

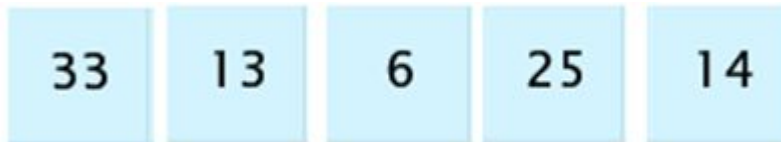
# Listas

## Algunas operaciones

- Una Lista es una estructura de datos en donde los objetos están ubicados en forma secuencial. A diferencia de la **Pila** y la **Cola**, en una **Lista** se puede “agregar” y “eliminar” en **cualquier** posición.
- Puede estar implementada a través de:
  - una estructura estática (arreglo)
  - una estructura dinámica (usando nodos enlazados)
- Puede estar ordenada o no:
  - Si está ordenada, los elementos se ubican siguiendo el orden de las claves almacenadas en la lista.



- Si está desordenada, los elementos pueden aparecer en cualquier orden.



# Listas

## Algunas operaciones

Por simplicidad comenzaremos con elementos de tipo enteros.

***elemento(int pos)***: retorna el elemento de la posición indicada

***incluye(Integer elem)***: retorna true si ***elem*** está en la lista, false en caso contrario

***agregarInicio(Integer elem)***: agrega al inicio de la lista

***agregarFinal(Integer elem)***: agrega al final de la lista

***agregarEn(Integer elem, int pos)***: agrega el elemento ***elem*** en la posición ***pos***

***eliminarEn(int pos)***: elimina el elemento de la posición ***pos***

***eliminar(Integer elem)***: elimina, si existe, el elemento ***elem***

***esVacía()***: retorna true si la lista está vacía, false en caso contrario

***tamaño()***: retorna la cantidad de elementos de la lista

***comenzar()***: se prepara para iterar los elementos de la lista

***proximo()***: retorna el elemento y avanza al próximo elemento de la lista.

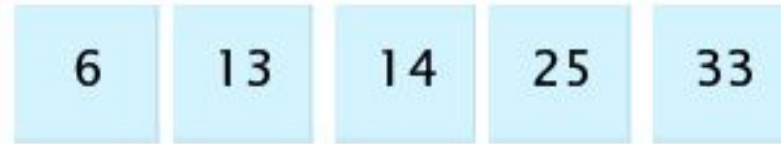
***fin()***: determina si llegó o no al final de la lista, retorna true si no hay mas elementos, false en caso contrario

# Listas sin Orden

## Algunas operaciones: agregar

- ***elemento(int pos)***: retorna el elemento de la posición indicada por ***pos***.

**elemento(3)**



resultado:



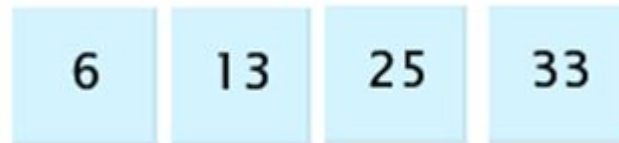
- ***incluye(Integer elem)***: retorna true si ***elem*** está contenido en la lista, false en caso contrario

**incluye(13)**

resultado: true

**incluye(7)**

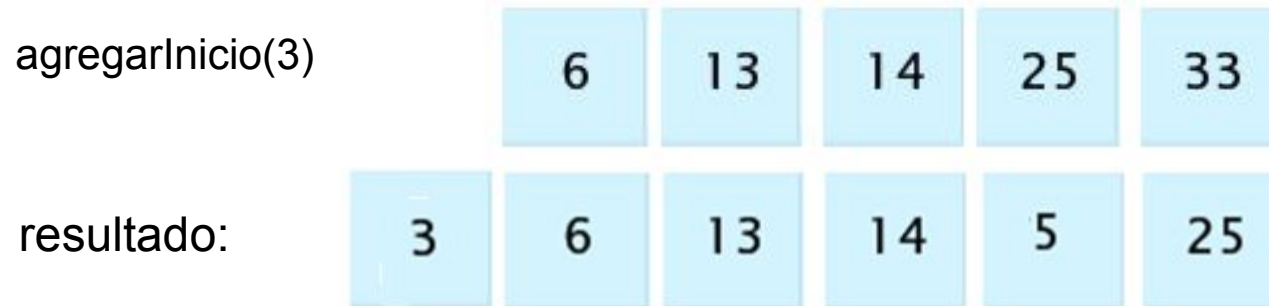
resultado: false



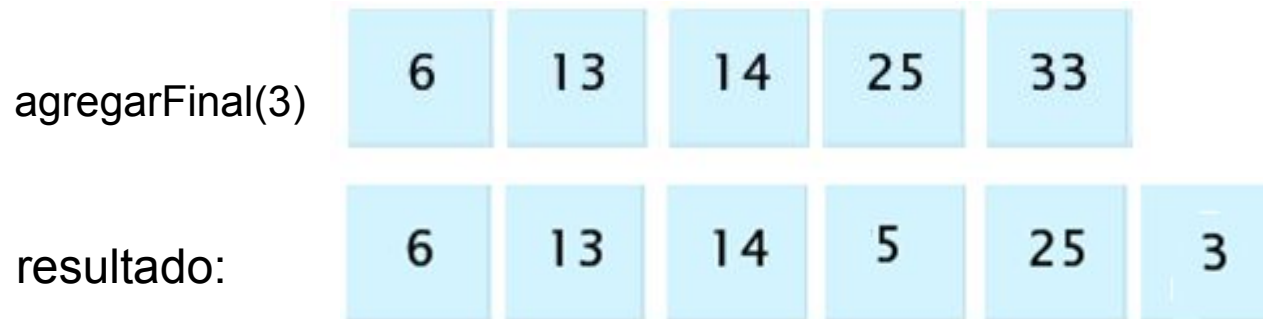
# Listas sin Orden

## Algunas operaciones: agregar

- ***agregarInicio(Integer elem)***: agrega el elemento ***elem*** al inicio de la lista



- ***agregarFinal(Integer elem)***: agrega el elemento ***elem*** al final de la lista



# Listas sin Orden

## Algunas operaciones: agregar

- ***agregarEn(Integer elem, int pos)***: agrega el elemento ***elem*** en la posición ***pos***.

agregarEn(5,4)

6	13	14	25	33
---	----	----	----	----

resultado:

6	13	14	5	25	33
---	----	----	---	----	----

¿Cómo se comportan los métodos ***agregarInicio(Integer elem)*** y ***agregarFinal(Integer elem)*** en términos de ***agregarEn(Integer elem, int pos)***?

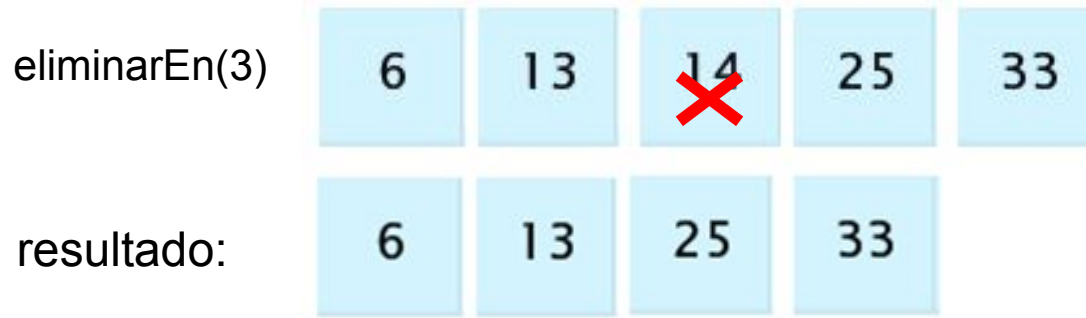
***agregarEn(elem,1)***

***agregarEn(elem, tamaño()+1)***

# Listas sin Orden

## Algunas operaciones: eliminar

- ***eliminarEn(int pos)***: elimina el elemento de la posición indicada



- ***eliminar(Integer elem)***: elimina el elemento “elem” indicado

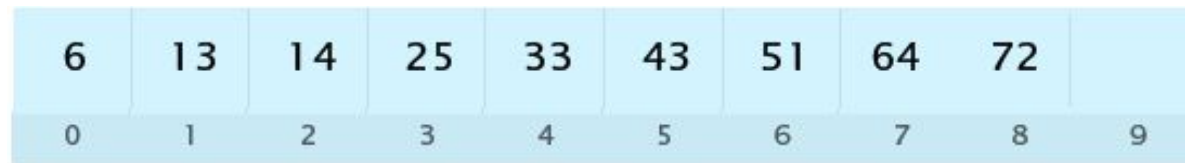


# Listas sin Orden

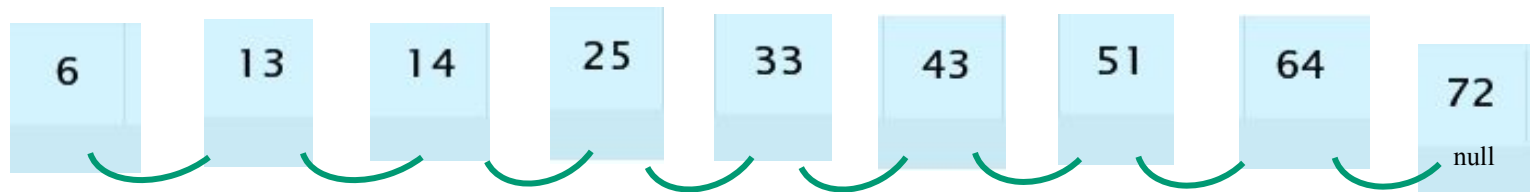
## Implementaciones

Una lista puede estar implementada a través de:

- una estructura estática (arreglo)



- una estructura dinámica (nodos enlazados)



# Listas sin Orden

## Implementaciones

Independientemente de la estructura de datos usada para implementar la lista, ambas responden al mismo conjunto de operaciones:

**lista1.elemento(2):** debe retornar 13, el valor del segundo nodo

**lista2.elemento(2):** debe retornar 13, el valor de la 2<sup>da</sup> componentes del arreglo (índice 1)

**lista1.agregarInicio(12):** debe agregar el 12 al inicio de la lista, actualizando referencias

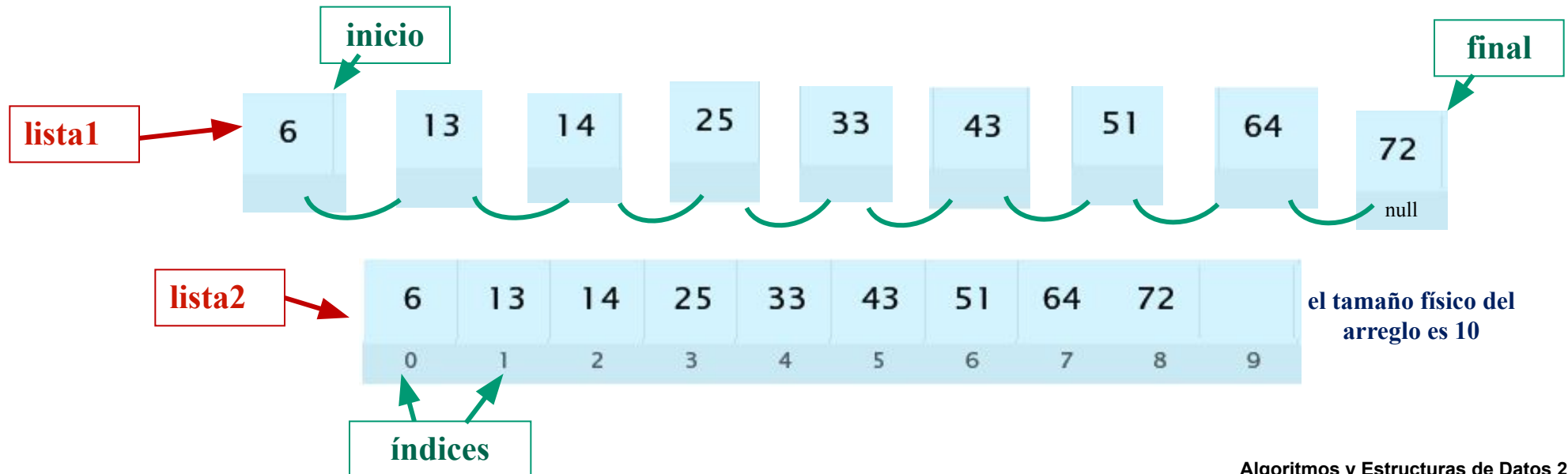
**lista2.agregarInicio(12):** debe agregar el 12 al inicio de la lista, haciendo corrimiento a la derecha

**lista1.agregarEn(8, 3):** debe agregar un nuevo nodo entre los nodos con valor 13 y 14

**lista2.agregarEn(8, 3):** debe agregar el valor 8 donde está el valor 14 previo corrimiento a la derecha a

**lista1.tamaño()** □ 9 y **lista2.tamaño()** □ 9

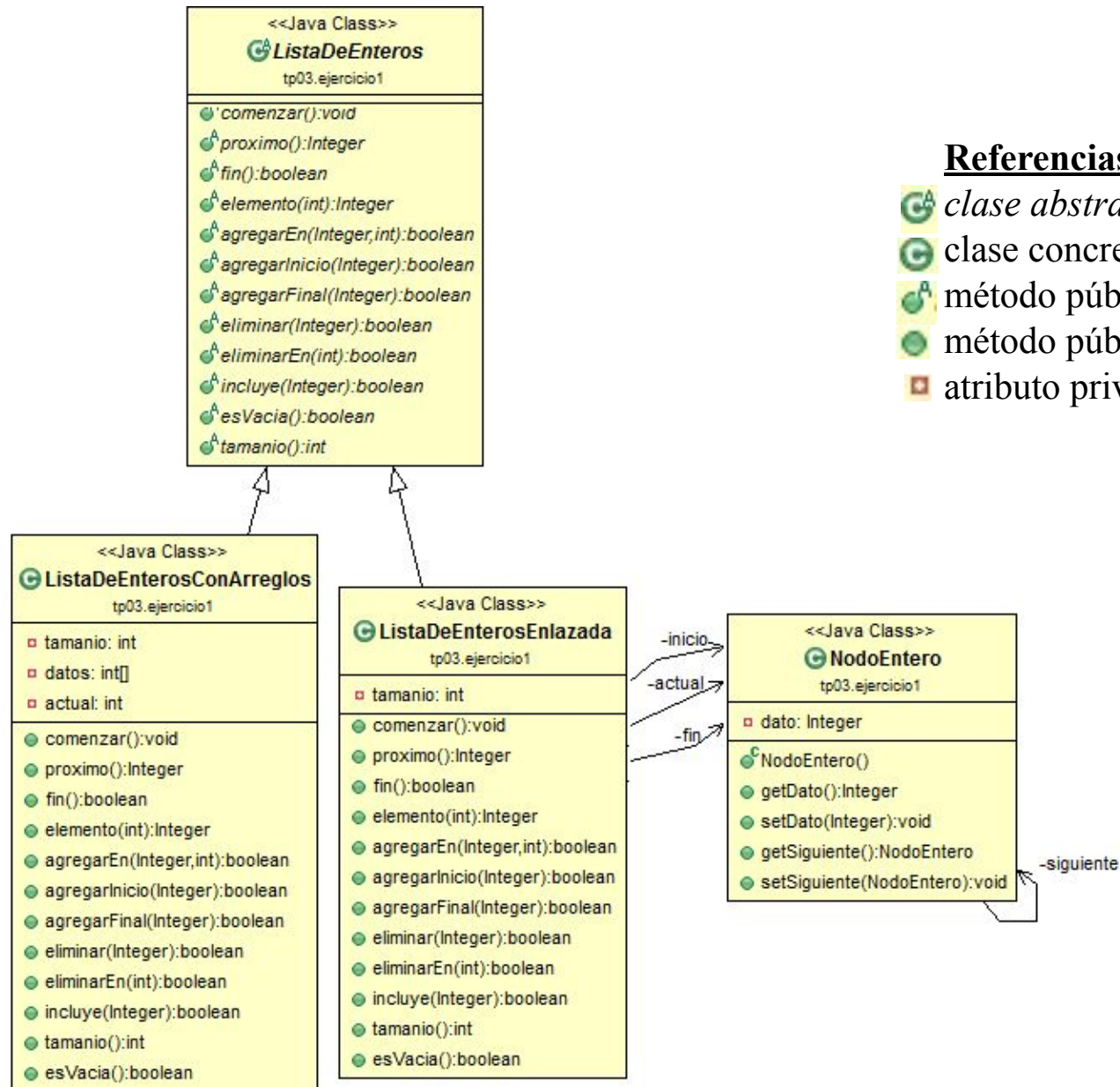
partir de ese valor.





# Trabajo Práctico 2

## Encapsulamiento y abstracción con Listas sin orden



### Referencias

- clase abstracta (letra cursiva)*
- clase concreta
- método público abstracto (+)
- método público concreto (+)
- atributo privado (-)

# Trabajo Práctico 2

## Encapsulamiento y abstracción

La clase abstracta **ListaDeEnteros**

```
package tp03.ejercicio2;
public abstract class ListaDeEnteros {
    public abstract void comenzar();
    public abstract Integer proximo();
    public abstract boolean fin();

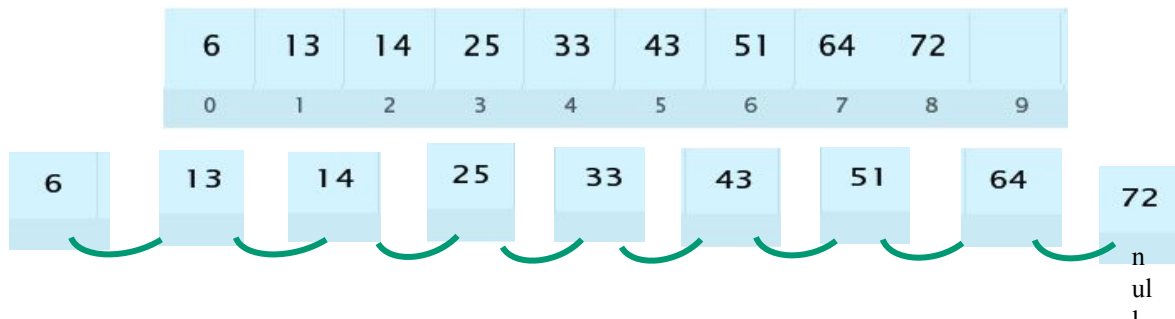
    public abstract Integer elemento(int pos);

    public abstract boolean agregarEn(Integer elem, int pos);
    public abstract boolean agregarInicio(Integer elem);
    public abstract boolean agregarFinal(Integer elem);

    public abstract boolean eliminar(Integer elem);
    public abstract boolean eliminarEn(int pos);

    public abstract boolean incluye(Integer elem);
    public abstract boolean esVacia();
    public abstract int tamanio();
}
```

¿Qué mecanismos podemos usar para crear subclases concretas de Lista?



# Trabajo Práctico 2

## Encapsulamiento y abstracción

Lista de enteros implementada con un arreglo

```
public class ListaDeEnterosConArreglos extends ListaDeEnteros {
    private int tamanio;
    private int[] datos = new int[200];
    private int actual = 0;
    @Override
    public void comenzar() {
        actual = 0;
    }
    @Override
    public Integer proximo() {
        return datos[actual++];
    }
    @Override
    public boolean fin() {
        return actual==tamanio;
    }
    @Override
    public Integer elemento(int pos) {
        return datos[pos-1];
    }
    @Override
    public boolean agregarEn(Integer elem, int pos) {
        if (pos<1 || pos>tamanio+1 || pos>datos.length || tamanio==datos.length)
            return false;
        tamanio++;
        for (int i = tamanio; i >= pos; i--)
            datos[i] = datos[i - 1];
        datos[pos-1] = elem;
        return true;
    }
    . . .}
```

```
public class ListaTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ListaDeEnterosConArreglos l = new ListaDeEnterosConArreglos();
        l.agregarFinal(1);
        l.agregarFinal(2);
        l.agregarFinal(3);
        l.agregarEn(25, 3);
        l.agregarEn(55, 1);

        System.out.println("Usando toString()");
        System.out.println(l);
        l.comenzar();
        System.out.println("Usando métodos");
        while (!l.fin()) {
            System.out.println(l.proximo());
        }
    }
}
```

**NOTA:** `@override` indica que se está sobrescribiendo un método de la superclase y el compilador informa un error en caso de no existir el método en la superclase

```
Usando toString()
55 -> 1 -> 2 -> 25 -> 3
Usando métodos
55
1
2
25
3
```

# Trabajo Práctico 2

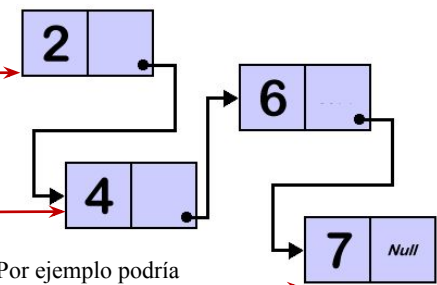
## Encapsulamiento y abstracción

Lista de enteros implementada con nodos enlazados

```
public class ListaDeEnterosEnlazada extends ListaDeEnteros {
    private NodoEntero inicio;
    private NodoEntero actual;
    private NodoEntero fin;
    private int tamano;
    @Override
    public void comenzar() {
        actual = inicio;
    }
    @Override
    public Integer proximo() {
        Integer elto = actual.getDato();
        actual = actual.getSiguiente();
        return elto;
    }
    @Override
    public boolean fin() {
        return (actual==null);
    }
    @Override
    public boolean incluye(Integer elem) {
        NodoEntero n = this.inicio;
        while (!(n == null) && !(n.getDato().equals(elem)))
            n = n.getSiguiente();
        return !(n == null);
    }
    @Override
    public boolean esVacia() {
        return (inicio == null);
    }
    . . . }
```

```
public class NodoEntero {
    private Integer dato;
    private NodoEntero siguiente;
    public Integer getDato() {
        return dato;
    }
    public void setDato(Integer dato) {
        this.dato = dato;
    }
    public NodoEntero getSiguiente() {
        return siguiente;
    }
    public void setSiguiente(NodoEntero siguiente) {
        this.siguiente = siguiente;
    }
}
```

Por ejemplo podría  
referenciar a este nodo



El uso es igual a la de Lista con arreglos, solo se cambia la instanciación. La clases concretas no agregan métodos nuevo, por ello los métodos que se pueden invocar son los definidos en la clase abstracta.

```
public class ListaTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ListaDeEnterosEnlazada l = new ListaDeEnterosEnlazada();
        l.agregarFinal(1);
        . . .
    }
}
```

# Trabajo Práctico 2

## Encapsulamiento y abstracción

Ejemplo de uso de una lista desde otra clase que está en otro paquete.

```
package tp03.ejercicio2;
import tp03.ejercicio1.ListaDeEnteros;
import tp03.ejercicio1.ListaDeEnterosEnlazada;

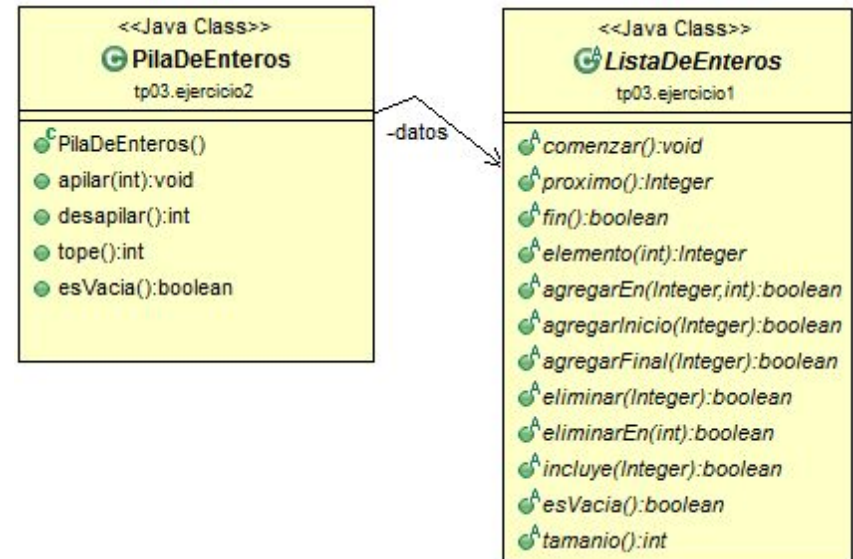
public class PilaDeEnteros {
    private ListaDeEnteros datos =
        new ListaDeEnterosEnlazada();

    public void apilar(int dato) {
        datos.agregarEn(dato, 1);
    }

    public int desapilar() {
        int x = datos.elemento(1);
        datos.eliminarEn(1);
        return x;
    }

    public int tope() {
        return datos.elemento(1);
    }

    public boolean esVacia() {
        return datos.tamano() == 0;
    }
}
```



```
package tp03.ejercicio2;

public class PilaTest {
    public static void main(String args[]) {
        PilaDeEnteros p = new PilaDeEnteros();
        p.apilar(10);
        p.apilar(20);
        p.apilar(30);
        System.out.print("Tope: " + p.tope());
    }
}
```

La salida es: **Tope: 30**

# Trabajo Práctico 2

## Encapsulamiento y abstracción

A continuación se muestra la clase `Lista` donde se pueden mantener elementos de tipo `Object`. Se podrán definir dos subclases `ListaConArreglos` y `ListaEnlazada` de manera que también puedan almacenar elementos de tipo `Object`.

```
package tp03.ejerciciox;

public abstract class Lista {
    public abstract void comenzar();
    public abstract Object proximo();
    public abstract boolean fin();

    public abstract Object elemento(int pos);

    public abstract boolean agregarEn(Object elem, int pos);
    public abstract boolean agregarInicio(Object elem);
    public abstract boolean agregarFinal(Object elem);

    public abstract boolean eliminar(Object elem);
    public abstract boolean eliminarEn(int pos);

    public abstract boolean incluye(Object elem);
    public abstract boolean esVacia();
    public abstract int tamanio();
}
```

### Ejemplo de uso:

```
ListaConArreglos lista = new ListaConArreglos();
lista.agregarEn(new Integer(2), 1);
lista.agregarEn(new Integer(4), 2);
lista.agregarEn(new String("Hola"), 3);

Integer x = (Integer)lista.elemento(2); // se debe castear
```

- ¿Podría guardar objetos de tipo `¿Alumno?`
- Y al recuperarlo, ¿puedo pedirle directamente su número de alumno?



# Generalizando Estructuras

Analizamos la implementación de Listas con elementos de tipo **Integer** y con elementos de tipo **Object**:

## Usando un tipo específico (Integer):

```
public class ListaDeEnterosConArreglos {  
    private Integer[] datos = new Integer[200];  
    private int actual;  
    . . .  
}
```

**Ventajas:** el compilador chequea el tipo de dato que se inserta. No se necesita hacer uso del *casting*

**Desventajas:** si se quisiera tener una estructura para cada tipo de datos, se debería definir una clase para cada tipo. Por ejemplo: **ListaDeEnteros**, **ListaDeAlumnos**, etc.

```
ListaDeEnterosConArreglos lista = new ListaDeEnterosConArreglos();  
lista.agregarFinal(new Integer(50));  
lista.agregarFinal(new String("Hola"));  
Integer x1 = lista.elemento(1);
```

- ❑ no deja poner otra cosa que no sea Integer
- ❑ no necesitamos castear cada vez

## Usando Object:

```
public class ListaConArreglos {  
    private Object[] datos = new Object[200];  
    private int actual;  
    . . .  
}
```

**Ventajas:** Se logra una estructura genérica

**Desventajas:** El compilador pierde la oportunidad de realizar chequeos y se debe hacer uso de *casting*

```
ListaConArreglos lista = new ListaConArreglos();  
lista.agregarFinal(new Integer(50));  
lista.agregarFinal(new String("Hola"));  
Integer x = (Integer)lista.elemento(1);
```

- ❑ deja poner cualquier tipo
- ❑ necesitamos castear y podría dar error en ejecución

# Generalizando Estructuras

J2SE 5.0 introduce varias extensiones al lenguaje java. Una de las más importantes, es la incorporación de los **tipos genéricos**, que le permiten al programador abstraerse de los tipos.

Usando tipos genéricos, es posible definir estructuras dónde la especificación del tipo de objeto a guardar se posterga hasta el momento de la instanciación.

Para especificar el uso de genéricos, se utiliza **<tipo>**.

```
package tp03.ejercicio6;
public class ListaEnlazadaGenerica<T> extends ListaGenerica<T>{
    private NodoGenerico<T> inicio;
    private NodoGenerico<T> actual;
    private NodoGenerico<T> fin;
    private int tamano;
    ...
}
```

```
package tp03.ejercicio6;
public class NodoGenerico<T> {
    private T dato;
    private NodoGenerico<T> siguiente;

    public T getDato() {
        return dato;
    }
    . . .
}
```

Cuando se instancian las estructuras se debe definir el tipo de los objetos que en ella se almacenarán:

```
ListaEnlazadaGenerica<Integer> lista = new ListaEnlazadaGenerica<Integer>();
lista.agregarFinal(new Integer(50));
lista.agregarFinal(new String("Hola")); ❑ error de compilación
lista.comenzar();
Integer x = lista.proximo(); ❑ no necesitamos castear
```

```
ListaEnlazadaGenerica<Alumno> lista = new ListaEnlazadaGenerica<Alumno>();
lista.agregarFinal(new Alumno("Peres, Juan", 3459);
lista.agregarFinal(new Alumno("Rios, Ivana", 3052);

lista.comenzar();
Alumno a = lista.proximo(); ❑ no necesitamos castear

Integer i = lista.proximo(); ❑ error en compilación
lista.agregarFinal(55); ❑ error de compilación
```



# ¿Cómo quedan las Listas con Tipos Genéricos?

Clase abstracta **ListaGenerica** y una subclases implementada como lista enlazada con tipos Genéricos:

```
package tp03.ejercicio6;

public abstract class ListaGenerica<T> {
    public abstract void comenzar();
    public abstract T proximo();
    public abstract boolean fin();

    public abstract T elemento(int pos);
    public abstract boolean agregarEn(T elem, int pos);
    public abstract boolean agregarInicio(T elem);
    public abstract boolean agregarFinal(T elem);

    public abstract boolean eliminar(T elem);
    public abstract boolean eliminarEn(int pos);

    public abstract boolean incluye(T elem);
    public abstract boolean esVacia();
    public abstract int tamano();
}
```

```
package tp03.ejercicio6;

public class ListaEnlazadaGenerica<T> extends
    ListaGenerica<T> {
    private NodoGenerico<T> inicio;
    private NodoGenerico<T> actual;
    private NodoGenerico<T> fin;
    private int tamano;

    @Override
    public void comenzar() {
        actual = inicio;
    }

    @Override
    public T proximo() {
        T elto = actual.getDato();
        actual = actual.getSiguiente();
        return elto;
    }
}
```

```
...
package tp03.ejercicio6;
public class NodoGenerico<T> {
    private T dato;
    private NodoGenerico<T> siguiente;

    public T getDato() {
        return dato;
    }
    . . .
}
```