

# Práctica 2

## Encapsulamiento y abstracción

La clase **ListaDeEnteros**

```
package ejercicio1;

public abstract class ListaDeEnteros {

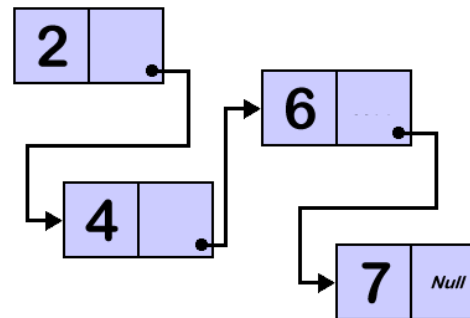
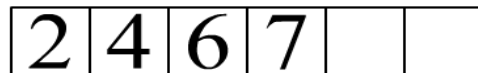
    public abstract void comenzar();
    public abstract void proximo();
    public abstract Integer elemento();
    public abstract Integer elemento(int pos);
    public abstract boolean agregar(Integer elem, int pos);
    public abstract boolean eliminar();
    public abstract boolean eliminar(int pos);
    public abstract int tamanio();
    public abstract boolean fin();
    protected int tamanio;

}
```

```
public boolean incluye(Integer elem) {
    this.comenzar();
    while (!this.fin() && !this.elemento().equals(elem))
        this.proximo();
    return !this.fin();
}
```

```
public String toString(){
    String str="";
    this.comenzar();
    while (!this.fin()) {
        str = str + this.elemento();
        this.proximo();
    }
    return str;
}
```

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





# Práctica 2

## Encapsulamiento y abstracción

Lista de enteros implementada con un arreglo

```
package ejercicio1;

public class ListaDeEnterosConArreglos extends ListaDeEnteros {
    private Integer[] datos = new Integer[200];
    private int actual = 0;

    @Override
    public void comenzar() {
        actual = 0; }

    @Override
    public void proximo() {
        actual++; }

    @Override
    public Integer elemento() {
        return datos[actual];
    }

    @Override
    public boolean agregar(Integer elem, int pos) {
        if (pos < 0 || pos > tamaño || pos >= datos.length)
            return false;
        tamaño++;
        for (int i = tamaño; i > pos; i--)
            datos[i] = datos[i-1];
        datos[pos] = elem;
        return true;
    }
    ...
}
```

2	4	6	7	.	.
---	---	---	---	---	---

Ejemplo de uso:

```
ListaDeEnterosConArreglos lista = new ListaDeEnterosConArreglos();
lista.agregar(new Integer(2));
lista.agregar(new Integer(4));
lista.agregar(new Integer(6));
lista.agregar(new Integer(7));
lista.comenzar();
Integer x = lista.elemento(); // Retorna un Integer
```

**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



# Práctica 2

## Encapsulamiento y abstracción

Lista de enteros implementada con nodos enlazados

```
package ejercicio1;

public class ListaDeEnterosEnlazada extends ListaDeEnteros {
    private NodoEntero inicio;
    private NodoEntero actual;

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

    @Override
    public void proximo() {
        actual = actual.getSiguiente();
    }

    @Override
    public Integer elemento() {
        return actual.getData();
    }

    @Override
    public Integer elemento(int pos) {
        this.comenzar();
        while (pos-- > 0)
            this.proximo();
        return actual.getData();
    }
    . . .
}
```

```
package ejercicio1;

public class NodoEntero {
    private Integer dato;
    private NodoEntero siguiente;

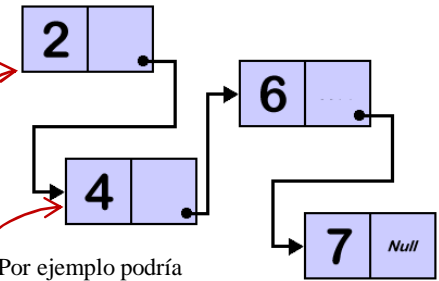
    public Integer getData() {
        return dato;
    }

    public void setData(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 interface es la de la clase abstracta.

# Práctica 2

## Encapsulamiento y abstracción

A la clase Lista y a las subclasses también podríamos definir las de manera que puedan almacenar elementos de tipo Object

```
package ejemplo;

public abstract class Lista {

    public abstract void comenzar();
    public abstract void proximo();
    public abstract Object elemento();
    public abstract Object elemento(int pos);
    public abstract boolean agregar(Object elem, int pos);
    public abstract boolean eliminar();
    public abstract boolean eliminar(int pos);
    public abstract int tamaño();
    public abstract boolean fin();
    protected int tamaño;

}
```

```
public boolean incluye(Object elem) {
    this.comenzar();
    while (!this.fin() && !this.elemento().equals(elem))
        this.proximo();
    return !this.fin();
}
```

```
public String toString(){
    String str="";
    this.comenzar();
    while (!this.fin()) {
        str = str + this.elemento();
        this.proximo();
    }
    return str;
}
```

### Ejemplo de uso:

```
ListaConArreglos lista = new ListaConArreglos();
lista.agregar(new Integer(2));
lista.agregar(new Integer(4));
lista.agregar(new Integer(6));
lista.agregar(new Integer(7));
lista.comenzar();
Integer x = (Integer)lista.elemento(); // 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 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.agregar(new Integer(50));  
lista.agregar(new String("Hola")); ➔ no deja poner otra cosa que no sea Integer  
Integer x1 = lista.elemento(0); ➔ 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.agregar(new Integer(50));  
lista.agregar(new String("Hola"));  
Integer x = (Integer)lista.elemento();
```

➔ 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 ejercicio6;
public class ListaEnlazadaGenerica<T> extends
        ListaGenerica<T> {
    private NodoGenerico<T> inicio;
    private NodoGenerico<T> actual;
    ...
}
```

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.agregar(new Integer(50));
lista.agregar(new String("Hola"));           → error de compilación
lista.comenzar();
Integer x = lista.elemento();                 → no necesitamos castear
```

```
ListaEnlazadaGenerica<Alumno> lista = new ListaEnlazadaGenerica<Alumno>();
lista.agregar(new Alumno("Peres, Juan", 23459));
lista.agregar(55);                           → error de compilación
lista.comenzar();
Alumno a = lista.elemento();                  → no necesitamos castear
Integer i = lista.elemento();                 → error en compilación
```

# ¿Cómo quedan las Listas con Tipos Genéricos?

La clase abstracta **ListaGenerica** y una subclases implementada como lista enlazada:

```
package ejercicio6;

public abstract class ListaGenerica<T> {
    protected int tamano;

    abstract public void comenzar();
    abstract public void proximo();
    abstract public boolean fin();
    abstract public T elemento();
    abstract public T elemento(int pos);
    abstract public boolean agregar(T elem, int pos);
    abstract public boolean eliminar();
    abstract public boolean eliminar(int pos);
    abstract public boolean esVacia();
    abstract public int tamano();

    public boolean incluye(T elem) {
        this.comenzar();
        while (!this.fin() && !this.elemento().equals(elem))
            this.proximo();
        return !this.fin();
    }
}
```

```
package ejercicio6;

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

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

    @Override
    public T elemento() {
        return actual.getDato();
    }

    ...
}
```

```
package ejercicio6;

public class NodoGenerico<T> {
    private T dato;
    private NodoGenerico<T> siguiente;

    public T getDato() {
        return dato;
    }

    ...
}
```