Unidad 5: Estructuras de datos básicas

Metodología de la Programación

Curso 2021-2022

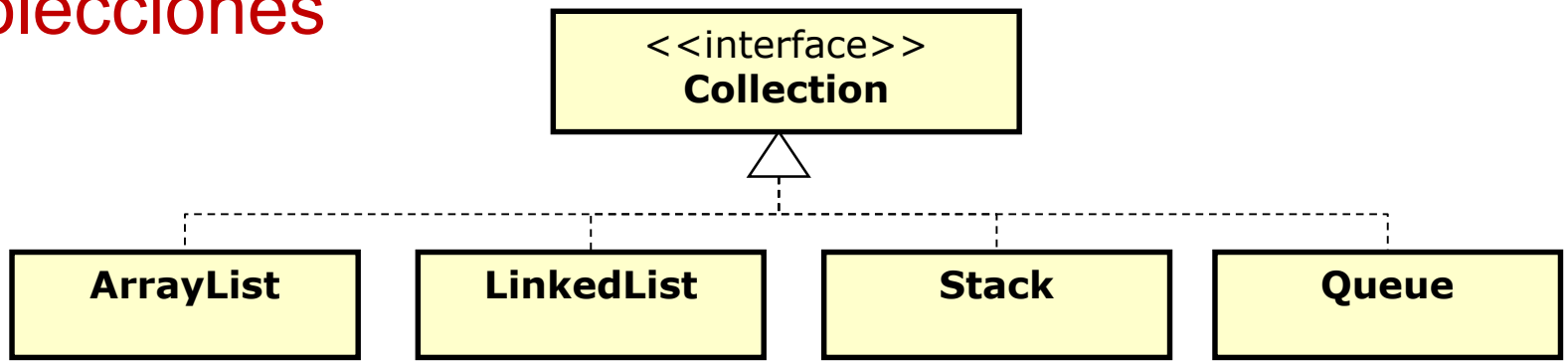
© *Candi Luengo Díez , Francisco Ortín Soler y José Manuel Redondo López, Alberto M. Fernandez Álvarez*

Principales conceptos

- Collections (*Colecciones*)
- Array Lists *(Listas Array)*
- Linked Lists *(Listas enlazadas)*
- Stacks *(Pilas)*
- Queues *(Colas)*

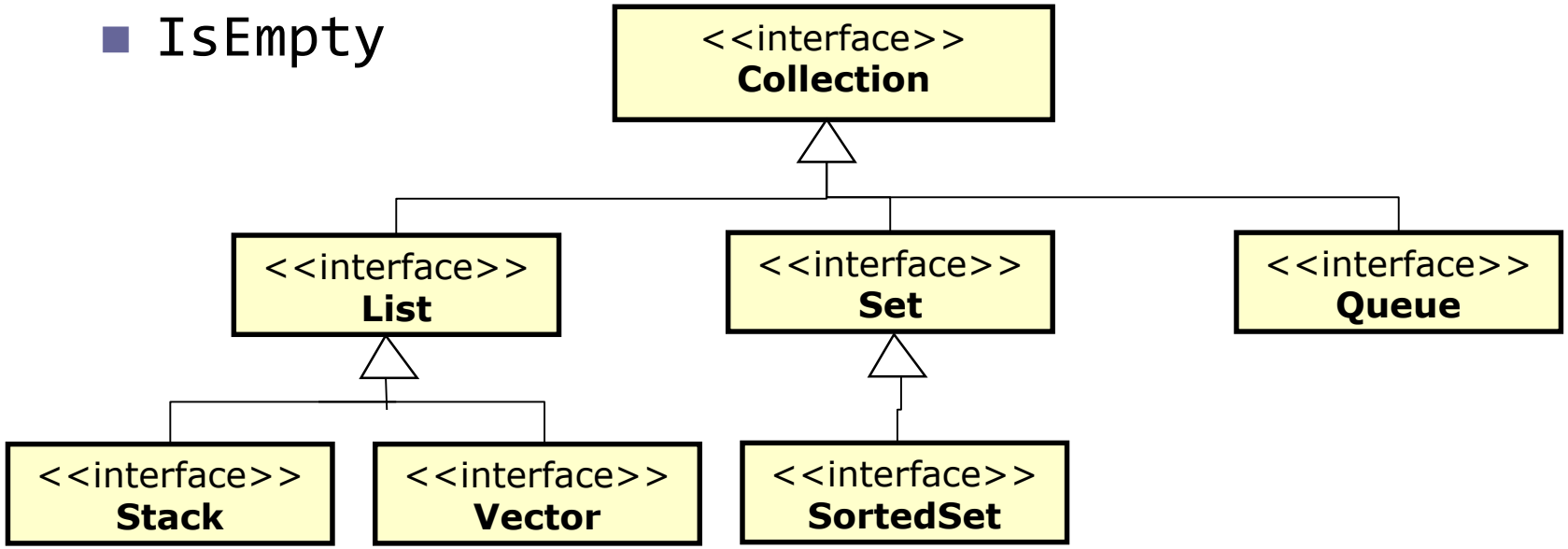
Colecciones

- Una **colección** es una secuencia o grupo de objetos
- Es una estructura de datos lineal: una dimensión
- En Java, una colección es una interfaz: **especificación**, **no implementación**
- Las estructuras de datos que vamos a ver en este capítulo son implementaciones específicas de colecciones



Collections

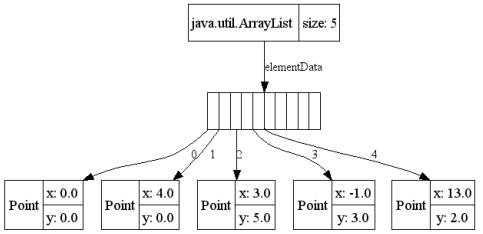
- Las operaciones básicas en una colección son:
  - `Add`
  - `Remove`
  - `Size`
  - `IsEmpty`



Listas

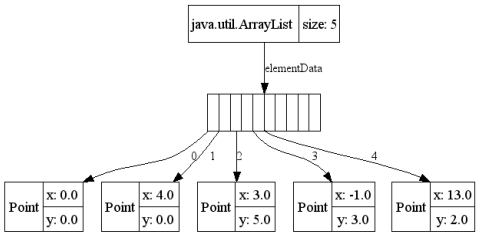
- Una **Lista** es una secuencia de elementos (una colección) donde el mismo elemento puede aparecer más de una vez
- **Array Lists** y **Linked Lists** son dos implementaciones de **List**
- Las operaciones básicas de una lista son:
  - `Add, Remove`
  - `Add(pos, obj)`
  - `Remove(pos)`
  - `Get(pos)` and `Set(pos, obj)`
  - `Size, IsEmpty`

Array Lists



- Las Array Lists son conocidas también como array dinámico, array de tamaño variable, array de longitud variable,…
- Ofrece los servicios de una Lista, implementada con un array
  - En Java, se denomina `ArrayList`
- Cada Array List
  - tiene un array
  - con una capacidad máxima
  - almacena el número de elementos

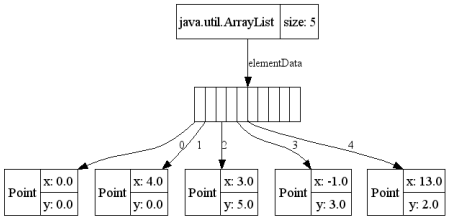
Array Lists




```
public class ArrayList {
  private final static int INITIAL_CAPACITY = 20;

  private Object[] elements;
  private int numberOfElements;

  public ArrayList(int capacity) {
    elements=new Object[capacity];
    numberOfElements = 0;
  }

  public ArrayList() {
    this(INITIAL_CAPACITY);
  }
```

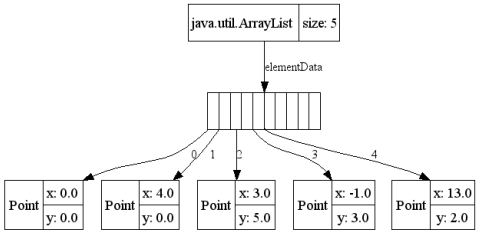
Un ArrayList vacío tiene un array de 20 elementos (de capacidad), siendo el numberOfElements == 0

Array Lists :: Size, Get, Set



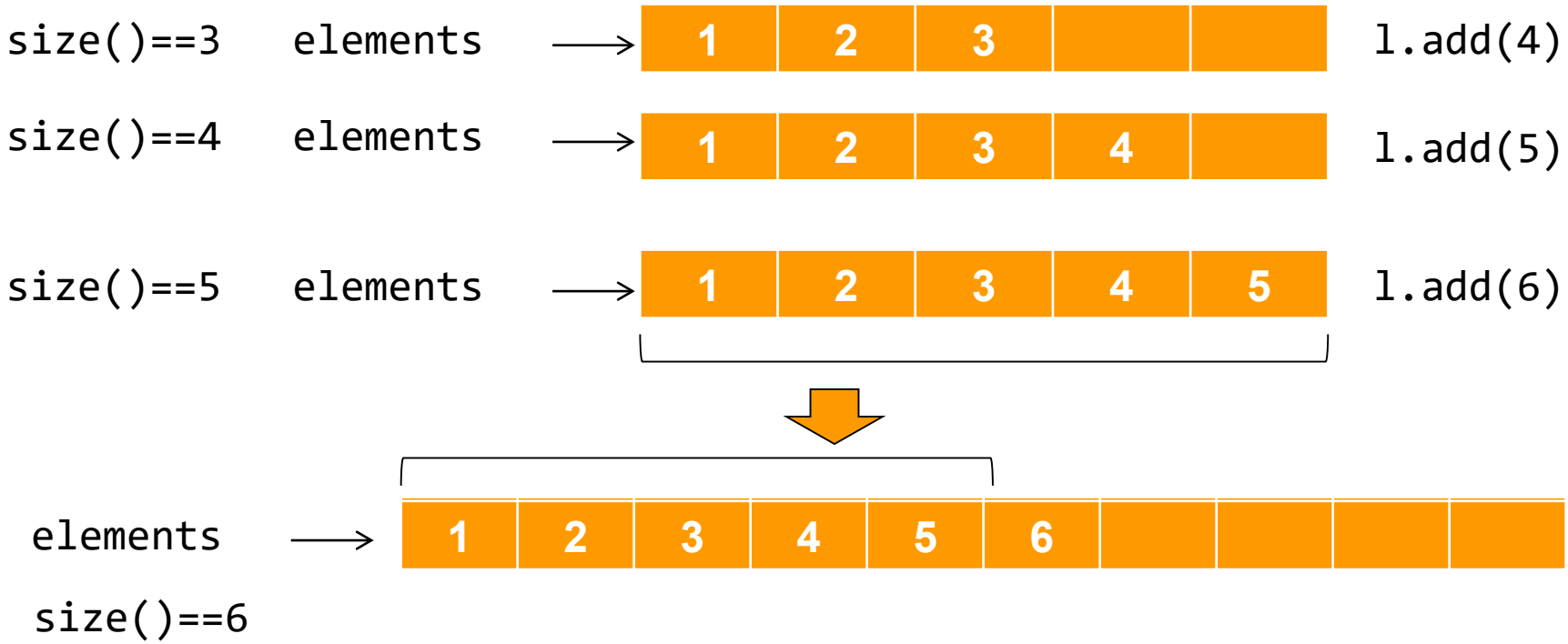

```
public int size() {
  return numberOfElements;
}

public Object get(int index) {
  return elements[index];
}

public void set(int index, Object element) {
  elements[index]=element;
}
```

Array Lists :: Add

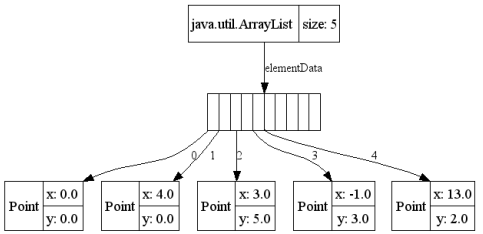


- Si se necesita añadir un elemento y el array está lleno …se crea un nuevo array y se mueven los elementos

```
private ArrayList<Object> l = new ArrayList<>(5);
```

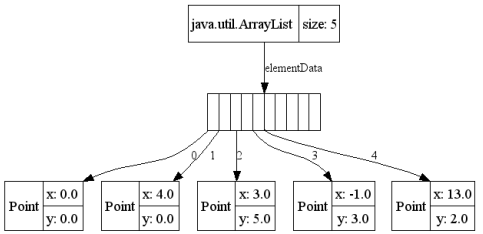


Array Lists :: Add



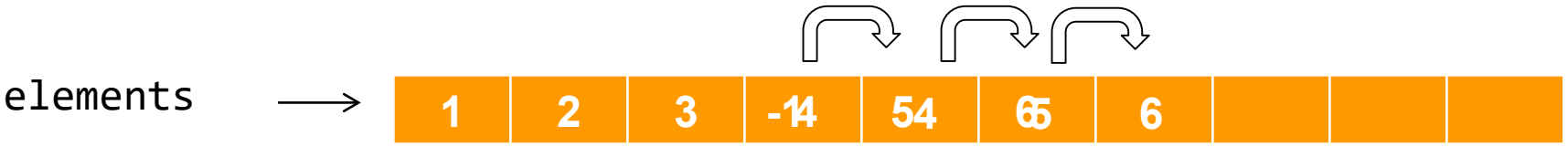

```
public void add(Object element) {
  if (size() >= elements.length)
    moreMemory(size()+1);
  elements[size()] = element;
  numberOfElements++;
}

private void moreMemory(int numElem) {
  if (numElem > elements.length) {
    Object[] aux = new Object[Math.max( numElem,
                                2*elements.length)];
    System.arraycopy(elements, 0, aux, 0,
                     elements.length);
    elements=aux;
  }
}
```

Array Lists :: Insertar



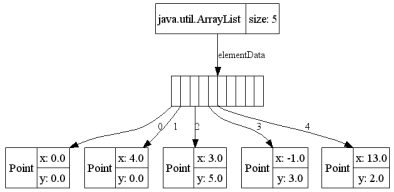
- **Añadir un elemento en una posición específica** (inserción) también puede necesitar la expansión de memoria
- Los elementos situados a la derecha de esa posición se mueven una posición en el array
- Se incrementa el número de elementos

```
l.add(3, -1)
```




```
size()==6
size()==7
```

Array Lists :: Add (index, element)




```
public void add(int index, Object element) {
  if (size() >= elements.length)
    moreMemory(size()+1);
  for(int i=size(); i>index; i--)
    elements[i] = elements[i-1];
  elements[index] = element;
  numberOfElements++;
}
```

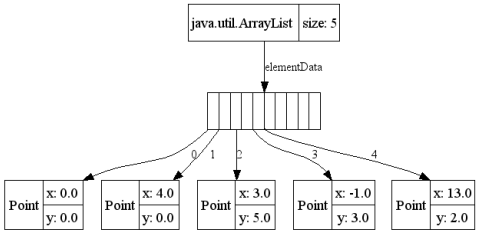
```
l.add(3, -1)
```

elements ⟶

1	2	3	-1	4	5	6			

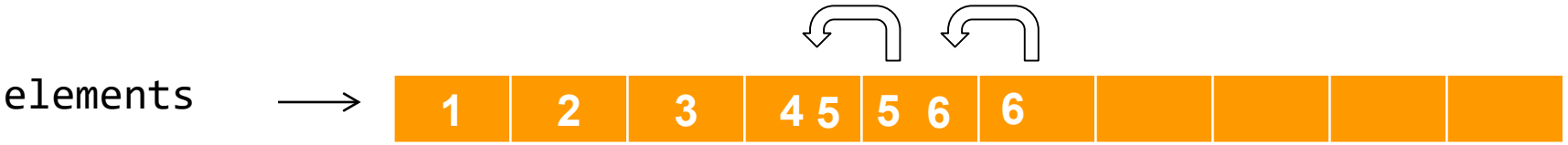
size()==7

Array Lists :: Remove



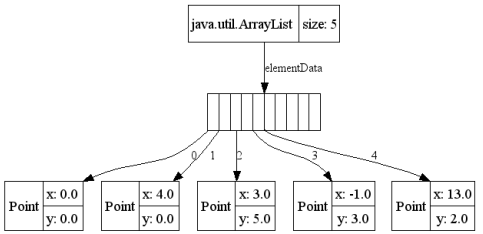
- Cuando se borra un elemento, también se desplazan los elementos situados a su derecha una posición hacia la izquierda
- El número de elementos se decrementa

```
l.remove(3)
```




```
size()==6
size()==5
```

Array Lists :: Remove



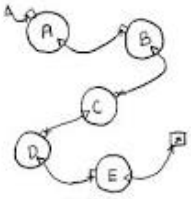

```
public Object remove(int index) {
  Object value = elements[index];
  for(int i=index; i<size()-1; i++)
    elements[i] = elements[i+1];
  numberOfElements--;
  return value;
}
```

l.remove(3)

elements ⟶

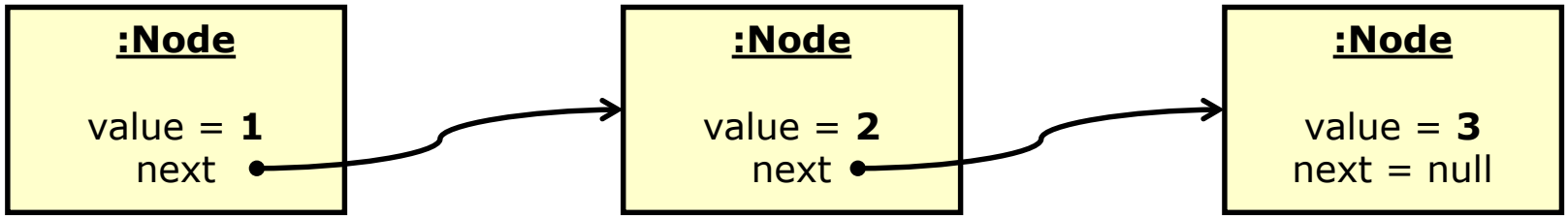
size()==5

Linked Lists

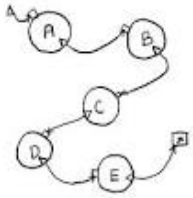


- Una lista enlazada **Linked List** es una Lista (una secuencia de elementos) formada por un grupo de nodos enlazados
- Es otra implementación de Lista
- Se crea un nodo para cada elemento de la colección
  - Hay tantos nodos como objetos en la colección
  - No se necesitan nodos adicionales
  - Están enlazados entre sí
  - No hay una longitud máxima

Una lista enlazada con los elementos 1, 2 y 3

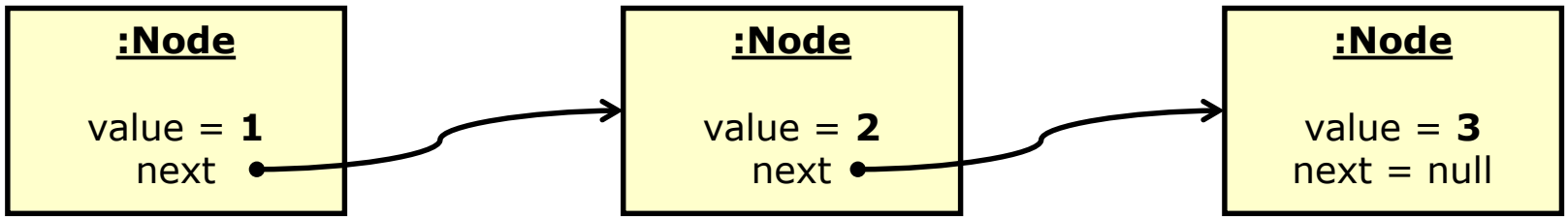


Los nodos en listas enlazadas

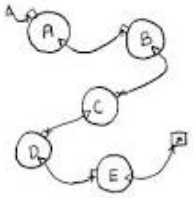


```
class Node {
  Object value;
  Node next;

  Node(Object value, Node next) {
      this.value = value;
      this.next = next;
  }
}
```

Una lista enlazada con los elementos 1, 2 y 3

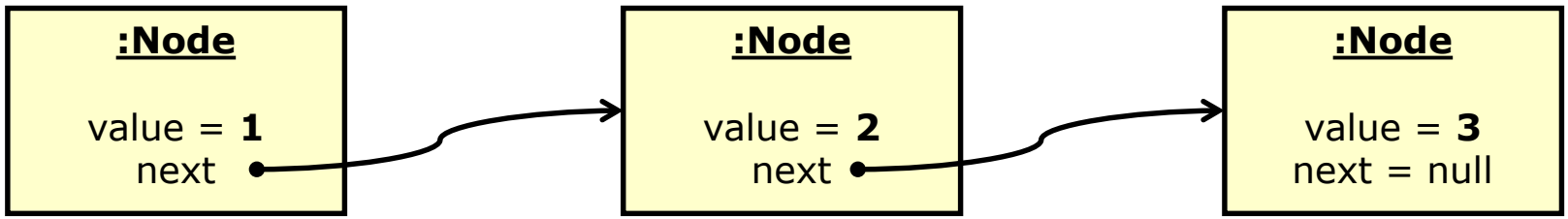


Linked Lists - Head

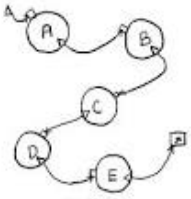


- Si queremos acceder a alguno de estos nodos, necesitamos una referencia que apunte a uno de ellos
- ¿A que nodo?

Una lista enlazada con los elementos 1, 2 y 3

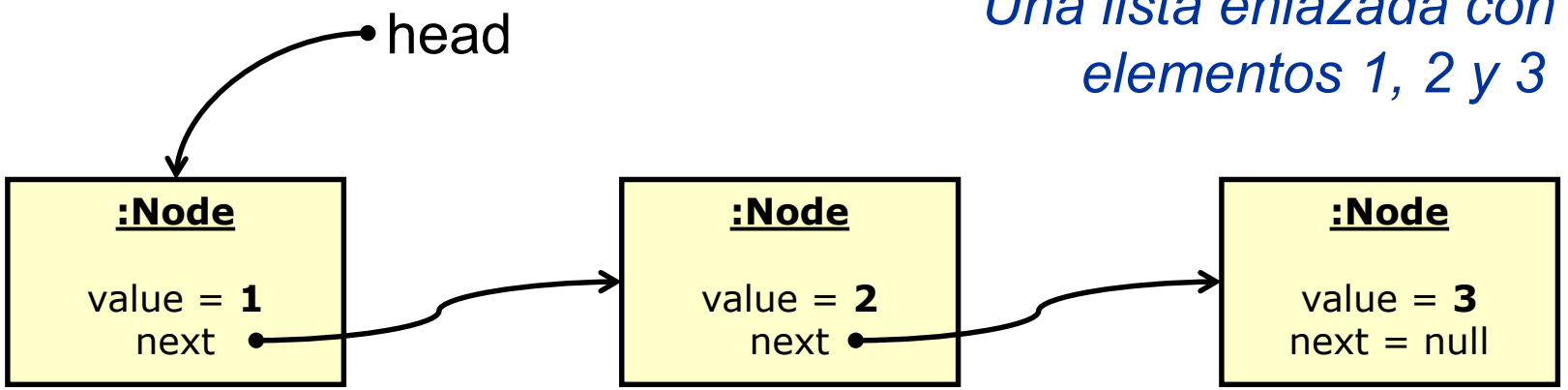


Linked Lists - Head

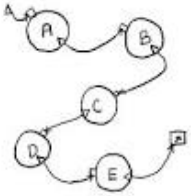


- A el primero
- Esta referencia, es la única que nos permite acceder al resto de los nodos, mediante su referencia `next` (de forma recursiva)
- Denominamos a esta referencia `head`

Una lista enlazada con los elementos 1, 2 y 3

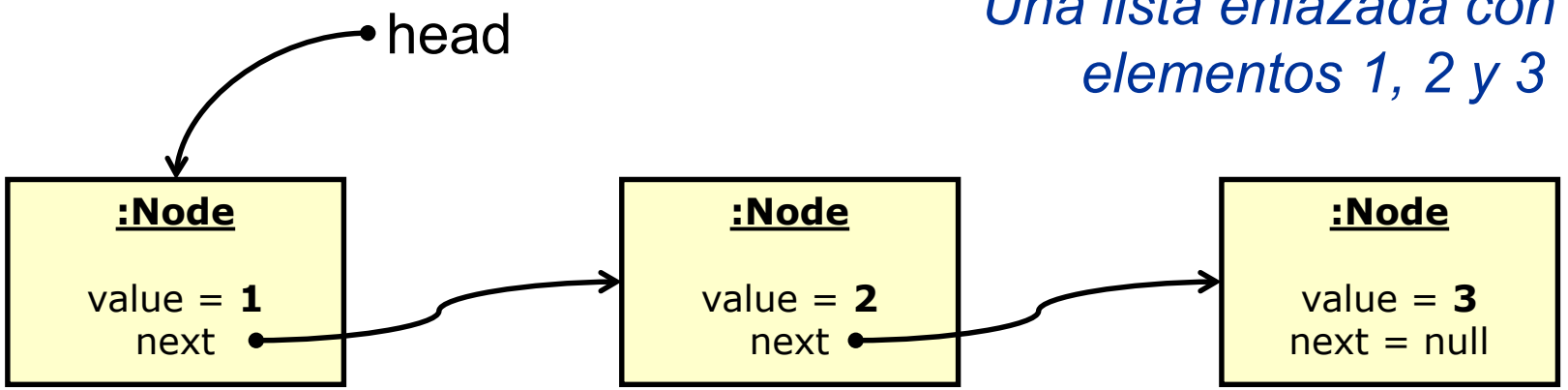


Linked Lists

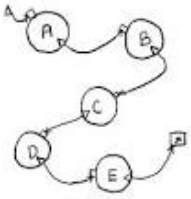


- La clase `Node` representa un elemento
- Es necesaria una clase `LinkedList` que ofrezca
  - Las operaciones `add`, `size`, `remove`...
  - Y los atributos `head` y `numberOfElements`

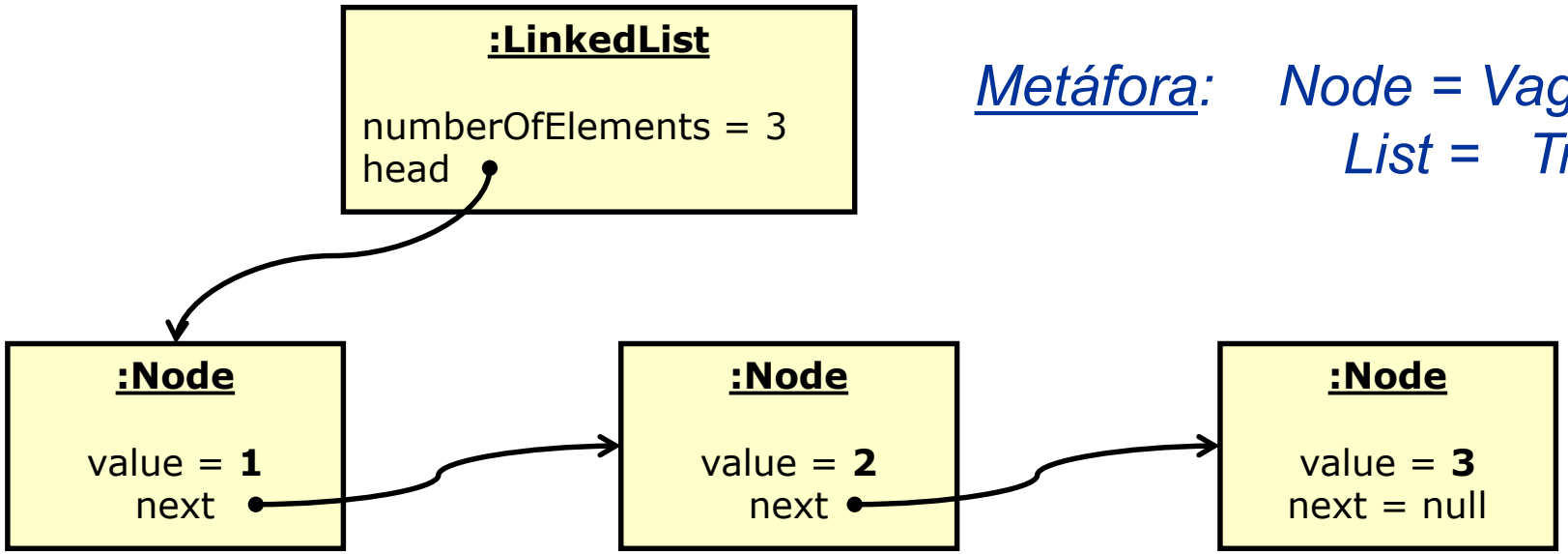
Una lista enlazada con los elementos 1, 2 y 3



Linked Lists

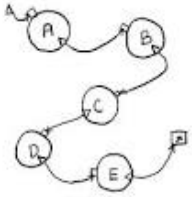


- La clase `Node` representa un elemento
- Es necesaria una clase `LinkedList` que ofrezca
  - Las operaciones `add`, `size`, `isEmpty`, `remove`...
  - Y los atributos `head` y `numberOfElements`



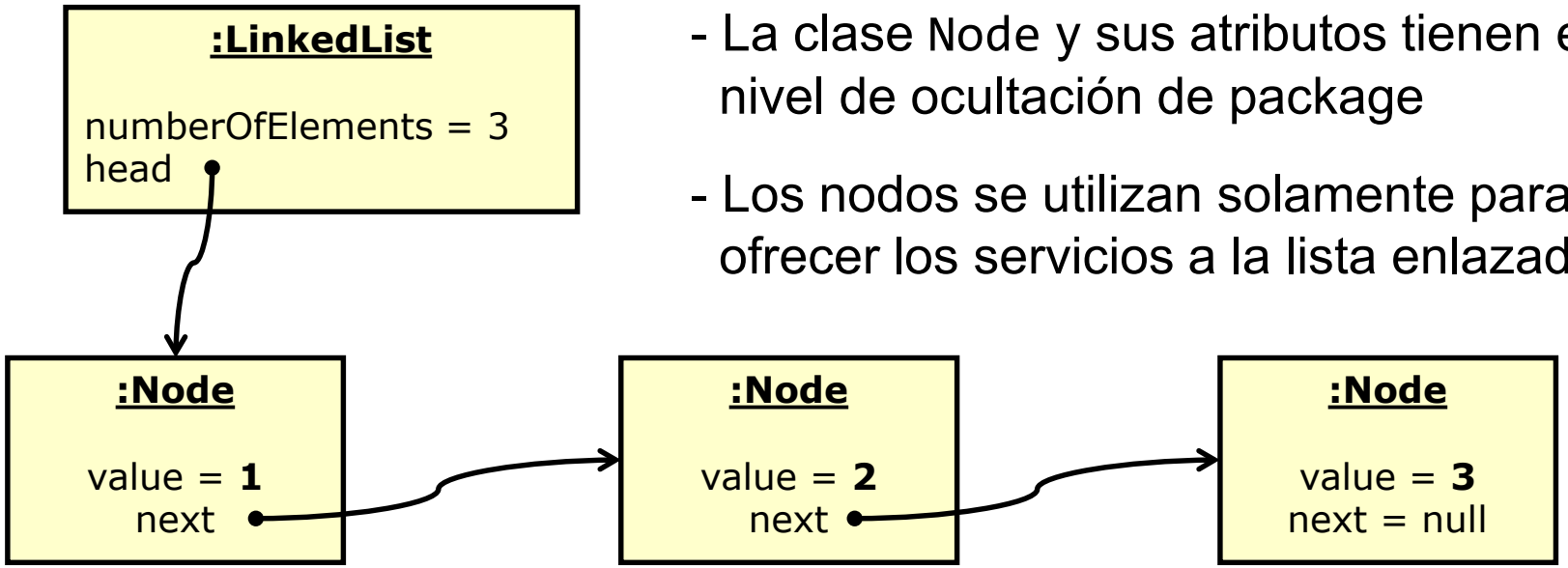
Metáfora: *Node = Vagón*
List = Tren

Linked Lists



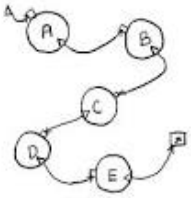
```
public class LinkedList {
   private Node head;
   private int numberOfElements;
   // ...
}
```

```
class Node {
   Object value;
   Node next;
   // ...
}
```



- La clase Node y sus atributos tienen el nivel de ocultación de package
- Los nodos se utilizan solamente para ofrecer los servicios a la lista enlazada

Linked Lists – Añadir al principio

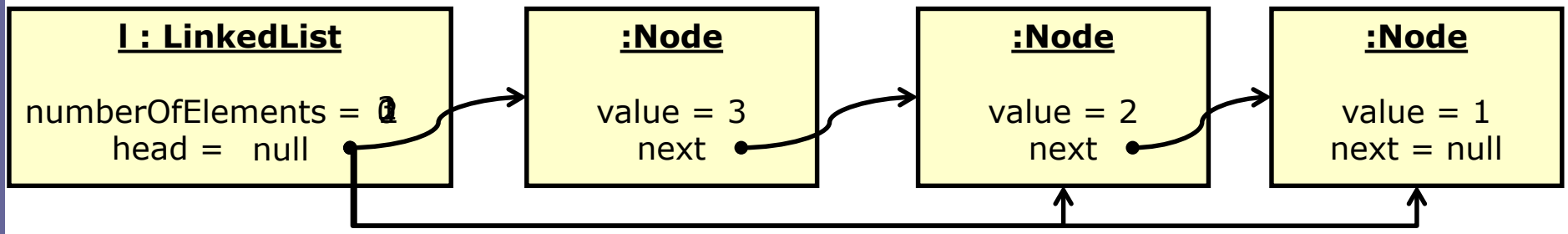


- Este método añade elementos al principio de la lista

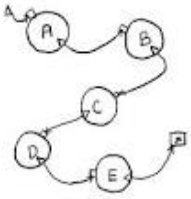
```
public void addFirst(Object value) {
  head = new Node(value, head);
  numberOfElements++;
}
```

```
LinkedList l = new LinkedList();
```

```
l.addFirst(1)
l.addFirst(2)
l.addFirst(3)
```

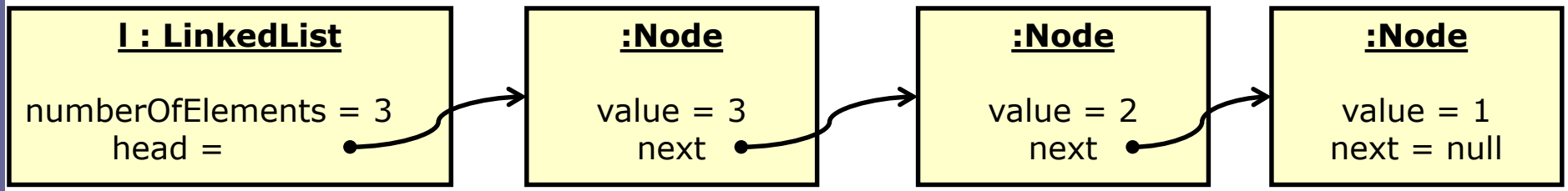


Linked Lists – Size - IsEmpty

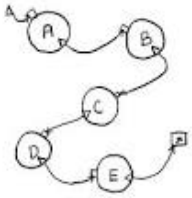


- **`NumberOfElements`** es un **atributo derivado**: se podría conocer contando los nodos
  - Se ha añadido para mejorar el rendimiento en tiempo de ejecución

```
public int size() {
  return numberOfElements;
}
public boolean isEmpty() {
  return size() == 0;
}
```

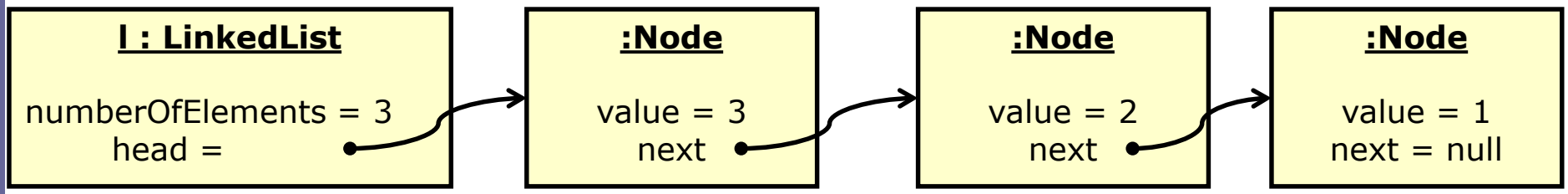


Linked Lists - Get y Set

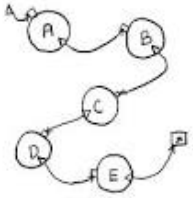


- Para las operaciones get y set sobre un elemento de la lista, buscamos el nodo con el índice especificado
  - Este comportamiento se realiza mediante el método privado `getNode`

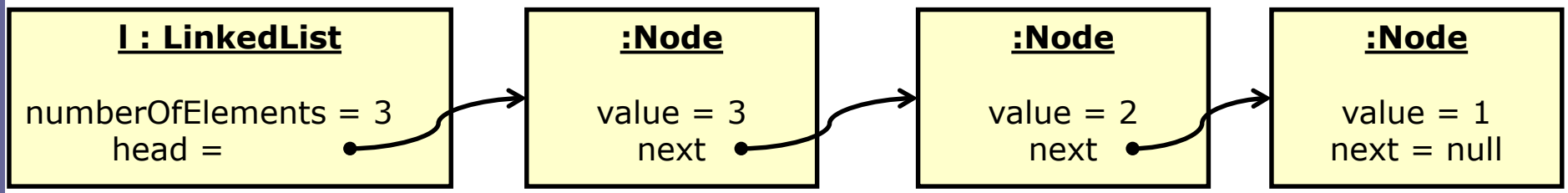
```
public Object get(int index) {
  return getNode(index).value;
}
public void set(int index, Object value)  {
  getNode(index).value = value;
}
```



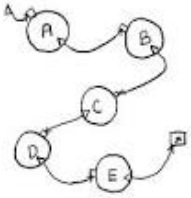
Linked Lists - Get y Set



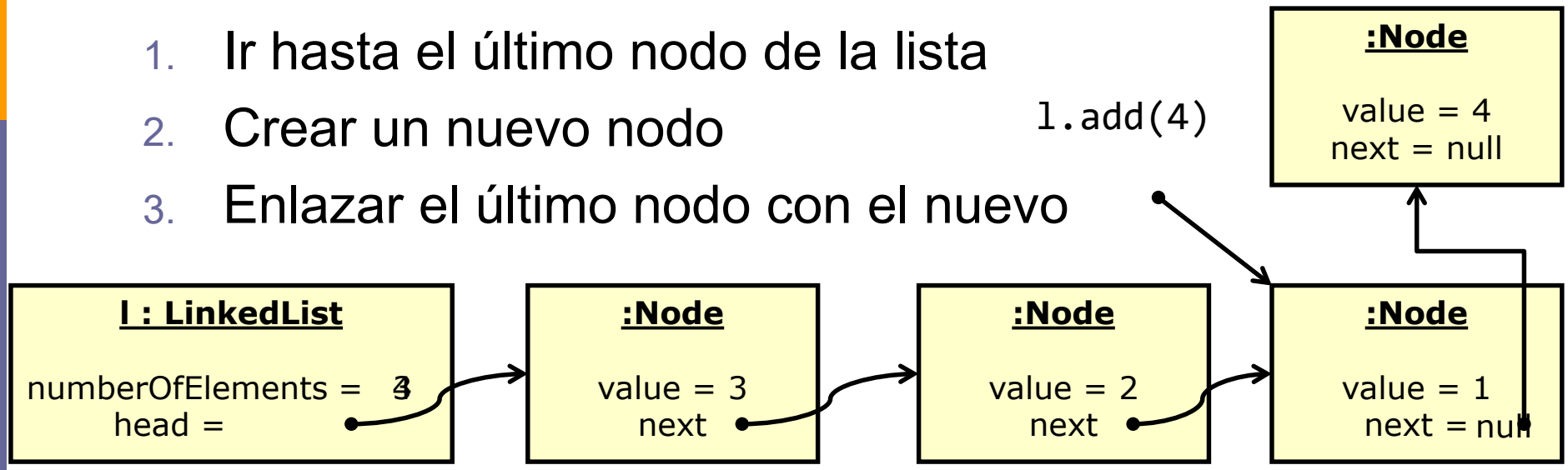
```
private Node getNode(int index) {
  int position = 0;
  Node node = head;
  while (position < index)  {
    node = node.next;
    position++;
  }
  return node;
}
```



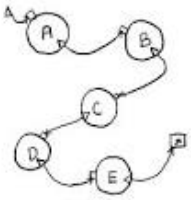
Linked Lists – Añadir al final



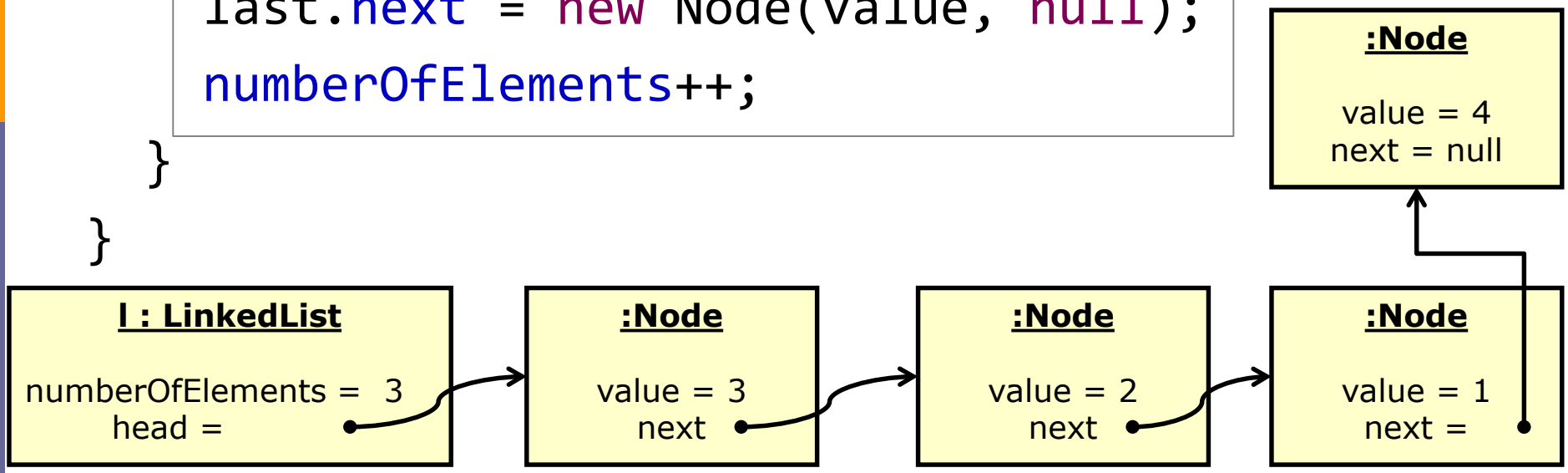
- El método `AddFirst` añade los elementos en orden inverso
- ¿Como se podrían añadir elementos en el mismo orden?
- Añadiendo los elementos al final de la lista
  1. Ir hasta el último nodo de la lista
  2. Crear un nuevo nodo
  3. Enlazar el último nodo con el nuevo



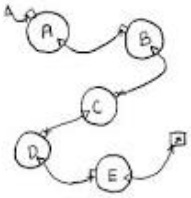
Linked Lists – Añadir al final



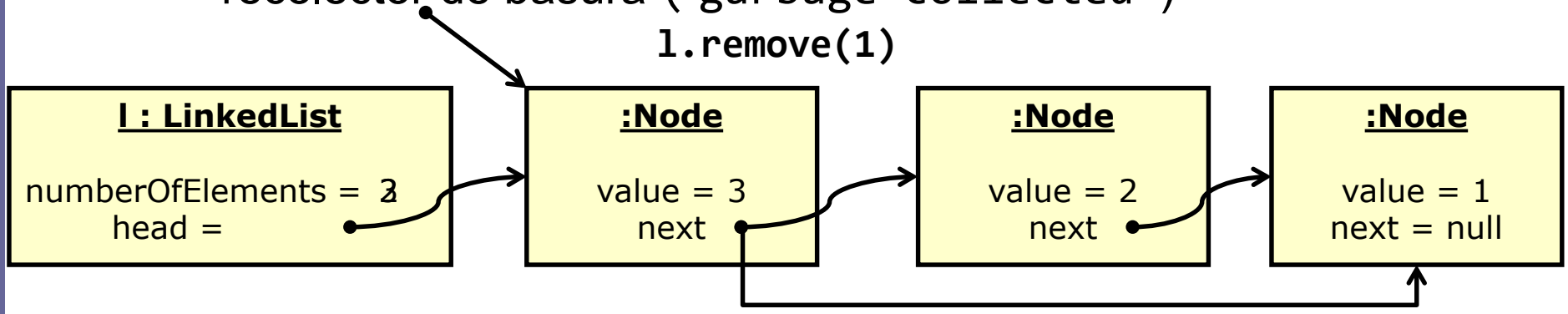
```
public void add(Object value) {
  if (isEmpty())
    addFirst(value);
  else {
    Node last = getNode(size()-1);
    last.next = new Node(value, null);
    numberOfElements++;
  }
}
```



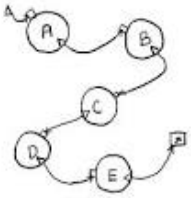
Linked Lists - Borrar



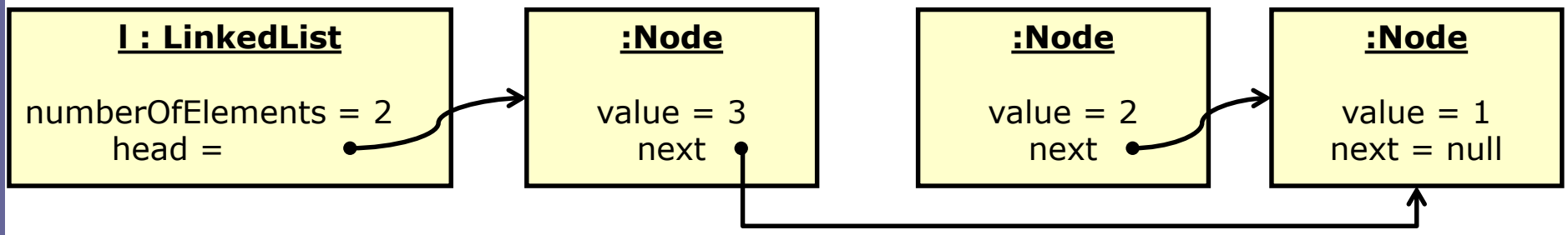
- Borrar un elemento que ocupa una determinada posición (identificada con `index`) se realiza de la siguiente manera
  1. Situarse en el nodo anterior (`index-1`)
  2. Cambiar la referencia `next` a la referencia `next` del nodo (`index+1`)
  3. Decrementar el número de elementos
  4. El elemento de la posición `index` es liberado por el recolector de basura (`garbage collected`)



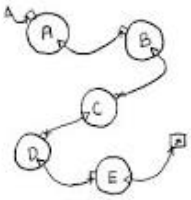
Linked Lists - Remove



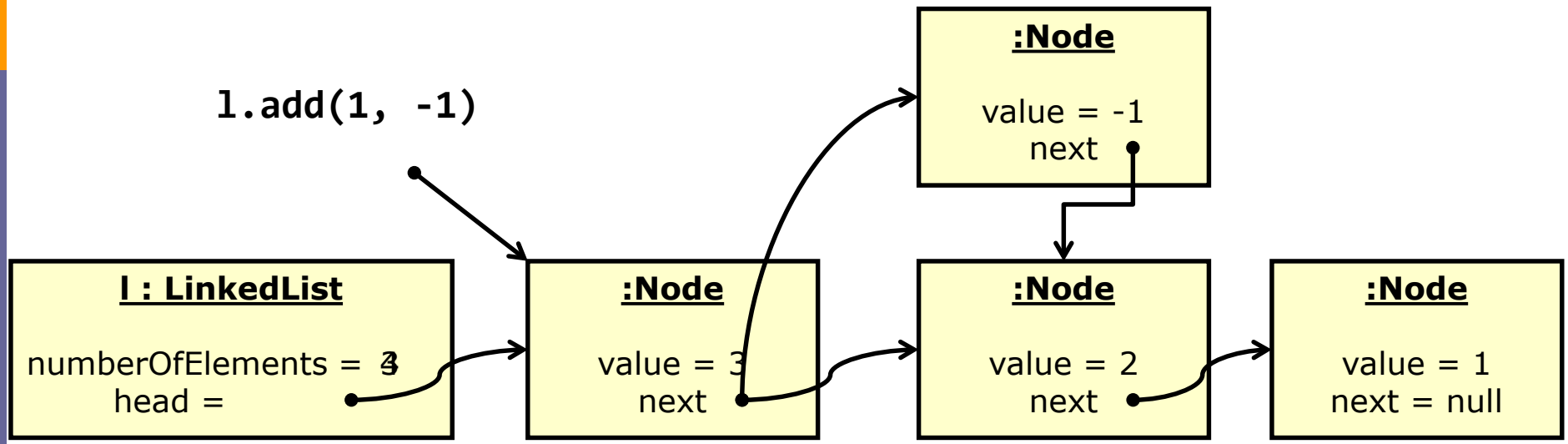
```
public Object remove(int index) {
  if (isEmpty()) return null;
  Object value;
  if (index == 0) {
    value = head.value;
    head = head.next;
  } else {
    Node previous = getNode(index - 1);
    value = previous.next.value;
    previous.next = previous.next.next;
  }
  numberOfElements--;
  return value;  }
```



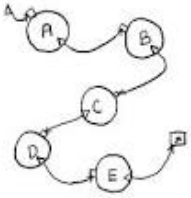
Linked Lists – Insertar un elemento



- Insertar un elemento en la posición indicada por `index`
  1. Situarse en el nodo anterior (`index-1`)
  2. Crear un nuevo nodo con `next` = nodo con posición `index` y el apropiado valor en `value`
  3. Ahora el nodo que sigue al nodo anterior es el nuevo
  4. Incrementar el número de elemntos

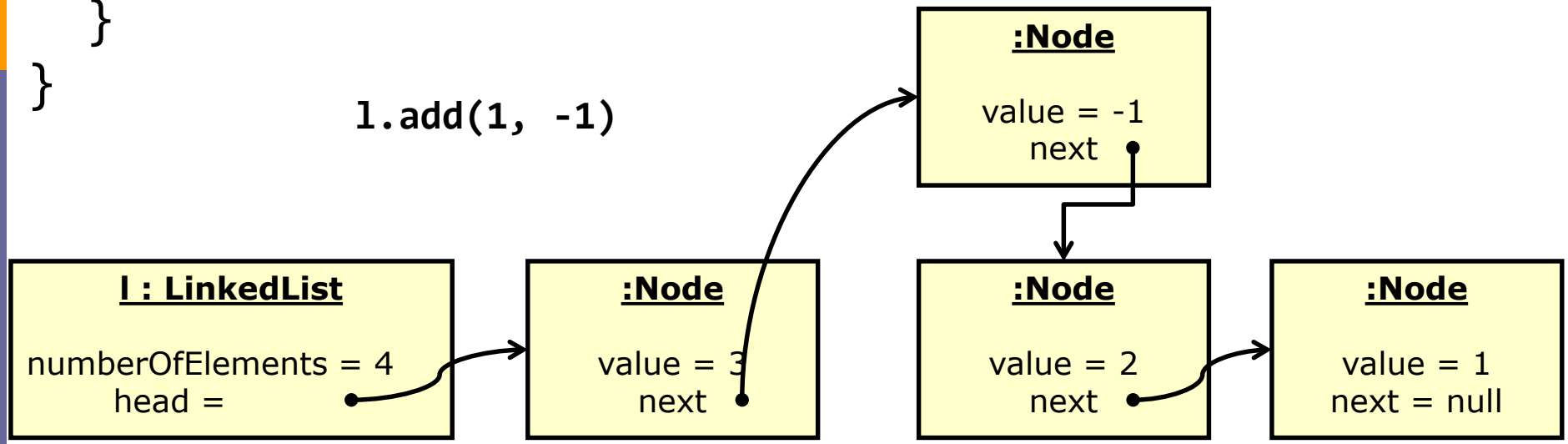


Linked Lists- Insertar un elemento

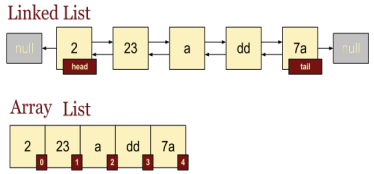


```
public void add(int index, Object value) {
  if (index==0)
    addFirst(value);
  else {
    Node previous = getNode(index-1);
    previous.next = new Node(value, previous.next);
    numberOfElements++;
  }
}
```

l.add(1, -1)



Array vs. Linked Lists



- Si las dos ofrecen la misma interface (`List`), ¿cuando elegimos una o la otra?
- Rendimiento en ejecución:
  - `Get` / `Set` en `ArrayList` no dependen del número de elementos (sí dependen en `LinkedList`)
  - `Remove` / `Insert` / `Add` en `LinkedList` no dependen del número de elementos (sí dependen en `ArrayList`)
- Por lo tanto
  - Usar **`ArrayList`** cuando se usen de forma iterativa las operaciones `get` y `set`
  - Usar **`LinkedList`** cuando el tamaño de la colección varíe mucho (p.e. Pilas (Stacks) y Colas (Queues))

Pilas (Stacks)

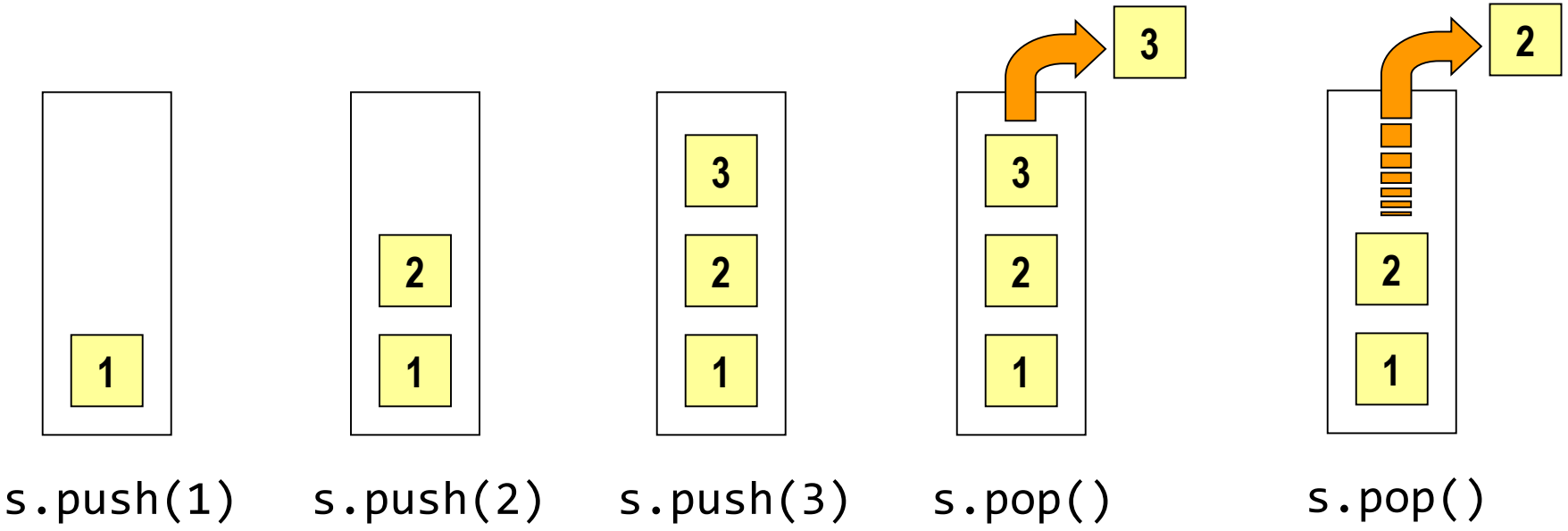
- Una pila (**Stack)** es una colección (secuencia linear) donde los elementos pueden ser añadidos o extraídos por un único extremo (denominado comúnmente top)
  - Los elementos no pueden ser insertados (añadidos en una posición específica)
  - Los elementos no pueden ser eliminados de una posición específica
- Una Pila posee un tratamiento **LIFO**: **L**ast **I**n, **F**irst **O**ut

Operaciones

- Las operaciones de una pila son:
  - Push: introduce un elemento en la parte superior de la pila
  - Pop: Elimina y devuelve el objeto de la parte superior (top) de la pila
  - Peek: Devuelve el elemento de la parte superior (top) de la pila sin eliminarlo
  - IsEmpty: Indica si la pila está vacía o no
  - Size: Devuelve el número de elementos de la pila

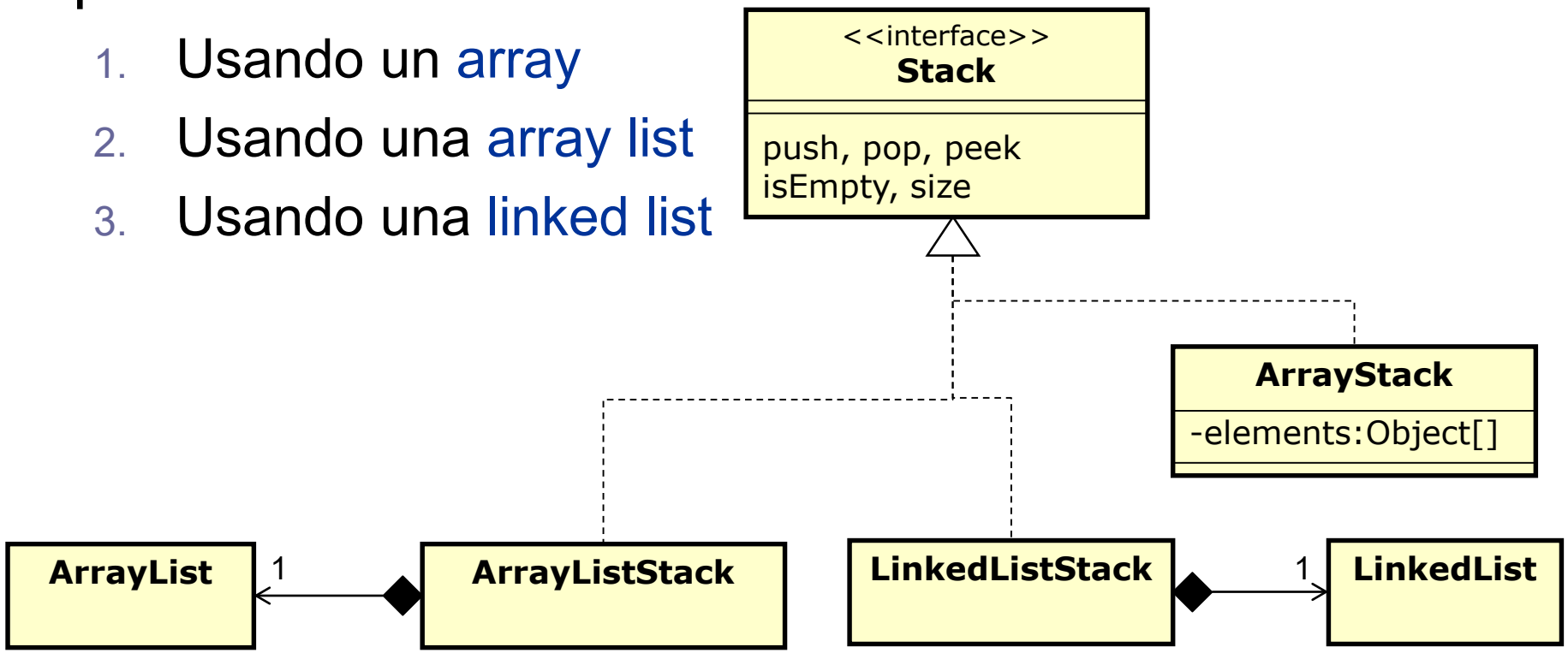
Push y Pop

- Este es un escenario de ejemplo del uso de una pila con valores enteros



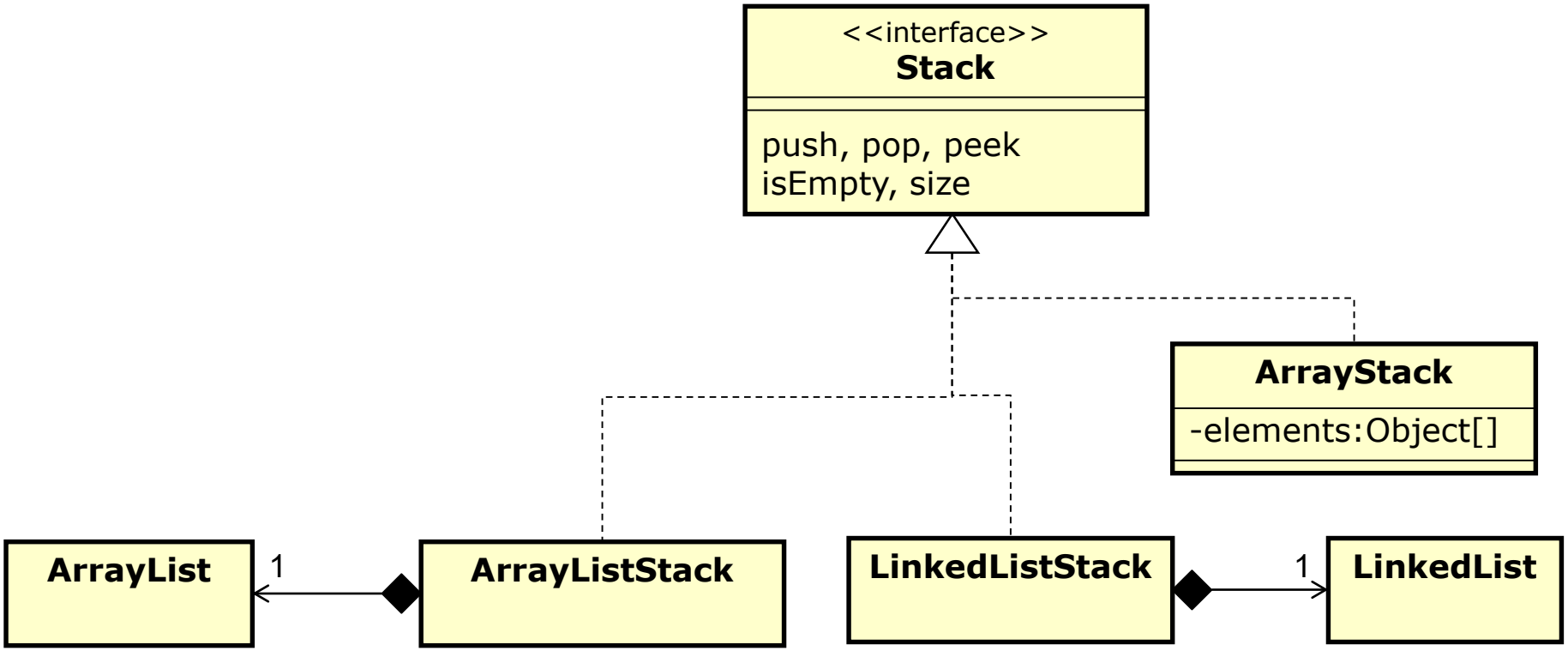
Implementación de una pila

- Las implementaciones más comunes de las pilas son:
  1. Usando un array
  2. Usando una array list
  3. Usando una linked list



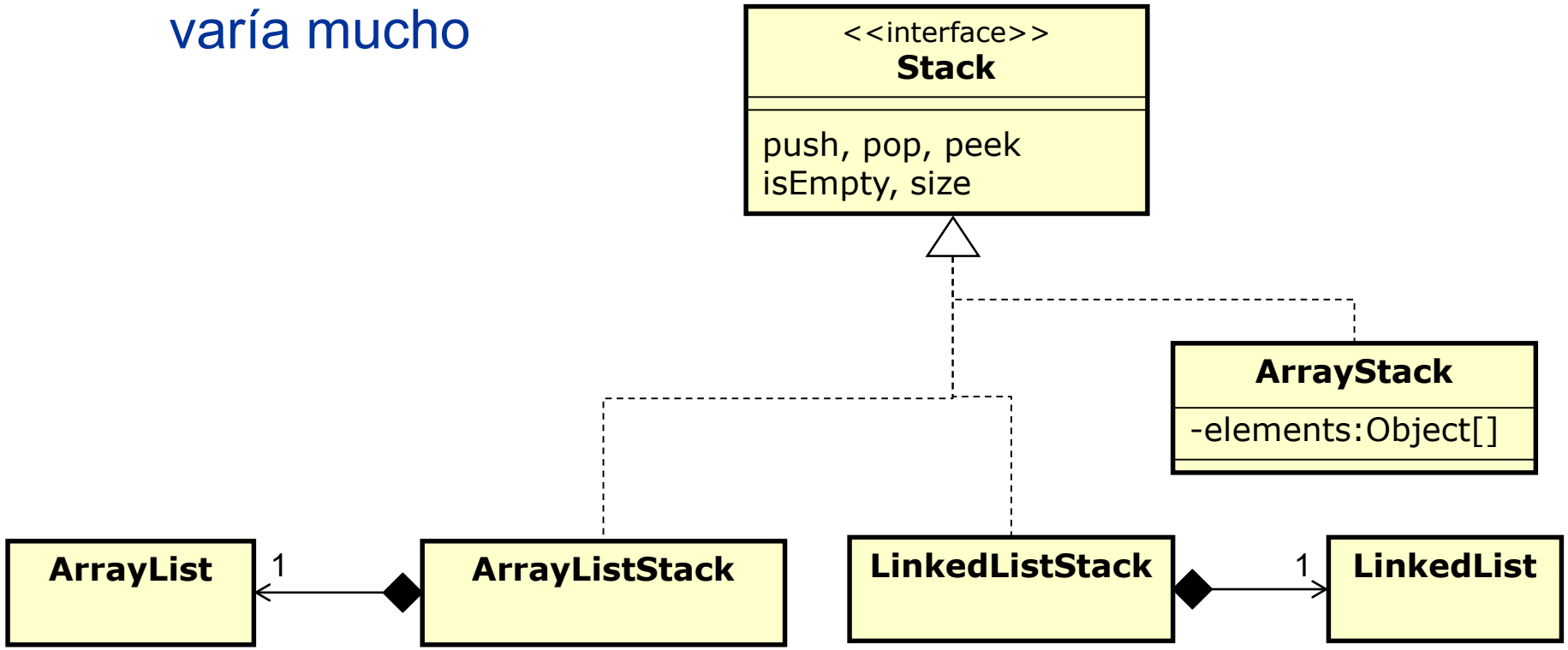
Implementación de una pila

- ¿Cuál es el más apropiado?

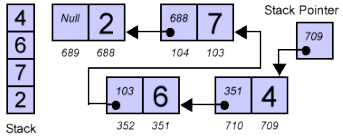


Implementación de una pila

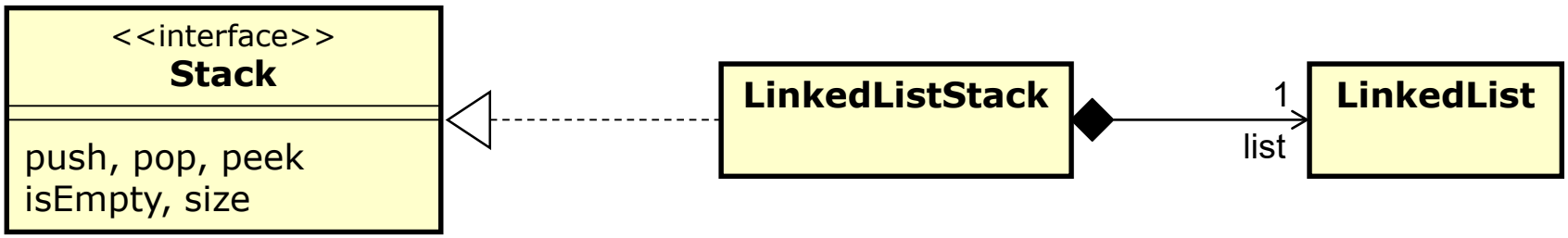
- Recordando
  - Usar **`ArrayList`** cuando se usen de forma iterativa las operaciones `get` y `set`
  - Usar **`LinkedList`** cuando el tamaño de la colección varía mucho



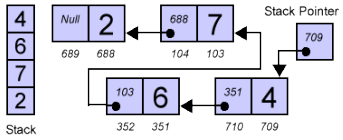
Pila con Linked List



- Usando una lista enlazada
  - Qué elemento sería la parte superior (top) de la pila ¿el primero o el último?
  - ¿Cómo podemos implementar método push?
  - ¿Cómo podemos implementar el método pop?
  - ¿Cómo podemos implementar el método peek?
  - ¿Cómo podemos implementar los métodos `isEmpty` y `size`?

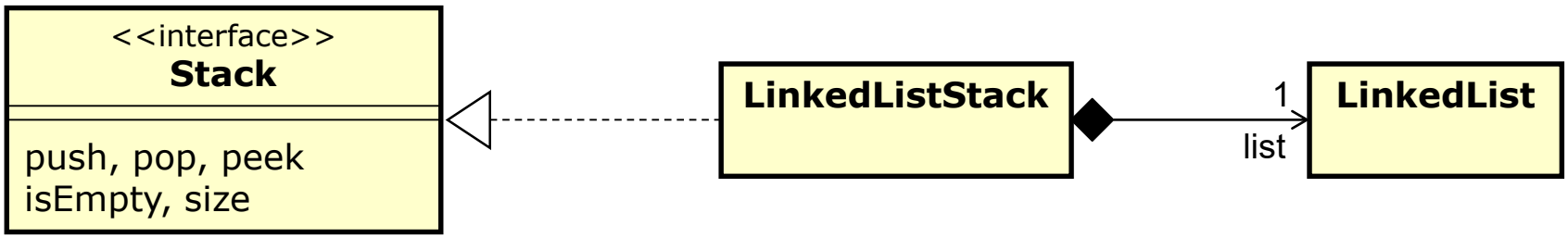


Pila con Linked List

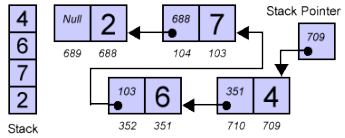


- Qué elemento sería la parte superior (top) de la pila ¿el primero o el último? El primero
- ¿Cómo podemos implementar método push?
- `list.addFirst(elemento);`
- o también `list.add(0, element);`
- ¿Cómo podemos implementar el método pop?

```
if (isEmpty())
  return null;
return list.remove(0);
```



Pila con Linked List



- ¿Cómo podemos implementar el método peek?

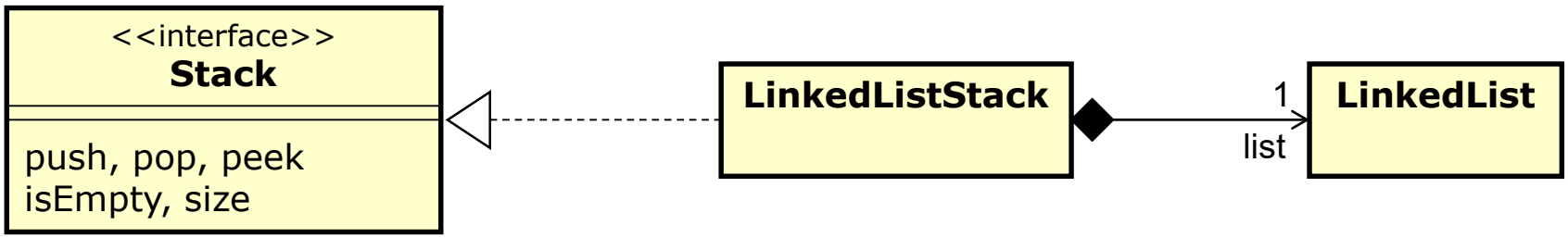
```
if (isEmpty()) return null;
return list.get(0);
```

- ¿Cómo podemos implementar el método size?

```
public int size() {
  return list.size();}
```

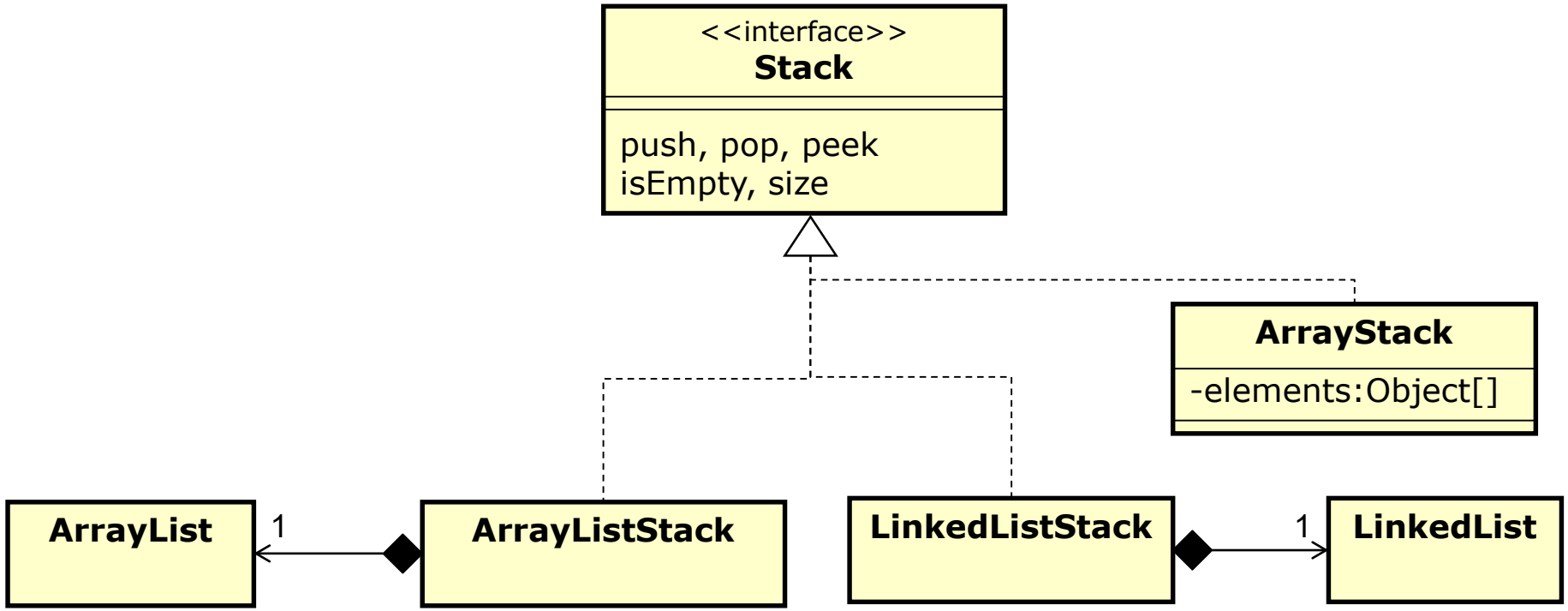
- ¿Cómo podemos implementar el método isEmpty?

```
public boolean isEmpty() {
  return size() == 0;}
```



Ejercicio Opcional

- Como **ejercicio opcional**, realiza tres implementaciones diferentes de la pila
  - Usando una linked list, una array list y un array



Colas (Queues)

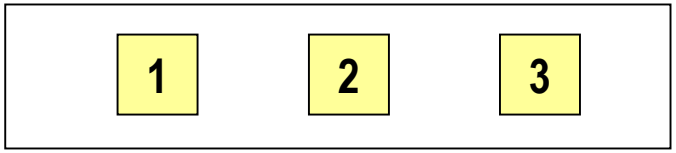
- Una Cola (**Queue)** es una colección (secuencia lineral) donde los elementos se añaden por la parte posterior y se extraen por la parte frontal
  - Los elementos no pueden ser insertados (añadidos en una posición específica)
  - No se pueden eliminar los elementos de una posición específica
- Las Colas tienen un tratamiento **FIFO**: **F**irst **I**n, **F**irst **O**ut
- Son utilizadas para almacenar entidades que son procesadas más tarde

Operaciones

- Las operaciones de una Cola son :
  - Enqueue: Añade un objeto por la parte posterior (final) de la cola
  - Dequeue: Elimina y devuelve un objeto por la parte frontal (inicio) de la cola
  - Peek: devuelve un objeto por la parte frontal (inicio) de la cola sin eliminarlo
  - IsEmpty: Indica si la cola está vacía o no
  - Size: Devuelve el número de elementos de la cola

Enqueue y Dequeue

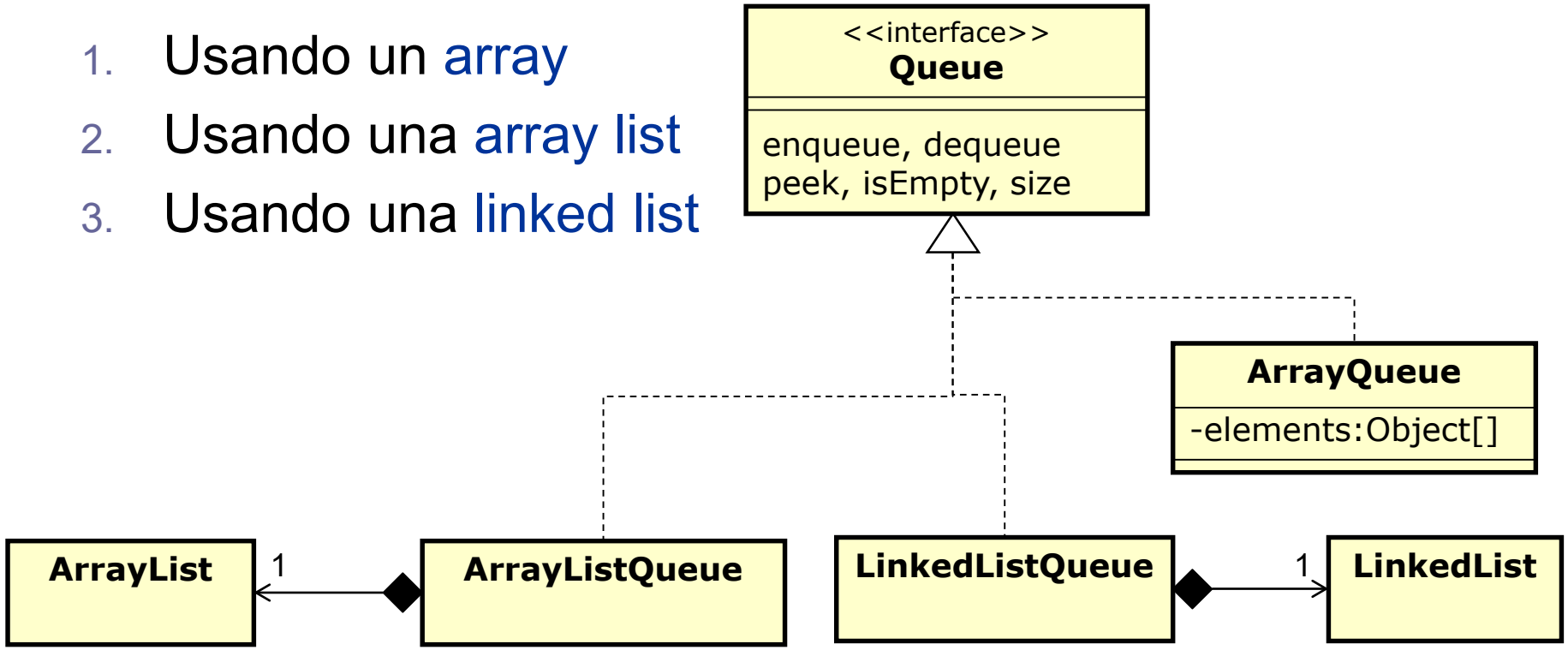
- Este es un escenario de ejemplo del uso de una cola con valores enteros




```
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
queue.dequeue()
queue.dequeue()
queue.dequeue()
```

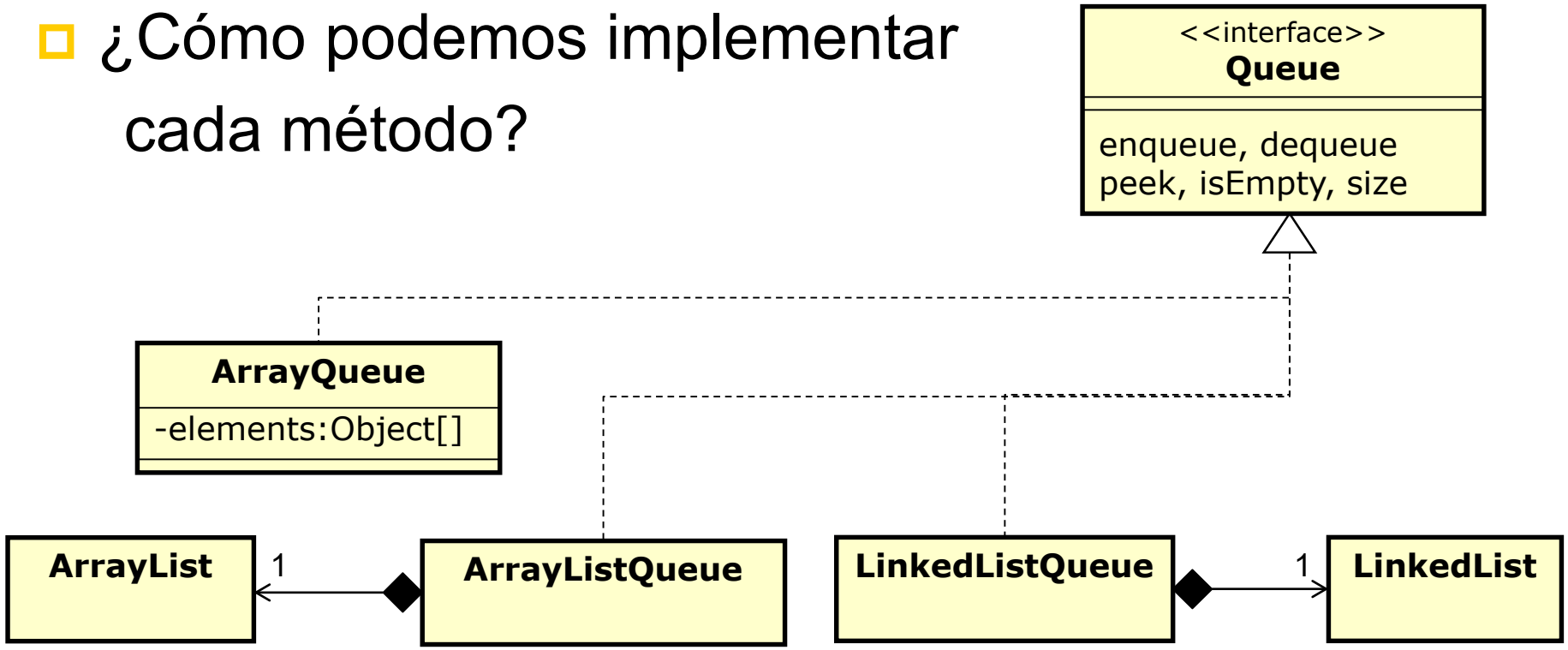
Implementación de una Cola

- Las implementaciones más comunes de las colas son:
  1. Usando un array
  2. Usando una array list
  3. Usando una linked list

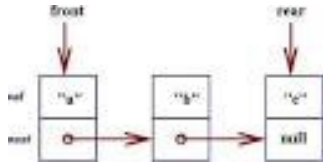


Implementación de una Cola

- ¿Cual es la más adecuada?
- ¿Qué extremo podría ser la parte posterior?
- ¿Qué extremo podría ser la parte frontal?
- ¿Cómo podemos implementar cada método?



Cola con Linked List

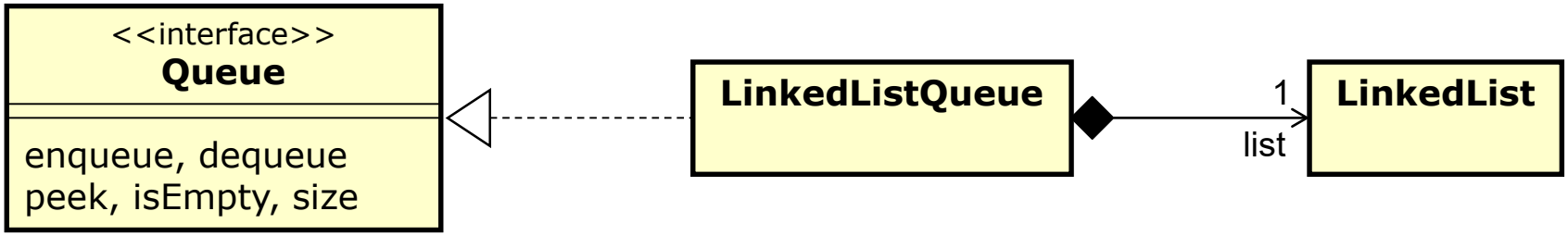


- Al igual que con las pilas, linked list es la mejor opción
- Parte posterior = último elemento (podría ser el primero)
- Parte frontal = primer elemento (podría ser el último)
- El método enqueue:

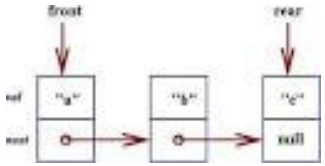
```
list.add(element);
```

- El método dequeue:

```
if (isEmpty())
  return null;
return list.remove(0);
```



Cola con Linked List



- El método `peek`:

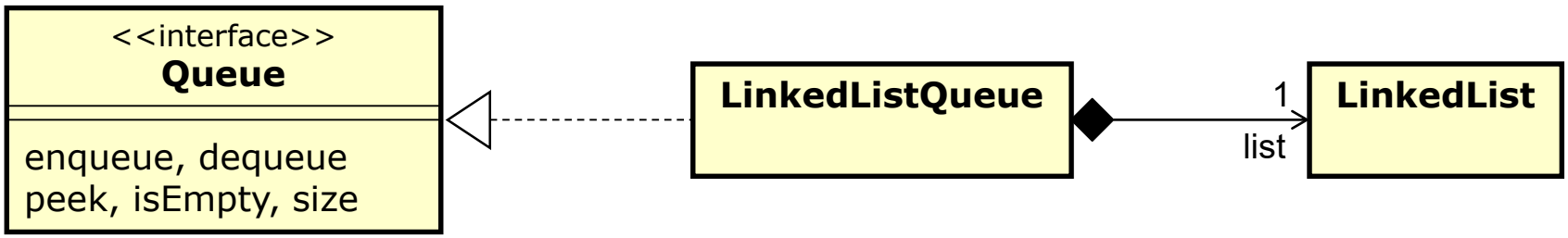
```
if (isEmpty())
   return null;
return list.get(0);
```

- El método `size`:

```
return list.size();
```

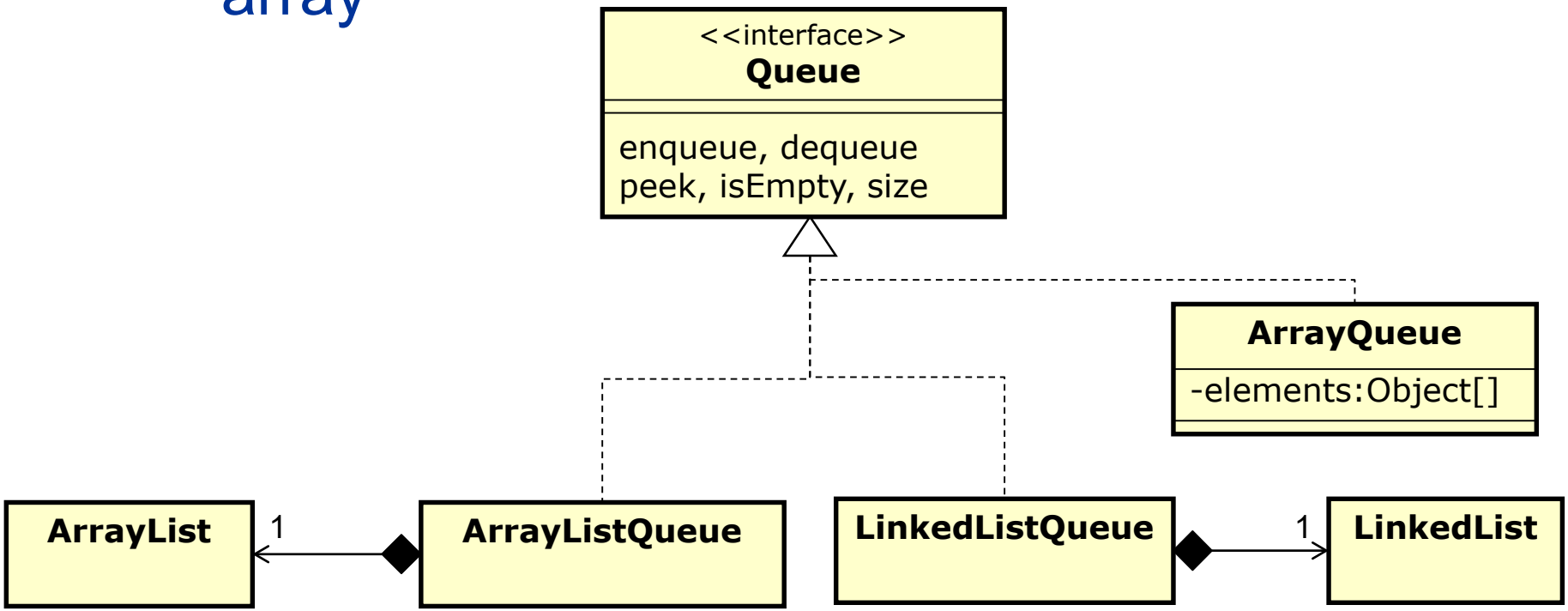
- El método `isEmpty`:

```
return size() == 0;
```



Ejercicio optional

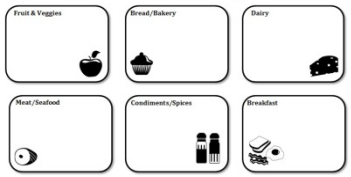
- Como **ejercicio opcional,** realiza tres implementaciones diferentes de la cola
  - Usando una linked list, una array list y un array



Revisión

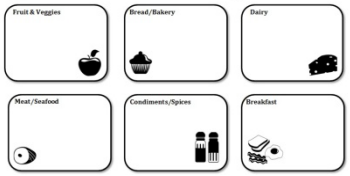
- Una **colección** es una secuencia o grupo de objetos
  - Es una estructura de datos lineal: una dimensión
- Una **Lista** es una colección en la que los elementos pueden estar repetidos
- **Array Lists** y **Linked Lists** son dos implementaciones diferentes de Lista
  - Usar array lists cuando el acceso a los elementos es una operación iterativa
  - Use linked list cuando el tamaño de la colección varía mucho
- Las **Pilas** son colecciones LIFO
- Las **Colas** son colecciones FIFO

Apéndice: ¿Usar colecciones genéricas o no?



- Todos estos tipos de colecciones están presentes en la API de Java.
- Se pueden declarar genéricas (1) o no genéricas (2). Ambas son válidas:
  1. `ArrayList arr = new ArrayList(…);` **(almacena Object)**
  2. `ArrayList<Item> arr = new ArrayList<Item>(…);`
- Usando (1):
  - Se pueden almacenar toda clase de objetos (Objects) en la misma colección, pero…
  - Se desconoce el tipo cuando se extrae un elemento
  - Es necesario hacer un cast y hay que adivinar el tipo en tiempo de ejecución → puede provocar un error.
  - Se puede preguntar con `instanceof`, pero…
    - Es más complejo
    - … no es una solución acertada.

Apéndice: ¿Usar colecciones genéricas o no?



- Usando (2):
  - Se pueden coleccionar `Items` y subclases de `Item` en la misma colección.
  - Se sabe de qué tipo será cuando se extrae un elemento.
  - Se tienen más métodos para invocar cuando se extrae el elemento.
  - ¿Es necesario hacer un cast si se necesita saber el tipo exacto del elemento? **SI**
  - Pero `Item` es mucho más específico que `Object`
    - Más métodos que `equals()`, `toString()` and `hashCode()`…
  - Al usar los métodos de `Item` se asegura que el algoritmo funciona para todas las subclases de Item (presentes y futuras)
    - Es mejor no tener que hacer un cast en tiempo de ejecución.