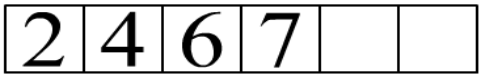
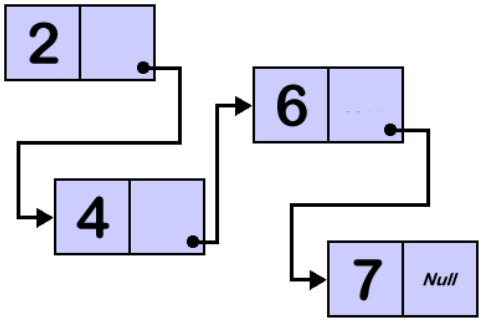
Práctica 2
Encapsulamiento y abstracción

La clase **`ListaDeEnteros`**

```
public abstract class ListaDeEnteros {

          protected int tamanio;

          abstract public void comenzar();
          abstract public void proximo();
          abstract public boolean fin();
          abstract public Integer elemento();
          abstract public Integer elemento(int pos);
          abstract public boolean agregar(Integer elem);
          abstract public boolean agregar(Integer elem, int pos);
          abstract public boolean eliminar();
          abstract public boolean eliminar(int pos);
          abstract public boolean esVacia();
          abstract public boolean incluye(Integer elem);
          abstract public int tamanio();
}
```

¿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 > this.tamanio || pos >= datos.length)
    return false;
 this.tamanio++;
 for (int i = tamanio; i > pos; i--)
   datos[i] = datos[i-1];
 datos[pos] = elem;
 return true;
}
…
}
```

Ejemplo de uso:

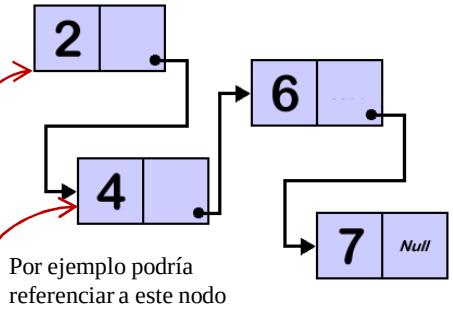
```
ListaDeEnterosConArreglos lista = new ListaDeEnterosConArreglos();
lista.agregar(new Integer(2));
lista.agregar(new Integer(4));
lista.agregar(new Integer(6));
lista.agregar(new Integer(7));
Lista.comenzar();
Integer x = lista.elemento();  // Retorna un Integer
```

NOTA: `@override` indica que se está sobrescribiendo un método de la superclase y el compilador informa un error en caso de no exisitir 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.getDato();
 }

 @Override
 public boolean incluye(Integer elem) {
   NodoEntero n = this.inicio;
   while (!(n==null) && !(n.getDato().equals(elem)) )
      n = n.getSiguiente();
   return !(n==null);
  }
 . . .
}
```




```
package ejercicio1;

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;
}
}
```

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

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

```
package tp02.ejercicio2;
import tp02.ejercicio1.ListaDeEnteros;
import tp02.ejercicio1.ListaDeEnterosEnlazada;

public class PilaDeEnteros {
 private ListaDeEnteros datos;

 public PilaDeEnteros() {
  datos = new ListaDeEnterosEnlazada();
 }

 public void poner(int dato) {
  datos.agregar(dato, 0);
 }

 public int sacar() {
  int x = datos.elemento(0);
  datos.eliminar(0);
  return x;
 }

 public int tope() {
  return datos.elemento(0);
 }

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

Ejemplo de uso

```
package tp02.ejercicio2;

public class PilaTest {

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

La salida es: **Tope: 30**

Práctica 2
Encapsulamiento y abstracción

A la clase Lista y a las subclases también podríamos definirlas de manera que puedan almacenar elementos de tipo Object:

```
public abstract class ListaDeEnteros {

          protected int tamanio;

          abstract public void comenzar();
          abstract public void proximo();
          abstract public boolean fin();
          abstract public Object elemento();
          abstract public Object elemento(int pos);
          abstract public boolean agregar(Object elem);
          abstract public boolean agregar(Object elem, int pos);
          abstract public boolean eliminar();
          abstract public boolean eliminar(int pos);
          abstract public boolean esVacia();
          abstract public boolean incluye(Object elem);
          abstract public int tamanio();
}
```

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"));        ➔ deja poner cualquier tipo
Integer x = (Integer)lista.elemento();    ➔ 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 tamanio;

   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);
   abstract public boolean agregar(T elem, int pos);
   abstract public boolean eliminar();
   abstract public boolean eliminar(int pos);
   abstract public boolean esVacia();
   abstract public boolean incluye(T elem);
   abstract public int tamanio();
}
```

```
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;
  }
  . . .
}
```