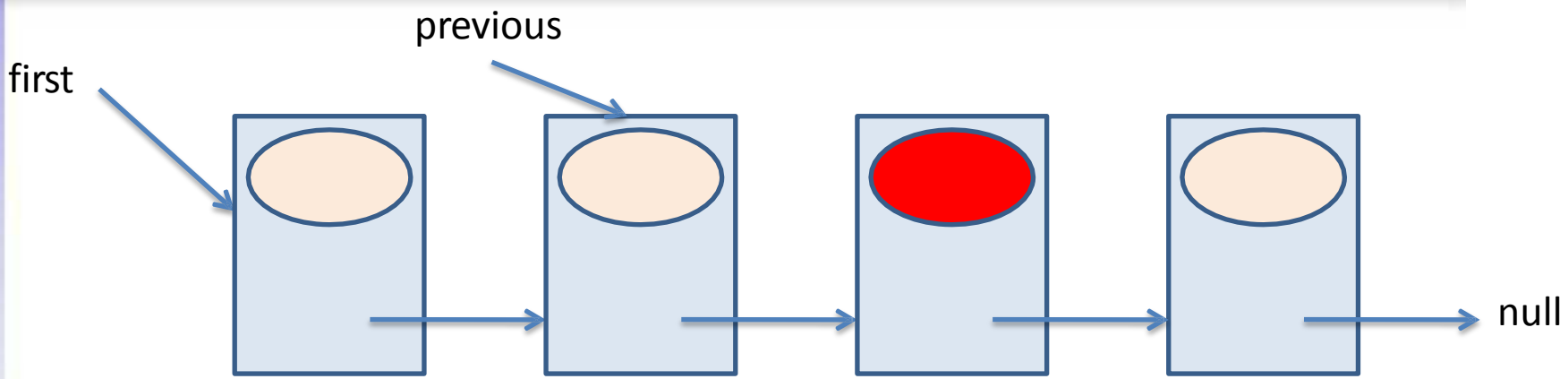


Paso 4: Asignar el nodo creado como el siguiente a *previous*

previous.setNext(newNode);

Eliminación de un dato intermedio

```
public E extract(Node<E> previous)
```

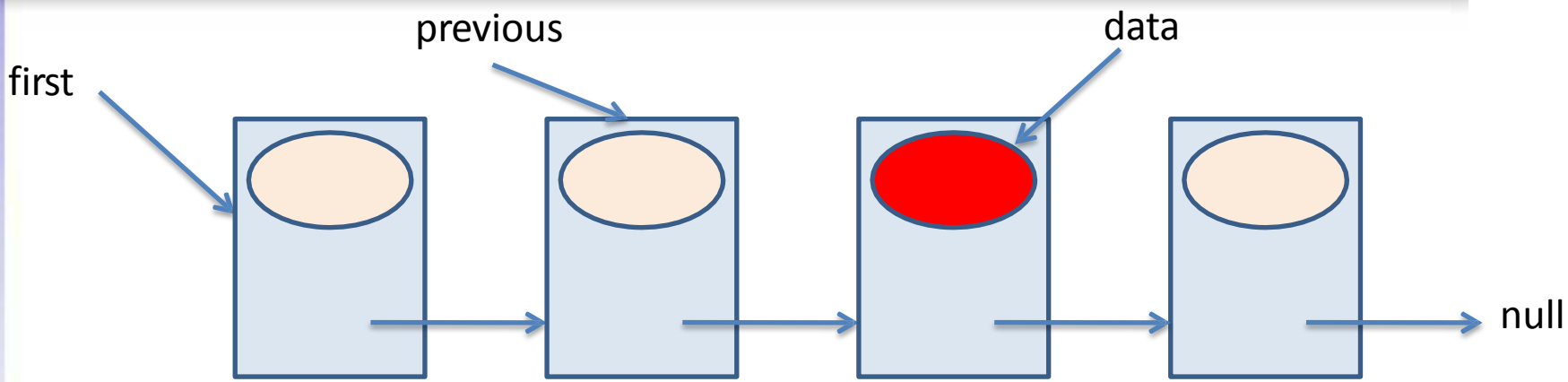


Paso 1: Comprobar que la lista no está vacía.
Comprobar que previous no es null.
Comprobar que el siguiente a previous no es null

```
if (first != null && previous!= null && previous.getNext() != null)
```

Eliminación de un dato intermedio

```
public E extract(Node<E> previous)
```

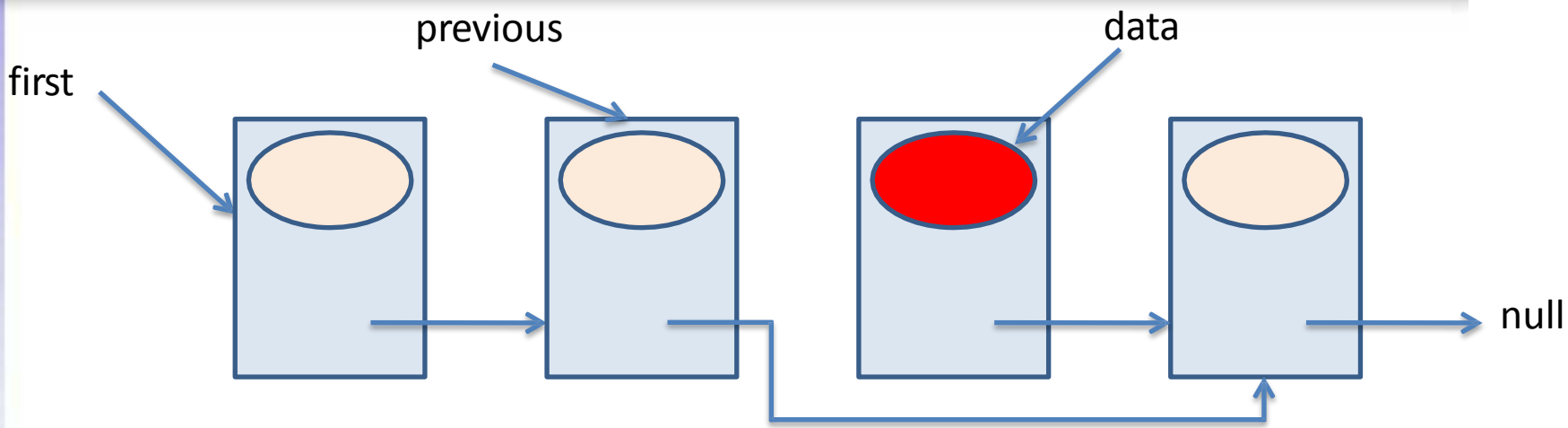


Paso 2: Recuperar la información del nodo a extraer

E data = previous.getNext().getInfo();

Eliminación de un dato intermedio

```
public E extract(Node<E> previous)
```

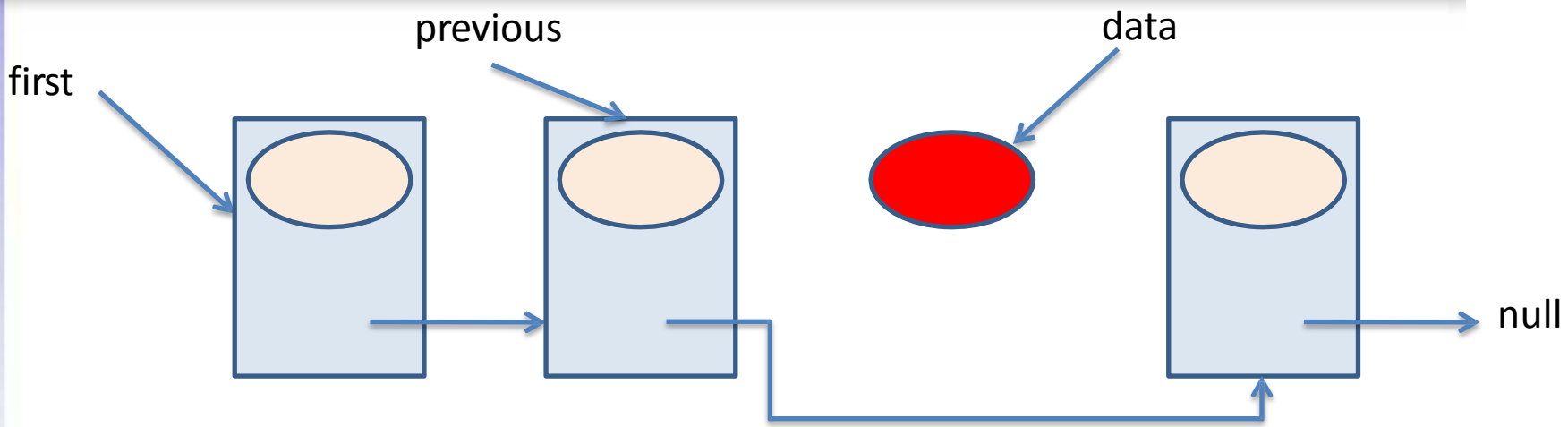


Paso 3: Asignar como siguiente a previous el siguiente al nodo a extraer

previous.setNext(previous.getNext().getNext())

Eliminación de un dato intermedio

```
public E extract(Node<E> previous)
```



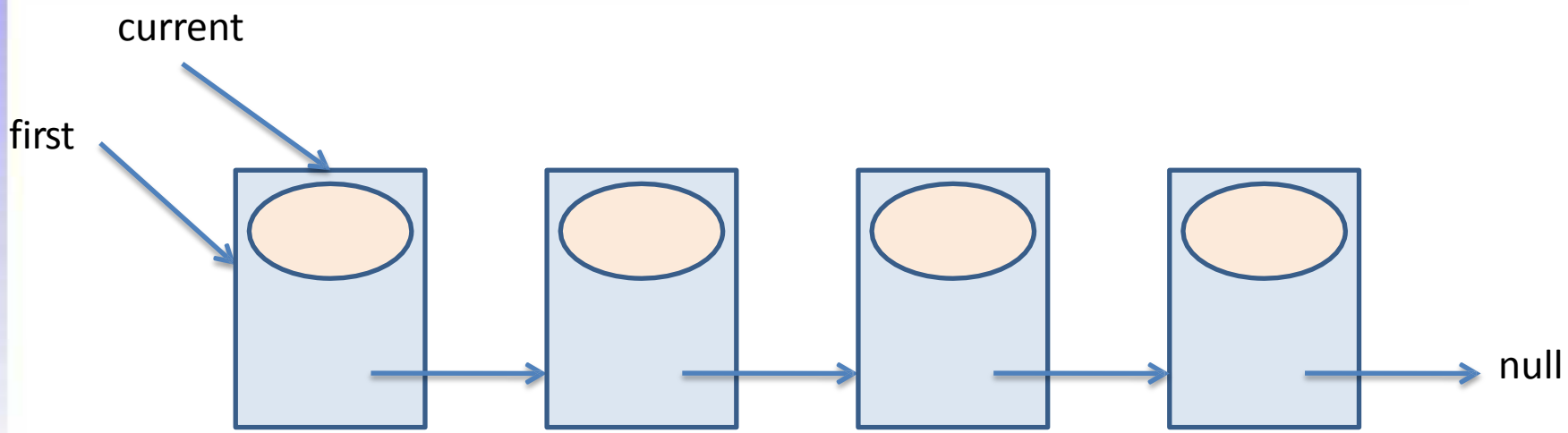
El nodo queda aislado y ya no pertenece a la lista

Paso 4: Devolver el dato

return data;

Recorrido de la lista e impresión

```
public void print()
```




```
Node<E> current = first;
while (current != null) {
    System.out.println(current.getInfo());
    current = current.getNext();
}
```

Recorrido: buscar el último nodo

`public Node<E> searchLastNode()`

- Se avanza una referencia hasta localizar un nodo cuyo siguiente sea null:

```
public Node<E> searchLastNode() {
    Node<E> last = null;
    Node<E> current = first;
    if (current != null) {
        while (current.getNext() != null) {
            current = current.getNext();
        }
        last = current;
    }
    return last;
}
```

Recorrido: buscar posición de un dato

`public int search(E info)`

- Se avanza una referencia hasta localizar el dato.

  Se va incrementando un contador al mismo tiempo:

```
public int search(E info) {
    int pos = 1;
    Node<E> current = first;
    while (current != null
        && !current.getInfo().equals(info)) {
        pos += 1;
        current = current.getNext();
    }
    if (current != null){
        return pos;
    } else{
        return -1;
    }
}
```

Ejercicio 3

- Crea el método `public int numberOfOcurrences(E info)`, que devuelva el número de nodos de una lista enlazada cuya información almacenada es la misma que la proporcionada como argumento

Ventajas de las listas enlazadas

- Inserción y extracción de nodos con coste independiente del tamaño de la lista
- Concatenación y partición listas con coste independiente del tamaño de las listas
- No hay necesidad de grandes cantidades de memoria contigua
- El uso de memoria se adapta dinámicamente al número de datos almacenados en la lista en cada momento

Desventajas de las listas enlazadas

- Acceso a posiciones intermedias con coste dependiente del tamaño de la lista
- Necesidad de memoria adicional para almacenar los objetos `Node` con sus atributos

Ejercicio 4

- Crea una Lista Enlazada de diez elementos de tipo entero e inicialízalos todos con el valor 0. Después inserta un elemento adicional con valor 1 en la cuarta posición de la Lista Enlazada. Asume que ya existen la clase **`Node`** y la clase **`MyLinkedList`** tal y como se han programado anteriormente.