# Práctica 2

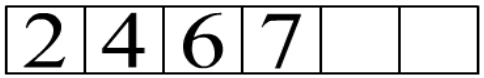
La clase **Lista**

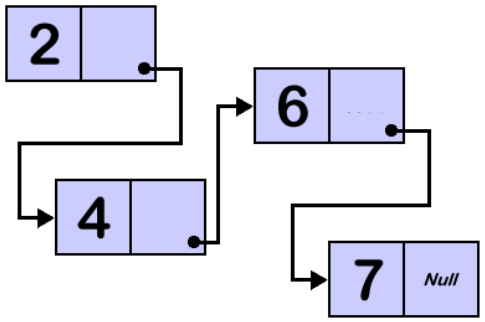
```
package tp03;

public abstract class Lista {

 public abstract void comenzar();
 public abstract void proximo();
 public abstract boolean fin();
 public abstract Object elemento();
 public abstract Object elemento(int pos);
 public abstract boolean agregar(Object elem);
 public abstract boolean agregar(Object elem, int pos);
 public abstract boolean eliminar();
 public abstract boolean eliminar(int pos);
 public abstract boolean esVacia();
 public abstract int tamaño();
 public abstract boolean incluye(Object elem);
 public abstract String toString();

}
```

```
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

Lista implementada con un arreglo

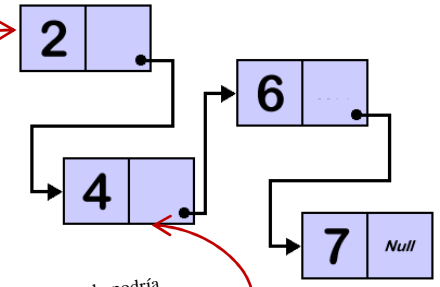
```
package tp03;
public class ListaConArreglos extends Lista {

 private Object[] datos = new Object[100];
 private int ultimo = -1 ;
 private int actual = 0;

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

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

 @Override
 public boolean fin() {
    return (actual>ultimo);
 }

 @Override
 public Object elemento() {
   return datos[actual];
 }
 . . .
}
```


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();
```

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

# Práctica 2




Lista implementada nodos enlazados

```
package tp03;

public class ListaEnlazada extends Lista {

 private Nodo primero;
 private Nodo actual;
 private int tamaño;

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

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

 @Override
 public boolean fin() {
    return (actual==null);
 }

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

`@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

```
package tp03;

public class Nodo  {
  private Object dato;
  private Nodo proximo;

  public Nodo(Object elem) {
   dato = elem;
  }
  public Object getDato() {
   return dato;
  }
  public void setDato(Object dato) {
   this.dato = dato;
  }
  public Nodo getProximo() {
   return proximo;
  }
  public void setProximo(Nodo proximo) {
    this.proximo = proximo;
  }
}
```

# Generalizando estructuras – Tipos Genéricos

Analizamos la implementación de Listas con elementos de tipo Object:

```
public class ListaConArreglos {
   private Object[] datos;
   private int tamaño;
   . . .
}
```

**Usando Object:**

```
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
```

**Usando un tipo específico:**

```
public class ListaDeEnterosConArreglos {
   private Integer[] datos;
   private int tamaño;
   . . .
}
```

```
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
```

# Generalizando estructuras

Cada definición tiene ventajas y desventajas:

```
public class ListaDeEnterosConArreglos {
   private Integer[] datos;
   private int tamaño;
   . . .
}
```

```
public class ListaConArreglos {
   private Object[] datos;
   private int tamaño;
   . . .
}
```

## Usando un tipo específico

**Ventajas:**

- El compilador chequea el tipo de datos 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`**, **`ListaDeCiudades`**, etc.

## Usando Object

**Ventajas:**

- Se logra una estructura genérica

**Desventajas**

- El compilador pierde la oportunidad de realizar chequeos
- Se debe hacer uso de *casting*

# 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>**.

```
public class ListaConArreglos<T> {
   private T[] datos;
   private int tamaño;
   . . .
}
```

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

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

```
ListaConArreglos<Alumno> lista = new ListaConArreglos<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 **Lista** y las subclases implementada con tipos genéricos:

1

```
public abstract class Lista <T> {
  public abstract void comenzar();
  public abstract void proximo();
  public abstract boolean fin();
  public abstract T elemento();
  public abstract boolean agregar(T elem);
  public abstract boolean eliminar();
  public abstract boolean esVacia();
  public abstract int tamaño();

  public boolean incluye(T elem){
    comenzar();
    while (!esVacia()) {
    T obj = proximo();
    if (elem.equals(obj))
       return true;
    }
    return false;
  }
  public String toString(){
    . . .
  }
}
```

2

```
public class ListaConArreglos<T> extends Lista<T> {
  private T[] datos;
  private int tamaño;
  . . .
}
```

```
public class ListaEnlazada<T> extends Lista<T> {
  private Nodo<T> primero;
  private Nodo<T> actual;
  private int tamaño;
  . . .
}
```

3

```
public class Nodo<T> {
 private T dato;
 private Nodo<T> proximo;
 . . .
 public String toString() {
   return getDato().toString();
 }
}
```

# ¿Cómo quedan las Listas con Tipos genéricos?

**Definición de subclases de Lista utilizando tipos genéricos**

```
public class ListaConArreglos<T> extends Lista<T>
{
  private T[] datos;
  private int tamaño;
  . . .
}
```

```
public class ListaEnlazada<T> extends Lista<T> {
  private Nodo<T> primero;
  private Nodo<T> actual;
  private int tamaño;
  . . .
}
```

```
public class Nodo<T> {
 private T dato;
 private Nodo<T> proximo;
 . . .
 public String toString() {
   return getDato().toString();
 }
}
```

**Instanciación de listas con tipos genéricos**

```
public class TestLista {
 public static void main(String[] args) {

  ListaEnlazada<Alumno> lista1 =
                    new ListaEnlazada<Alumno>();
  lista1.agregar(new Alumno("23456/8"));
  lista1.agregar(new Alumno("23356/2"));
  lista1.agregar(new Alumno("23405/1"));
  lista1.agregar(new Alumno("24536/6"));
  System.out.println(lista1.toString());

  ListaEnlazada<String> lista2 =
                    new ListaEnlazada<String>();
  lis2.agregar("La ");
  lis2.agregar("vida ");
  lis2.agregar("de ");
  lis2.agregar("los ");
  lis2.agregar("otros ");
  System.out.println(lis2.toString());
 }
}
```