Estructuras dinámicas

Estructura de datos: Los datos se agrupan entre sí formando un todo o "estructura". Cuando trabajamos con objetos más que estructura de datos deberíamos hablar mejor de estructura de objetos.

¿Como se clasifican las estructuras?
Hay muchos criterios de clasificación, ahora interesa clasificarlas entre estáticas y dinámicas.
Estáticas: una vez que se crea su tamaño es fijo. Ejemplo Arrays y Strings, ambos son objetos en Java (no es así en todos los lenguajes) y por tanto se crea en tiempo de ejecución, pero una vez que se crean su tamaño es fijo
Dinámicas: Son estructuras que pueden cambiar su tamaño en tiempo de ejecución. Ejemplos del API Java: ArrayList, HashSet, ...

¿Cómo se crean las estructuras dinámicas?
Basándose principalmente en dos recursos base:

- arrays
- clases autorrefenciadas

Clases autorreferenciadas
Es una clase que contiene un atributo que hace referencia a una clase del mismo tipo.

Ejemplo: una clase Persona que tiene un atributo pareja(novio/compañero) que también será de tipo Persona

```
class Persona{
        String nombre;
        int edad;
        Persona pareja;
}
```

y luego en código ….

```
Persona p1 = new Persona();
……
Persona p2 = new Persona();
…..
p1.pareja=p2;
```

esto hace que una persona se "enlace" o "enganche" con otra.

Un ejemplo de clase autoreferenciada que usaremos para crear estructuras dinámicas, la clase Nodo.
La siguiente clase *Nodo*, contiene un atributo *sig* que es de tipo Nodo, esto la convierte en una clase autoreferenciada. Clases Nodo similares a las de este ejemplo son la base de muchas estructuras dinámicas

```
class Nodo{
  private Nodo sig;
  private int dato;
   //crea un nodo y se enlaza con otro
  public Nodo(int dato, Nodo sig) {
    this.dato = dato;
    this.sig = sig;
  }
  public void setSiguiente(Nodo sig) {
    this.sig = sig;
  }
  public Nodo getSiguiente() {
```

```
    return sig;
  }
  public int getDato() {
    return dato;
  }
}
```

Esto permite que un nodo referencie a otro nodo o dicho de otro modo que un nodo "se enganche a otro".

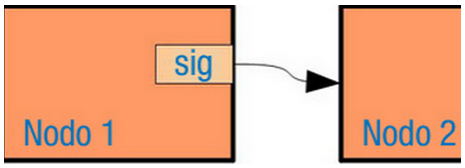
Observa detenidamente `private Nodo sig;`

Si en el primer ejemplo, el atributo pareja enlaza a un objeto persona(p1) con otro objeto persona(p2)

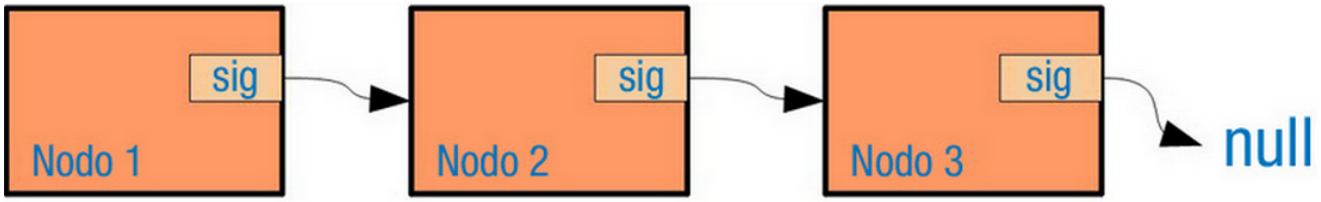
Ahora, de la misma forma en

```
class Nodo{
  private Nodo sig;
  private int dato;
 }
```

sig enlaza a un nodo con otro



si ahora al sig de Nodo2 le enlazo otro nodo y así sucesivamente….



tengo una lista de nodos enlazados. ENLAZADOS POR VARIABLES REFERENCIAS. Ya que la lista no puede ser infinita, el sig del último nodo de la lista no referencia a ningún objeto Nodo y por tanto valdrá null

¿Cómo sería una estructura dinámica basada en Nodos como los anteriores?

Sería una estructura basada en la interconexión de nodos. Hay muchas estructuras famosas basadas en enlazar nodos muy conocidas y usadas:

- listas
- pilas
- colas
- árboles
- otras

Ejemplo de una estructura dinámica: Una lista.

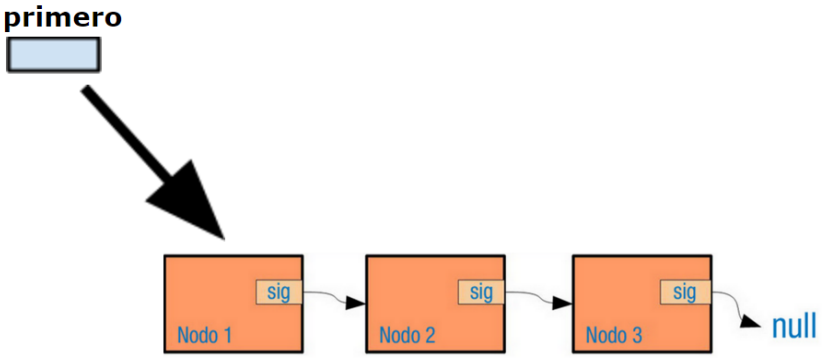
Es una de las estructuras de datos más genérica, la idea es que se tienen una lista de cosas (como datos primitivos u objetos) y puedo ir añadiendo o borrando cosas a la lista. Veremos cómo hacer una lista sencilla de dos formas distintas:

- basándonos en nodos: una lista enlazada

- basándonos en arrays.

EJEMPLO DE LISTA ENLAZADA (basada en nodos autorerenciados)

Puede ser más o menos compleja, veremos el ejemplo más básico. Una lista enlazada se consigue enlazando una serie de nodos y creando una forma de entrar en dicha lista. En el ejemplo accedemos a través de cómo no, una variable referencia a un nodo



Un código ejemplo para implantar un lista enlazada

Hay tres clases implicadas:

- La clase *Nodo,* me permite crear un objeto capaz de almacenar información y de encadenarse con otro objeto similar.
- la clase MiListaEnlazada tiene la entrada a la lista y un método insertar() que gestiona cómo se insertan los nuevos nodos a la estructura de nodos ya existente, en este caso, cada nuevo nodo siempre se inserta por el principio de la lista, es decir, un nodo nuevo pasa a ser el primero de la lista.
- la clase App, se limita a usar la lista pero a un nivel de abstracción tal, que ignora el concepto de Nodo. Usar la lista en este caso es crearla e insertar elementos en ella.

```
class Nodo{
    private Nodo sig;
    private int dato;
    public Nodo(int dato, Nodo sig) {
      this.dato = dato;
      this.sig = sig;
    }
    public void setSiguiente(Nodo sig) {
      this.sig = sig;
    }
    public Nodo getSiguiente() {
      return sig;
    }
    public int getDato() {
      return dato;
    }
  }
  class MiListaEnlazada{
      private Nodo primero=null;
      //en este mismo boletín veremos más adelante  una versión más compacta de insertar()
      public void insertar(int dato){
          if(primero==null){
              primero=new Nodo(dato,null);
          }else{
              Nodo temp= new Nodo(dato,primero);
              primero=temp;
          }

      }
```

```
 }
class App {
   public static void main(String[] args) {
       MiListaEnlazada miLista= new MiListaEnlazada();
       miLista.insertar(5);
       miLista.insertar(67);
       System.out.println("chao");
   }
}
```

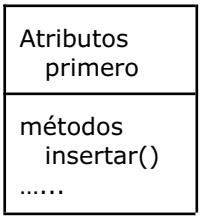
Para darle sentido a este código, supongamos que hay dos programadores. Un programador escribe la clase Nodo y la clase MiListaEnlazada. Otro programador escribe App. La clase nodo y la clase MilistaEnlazada tiene una relación de composición. El programador de App sólo accede a los métodos públicos de la clase MiListaEnlazada, no le interesan los entresijos de la clase Nodo

si "quitamos una foto" a la memoria del programa anterior podríamos ver la siguiente estructura

Ejercicio: con el profesor pon un punto de interrupción en el último println() y examina con el debugger los enlaces generados.

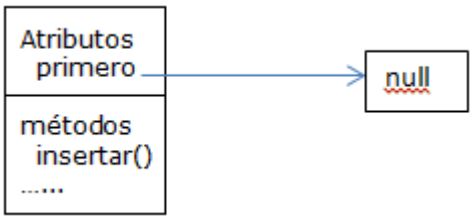
A continuación examinaremos la construcción de la lista paso a paso.
Primer paso construir una lista enlazada de enteros: Crear una lista vacía.

Un objeto de clase lista tiene el siguiente aspecto



Nuestra lista es muy sencilla, pero otras listas como las doblemente enlazadas tendrían más atributos para tener más de un enlace.

Al hacer `MiListaEnlazada miLista= new MiListaEnlazada();`
Creamos una lista vacía, sin nodos.
miLista



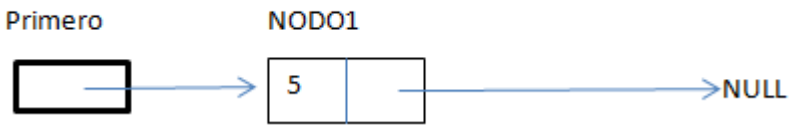
Insertar un nodo en la lista.
Debes observar a partir de aquí que realmente un objeto *Lista* no contiene una Lista, si no una variable *primero* que referencia al primer elemento de la lista, y por tanto, podemos considerarlo como la entrada de la lista.

Nuestra lista, la más simple posible, sólo tiene capacidad para insertar un nuevo nodo por la cabeza de la lista(no hay código para insertar al final ni por el medio de la lista)
La primera inserción es la siguiente

```
miLista.insertar(5);
```

esto implica que se ejecuta la siguiente instrucción del método insertar

```
primero= new Nodo(dato,null);
```

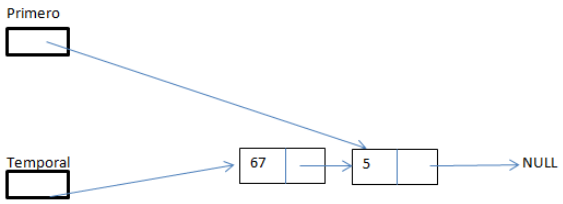


Cuando ya existe algún nodo(la lista no está vacía) la inserción consiste en añadir el nuevo nodo por la cabeza de la lista

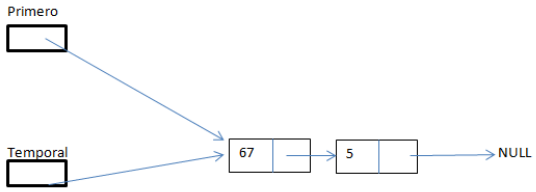
```
temporal= new Nodo(dato,primero);
primero=temporal;
```

por lo tanto, con la instrucción miLista.insertar(67), se inserta un nuevo nodo por la cabeza de la lista que gráficamente podemos verlo como

```
temporal= new Nodo(dato,primero);
```




```
primero=temporal;
```



Conclusión "jugando" con las variables referencia conseguimos ir enlazando un nodo con otro

Ejercicio U6_B2A_E1: Escribe los métodos tamano() y obtener() para que funcione el siguiente main()
nota: podemos usar la ñ y llamar al método tamaño(). Sin problema con JDK pero puede fallar con algún plugin de ide

```
class App {
   public static void main(String[] args) {
       MiListaEnlazada miLista= new MiListaEnlazada();
       miLista.insertar(8);
       miLista.insertar(88);
       miLista.insertar(888);
       for(int i=0;i<miLista.tamano();i++){
         System.out.print(miLista.obtener(i)+" ");
       }
  }
}
```

EJEMPLO DE LISTA BASADA EN ARRAYS

```
class ArrayListCasero{
   int[] arrayInterno;

   void insertar(int nuevoElemento){
      int[] nuevoArray;
      if(arrayInterno==null){
         nuevoArray = new int[1];
         nuevoArray[0]=nuevoElemento;
      }else{
         nuevoArray = new int[arrayInterno.length+1];
         for(int i=0;i<arrayInterno.length;i++){
            nuevoArray[i]=arrayInterno[i];
         }
         nuevoArray[nuevoArray.length-1]=nuevoElemento;
      }
      arrayInterno=nuevoArray;
   }

}

public class App{
   public static void main(String[] args){
      ArrayListCasero alc= new ArrayListCasero();
      alc.insertar(1);
      alc.insertar(2);
   }
}
```

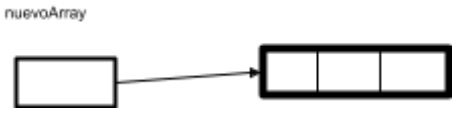
En el código anterior asumimos que si el array es null, el array está vacío. Otro enfoque es crear un array de tamaño 0 en lugar de utilizar null para representar una lista vacía.

```
class ArrayListCasero {

   int[] arrayInterno = new int[0];

   void insertar(int nuevoElemento) {
      int[] nuevoArray = new int[arrayInterno.length + 1];
      for (int i = 0; i < arrayInterno.length; i++) {
         nuevoArray[i] = arrayInterno[i];
      }
      nuevoArray[nuevoArray.length - 1] = nuevoElemento;

      arrayInterno = nuevoArray;
   }
}
```

Preferimos este segundo enfoque porque nos ahorramos un if. Puedes hacer la copia con System.arraycopy() en lugar de manualmente

El uso de arrays en comparación con los nodos enlazados son mucho más fáciles de entender, ya que consiste en una simple operación de copia entre un array viejo y otro nuevo con un elemento adicional recurso que ya utilizamos en ejemplos anteriores para "hacer crecer un array" .

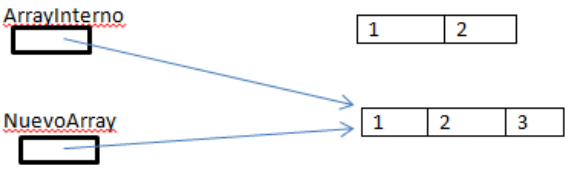
Supongamos que a la lista del main anterior quiero añadirle un nuevo elemento
alc.insertar(3);
esto provoca que se cree un nuevoArray con 3 elementos



si copiamos arrayInterno en nuevoArray y añadimos el nuevo elemento al final de nuevoArray


```
for(int i=0;i<arrayInterno.length;i++){
    nuevoArray[i]=arrayInterno[i];
}
nuevoArray[nuevoArray.length-1]=nuevoElemento;
```

Quedaría actualizar la referencia de arrayInterno. Finalmente el array viejo quedará en memoria perdido y esperando a que el recolector de basura lo detecte y elimine.



Ejercicio U6_B2A_E2: Escribe los métodos tamano() y obtener para que funcione el siguiente main()

```
public class App{
   public static void main(String[] args){
      ArrayListCasero alc= new ArrayListCasero();
      alc.insertar(1);
      alc.insertar(2);
      alc.insertar(3);

      for(int i=0;i<alc.tamano();i++)
         System.out.println(alc.obtener(i));

   }
}
```

escribir un método para eliminar un nodo en la lista enlazada y un método insertar más conciso

Lo mejor es hacer un dibujo, para saber como "jugar" con las referencias para conseguir el efecto deseado. Se puede escribir un método eliminar de muchas formas, por ejemplo

```
class Nodo {

    private Nodo sig;
    private int dato;


    public Nodo(int dato, Nodo sig) {
        this.dato = dato;
        this.sig = sig;
    }

    public void setSiguiente(Nodo sig) {
        this.sig = sig;
    }

    public Nodo getSiguiente() {
        return sig;
    }

    public int getDato() {
        return dato;
    }
}

class MiListaEnlazada {

    private Nodo primero = null;
    /*
    public void insertar(int dato) {
        if (primero == null) {
            primero = new Nodo(dato,null);
        } else {
            Nodo temp = new Nodo(dato, primero);
            primero = temp;
        }

    }
    */

    //este insertar es equivalente al anterior. Quizá se entienda peor
    public void insertar(int dato) {
        primero = new Nodo(dato,primero);
    }



    public void eliminar(int posicion) {
        //suponemos que la posicion es de 0 a n -1 siendo n el tamaño de la lista

        //el borrado del primero es un caso especial, consiste en hacer primero el segunod
        if (posicion == 0) {
            primero = primero.getSiguiente();
        } else {

            //para borrar un nodo que no es el primero
            //localizo el nodo justo anterior, es decir el de que está en posicion-1
             Nodo ant = primero;
            for (int i = 0; i < (posicion - 1); i++) {
                ant = ant.getSiguiente();
            }
            //tengo en ant el nodo justo anterior al que quiero borrar
            //borrar es puentear el elemnto i

            ant.setSiguiente(ant.getSiguiente().getSiguiente());
        }

    }

    public int tamano() {
        int i = 0;
        Nodo temp = primero;
        while (temp != null) {
            i++;
            temp = temp.getSiguiente();
        }
        return i;
    }

    public int obtener(int indice) {
        Nodo temp = primero;
        int i = 0;
        while (i < indice) {
            temp = temp.getSiguiente();
            i++;
        }

        return temp.getDato();
    }
}

class App {

    public static void main(String[] args) {
        MiListaEnlazada miLista = new MiListaEnlazada();
```

```
        miLista.insertar(1);
        miLista.insertar(2);
        miLista.insertar(3);
        miLista.insertar(4);
        miLista.eliminar(1);
        for (int i = 0; i < miLista.tamano(); i++) {
            System.out.print(miLista.obtener(i) + " ");
        }
    }
}
```

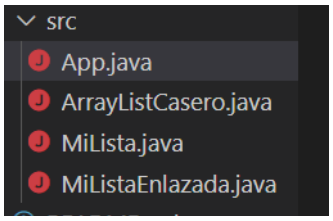
Ayuda mucho dibujar la lista de nodos en un papel, usando flechas para las referencias para razonar el proceso de eliminación

Un interface muchas implementaciones

Como ves hemos creado una lista con dos implementaciones diferentes, una basada en enlaces y otra en Arrays. Con diferentes técnicas de almacenamientos, hemos conseguido una Lista que puede crecer dinámicamente. Observa que ambas ofrecen los métodos insertar(), tamano() y obtener(), si sacamos factor común podemos subir estos métodos a una clase abstracta o a un interface. ¿Porqué utilizar un Interface?:

- los usuarios de nuestras clases prefieren a menudo "el resumillo" del interface.
- el código de los usuarios es más genérico ya que está aislado por el Interface y lo hace más inmune a los cambios de las implementaciones (similar a lo de set/get pero con más implicaciones todavía)
- se obliga a las clases que lo implementan a que cubran un "contrato" de mínimos
- etc…

Ejercicio U6_B2A_E3: Crea un interface MiLista para que funcione el siguiente main()




```
class App{
    public static void main(String[] args){
        MiLista ml1=new MiListaEnlazada();
        MiLista ml2= new ArrayListCasero();

        for(int i =20;i <30;i++){
            ml1.insertar(i);
            ml2.insertar(i);
        }
        System.out.print("Lista 1: ");
        for(int i=0;i<ml1.tamano();i++){
            System.out.print(ml1.obtener(i)+" ");
        }
        System.out.print("\nLista 2: ");
        for(int i=0;i<ml2.tamano();i++){
            System.out.print(ml2.obtener(i)+" ");
        }
    }
}
```

En este caso usar desde App el interface es una buena práctica de programación ya que aisla al código de App de una implementación concreta. Si se utiliza una implementación basada en array y el programador se arrepiente porque no obtiene el rendimiento esperado simplemente cambiando el new su código pasa a utilizar otra implementación.