Unidad III

TIPOS DE DATOS ABSTRACTOS

Programación Orientada a Objetos

Lenguaje Java

Año 2005

> "Pesemos la ganancia y la pérdida, considerando cara que Dios existe. Si ganáis, ganáis todo. Si perdéis, no perdéis nada. Apostad, pues, a que Dios existe, sin vacilar."
>
> Blas Pascal

Autor: Ing. Tymoschuk, Jorge
Colaboración: Ing. Guzmán, Analía

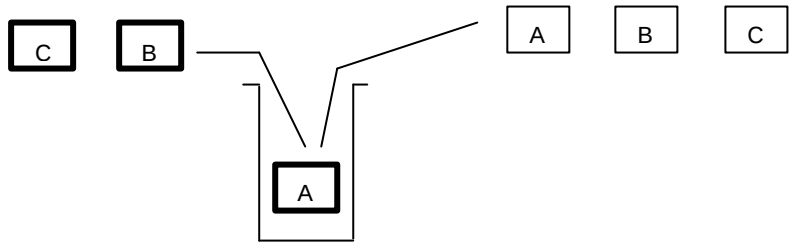
Indice (Unidad III, AED 2005)

PILAS, COLAS y LISTAS

Introducción: En este capítulo vemos en detalle las estructuras de datos pilas, colas y listas.
Las dos primeras son estructuras de datos que almacenan y recuperan sus elementos atendiendo a un estricto orden. Las pilas se conocen también como estructuras **LIFO** *(Last in, First-out),* último en entrar-primero en salir), las colas como estructuras **FIFO** *(First in, First out),* primero en entrar-primero en salir.

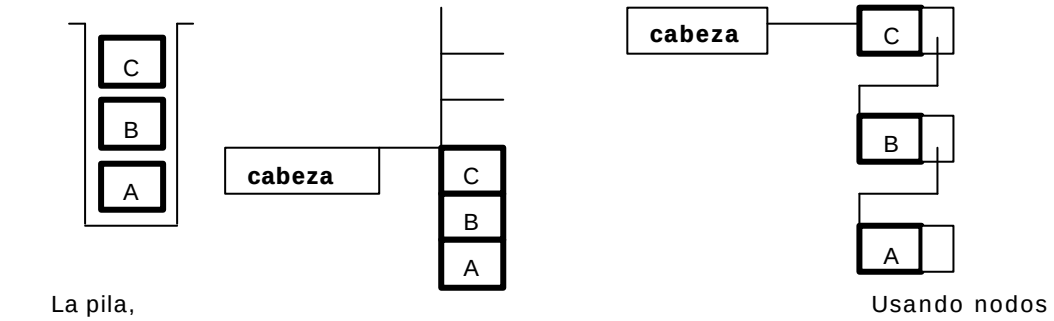
CONCEPTO DE PILA: Una **pila (stack)** es una colección ordenada de elementos a los que solo se puede acceder por un único lugar o extremo. Los elementos de la pila se añaden o se quitan (borran) de la misma sólo por su parte superior **(cima).** Entonces**, Insertar y Excluir son las únicas operaciones lícitas (posibles) en una pila.**

Al operarse únicamente por la parte superior de la pila, al excluir elementos retiramos el que está en la cima, el último que incluimos. Si repetimos esta operación retiraremos el penúltimo, el antepenúltimo, o sea en orden inverso al que los incluimos. Supongamos que tenemos las letras 'A', 'B' y 'C', las incluimos y luego excluimos dos de ellas, necesariamente 'C' Y 'B'. Gráficamente:



La operación de **Insertar** sitúa un elemento dado en la cima de la pila y **Excluir** lo elimina o quita

El lenguaje C (o C++) no tiene sentencias específicas para tratamiento de pilas, a diferencia de los arrays, donde si tiene, y mucho. Para trabajar con pilas debemos "pedir prestado" recursos a otras estructuras. Se pueden implementar pilas en arrays, en archivos, con nodos vinculados por punteros. Gráficamente, si tenemos 'A', 'B', 'C' "apiladas":



La pila, Conceptualmente — Usando Array — Usando nodos vinculados

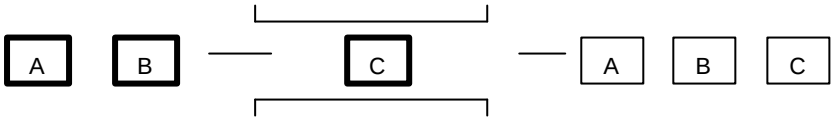
En la implementación con arrays, **cabeza** es el índice del último casillero ocupado, (o del primero disponible). Es simple, pero tiene el inconveniente de soportar una estructura típicamente dinámica (la pila) con una estática (el array): El array puede ser demasiado grande, o pequeño.

En la implementación con nodos vinculados **cabeza** es un puntero al primer nodo de esta estructura lineal, el cual contiene siempre el último elemento insertado. Su implementación requiere un poco mas de trabajo, pero al ser ambas estructuras del mismo tipo no existen los inconvenientes de la anterior. Por ello, esta es la que veremos en detalle.

La pila es una estructura importante en computación. Algunos de sus usos, transparentes para usuarios de un lenguaje de programación, es el retorno automático al punto en que una función fue invocada. Supongamos que main() invoca a func1(); dentro de func1() se invoca a func2(); dentro de func2() se invoca a func3(), allí procesamos y llegamos a la sentencia return. Una pila es la que contiene la información para que el retorno sea a un punto del cuerpo de func2(), luego de dirigirá nuestro retorno a func1(), y finalmente al main().

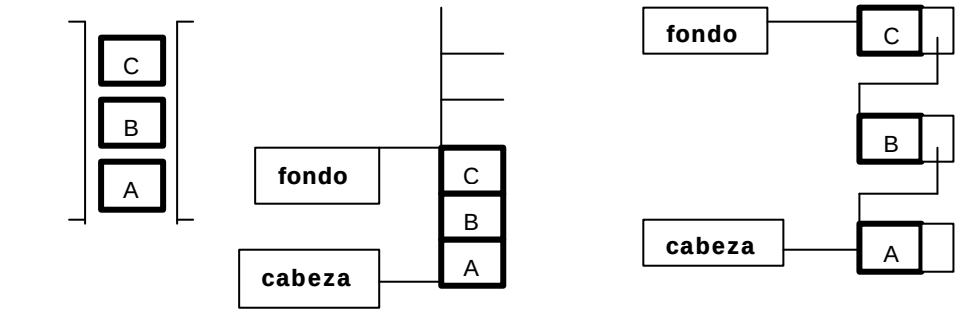
CONCEPTO DE COLA: Una **cola** es una estructura de datos que almacena elementos en una fila (uno a continuación de otro). Cada elemento se inserta en la cola (parte final) de ¡a fila y se suprime o elimina por el frente (Cabeza). de la fila Las aplicaciones utilizan una cola para almacenar y liberar elementos en su orden de aparición o concurrencia. **Insertar y Excluir son las únicas operaciones lícitas (posibles) en una cola.** La única diferencia con la pila es que estas operaciones se realizan en extremos opuestos. **Insertar** por el fondo, **excluir** por el frente.

Entonces **Insertar y Excluir** se realizan en el mismo orden. Un buen ejemplo de cola es el de personas esperando utilizar el servicio público de transporte. (Si dijéramos un mal ejemplo también es verdad). La gestión de tareas de impresión en una única impresora de una red de computadoras es otro ejemplo. Supongamos que tenemos las letras 'A', 'B' y 'C', las incluimos y luego excluimos dos de ellas, necesariamente 'A' Y 'B'. Gráficamente:



frente (cabeza) fondo

Como en el caso anterior, el lenguaje C (o C++) tampoco tiene sentencias específicas para tratamiento de colas. Para trabajar con colas debemos "pedir prestado" recursos a otras estructuras. Se pueden implementar colas en arrays, en archivos, con nodos vinculados por punteros. Gráficamente, si tenemos 'A', 'B', 'C' "encoladas":



La cola, conceptualmente

Usando array

Usando nodos vinculados

En la implementación con arrays, **fondo** es el índice del último casillero ocupado, (o del primero disponible). Es simple, pero tiene dos inconvenientes:

- de soportar una estructura típicamente dinámica (la cola) con una estática (el array)
- al excluir elementos del array lo hacemos desde el lado de la cabeza, con lo que comienzan a existir casilleros libres al inicio del array. Para el array, la cabeza de la cola se desplaza hacia atrás. Este inconveniente se soluciona "simulando" que el arrray es circular, pero sigue vigente el primer inconveniente.

En la implementación con nodos vinculados **cabeza** es un puntero al primer nodo de esta estructura lineal, el cual contiene siempre el primer elemento insertado y **fondo** al último. Esta implementación requiere un poco mas de trabajo, pero al ser ambas estructuras del mismo tipo no existen los inconvenientes de usar arrays. Por ello, esta es la que veremos en detalle.

En general, la cola es la estructura de gestión de recursos de alguna manera escasos. Impresoras que atienden a un grupo de estaciones de trabajo en una red, tablas de bases de datos que varios usuarios pretenden actualizar en forma exclusiva, etc, etc.

CONCEPTO DE LISTA.

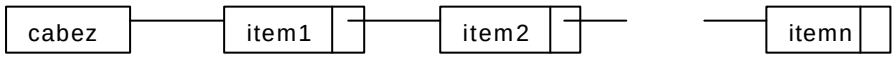
Una lista es una colección o secuencia de elementos dispuestos uno detrás de otro.
Una lista, por ejemplo la que hacemos para hacer las compras de la semana (quincena, mes) en el súper. Si somos ordenados y eficientes, hacemos esta lista de manera que en una sola recorrida, con un mínimo de tiempo y pasos, compramos todo lo anotado. Pero he aquí que llega nuestro cónyuge, ve nuestra lista, tacha las dos primeras líneas y las tres ultimas, luego incluye nuevas líneas en diferentes puntos de la lista, modifica otras líneas (esta marca es mas barata, solo esta sirve), etc. Una lista debe soportar un comportamiento de este tipo. De hecho, ni pensar en un array. (Si Ud. cree que es fácil, pruebe hacerlo...) En cambio, una estructura lineal de nodos vinculados por punteros no tiene ningún problema. Es mas, es tan adecuada que se confunden los conceptos: la **lista de los items** que necesitamos comprar con el de **lista enlazada** de nodos donde vamos a implementar la de items.

Una vez planteado que es lo que vamos a tratar (Pilas, Colas, Listas) y como (Implementándolas en estructuras lineales de nodos vinculados por punteros), definamos un par de puntos.

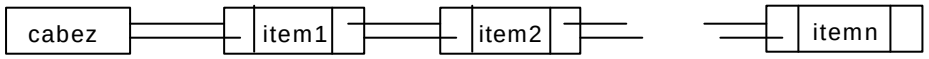
Nodo: Un nodo es una combinación de una parte de datos y un enlace (puntero) al siguiente nodo. Para la parte de datos, ya estamos hablando de items, nos viene bien usar la clase Items (Código/valor), que ya usamos en búsqueda y ordenamiento. Con lo cual, nuestra clase Nodo contendrá (relación **tiene un**) un objeto Item y un puntero al próximo nodo.

Lista enlazada: se compone de una serie de nodos enlazados mediante punteros.
Constructivamente, pueden ser de varios tipos.

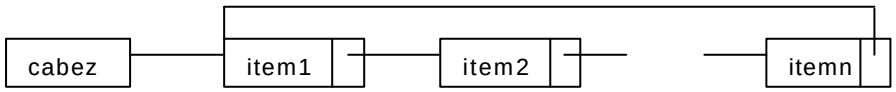
Lista simplemente enlazada: Cada nodo contiene un puntero que vincula al próximo. En el último, el puntero es nulo. Solo podemos avanzar. Gráficamente:



Lista doblemente enlazada: Cada nodo contiene dos enlaces, uno a su nodo predecesor y otro a su sucesor. Podemos avanzar y retroceder. Gráficamente:

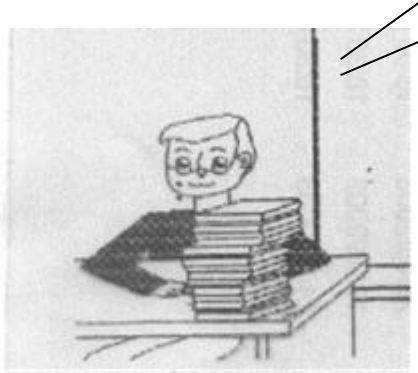


Lista circular simplemente enlazada: Variante de la lista simplemente enlazada, su último nodo enlaza al primero.

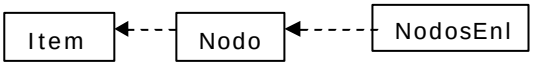


Por simplicidad, y porque raramente se necesita de otra cosa, usaremos la **Lista simplemente enlazada**.
La clase NodosEnl es entonces el ancestro común, la clase base de Pila, Cola y Lista. Y que

Bueno, y como hacemos todo esto en objetos, profe ?



*Bueno, hay que ir viendo que relaciones tenemos. Ya dijimos que un Nodo **tiene un** Item. Y que una lista enlazada se compone, es decir, tiene nodos, de nuevo relación **tiene un.** Hasta aquí tendríamos:*



*La lista enlazada (class NodosEnll) debe ser el **ancestro común** de Pila, Cola, Lista, para que estas clases cumplan la relación **es un** con ListEnl. Y que atributo común tienen todas ellas ? Sin duda, el puntero de entrada, puntero a nodo, **cabeza o frente**, como prefieran llamarlo ...*

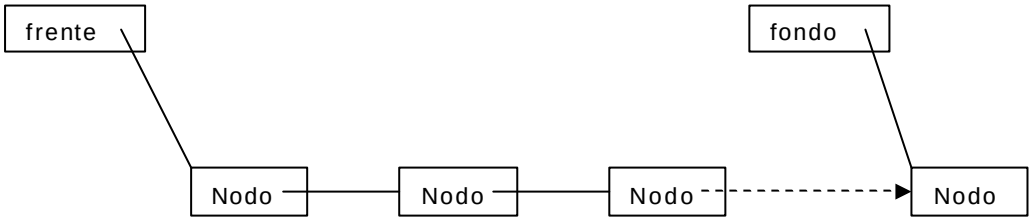
comportamiento común tienen todas ellas ?. Veamos un poco:

	Pila	Cola	Lista
Recorrer nodos, procesando información contenida en ellos ?	no	no	si
Incluir nodos ?	frente	fondo	Todo
Excluir nodos ?	frente	frente	Todo
Verificar si la estructura está vacía ?	si	si	si

Vemos que hay bastante comportamiento común pero también diferenciado.

Implementando pilas, colas y listas

Usaremos, para todo, una única estructura de nodos enlazados.



Nuestra clase Nodo

```
import Item;
class Nodo{   // Nodo de una estructura enlazada simple
        protected Item item; //  Nodo tiene un Item y
        protected Nodo prox;  //  una referencia al próximo Nodo...

        public Nodo(){              // Constructor default
                this(0, 0, null);
        }

        public Nodo(int cod, float val, Nodo sig) {// Constructor parametrico
                item = new Item();
                item.setCodigo(cod);
                item.setValor(val);
                prox = sig;
        }

        public Item getItem(){return item;}      // Retornamos el objeto item
                                                 // almacenado en Nodo

        public void setItem(Item it){item = it;};        // Inicializamos idem

        public Nodo getProx(){return prox;} // Retornamos puntero al proximo nodo

        public void setProx(Nodo ptr){prox = ptr;}// Inicilizamos puntero prox

        public String toString(){   //       Exhibimos el contenido de Nodo
                String aux = ""+item.toString()+"\n";   // Usamos toString() de Item
                return aux;
        }
}
```

La clase NodosEnl // Nodos enlazados

```
import Nodo;
class NodosEnl{    //      Nodos enlazados mediante referencias
        protected Nodo frente, fondo;  //  Referencia de entrada
        protected boolean trace; // Trazamos ?
        public NodosEnl(){
                frente = null; trace = false;}   //   Un constructor sin argumentos;
        public boolean vacia(){return frente == null;} // No hay nodos
        public String toString(){
                String aux = "NodosEnl contiene \n";
                for(Nodo ptr = frente; ptr!=null; ptr=ptr.getProx())
                        aux+= ptr.toString();
                return aux;
        }
        public void setTrace(boolean trazar){trace = trazar;}
        public void inclFirst(int cod, float val){
                frente = new Nodo(cod,val,frente);
                if(frente.getProx() == null)  // Primer nodo
                        fondo = frente;
                if(trace)
                        System.out.print("incluido "+frente);
        }
        public Nodo exclFirst(){
                if(vacia()) return null; // Retornamos referencia nula
                Nodo ptr = frente;    // Guardamos en aux la referencia del primer nodo
                frente = frente.getProx(); // Redirecionamos frente, ➔ siguiente nodo
                if(frente == null)      //      No hay ya nodos ...
                        fondo = null;
                if (trace)
                        System.out.print("excluido "+ptr);
                return ptr;                     //      Retornamos referencia nodo liberado
        }
        public void demo(){
                System.out.println("Demo class NodosEnl");
                setTrace(true);
                inclFirst(10,20);
                inclFirst(15,15);
                inclFirst(20,15);
                exclFirst();
                System.out.print(this);
                System.out.println("Demo NodosEnl finished !!!");
        }

        public static void main(String args[]){
                NodosEnl obj = new NodosEnl();
                obj.demo();
        }
}
```

```
Demo class NodosEnl

incluido 10 - 20.0
incluido 15 - 15.0
incluido 20 - 15.0
excluido 20 - 15.0
NodosEnl contiene
15 - 15.0
10 - 20.0
Demo NodosEnl finished !!!
Process Exit...
```

La clase Pila

```
import NodosEnl;
class Pila extends NodosEnl{ //  Implementando Pila a partir de NodosEnl

        public Pila(){
                super();
        }

        public void demo(){
                System.out.println("Demo class Pila");
                setTrace(true);
                for(int i= 1;i<=5;i++)
                        inclFirst(i,(float)i);
                for(int i = 1;i<=3;i++)
                        exclFirst();
                System.out.print(this);
                System.out.println("Demo Pila finished !!!");
        }

        public static void main(String args[]){
                Pila pila = new Pila();
                pila.demo();
        }
}
```

```
Demo class Pila

incluido 1 - 1.0
incluido 2 - 2.0
incluido 3 - 3.0
incluido 4 - 4.0
incluido 5 - 5.0
excluido 5 - 5.0
excluido 4 - 4.0
excluido 3 - 3.0
NodosEnl contiene
2 - 2.0
1 - 1.0
Demo Pila finished !!!

Process Exit...
```

La clase Cola

```
import Pila;
class Cola extends Pila {   //  Implementando Cola a partir de Pila
        public Cola(){              //      Un constructor sin argumentos
                super();
        }
        public void inclLast(int cod,float val){
                Nodo aux;
                if (frente == null){ //         no teníamos nodos, este es el primero
                        frente = aux = new Nodo(cod,val,null);
                        fondo = frente;
                }else{          //  Ya teniamos, este es uno mas ...
                        aux = new Nodo(cod,val,null);
                        fondo.setProx(aux);
                        fondo = aux;
                }
                if(trace)
                        System.out.print("incluido "+aux);
        }
        public void demo(){
                System.out.println("Demo class Cola");
                setTrace(true);
                for(int i= 1;i<=5;i++)                  // Incluyo
                        inclLast(i,(float)i);                   // 5 nodos
                for(int i = 1;i<=3;i++)                 // Excluyo
                        exclFirst();                            // solo 3 nodos
                for(int i= 1;i<=2;i++)                  // Incluyo
                        inclLast(i+10,(float)i+10);             // mas 2 nodos
                System.out.print(this);
                for(int i = 1;fondo!=null;i++)                  // excluyo
                        exclFirst();                            // Toda la co
                System.out.println("Demo Cola finished !!!");
        }
        public static void main(String args[]){
                Cola cola = new Cola();
                cola.demo();
        }
}
```

```
Demo class Cola

incluido 1 - 1.0
incluido 2 - 2.0
incluido 3 - 3.0
incluido 4 - 4.0
incluido 5 - 5.0
excluido 1 - 1.0
excluido 2 - 2.0
excluido 3 - 3.0
incluido 11 - 11.0
incluido 12 - 12.0
NodosEnl contiene
4 - 4.0
5 - 5.0
11 - 11.0
12 - 12.0
excluido 4 - 4.0
excluido 5 - 5.0
excluido 11 - 11.0
excluido 12 - 12.0
Demo Cola finished !!!
Process Exit...
```

La clase Lista

```
import NodosEnl;
class Lista extends Pila{ //  Implementando Lista a partir de Pila
        private Nodo shadow;

        public Lista(){
                super();
                shadow = null;
        }

        protected boolean existe(int codPar){
                boolean exis = false;                   // A priori, suponemos no existe
                Nodo ptr = frente; // Puntero para recorrer la lista.
                Nodo sombra = frente; // Puntero sombra, irá siempre atras de ptr
                int codNod;
                if (vacia())  shadow = null;      //  La lista no tiene nodos
                else{             //  Tenemos una lista con nodos
                        for(;ptr!=null;sombra = ptr, ptr = ptr.getProx()){
                                codNod = ptr.getItem().getCodigo();
                                if(codNod == codPar){ // hemos encontrado igual, basta
                                                      // de buscar ...
                                        exis = true; // existencia verdadera ...
                                        if(ptr == frente)       // ptr apunta al primer nodo,
                                                shadow = null; // código parámetro es ==
                                                               // que el del 1er nodo
                                        else    // ptr apunta a un nodo intermedio o final,
                                                shadow = sombra;// Referenciamos la
                                                                //  posicion del nodo anterior
                                        break;
                                }     //  if(codNod == codPar)
                                if(codNod > codPar){ // ya no vamos a encontrar igual,
                                                     // basta de nuevo
                                        if(ptr == frente)       // ptr apunta al primer nodo,
                                                shadow = null; // código parámetro es < que
                                                               // el del 1er nodo
                                        else    // ptr apunta a un nodo intermedio o final,
                                                shadow = sombra;// Referenciamos la
                                                                // posicion del nodo anterior
                                        break;
                                }     // if(codNod > codPar
                        } // for
                        if(ptr == null)  // Recorrimos lista completa, codPar es mayor al
                                         // del último nodo
                                shadow = sombra; //  posicion del último nodo
                }       // else if (vacia())
                return exis;
        }


/* el método existe() tiene una doble funcionalidad:

        1) retorna
                true si el  código del item parámetro existe
                false en caso contrario
```

```
        2) inicializa la referencia shadow

                        shadow = null cuando:
                                la lista está vacia
                                el código del item parámetro es menor al del primer nodo
                                el codigo parametro = codigo nodo en el primer nodo

                        shadow = sombra (puntero que sique "pegado" a ptr)
                                el código nodo  = código parámetro, no en el primer nodo
                                el código nodo  > código parámetro
                                el código parámetro > al del último nodo        */

private void inclOrd(int codPar, float vlr){ // Inserta nodos. Lista ordenada
        Nodo ptr;
        boolean exist = existe(codPar);
        if(exist){      // ya tenemos ese código, no podemos re incluirlo
                if(trace)System.out.println("inclOrd(), codigo " + codPar + " ya existe");
        }else{          // No existe el código del item parámetro, podemos incluir
                        if(shadow != null){    // es una inserción intermedia o al fin
                                ptr = shadow.getProx(); // puntero al próximo
                                shadow.setProx(new Nodo(codPar,vlr,ptr));
                                if(trace) System.out.println("inclOrd() " + codPar + "
                                                                        incluido");
                        }else  // Es una inserción al inicio de la lista o la lista está vacía
                                inclFirst(codPar, vlr);// usamos el método heredado
        }   // else del if(exist)
}

protected Nodo exclOrd(int codigo){    //Excluye nodos.Lista ordenada por código
        Nodo ptr;               // para puntero al nodo
        boolean exist = existe(codigo);  // Verificamos su existencia
        Nodo nodo = null;               // Lo necesitamos para almacenar el nodo excluido
        if(exist)       // Tenemos igual codigo en lista, entonces podemos excluir
                if(shadow != null){     // Debemos excluir un nodo intermedio o final
                        nodo = shadow.getProx(); // puntero al nodo a excluir
                        ptr  = nodo.getProx();          // puntero al siguiente al excluido
                        shadow.setProx(ptr);// Vinculamos anterior al siguiente al excl.
                        if(trace) System.out.println("exclOrd(), codigo " + codigo + "
                                                                        excluido");
                }else                           // El nodo a excluir es el primero
                        nodo = exclFirst();    // usamos el método adecuado, el heredado
                if(trace) System.out.print("exclFirst(), codigo " + codigo + " excluido");
        else                            // No existe el item buscado
                if(trace) System.out.print("exclOrd(), codigo " + codigo + " no existe");
        return nodo;                    // retornamos referencia al nodo excluido (o no)
}

protected void actValor(int codigo, float val){ // Actualiza valor del item del nodo.
        Nodo ptr;                               // para puntero al nodo
        boolean exist = existe(codigo);         // Verificamos su existencia
        Nodo ptrAct;            // Puntero al nodo cuyo valor de item actualizaremos
        if(exist){      // Tenemos código en lista, podemos actualizar valor asociado
                if(shadow != null)      // Actualizar en un nodo intermedio o final
                        ptrAct = shadow.getProx(); // ptrAct con la dirección correcta
                else            // El valor a actualizar está en el item del primer nodo
```

```
                ptrAct = frente;
            //  ptrAct apunta al nodo cuyo valor queremos actualizar
            Item it = new Item(codigo,val); // Generamos objeto item
            ptrAct.setItem(it);          // reemplazar el del nodo
            if(trace) System.out.println("actValor(), código " + codigo + " a " + val);
        }else                            // No existe el item buscado
            if(trace)System.out.print("actValor(), código " + codigo + " no existe");
    }

    protected void implDemo1(){
        System.out.println("implDemo1() class Lista");
        System.out.println("Incluimos 4 nodos - inclFirst()");
        for(int i=4;i>0;--i)inclFirst(i*2,(float)(i*4));
        System.out.println(this);
        System.out.println("Borramos el primero");
        exclFirst();
        System.out.println(this);
        System.out.println("Demo implDemo1() terminado !!!");
    }

    public static void main(String args[]){
        Lista list = new Lista();
        list.implDemo1();
    }
} //  Class Lista
```

```
implDemo1() class Lista
Incluimos 4 nodos - inclFirst()

NodosEnl contiene
2 - 4.0
4 - 8.0
6 - 12.0
8 - 16.

Borramos el primero

NodosEnl contiene
4 - 8.0
6 - 12.0
8 - 16.0

Demo implDemo1() terminado !!!
Process Exit...
```

```
    protected void implDemo2(){
        System.out.println("implDemo2() class Lista");
        System.out.println("Incluimos 4 nodos - inclOrd()");
        for(int i=1;i<5;++i)inclOrd(i*2+1,(float)(i*4.2));
        System.out.println(this);
        System.out.println("Borramo s el primero");
        exclFirst();
        System.out.println("Borramos el código 7");
        exclOrd(7);
        System.out.println(this);
        System.out.println("Demo implDemo2() terminado !!!");
    }

    public static void main(String args[]){
        Lista list = new Lista();
        list.implDemo 2();
    }
```

```
implDemo2() class Lista
Incluimos 4 nodos - inclOrd()
NodosEnl contiene
3 - 4.2
5 - 8.4
7 - 12.6
9 - 16.8

Borramos el primero
Borramos el código 7

NodosEnl contiene
5 - 8.4
9 - 16.8
Demo implDemo2() terminado !!!
Process Exit...
```

```
protected void implDemo3(){
        System.out.println("implDemo3() class Lista");
        System.out.println("Incluimos 4 nodos - inclFirst()");
        for(int i=4;i>1;--i)inclFirst(i*2,(float)(i*4));
        System.out.println(this);
        System.out.println("Incluimos 3 nodos - inclOrd()");
        for(int i=1;i<5;++i)inclOrd(i*2+1,(float)(i*4.2));
        System.out.println(this);
        System.out.println("Demo implDemo3() terminado !!!");
}

public static void main(String args[]){
        Lista list = new Lista();
        list.implDemo3();
}
```

```
implDemo3() class Lista
Incluimos 3 nodos - inclFirst()
NodosEnl contiene
4 - 8.0
6 - 12.0
8 - 16.0
Incluimos 4 nodos - inclOrd()
NodosEnl contiene
3 - 4.2
4 - 8.0
5 - 8.4
6 - 12.0
7 - 12.6
8 - 16.0
9 - 16.8
Demo implDemo3() terminado !!!
Process Exit...
```

```
protected void implDemo4(){
        System.out.println("implDemo4() class Lista");
        System.out.println("Incluimos 5 nodos - inclOrd()");
        for(int i=1;i<6;++i)inclOrd(i*2+1,(float)(i*4.2));
        System.out.println(this);
        System.out.println("Borraremos el codigo 5");
        exclOrd(5);
        System.out.println("Borraremos el código 9");
        exclOrd(9);
        System.out.println("Borraremos el código 8 ");
        exclOrd(8);
        System.out.println(toString());
        System.out.println("Demo implDemo4() terminado !!!");
}

public static void main(String args[]){
        Lista list = new Lista();
        list.implDemo4();
}
```

```
implDemo4() class Lista
Incluimos 5 nodos - inclOrd()
NodosEnl contiene
3 - 4.2
5 - 8.4
7 - 12.6
9 - 16.8
11 - 21.0
Borraremos el codigo 5
Borraremos el código 9
Borraremos el código 8
NodosEnl contiene
3 - 4.2
7 - 12.6
11 - 21.0
Demo implDemo4() terminado !!!
Process Exit...
```

```
protected void implDemo5(){
        System.out.println("implDemo5() class Lista");
        System.out.println("Incluiremos 3 nodos - inclOrd()");
        for(int i=1;i<4;++i)inclOrd(i*2+1,(float)(i*4.2));
        System.out.println(this);
        System.out.println("Actualizaremos el código 5");
        actValor(5,10);
        System.out.println("Actualizamos el código 3");
        actValor(3,20);
        System.out.println("Actualizamos el código 8");
        actValor(8,20);
        System.out.println(this);
        System.out.println("Demo implDemo5()terminado !!!");
}
public static void main(String args[]){
        Lista list = new Lista();
```

```
implDemo5() class Lista
Incluiremos 3 nodos - inclOrd()
NodosEnl contiene
3 - 4.2
5 - 8.4
7 - 12.6
Actualizaremos el código 5
Actualizamos el código 3
Actualizaremos el código 8
NodosEnl contiene
3 - 20.0
5 - 10.0
7 - 12.6
Demo implDemo5()terminado !!!
Process Exit...
```

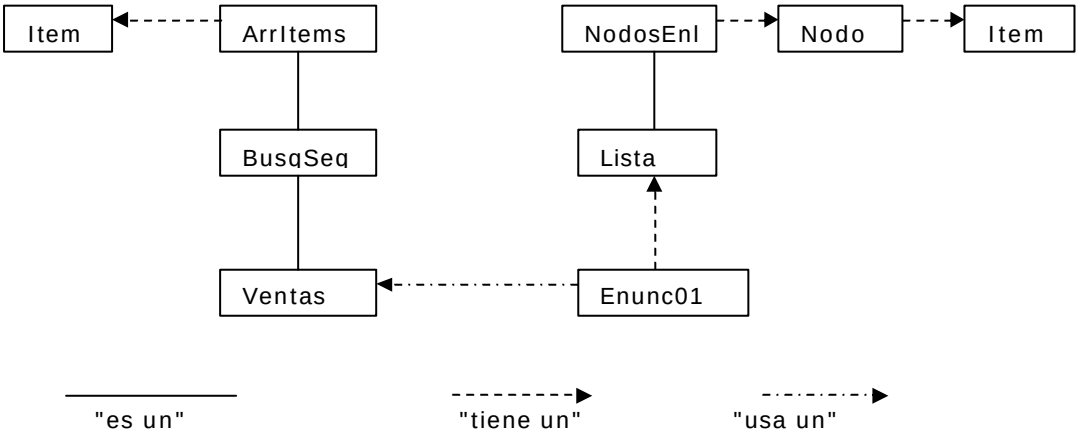
```
        list.implDemo5();
}
```

Todo está muy bonito, profe, pero como hacemos esos programas de antes, por ejemplo tenemos un array y debemos generar una lista con los códigos que ...

Me quieren poner en un aprieto, me parece ... Hum... veamos... tenemos 10 minutos ? Supongamos que tenemos las ventas de un comercio, las del viernes, las del sábado, cada una es un array de items código - valor. Dos arrays como entrada de datos. Y el tema podía ser detectar que se vende siempre, que se vende en fin de semana, algo así. Dos listas que se generan en el confronto de los arrays. Les parece que puede ser ? ...

COMPOSICION USANDO ARREGLOS Y LISTAS

El diagrama de clases:



/* **Enunciado 01**

Se tienen dos objetos arrays de Items, class ArrItems
Ambos contienen las ventas de un comercio, en dos días sucesivos de una semana cualquiera.
El objeto **vtasVier** contiene las ventas del viernes.
El objeto **vtasSab** contiene las ventas del sábado.

Se pide:
1) - Generar un objeto lista de lo vendido el sábado que no se vendió el viernes. Estamos queriendo detectar que es lo que se vende exclusivamente el fin de semana. Llamamos a este objeto **vtasFSem**. Queremos que en este objeto **vtasFSem** se registren por código. Si en un mismo código hay varias, registrar la última.

2) - Generar un objeto lista de lo vendido el sábado que tambien se vendió el viernes. Estamos queriendo detectar que es lo que se vende habitualmente, lo que la gente consume siempre. Llamamos a este objeto **vtasEver**. A diferencia del punto anterior, queremos que cada item del objeto **vtasSab** se incluya

en **vtasEver** directamente, no solo la última.
*/

/* **Estrategia de resolución.**

En principio, tenemos gran parte de comportamiento listo, desarollado en esta unidad y anteriores.
En la parte de array de items, podemos usar **protected boolean existe(int clave)** // Existe la clave parámetro, class BusqSec.) ?
En la parte de listas, en class lista, tambien tenemos protected **void inclOrd(int codPar, float vlr)** //Inserta nodos. Lista ordenada por código. Se preocupa por la unicidad del codigo del item.)

Evidentemente, es un problema que se resuelve con el concurs punto01o de bastantes clases. Para ordenar un poco esto, definimos que todo lo que haremos está dentro de una clase, naturalmente class Enunc01. Esta clase tendrá por lo menos dos métodos, que implementarán lo que pide en enunciado, que llamaremos **punto01** y **punto02**.

Y como relacionamos estas clases? Comencemos por la salida, las listas.
Nuestra class **Enunc01** debe
. heredar (Relación "es un"), de class Lista ?.
. debe tener (Relación "tiene un") class Lista ?.
. simplemente usar objetos lista ?

Es evidente que class **Enunc01** no es una especialización de Lista, entonces podríamos optar por cualquiera de las otras dos alternativas. Si suponemos que los objetos **vtasFSem** (punto01) y/o **vtasEver** (punto02) serán usados por otros futuros métodos de esta clase o herederas, necesitamos de la relación "**tiene un**"

La entrada de nuestro programa son los arrays de items. Una vez procesados, la información como la queremos está en las listas, no los necesitamos mas, entonces deberíamos deshacernos de ellos. Un buen lugar de definirlos es en los metodos **punto01** y **punto02**. O sea que **usaremos** objetos arrays de items, de la clase derivada **Ventas**, porque precisaremos de sus metodos.

Al trabajo.

```
import Ventas;
import Lista;
class Enunc01Pto01{
    protected Lista vtasFSem;

    public void punto01(){
        int codVSab;        // Codigo de un item vendido el sábado ...
        float valVSab;      // Y su valor
        boolean vendVie, yaVend;// Lo inicializa existe() de Lista al retornar
        Ventas vtasVie = new Ventas(10,'R'); // ventas del viernes,
        Ventas vtasSab = new Ventas(20,'R'); // ventas del sabado
        vtasFSem = new Lista();
        int venSab = vtasSab.getTalle(); // cantidad de ventas del sábado
        Item  item;
        vtasFSem.setTrace(true);
        for(int i=0;i<venSab;++i){ // Recorremos esas ventas de sábado
                item = vtasSab.item[i].getItem(); // Obtenemos un item vendido el sabado
                codVSab = item.getCodigo();         // ahora su código
                valVSab = item.getValor();   // y su valor
                vendVie   = vtasVie.existe(codVSab); // Veamos si tambien vendio el viernes
                if (!vendVie){ // No vendido el viernes, es venta fin de semana
                        yaVend = vtasFSem.existe(codVSab);// ya lo tengo en vasFSem
                        if (yaVend)                  // Este código ya estaba en vtasFSem
                                vtasFSem.actValor(codVSab,valVSab);  //lo actualizamos
                        else                    // No estaba,
```

```
                        vtasFSem.inclOrd(codVSab,valVSab);  // lo incluimos
                }
        } // for
        System.out.println("Demo Enunc01(), pto 1");
        System.out.println("Ventas tipo fin de semana");
        System.out.println(vtasFSem);
        System.out.println("Demo Enunc01, pto 1, finish !!!");
    }
    public static void main(String args[]) {
        Enunc01Pto01 pto = new Enunc01Pto01();
        pto.punto01();
    }
}
```

```
Primeros 10 de 10
 elementos Item
4 - 4.110216
8 - 5.5699077
9 - 4.902982
4 - 4.084873
8 - 7.1807923
3 - 1.6490803
3 - 0.8555745
1 - 4.2965527
4 - 8.453937
2 - 3.8001635
 Primeros 10 de 20
 elementos Item
13 - 14.013713
1 - 1.1455365
2 - 19.92075
19 - 9.822017
9 - 4.592097
10 - 3.9259176
15 - 0.52571803
11 - 5.8398447
10 - 11.2952385
10 - 14.728624
incluido 13 - 14.013713
inclOrd() 19 incluido
incluido 10 - 3.9259176
```

```
inclOrd() 15 incluido
inclOrd() 11 incluido
actValor(), código 10 a 11.2952385
actValor(), código 10 a 4.728624
inclOrd() 14 incluido
inclOrd() 18 incluido
actValor(), código 10 a 8.3007145
incluido 6 - 15.982461
incluido 0 - 9.244863
actValor(), código 14 a 5.828643

Demo Enunc01(), pto 1
Ventas tipo fin de semana
NodosEnl contiene
0 - 9.244863
6 - 15.982461
10 - 8.3007145
11 - 5.8398447
13 - 14.013713
14 - 5.828643
15 - 0.52571803
18 - 0.33889037
19 - 9.822017

Demo Enunc01, pto 1, finish !!!
Process Exit...
```

```
import Ventas;
import Lista;
class Enunc01Pto02{
    protected Lista vtasEver;

    public void punto02(){
        int codVSab;        // Codigo de un item vendido el sábado ...
        float valVSab;      // Y su valor
        boolean vendVie, yaVend;// Lo inicializa existe() de Lista al retornar
        Ventas vtasVie = new Ventas(10,'R'); // ventas del viernes,
        Ventas vtasSab = new Ventas(20,'R'); // ventas del sabado
        vtasEver = new Lista();
        int venSab = vtasSab.getTalle(); // cantidad de ventas del sábado
        Item  item;
        vtasEver.setTrace(true);
        for(int i=0;i<venSab;++i){ // Recorremos esas ventas de sábado
                item = vtasSab.item[i].getItem(); // Obtenemos un item vendido el sabado
                codVSab = item.getCodigo();         // ahora su código
                valVSab = item.getValor();   // y su valor
```

```
                vendVie   = vtasVie.existe(codVSab); // Veamos si tambien vendio el viernes
                if (vendVie)   // Vendido el viernes, es venta ever (Siempre)
                               vtasEver.inclFirst(codVSab,valVSab);  // lo incluimos

        } // for
        System.out.println("Demo Enunc01(), pto 2");
        System.out.println("Ventas tipo ever (Siempre)");
        System.out.println(vtasEver);
        System.out.println("Demo Enunc01, pto 2, finish !!!");
    }

    public static void main(String args[]) {
        Enunc01Pto02 pto = new Enunc01Pto02();
        pto.punto02();
    }
}
```

```
Primeros 10 de 10
 elementos Item
5 - 6.34369
8 - 5.499295
1 - 2.6919718
4 - 1.7708089
8 - 3.7486262
9 - 3.7431767
9 - 4.7811975
6 - 1.4552411
7 - 1.5517539
3 - 6.4907618
 Primeros 10 de 20
 elementos Item
2 - 8.641394
2 - 7.1809883
19 - 11.150965
1 - 5.7408614
18 - 14.990596
6 - 7.011776
0 - 2.3762622
10 - 19.34552
13 - 12.149126
19 - 9.562794
```

```
incluido 1 - 5.7408614
incluido 6 - 7.011776
incluido 5 - 3.6529748
incluido 9 - 16.026426
incluido 9 - 2.2708728

Demo Enunc01(), pto 2
Ventas tipo ever (Siempre)
NodosEnl contiene
9 - 2.2708728
9 - 16.026426
5 - 3.6529748
6 - 7.011776
1 - 5.7408614
Demo Enunc01, pto 2, finish !!!
Process Exit...
```

Tipo de dato abstracto grafo

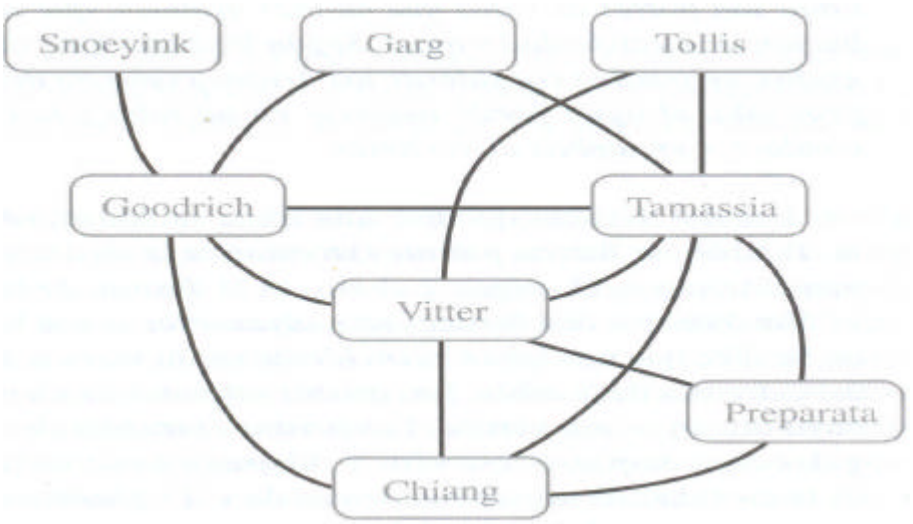
Desde el punto de vista abstracto, un *grajo* G no es más que un conjunto *V* de nodos y una colección *E* de arcos que unen los nodos. Así, un grafo es una forma de representar conexiones o relaciones entre pares de objetos de algún conjunto *V.*
Los arcos en un grafo pueden ser *dirigidos* o *no dirigidos.* Se dice que un arco *(u, v)* es *dirigido* de *u* a *v* si el par *(u,v)* es ordenado y *u* precede a *v.* Se dice que un arco *(u,v)* es *no dirigida* si el par *(u, v)* no es ordenado.

Ejemplo: *Se pueden visualizar las colaboraciones, entre los investigadores de cierta disciplina, si se forma un grafo cuyos nodos estén asociados con los investigadores mismos y cuyos arcos unen pares de nodos asociados con investigadores que fueron coautores de una publicación o un libro (véase la figura*). *Esas aristas son no dirigidas porque la coautoría es una relación* simétrica, *esto es, si A es coautor de algo con B, necesariamente B es coautor de algo con A. Diagrama en la pg. Siguiente.*

Tambien se puede asociar un programa orientado a objetos con un grafo cuyos nodos representan las clases definidas en el programa, y cuyos arcos indican la herencia entre clases. Hay un arco de un nodo v *a un nodo u si la clase para* v *extiende a la clase u. Esos arcos son dirigidos, porque la*

relación de herencia tiene sólo una dirección (es decir, es asimétrica).

Si todos los arcos de un grafo son no dirigidos, se dice del grafo que es un **grafo no dirigido**. En el caso de un ***grafo dirigido***, llamado también *dígrafo,* todas sus arcos son dirigidos. A un grafo que tiene arcos dirigidos y no dirigidos al mismo tiempo se le llama ***grafo mezclado***. Nótese que un grafo no dirigido o mezclado se puede convertir en uno dirigido, reemplazando a cada arco no dirigido *(u,v)* por el par de aricos dirigidos *(u,v)* y *(v,u).*



Ejemplo: *Se puede modelar un mapa de ciudad con un grafo cuyos nodos son los cruceros o las calles cerradas, y cuyas arcos son tramos de calles sin cruceros. Este grafo tiene arcos no dirigidos, que corresponden a calles con doble circulación, así como arcos dirigidos que corresponden a calles con un sentido de circulación. De esta forma, el grafo que modela un mapa de ciudad es un grafo mezclado.*

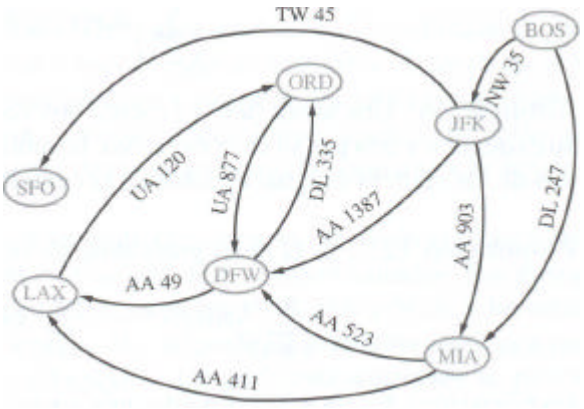
Los dos nodos que une un arco se llaman nodos *terminales* de ese arco. También se llaman *puntos extremos* del arco. Si un arco es dirigido, su primer extremo es el ***origen*** y el otro es el ***destino*** del arco.
Se dice que dos nodos son ***adyacentes*** si son los extremos del mismo arco. Se dice que un arco es ***incidente*** en un nodo, si el nodo es uno de los puntos extremos del arco. Los ***arcos salientes*** de un nodo son los arcos dirigidos cuyo origen es ese nodo. Los ***arcos entrantes*** de un nodo son los arcos dirigidos cuyo destino es ese nodo. El ***entreGrado*** de un nodo v, representado por deg(v), es la cantidad de arcos incidentes de v. El entrGr*ado* y el fueraDeGrado de un nodo v son la cantidad de arcos entrantes y salientes de v, que se representan por indeg(v) y outdeg(v), respectivamente.

Ejemplo: *Se puede estudiar el transporte aéreo trazando un grafo* G, *llamado* red de vuelo, *cuyos nodos se asocian con aeropuertos y cuyos arcos se asocian con vuelos (véase la figura*). *En el grafo* G, *los arcoss son dirigidos, porque determinado vuelo tiene una dirección específica de avance (desde el aeropuerto de origen hasta el aeropuerto de destino). Los extremos de un arco e en* G *corresponden, respectivamente, al origen y al destino del vuelo correspondiente a e. Dos aeropuertos son adyacentes en* G *si hay un vuelo que los una, y una arco e es incidente en un vértice* v *en* G *si el vuelo para e sale o va hacia el aeropuerto para* v. *Los arcos salientes de un nodo* v *corresponden a los vuelos de salida del aeropuerto de* v, *y las arcos entrantes corresponden a los vuelos hacia el aeropuerto de* v. *Por último, el grado entrante*(**Entregrado**) *de un nodo* v *de* G *corresponde a la cantidad de vuelos que tienen como destino al aeropuerto de* v, *y el grado saliente* **(FueraDe Grado)** *de un nodo* v *en* G *corresponde a la cantidad de vuelos que salen de él.*

La definición de un grafo se refiere al grupo de arcos como ***colección***, y no como *conjunto,* para permitir así que dos arcos no dirigidos tengan los mismos nodos extremos, y que dos arcos dirigidos tengan el mismo origen y el mismo destino. Esas arcos se llaman ***arcos paralelos* o *arcos múltiples***. Los arcos paralelos pueden estar en una red de vuelos (ver ejemplo), en cuyo caso los arcos múltiples entre el mismo par de nodos podrían indicar vuelos distintos que tienen la misma ruta a distintas horas del día. Otra clase especial de arcos es la que conecta un nodo consigo mismo. Esto es, se dice que un arco (no dirigido o

dirigido) es un ***autociclo*** si coinciden sus dos puntos extremos. Un autociclo puede presentarse en un grafo asociado a un mapa de ciudad (ver el ejemplo), y en ese caso correspondería a una calle que se curva y regresa a su punto de partida).

Figura: Ejemplo de un grafo dirigido que representa una red de vuelos. Los extremos del arco UA 120 son LAX y ORO; (LAX y ORO son adyacentes). El entreGrado de DFW es 3, y el fueraDeGrado de DFW 2.



Con pocas excepciones, como las que se acaban de mencionar, los grafos no tienen arcos paralelos ni autocic1os. Se dice que estos grafos son ***simples***. En consecuencia, se puede decir en cosas normales que los arcos de un grafo simple son un ***conjunto*** de pares de nodos (y no tan sólo una colección). En este capítulo supondremos que un grafo es simple a menos que se especifique otra cosa. Esta hipótesis simplifica la presentación de estructuras de datos y de algoritmos para grafos. Es fácil (y laboriosa) la extensión de los resultados de este capítulo a los grafos generales que admiten autocic1os y/o arcos paralelos.

En las proposiciones que siguen se examinan algunas propiedades importantes de los grafos y de la cantidad de arcos de un grafo. Estas propiedades relacionan la cantidad de nodos y de arcos entre sí, y a los grados de los nodos de un grafo.

Proposición 1 : *Si* G *es un grajo con m arcos,*

$$Sumatoria\ Grado(v) = 2m$$

Justificación': Un arco *(u, v)* se cuenta dos veces en la suma de arriba: una por su extremo *u* y una por su extremo *v.* Así, la aportación total de las arcos a los grados de los nodos es igual al doble de la cantidad de arcos. .

Proposición 2 : Si G *es un grafo dirigido con m arcos,*

$$Sumatoria\ entreGrado(v) = Sumatoria\ fueraDeGrado(v) = m$$

Justificación: En un grafo dirigido, un arco *(u,v)* contribuye con uno al grado saliente de su origen *u,* y con uno al grado entrante de su destino *v.* Así, la aportación total de los arcos a los grados salientes de los arcos (fueraDeGrado) es igual a la cantidad de arcos, y de igual manera para los grados entrantes (entreGrado). .

Algunas definiciones

Trayectoria **de un grafo:** secuencia de nodos altemantes que comienza en un nodo y termina en otro, en forma tal que cada arco es incidente a su nodo predecesor y sucesor.

Ciclo es una trayectoria tal que sus vértices inicial y final son iguales.

Trayectoria *simple* si cada vértice de ella es distinto.

Ciclo *simple* si cada vértice del ciclo es distinto, excepto el primero y el último.

Trayectoria dirigida es aquella en la que todas las aristas son dirigidas y se recorren a lo largo de su dirección.

Ciclo dirigido se define en forma análoga. Por ejemplo, en la red de vuelos de la figura 12.2, (BaS, NW 35, JFK, AA 1387, DFW) es una trayectoria simple, y (LAX, UA 120, aRD, UA 877, DFW, AA 49, LAX) es un ciclo simple dirigido.

Subgrafo de un grafo G es un grafo *H* cuyos nodos y arcos son subconjuntos de los vértices y aristas de G, respectivamente.

***Subgrafo de extensión* de G** es aquel que contiene a todos los vértices del grafo G.

Un grafo está *conectado* si, para dos vértices cualesquiera, hay una trayectoria entre ellos.

Si un **grafo G no está conectado**, sus subgrafos de conexiones máximas se llaman *componentes conectados* de G.

Un *bosque* es un grafo sin ciclos.

Un *árbol* es un bosque conectado, esto es, un grafo conectado sin ciclos. Este árbol no tiene raíz definida y se le llama *árbol libre.* Los componentes conectados de un bosque son árboles (libres). Un *árbol de extensión* de un grafo es un subgrafo de extensión que es un árbol (libre).

*Quizá el grafo del que más se habla hoy es **Internet**, cuyos nodos son computadoras y cuyos arcos (no dirigidos) son conexiones de comunicación entre pares de computadoras. Las computadoras y las conexiones entre ellas de un solo dominio forman un subgrafo de Internet.*

Métodos de grafo

Los grafos son un tipo de dato abstracto mucho más rico que los que se han descrito hasta ahora. La principal riqueza se debe a las diferentes clases de objetos que ayudan a definir un grafo, como son los nodos y los arcos. Para presentar los métodos del TDA grafo en forma lo más organizada posible, se dividen los métodos de grafo en tres categorías principales: métodos generales, métodos acerca de arcos dirigidos y métodos para actualizar y modificar grafos. Además, para simplificar la presentación, se representa con *v* una posición de nodo (o vértice), siendo *e* una posición de arco (o arista), y con *o* se representa un objeto (elemento) almacenado en un nodo o en un arco.

Como tipo de dato abstracto, un grafo es un contenedor posicional de elementos que se guardan en las arcos y los nodos del grafo. Es decir, las *posiciones* en un grafo son sus nodos y arcos. Se pueden guardar elementos en un grafo en sus arcos, nodos o ambos.

Estructuras de datos para grafos

El lenguaje java no tiene una estructura de datos para grafos, como la tiene para arreglos. Por ello es que el grafo es un tipo abstracto de datos. Hay varias formas de ver el TDA grafo con una estructura concreta de datos. Citamos algunas y como implementan el TDA grafo:

Lista de arcos:
- Un contenedor (una lista o un vector) para almacenar los nodos del grafo.
- Un contenedor de los arcos presentes

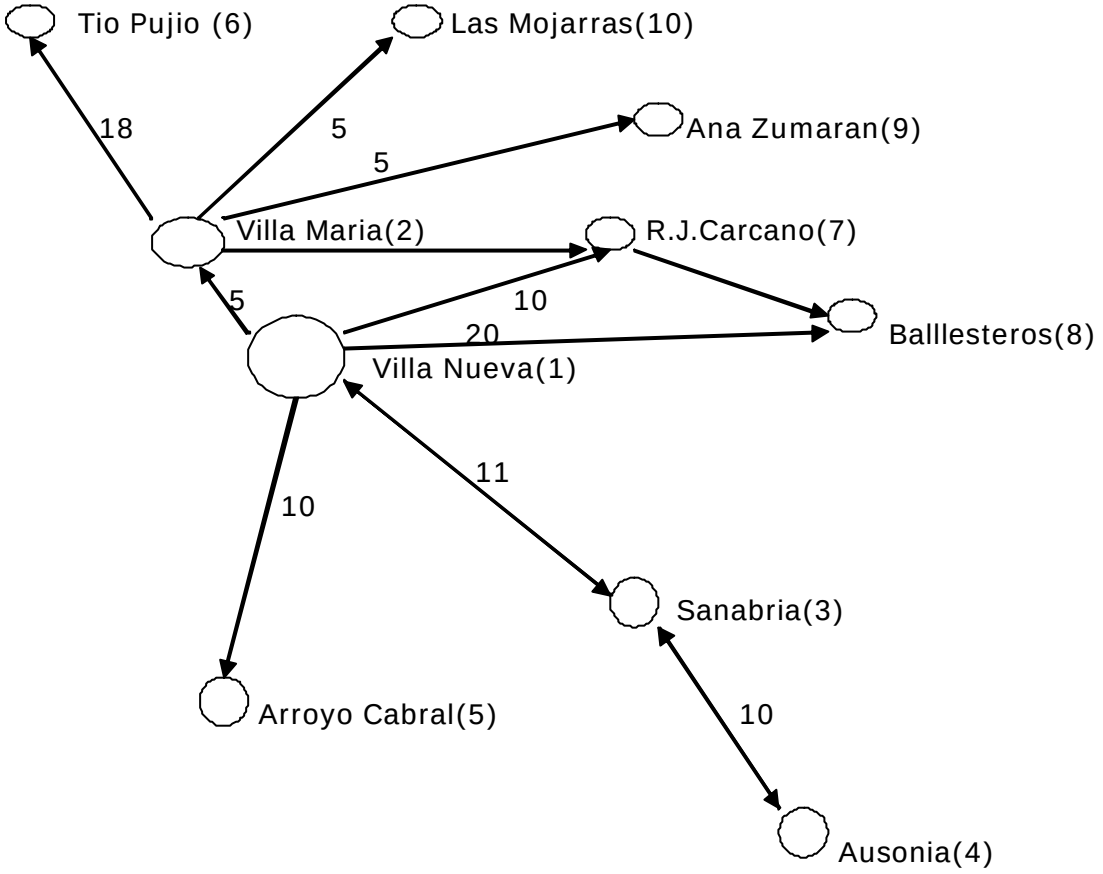
Lista de adyacencias
- Un contenedor (una lista o un vector) para almacenar los nodos del grafo.
- Un contenedor de las adyacencias presentes

Matriz de adyacencias.
- Un contenedor (una lista o un vector) para almacenar los nodos del grafo.
- Un casillero de la matriz por cada par de nodos (haya o no un arco entre ellos)

El lenguaje Java provee clases, interfaces, iteradores para implementar el TDA grafo en una forma muy profesional. (Lo mismo para arreglos, listas etc.) Esto excede ampliamente el propósito de esta

asignatura, que es enseñar a programar algoritmos. Asismismo, el tema Grafos visto en profundidad es toda una asignatura. Entonces lo que vamos a hacer es una **implementación práctica nuestra del TDA grafo** mediante una ***Matriz de adyacencias***.

Tomamos un conjunto de ciudades y pueblos del centro de la provincia de Córdoba. Sus nombres, distancias, son reales. No lo son los sentidos de circulación, esto para ejemplificar mejor algunos conceptos como los de entreGrado, fueraDeGrado, etc. En la implementación que sigue distancia no nula significará adyacencia, o sea existencia de un tramo de camino que se puede recorrer en el sentido indicado.



En nuestra implementación usaremos 3 clases.
La clase Ciudades (Nodos) contiene información y comportamiento relativo a ciudades.
La clase Tramos (Arcos) hace lo propio con los tramos que vinculan ciudades.
Grafo tiene como atributos Ciudades y Tramos.

En los constructores se usan arrays y matrices externas para evitar la carga de datos vía teclado.

Vamos al código.

```
class Ciudades{
        private int cuantas;
        private String[] ciudad;                    //  Array de cadenas

        public Ciudades(){   // Tomamos de un array de ctes literales
                String city[]={"","Villa Nueva","Villa Maria","Sanabria",
                                                "Ausonia","Arroyo Cabral","Tio Pujio",
                                                "Ramon J. Carcano","Ballesteros",
                                                "Ana Zumaran","Las Mojarras"};
                cuantas = city.length;
                ciudad = new String[cuantas];
                for (int i=0; i<cuantas; i++)
                        ciudad[i] = city[i];
```

```
        }

        public int existe(String city){
                for (int i=1; i<cuantas; i++)
                        if (ciudad[i] == city) return i;
                return 0;
        }

        public String toString(){
                String aux = "-- Ciudades --\n";
                aux+="nodo  ciudad\n";
                for(int i=1;i<cuantas;i++)
                        aux+=i + " - "+ciudad[i]+"\n";
                return aux;
        }

        public int getCuantas(){
                return cuantas;
        }

        public String getNombre(int nroCity){
                return ciudad[nroCity];
        }

        public int getNumero(String city){
                return existe(city);
        }

        public static void main(String args[]){
                Ciudades ciudades = new Ciudades();
                System.out.println(ciudades);
                System.out.println("Ausonia es la nro "+ciudades.getNumero("Ausonia"));
                System.out.println("La nro 8 es "+ciudades.getNombre(8));
        }
}
```

```
-- Ciudades --
nodo  ciudad
1 - Villa Nueva
2 - Villa Maria
3 - Sanabria
4 - Ausonia
5 - Arroyo Cabral
6 - Tio Pujio
7 - Ramon J. Carcano
8 - Ballesteros
9 - Ana Zumaran
10 - Las Mojarras
Ausonia es la nro 4
La  nro  8  es  Ballesteros
Process Exit
```

```
class Tramos{
        private int cuantas;       //     Cuantos nodos (Ciudades) tenemos
        private int[][] kms;
        public Tramos(){      // tomamos las distancias de una mtriz de ctes
                // Origenes     0  1  2  3  4  5  6  7  8  9 10    Destinos
                int matt[][] = {{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},   // 0
                                                { 0, 0, 5,11, 0,10, 0,10,20, 0, 0},   // 1 Villa Nueva
                                                { 0, 0, 0, 0, 0, 0,18, 0, 0, 5, 5},   // 2 Villa Maria
                                                { 0,10, 0, 0, 0, 0, 0, 0, 0, 0, 0},   // 3 Sanabria
                                                { 0, 0, 0,10, 0, 0, 0, 0, 0, 0, 0},   // 4 Ausonia
                                                { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},   // 5 Arroyo Cabral
                                                { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},   // 6 Tio Pujio
                                                { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},   // 7 Ramon J. Carc
                                                { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},   // 8 Ballesteros
                                                { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},   // 9 Ana Zumaran
                                                { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};  //10 Las Mojarras
                cuantas=matt.length;
                kms = new int[cuantas][cuantas];
                for(int dest,orig=0;orig<cuantas;orig++)
                        for (dest=0;dest<cuantas;dest++)
                                kms[orig][dest]=matt[orig][dest];
        }
        public boolean adyacent(int orig, int dest) {
                boolean auxAdy = false;
                if (kms [orig][dest] > 0)auxAdy = true;
                return auxAdy;
        }
        public void unir(int orig, int dest, int dist){
                kms[orig][dest] = dist;
```

```
        }
        public void separar(int orig, int dest) {
                kms[orig][dest] = 0;
        }


         public String toString(){
                String aux = "Origen  Destino (Distancia en kms)\n";
                aux+="    ";
                for(int i=1;i<cuantas;i++){
                        if(i<10)aux+="   ";
                        else   aux+="  ";
                        aux+=i;
                }
                aux+="\n";
                for(int dest,orig=1;orig<cuantas;orig++){
                        if(orig<10)aux+="   "+orig;
                        else     aux+="  " +orig;
                        for (dest=1;dest<cuantas;dest++){
                                if(kms[orig][dest]<10)aux+="
                                else             aux+="  ";
                                aux+=kms[orig][dest];
                        }
                        aux+="\n";
                }
                aux+="\n";
                return aux;
        }

        public static void main(String args[]) {
                Tramos tramos = new Tramos();
                System.out.println(tramos);
        }
}

import Ciudades;
import Tramos;
class Grafo{
        private Ciudades ciud;
        private Tramos tram;

        public Grafo(){
                ciud = new Ciudades();
                tram = new Tramos();
        }

        public int  entreGrado(String city){
                int arcAdy = 0, aux;
                aux = ciud.existe(city);
                if(aux == 0)return aux;                 // No encontró ciudad buscada
                for (int i=0; i<ciud.getCuantas(); i++)
                        if(tram.adyacent(i,aux))arcAdy++;
                return arcAdy;
        }

        public int fueraDeGrado(String city){
                int arcAdy = 0;
                int aux = ciud.existe(city);
                if(aux == 0)return aux;                 // No encontró ciudad buscada
                for (int i=0; i<ciud.getCuantas(); i++)
                        if(tram.adyacent(aux,i))arcAdy++;
                return arcAdy;
        }

        public String toString(){
```

```
Origen  Destino (Distancia en kms)
     1  2  3  4  5  6  7  8  9 10
  1  0  5 11  0 10  0 10 20  0  0
  2  0  0  0  0  0 18  0  0  5  5
  3 10  0  0  0  0  0  0  0  0  0
  4  0  0 10  0  0  0  0  0  0  0
  5  0  0  0  0  0  0  0  0  0  0
  6  0  0  0  0  0  0  0  0  0  0
  7  0  0  0  0  0  0  0  0  0  0
  8  0  0  0  0  0  0  0  0  0  0
  9  0  0  0  0  0  0  0  0  0  0
 10  0  0  0  0  0  0  0  0  0  0
Process Exit...
```

```
        String aux = "";
        aux+=ciud.toString();
        aux+=tram.toString();
        return aux;
}

public void unir(String desde, String hasta, int distan){
        int des=ciud.existe(desde);
        int has=ciud.existe(hasta);
        if (des != 0 && has != 0)       // Si ambas existen
                tram.unir(des,has,distan);
}

public void separar(String desde, String hasta, int distan){
        int des=ciud.existe(desde);
        int has=ciud.existe(hasta);
        if (des != 0 && has != 0)       // Si ambas existen
                tram.separar(des,has);
}

public boolean findPath(String desde, String hasta, int tramos){
        int des=ciud.existe(desde);
        int has=ciud.existe(hasta);
        if (des == 0 || has == 0)       // Si ningua existe
                return false;
        return hayCamino(des, has, tramos);
}

private boolean hayCamino(int des, int has, int tramos){
        if(tramos == 1) return tram.adyacent(des, has);
        else{
                for(int i=1;i<ciud.getCuantas();i++)
                        if(tram.adyacent(des,i) && hayCamino(i, has, tramos - 1))
                                return true;
        }       // else
        //      Formulación recursiva
        // Si existe adyacencia entre origen y un dest. intermedio y ademas
        // hay un camino de t-1 tramos desde dest. inter. al dest. final
        // Entonces tenemos el camino de t tramos pedido.
        return false;
}
public void demo(){
        boolean exist;
        System.out.println("\nConectando Tio Pujio a Ballesteros, 35 kms");
        unir("Tio Pujio","Ballesteros",35);
        System.out.println("El entreGrado de Villa Maria es "+entreGrado("Villa Maria"));
        System.out.println("y su fueraDeGrado es . . . . "+fueraDeGrado("Villa Maria"));
        System.out.println("\nBuscando caminos de k tramos ...");
        System.out.println("De Villa Nueva a Villa Maria");
        for(int k = 1;k<8;k++){
                exist = findPath("Villa Nueva","Villa Maria",k);
                if(exist)System.out.print(k+" si, ");
                else    System.out.print(k+" no, ");
        }
        System.out.println("\nDe Villa Maria a Villa Nueva");
        for(int k = 1;k<8;k++){
                exist = findPath("Villa Maria","Villa Nueva",k);
                if(exist)System.out.print(k+" si, ");
                else    System.out.print(k+" no, ");
        }
        System.out.println("\nAusonia a Arroyo Cabral");
        for(int k = 1;k<8;k++){
                exist = findPath("Ausonia","Arroyo Cabral",k);
                if(exist)System.out.print(k+" si, ");
                else    System.out.print(k+" no, ");
```

```
            }
            System.out.println("\n");
        }

        public static void main(String args[])
            Grafo grafo = new Grafo();
            grafo.demo();
        }
}
```

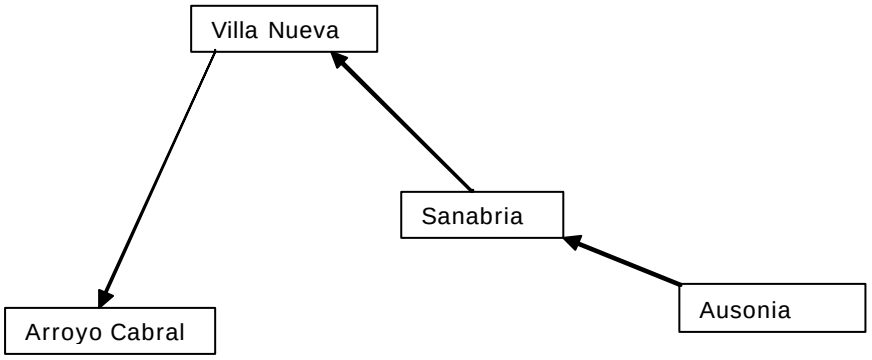
```
Conectando Tio Pujio a Ballesteros, 35 kms
El entreGrado de Villa Maria es 1
y su fueraDeGrado es .  .  .  . 3

Buscando caminos de k tramos ...
De Villa Nueva a Villa Maria
1 si, 2 no, 3 si, 4 no, 5 si, 6 no, 7 si,
De Villa Maria a Villa Nueva
1 no, 2 no, 3 no, 4 no, 5 no, 6 no, 7 no,
Ausonia a Arroyo Cabral
1 no, 2 no, 3 si, 4 no, 5 si, 6 no, 7 si,

Process Exit...
```

*Que es eso de **Formulación Recursiva** ?. Y como es que hayCamino(..) se llama desde el mismo método ?*

***Formulación Recursiva**, o sea la forma de plantear recursivamente bloques de código que deben ejecutarse varias veces para resolver un problema. Lo vimos en la Unid… hem. no, no lo vimos … Tenemos 10 minutos ?*

Formulación Recursiva.

Cuando en la práctica se habla de recursividad, se está haciendo referencia a una muy particular forma de expresar la definición de un objeto o un concepto. Esencialmente, una definición se dice recursiva si el objeto que está siendo definido aparece a su vez en la propia definición, o mejor, **se lo define en términos de si mismo.** Un ejemplo ayuda a entender. En nuestro grafo de ciudades, entre Ausonia y Arroyo Cabral:



Recursivamente esto se plantea:

Tenemos un camino de tres tramos entre Ausonia y Arroyo Cabral:

a) Si tenemos una adyacencia entre Ausonia y Sanabria (Esto se deduce de la matriz directamente)
b) Y tenemos un camino de dos tramos entre Sanabria y Arroyo Cabral (Esto hay que verlo)

El punto b, hay que verlo. Planteandolo de la misma manera:
Tenemos un camino de 2 tramos entre Sanabria y Arroyo Cabral:

a) Si tenemos una adyacencia entre Sanabria y Villa Nueva
b) Y tenemos un camino de un tramo entre Villa Nueva y Arroyo Cabral (Esto hay que verlo)

El punto b, nuevamente:
Tenemos un camino de 1 tramo entre Villa Nueva y Arroyo Cabral:

a) Si tenemos una adyacencia entre Villa Nueva y Arroyo Cabral
Y como no hay mas camino a recorrer, **recursivamente hemos concluido** que:
Tenemos un camino de tres tramos entre Ausonia y Arroyo Cabral

La **formulación recursiva** siempre consta de estos dos puntos.
a) El caso conocido, o por lo menos directamente deducible.
b) El planteo en términos de si mismo, siempre de un grado menor que el anterior.

El punto b) debe plantearse de forma que mas tarde o temprano llegue a ser el punto a.

Y no hay más que decir de la recursividad. Pero para que quede mas claro vamos a ilustrarlo con el ejemplo clásico, que sale en todos los libros: El **factorial de n**, siendo n un número entero positivo.

```
class Factorial{
        public Factorial(){;}
        public int factorial(int n){
                int fact;
                if (n == 0){
                        System.out.println("Llegamos al caso conocido: 0! = 1");
                        System.out.println("Retorno 1 a las invocaciones pendientes");
                        return 1;
                }else{
                        System.out.println("Invocando a factorial("+(n-1)+")");
                        fact = n*factorial(n - 1);
                        System.out.println("Tengo n = "+n + ", fact = "+ fact);
                        return fact;
                }
        }
        public static void main(String args[]){
                Factorial calculo = new Factorial();
                System.out.println("El factorial de 5 es:"+calculo.factorial(5));
        }
}
```

```
Invocando a factorial(4)
Invocando a factorial(3)
Invocando a factorial(2)
Invocando a factorial(1)
Invocando a factorial(0)
Llegamos al caso conocido: 0! = 1
Retorno 1 a las invocaciones pendientes
Tengo n = 1, fact = 1
Tengo n = 2, fact = 2
Tengo n = 3, fact = 6
Tengo n = 4, fact = 24
Tengo n = 5, fact = 120
El factorial de 5 es:120
Process Exit...
```

Interpretando calculo.factorial(5)

Cuando dentro del cuerpo de un método invocamos a otro, el lenguaje (Java, C, todos los lenguajes imperativos) guarda en una pila la situación de todas sus variables locales y el punto de la instrucción desde la cual se hace la llamada. Gracias a esto es que la sentencia return nos lleva al punto exacto en el cuerpo del método llamador y allí retomamos los cálculos. Y así podemos ir retornando y

retornando, haciendo el camino inverso al que recorrimos cuando fuimos invocando los métodos. Un detalle de esto vimos cuando estudiamos excepciones, que podían ser tratadas en la lista catch del método donde se produjo la excepción o en otra lista perteneciente a un método mas abajo en la pila de llamadas.

En la recursividad **no es diferente**. Al llamar apilamos los valores de las variables locales y el punto de llamada. Al retornar, desapilamos, los valores guardados en la pila reemplazan sus respectivos valores en las variables corrientes y el procesamiento continúa en el punto inmediato siguiente al de llamada. Una particularidad, no una diferencia, es que podemos retornar varias veces al mismo punto, tantas como desde allí llamamos.

Interpretemos calculo.factorial(5)

Invocaciones (Camino hacia delante).
"Invocando a factorial(4)": dentro de la intrucción **fact = n*factorial(n - 1)** hay un llamado a factorial pasando un argumento n – 1. El **producto no llega a concretarse**, tiene prioridad la resolución de llamada a método. Y esto ocurre cuatro veces mas, la última le pasamos argumento = 0. Aquí la lógica va por la otra rama. Estamos en el

Caso conocido
Factorial de 0 es 1. Retornamos 1, entonces. Y ese es el valor que aportará factorial() al primer producto pendiente de resolución. Esto ya ocurre en

Retornos (Camino de vuelta atras).
Se termina **de concretar el producto pendiente**. Podemos apreciar que el valor retornado es multiplicado por el valor de n "reposicionado" a partir del valor apilado, que es el que n tenía en el momento de la invocación. Esto es exhibido por **System.out.println("Tengo n = "+n + ", fact = "+ fact);** Llegamos al fin del cuerpo del método factorial(). La pila comanda el próximo retorno. Puede ser de nuevo al producto pendiente, llegará un momento que retornaremos al metodo llamador, el main() en este caso.