

# Taller de programación - Resumen

*Compartimos el mismo horizonte de progreso, y se llama Universidad Pública*

## ÍNDICE

<b>IMPERATIVO</b>	<b>3</b>
Definiciones	3
Arreglo:	3
Lista:	3
Recursión:	3
Árbol Binario de Búsqueda (ABB):	3
Ordenar un vector	4
Método de inserción	4
Método de Selección	4
Función recursiva que retorna el máximo valor de un vector	5
Búsqueda dicotómica recursiva	5
Eliminar elementos de un vector ordenado, que estén comprendidos en un rango pasado por parámetro	6
Insertar elemento a un Árbol de registros	6
Imprimir datos de un Árbol	7
En orden	7
Pre-orden	7
Post-orden	7
Insertar elemento a un Árbol de Listas	8
Buscar si un elemento se encuentra en un Árbol aprovechando el orden	9
Buscar y retornar un nodo de un Árbol	9
Buscar el nodo con valor más chico del Árbol (función)	10
Buscar un máximo en un Árbol (criterios de búsqueda y orden diferentes)	10
Recorrido acotado – contar en rango	10
Preguntas múltiple choice	11
Ordenación de vectores	11
Recursión	12
<b>Programación Orientada a Objetos (POO)</b>	<b>15</b>
Definiciones	15
Objeto:	15
Encapsulamiento:	15
Clase:	15
Constructor:	16
Herencia:	16
Polimorfismo:	16
Clase abstracta:	16

Métodos abstractos: _____	16
Super _____	17
Binding dinámico _____	17
This _____	17
Comparaciones _____	18
Casting o “casteo” _____	18
Vectores _____	18
Carga de datos en un vector de enteros leídos desde teclado: _____	18
Matrices _____	19
Cargar una matriz en orden, que puede no completarse (usando DIV y MOD) _____	19
Cargar una matriz en orden que puede no completarse (sin usar DIV y MOD) _____	20
Cargar matriz llevando dimensiones lógicas por columna (ejemplo) _____	21
Imprimir una matriz de enteros _____	21
Sumar los elementos de la fila 1: _____	21
Buscar un elemento en una matriz, si está retornar su posición: _____	22
Preguntas múltiple choice _____	23
Matrices y vectores _____	23
Conceptos de POO _____	25
Herencia y polimorfismo _____	27
<b>Programación concurrente _____</b>	<b>29</b>
Definiciones _____	29
Tipos de áreas _____	30
Patrones de resolución de problemas _____	30
Sincronización por barrera _____	30
Passing the baton _____	30
Productor / Consumidor _____	30
Cliente / Servidor _____	31
Master / Slave _____	31
Preguntas múltiple choice _____	32
Instrucciones _____	44

*Este resumen lo hizo un estudiante con la idea de organizar los conceptos básicos de la materia y reunirlos en un lugar, apuntando a un repaso más ordenado de los contenidos a la hora de preparar el final.*

*Si te llega este material y todavía no cursaste Taller, esto No reemplaza las teorías ni las prácticas, solo reúne información. Conviene no perderse ninguna clase porque el ritmo de cursada es rápido y furioso, y siempre corregí tus códigos con profesores, que funcione no significa que esté bien. Recordá que esta materia es correlativa a todo 2do año, no la cuelgues.*

**Defendamos la educación pública, educación para saber elegir y pública para que todos elijan.**

# IMPERATIVO

## Definiciones

### Arreglo:

Un arreglo es una estructura de datos compuesta que permite acceder a cada componente por una variable índice, que da la posición de la componente dentro de la estructura de datos. La estructura arreglo se almacena en posiciones contiguas de memoria.

Características: **homogénea –estática -acceso directo indexado -lineal**

Trabaja con dimensión física y dimensión lógica.

### Lista:

Una lista es una estructura de datos lineal compuesta por nodos. Cada nodo posee el dato que almacena la lista y la dirección del siguiente nodo. Toda lista puede recorrerse a partir de su primer elemento. Los elementos se almacenan en memoria dinámica y no necesariamente están en posiciones contiguas.

Para generar nuevos elementos en la lista se utiliza la operación *new*, y para eliminar elementos la operación *dispose*.

Características: **homogénea –dinámica -lineal -acceso secuencial**

### Recursión:

La recursividad es una técnica para resolver problemas que se basa en dividir un problema en instancias más pequeñas del mismo (denominadas subproblemas), hasta alcanzar un subproblema lo suficientemente simple que posea una solución trivial o directa.

Cuando el problema se va achicando llega a un punto que no puede achicarse más, esa instancia se denomina **caso base**.

Hay problemas en los cuales debe realizarse alguna tarea cuando se alcanza el caso base y otros que no. Hay problemas que pueden tener más de un caso base.

### Árbol Binario de Búsqueda (ABB):

Es una estructura de datos jerárquica (no lineal), homogénea y dinámica. Está formada por nodos donde cada nodo tiene a lo sumo 2 hijos. El nodo principal se denomina raíz (es el primer dato en ser ingresado a la estructura) y los nodos que no tienen hijos se denominan hojas del árbol.

```
Type
  arbol = ^nodoArbol;

  nodoArbol = record
    dato: tipoDeDato;
    HI: arbol;
    HD: arbol; rbol = ^nodoArbol;
  end;
```

Los nodos de un ABB respetan un criterio uniforme: los hijos ubicados a la izquierda son menores que el nodo padre, o viceversa, dependiendo de la configuración del árbol (si es de menor a mayor o al revés). Cuando un enunciado pide una estructura “*eficiente para la búsqueda ...*” se refiere a un ABB.

## Ordenar un vector

### Método de inserción

```
1. procedure ordenarPorInsercion(var v:vector; dimLog:integer);1
2. var
3.   i,j:integer;
4.   actual:registro; //actual es del tipo de datos que guarda el vector
5. begin
6.   for (i:= 2 to dimLog)2 do begin
7.     actual:=v[i];
8.     j:=i-1;
9.     while (j>0) and (v[j].numero >3 actual.numero)do begin
10.      v[j+1]:=v[j];
11.      j:=j-1;
12.    end;
13.    v[j+1]:=actual;
14.  end;
15.end;
```

### Método de Selección

```
1. Procedure ordenarPorSeleccion(var v:vector; dimL:integer);
2. Var
3.   i,j,pos:integer;
4.   item:registro; //item es del tipo de datos que guarda el vector
5. begin
6.   for i:=1 to dimL -1 do begin
7.     pos:=i;
8.     for j:= i+1 to dimL do begin
9.       if (v[j].num < v[pos].num) then
10.        pos:=j;
11.      item:=v[pos];
12.      v[pos]:=v[i];
13.      v[i]:=item;
14.    end;
15.  end;
16. end;
```

### Otra forma de codificar el método por selección

---


<sup>1</sup> Si los datos ya están ordenados de menor a mayor, el tiempo de ejecución resulta de orden  $n$ . Si están ordenados de mayor a menor hace todas las comparaciones y todos los intercambios, el tiempo sería de orden  $n^2$

<sup>2</sup> El algoritmo se completa en (dimensión Lógica - 1) pasadas.

<sup>3</sup> La comparación de la línea 9 ordena al vector de menor a mayor, si se quisiera ordenar de mayor a menor la comparación debería ser `(v[j].numero < actual.numero)`

## Función recursiva que retorna el máximo valor de un vector <sup>4</sup>

```
1.function obtenerMaximo(v:vector; dimL:integer):integer;5
2.begin
3.  if (dimL = 0) then
4.    obtenerMaximo:= -1
5.  else
6.    obtenerMaximo:=max(v[dimL],obtenerMaximo(v,dimL-1));
7.end;
```



```
1. function max(num1, num2: integer):integer;
2. begin
3.  if (num1 > num2) then
4.    max:=num1
5.  else
6.    max:= num2;
7. end;
```

## Búsqueda dicotómica recursiva <sup>6</sup>

```
1.procedure dicotomica(v:vector; ini,fin,dato:integer; var pos:integer);7
2. var
3.  medio:integer;
4. begin
5.  if (ini >= fin ) then
6.    pos:=-1
7.  else begin
8.    medio:= (ini + fin) div 2;
9.    if (dato = v[medio]) then
10.      pos:=medio
11.    else begin
12.      if (dato < v[medio]) then begin8
13.        fin:=medio -1;
14.        busquedaDicotomica(v,ini,fin,dato,pos);
15.      end
16.    else begin
17.      ini:=medio +1;
18.      busquedaDicotomica(v,ini,fin,dato,pos)
19.    end;
20.  end;
21. end;
22. end;
```

<sup>4</sup> Esta función solo requiere que se le pasen por parámetro el vector y la dimensión lógica, pero hace uso de una función auxiliar max, la cual recibe dos números enteros y retorna el más grande.

<sup>5</sup> Video explicativo de la función: <https://youtu.be/sNCHkSpS3zE?feature=shared>

<sup>6</sup> La búsqueda dicotómica solo se puede aplicar sobre un vector ordenado.

<sup>7</sup> Ini debe estar inicializado en 1 y fin en dimensión lógica

<sup>8</sup> En este caso el vector está ordenado de menor a mayor, si fuese al revés la comparación de línea 12 sería `dato>v[medio]`

## Eliminar elementos de un vector ordenado, que estén comprendidos en un rango pasado por parámetro

```
1.procedure eliminarEnRango(var v:vector;vardimL:integer;n1,n2:integer);
2.var
3.  i,ini,fin,aBorrar:integer;
4.begin
5.  ini:=1;
6.  while (ini <= dimL) and (v[ini].numero <= n1) do 9
7.    ini:= ini + 1;
8.  fin:=ini;
9.  while (fin <= dimL) and (v[fin].numero >= n2) do
10.    fin:= fin +1;
11.  aBorrar:= fin - ini;
12.  if (aBorrar > 0) then begin
13.    for i:=ini to dimL - aBorrar do
14.      v[i]:= v[i+aBorrar];
15.    dimL:=dimL-aBorrar;
16.  end;
17.end;
```

## Insertar elemento a un Árbol de registros

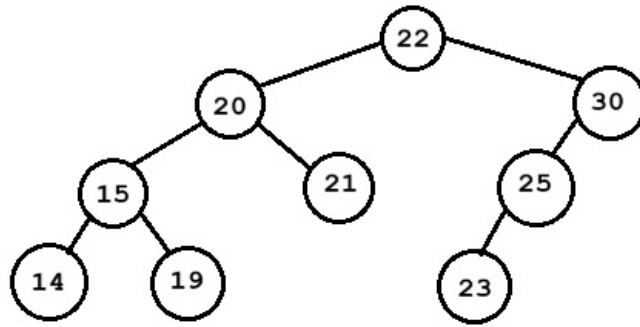
```
1. procedure insertarEnArbol(var arbol; elem:registro);
2. begin
3.   if (a = nil) then begin
4.     new(a);
5.     a^.dato:=elem;
6.     a^.HI:=nil;
7.     a^.HD:=nil;
8.   end
9.   else
10.    if (elem.num <= a^.dato.num)10 then
11.      insertarEnArbol(a^.HI,elem)
12.    else
13.      insertarEnArbol(a^.HD,elem)
14. end;
```

---

<sup>9</sup> Este algoritmo borra también los límites del rango, para no incluir los límites dentro de la eliminación en líneas 6 y 9 la segunda comparación debe quedar ' $<$ ' y ' $>$ ' estrictos.

<sup>10</sup> Árbol en orden ascendente con elementos repetidos a izquierda. Si se quisiera ordenar de mayor a menor la comparación debería ser (elem.num  $>=$  a^.dato.num)

## Imprimir datos de un Árbol



### En orden

```
1. Procedure imprimirEnOrden(a:arbol); 11
2. Begin //input 22-30-20-21-15-19-25-14-23
3.   If (a<>nil) then begin
4.     imprimirEnOrden(a^.HI);
5.     writeln(a^.dato);
6.     imprimirEnOrden(a^.HD);
7.   end;
8. end; //output 14-15-19-20-21-22-23-25-30
```

### Pre-orden

```
1. Procedure imprimirPreOrden(a:arbol);
2. begin //input 22-30-20-21-15-19-25-14-23
3.   if (a<>nil) then begin
4.     writeln(a^.dato);
5.     imprimirPreOrden(a^.HI);
6.     imprimirPreOrden(a^.HD);
7.   end;
8. end; //output 22-20-15-14-19-21-30-25-23
```

### Post-orden

```
1. Procedure imprimirPostOrden(a:arbol);
2. begin //input 22-30-20-21-15-19-25-14-23
3.   if (a<>nil) then begin
4.     imprimirPostOrden(a^.HI);
5.     imprimirPostOrden(a^.HD);
6.     writeln(a^.dato);
7.   end;
8. end; //output 14-19-15-21-20-23-25-30-22
```

---

<sup>11</sup> Imprime el Árbol en el orden en el que está armado, si se quisiera imprimir en el orden inverso en el primer llamado recursivo (línea 4) debe pasársele por parámetro el **a^.HD** y en el segundo llamado (línea 6) el **a^.HI**

## Insertar elemento a un Árbol de Listas

```

1. procedure agregarAlArbolDeListas(var a:arbol; elem:registro);12
2. begin
3.   if (a = nil) then begin
4.     new(a);
5.     a^.dato.numero:=elem.numero;
6.     a^.dato.lista:=nil;
7.     agregarAdelante(a^.dato.lista,elem.info);13
8.     a^.HI:=nil;
9.     a^.HD:=nil;
10.  end
11.  else
12.    if (a^.dato.numero = elem.numero) then
13.      agregarAdelante(a^.dato.lista,elem.info)
14.    else
15.      if (elem.numero < a^.dato.numero) then
16.        agregarAlArbolDeListas(a^.HI,elem)
17.      else
18.        agregarAlArbolDeListas(a^.HD,elem)
19.    end;

```

Type

```

infoRegistro=record
  nombre:String; anio:integer; etc...
end;

```

```

Registro=record
  Numero:integer;14
  Info:infoRegistro;
end;

```

```

Lista=^nodoLista;

```

```

nodoLista=record
  Dato:infoRegistro;
  Sig:lista;
end;

```

```

datoArbol=record
  numero:integer;
  l:lista;
end;

```

```

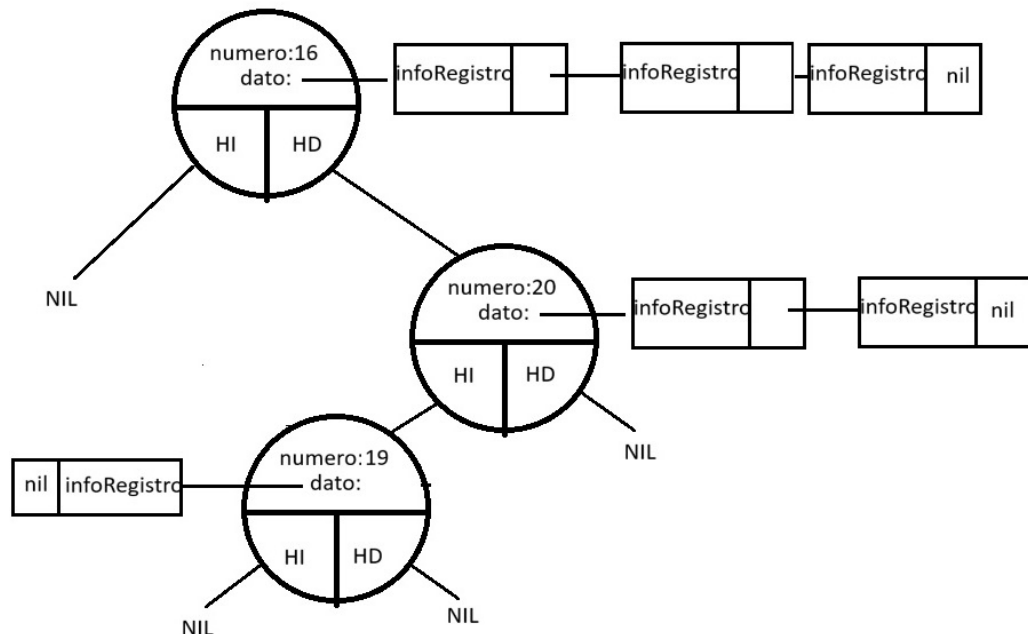
Arbol =^nodoArbol;

```

```

nodoArbol=record
  dato:datoArbol;
  HD,HI:arbol;
end;

```



<sup>12</sup> En cada nodo guarda el número por el cual se ordena y el puntero inicial a una lista con orden LIFO (Last In First Out).

<sup>13</sup> Si se quiere que las listas del árbol estén ordenadas ascendente o descendentemente se debe hacer un *insertarOrdenado*

<sup>14</sup> Dato por el que se va a ordenar el Árbol, debe estar por fuera del dato que guarda la lista, para no repetir información



## Buscar si un elemento se encuentra en un Árbol aprovechando el orden

```
1.function estaEnElArbol(a:arbol; num:integer):boolean;
2. begin
3.   if (a = nil) then
4.     estaEnElArbol:=False
5.   else
6.     if (a^.dato = num) then
7.       estaEnElArbol:=True
8.     else
9.       if (num < a^.dato) then
10.        estaEnElArbol:=estaEnElArbol(a^.HI,num)
11.      else
12.        estaEnElArbol:=estaEnElArbol(a^.HD,num);
13. end;
```

## Buscar y retornar un nodo de un Árbol

```
1. function buscarNodo(a:arbol; num:integer):arbol;
2. begin
3.   if (a = nil) then
4.     buscarNodo:=nil
5.   else
6.     if (num = a^.dato) then
7.       buscarNodo:=a
8.     else
9.       if (num < a^.dato) then
10.        buscarNodo:= buscarNodo(a^.HI,num)
11.      else
12.        buscarNodo:= buscarNodo(a^.HD,num)
13. end;
```

## Buscar el nodo con valor más chico del Árbol (función)<sup>15</sup>

Aprovecha el orden y solo recorre hijos izquierdos

```
1. function menorElemento(a:arbol):integer;
2. begin
3.   if (a=nil) then
4.     menorElemento:= 9999; //codigo de arbol vacio
5.   else
6.     if (a^.HI=nil) then
7.       menorElemento:=a^.dato.num
8.     else
9.       menorElemento:=menorElemento(a^.HI);
10. end;
```

## Buscar un máximo en un Árbol (criterios de búsqueda y orden diferentes)

Recorre el arbol completo

```
1.procedure buscarMax(a:arbol; var max:integer);
2.begin //max debe inicializarse previamente con un valor muy chico
3.  if (a<>nil) then
4.    if (a^.dato.num > max) then
5.      max:=a^.dato.num;
6.    buscarMax(a^.HI,max);
7.    buscarMax(a^.HD,max);
8.end;
```

## Recorrido acotado – contar en rango

```
1.function cantNodos(a:arbol; n1,n2:integer):integer; 16
2.begin
3.  if (a = nil) then
4.    cantNodos:=0
5.  else begin
6.    if (a^.dato.numero > n1) then 17
7.      if (a^.dato.numero < n2) then
8.        cantNodos:= 1 + cantNodos(a^.HI,n1,n2) + cantNodos(a^.HD,n1,n2)
9.      else //si es mas grande que n2
10.        cantNodos:= cantNodos(a^.HI,n1,n2)
11.    else //si es más chico que n1
12.      cantNodos:= cantNodos(a^.HD,n1,n2)
13.  end;
14.end;
```

<sup>15</sup> Si se quiere buscar el nodo **más grande** en líneas 6 y 9 se debe poner **a^.HD**

<sup>16</sup> Función que recibe un Árbol, números inferior y superior de un rango, y retorna cuántos nodos hay en ese rango sin contar los límites. N1 debe ser MENOR que N2

<sup>17</sup> Si se quisieran incluir los límites del rango las comparaciones deberían ser (a^.dato.num>=n1) y (a^.dato.num <=n2)

## Preguntas múltiple choice

### Ordenación de vectores

- Dado un vector con registros Producto (con campos código, nombre y stock) ordenado por stock de manera ascendente.

Para encontrar un producto con un determinado nombre (que seguro existe) es necesario hacer una búsqueda lineal que recorra el vector hasta encontrarlo.	Verdadero porque el vector está ordenado por stock y no por nombre
Puedo buscar eficientemente en el vector por stock usando búsqueda binaria o dicotómica.	Verdadero porque el vector está ordenado por ese criterio
Si se requiere hacer múltiples búsquedas de productos por código, se justifica primero ordenar el vector por dicho campo y luego aplicar la búsqueda binaria o dicotómica bajo el mismo criterio.	Verdadero
La búsqueda binaria o dicotómica se puede utilizar si necesitamos buscar un producto por código.	Falso, la búsqueda dicotómica solo se puede utilizar en un vector ordenado por tipo de dato buscado

- Dados un vector de enteros, con dimensión lógica DL y dimensión física DF, y el algoritmo de ordenación por selección para ordenar vectores de manera ascendente:

El peor caso de entrada tiene un tiempo de ejecución de orden $N^2$ , siendo N el tamaño del arreglo.	Verdadero
Si los datos ya se encuentran inicialmente ordenados de manera ascendente, el tiempo de ejecución es de orden N.	Falso, es de orden $N^2$
En cada pasada, el algoritmo busca el mínimo elemento entre una posición i y la última posición del vector.	Verdadero
No requiere estructuras auxiliares.	Verdadero

- Dados un vector de enteros, con dimensión lógica DL y dimensión física DF, y el algoritmo de ordenación por inserción para ordenar vectores de manera descendente.

El peor caso de entrada que puede llegar al algoritmo es cuando el vector ya está ordenado descendientemente.	Falso, el peor caso de entrada en este ejemplo es cuando el vector está ordenado ascendentemente
El mejor caso de entrada que puede llegar al algoritmo es cuando el vector está desordenado.	Falso
Para ordenar el vector el algoritmo hace DF iteraciones o pasadas.	Falso
El mejor caso de entrada tiene un tiempo de ejecución de orden N, siendo N el tamaño del arreglo..	Verdadero, el mejor caso de entrada en este ejemplo es cuando el vector está ordenado de manera descendente.

- Dado un vector  $v$  de enteros, con dimensión lógica  $DL$  y dimensión física  $DF$ . Indique VERDADERO o FALSO. ¿El siguiente código que ordena por selección es correcto?

```

1.for i:=1 to DL-1 do begin
2.  p := i;
3.  for j := i+1 to DL do
4.    if v[ p ] > v[ j ] then //ordena de menor a mayor
5.      p:=j;
6.  if (i <> p) then begin
7.    item := v[ i ];
8.    v[ i ] := v[ p ];
9.    v[ p ] := item;
10. end;
11.end;

```

Verdadero, el código es correcto (se asume que  $i, j, p, item$  están declaradas)

## Recursión

- Marque la afirmación VERDADERA sobre la metodología Recursión:

Un módulo recursivo puede no tener un caso base	Falso
Un módulo recursivo puede incluir más de una llamada recursiva	Verdadero
Un módulo recursivo no puede utilizar un for ya que es una estructura de control repetitiva	Falso
Un módulo recursivo luego de la autoinvocación no puede ejecutar más código.	Falso

- Se desea calcular el máximo común divisor de dos números enteros. El MCD de dos números enteros es el mayor número entero que los divide sin dejar resto. Ejemplos:  
MDC 4 y 8: 4 - MDC 18 y 24: 6 - MDC 10 y 81: 1  
Responda VERDADERO o FALSO: ¿el módulo recursivo presentado es CORRECTO para tal fin?.

```

1. function mcd(a,b:integer): integer;
2. begin
3.   if (b = 0) then
4.     mcd:= a
5.   else
6.     mcd:= mcd(a, b mod a);
7. end;

```

Falso, no es correcto, en el llamado recursivo (línea 6) hay ejemplos en los que  $b$  nunca llega a 0 (por ejemplo si se ingresan 18 y 24)  $24 \bmod 18 = 6$ ;  $6 \bmod 18 = 6$ ; etc ...

- Sea el procedure ImprimirVector y v = ('E', 'D', 'F', 'C', 'G', 'B', 'H','A');

```

1. procedure ImprimirVector (v: vector; ini, fin: integer);
2. begin
3.   if (ini < fin) then begin
4.     writeln (v[fin]);
5.     imprimirVector (v, ini + 1, fin - 1); //llamados (2,7) (3,6) (4,5) (5,5)
6.     writeln (v[ini]);
7.   end;
8. end;

```

Imprime E D F C G B H A.	Falso
Imprime A H B G C F D E.	Verdadero
Imprime A B C D E F G H.	Falso
Ninguna de las anteriores es verdadera	Falso

Prueba de escritorio:

```

Imprime v[8] = 'A'
Llamado1 con ini=2 fin=7
Imprime v[7] = 'H'
Llamado2 con ini= 3 fin =6
Imprime v[6] = 'B'
Llamado3 con ini = 4 fin = 5
Imprime v[5] = 'G'
Llamado4 con ini = 5 fin = 5 , se llega al caso base ini < fin da falso
Continúa con la sentencia que sigue al llamado4 (linea6)-imprime v[4]= 'C'
Continúa con la sentencia que sigue al llamado3 - imprime v[3] = 'F'
Continúa con la sentencia que sigue al llamado2 - imprime v[2] = 'D'
Continúa con la sentencia que sigue al llamado1 - imprime v[1] = 'E'

```

- Marque la opción verdadera:

La recursividad consiste en resolver un problema por medio de un módulo (procedimientos o funciones) que no se llama a sí mismo, evitando el uso de bucles y otros iteradores.	Falso
La recursividad es una técnica de resolución de problemas que consiste en dividir un problema en subproblemas más pequeños.	Falso
En un módulo recursivo siempre se realiza una tarea cuando se llega al caso base..	Falso
Ninguna de las anteriores es verdadera.	Verdadero

- Dada la siguiente función:

```

1. function Recursion (valor: integer): integer;
2. var digito:integer;
3. begin
4.   if (valor = 0) then
5.     Recursion:= 0
6.   else begin
7.     digito:= (valor mod 10);
8.     if (digito mod 2 = 0) then
9.       Recursion:= digito * Recursion (valor div 10)
10.    else
11.      Recursion:= Recursion (valor div 10)
12.    end;
13.  end;
14. end;

```

La función pretende multiplicar los dígitos pares de un número que se pasa por parámetro

Con la sentencia writeln (Recursion(1234)), se imprime 24.	Falso
Con la sentencia writeln (Recursion(7236)), se imprime 12.	Falso
Con la sentencia writeln (Recursion(3832)), se imprime 16.	Falso
Ninguna de las anteriores es verdadera.	Verdadero, el caso base retorna 0 lo que provoca que siempre de cero el producto de la línea 9, y siempre imprima cero. Para solucionarlo el caso base debería retornar 1 (línea 5)

# Programación Orientada a Objetos (POO)

## Definiciones

### Objeto:

Abstracción de un objeto del mundo real, definiendo qué lo caracteriza (estado interno) y qué acciones sabe realizar (comportamiento). ¿Qué cosas son objetos? Todo es un objeto.

- Estado interno: compuesto por datos/atributos que caracterizan al objeto y relaciones con otros objetos con los cuales colabora. Se implementan a través de **variables de instancia**.
- Comportamiento: acciones o servicios a los que sabe responder el objeto. Se implementan a través de métodos de instancia que operan sobre el estado interno. Los servicios que ofrece al exterior constituyen la **interfaz**.

Cuando decimos la variable x es un objeto de clase Persona, o es instancia de Persona: estamos diciendo que en la variable x se guarda una dirección que apunta a un lugar de la memoria (dinámica) donde se encuentran almacenados los atributos y métodos propios de un objeto de tipo *Persona*.

### Encapsulamiento:

Se oculta la implementación del objeto<sup>18</sup> hacia el exterior. Se antepone a los atributos la palabra reservada **private**. Desde el exterior sólo se conoce la interfaz del objeto. Facilita el mantenimiento y evolución del sistema ya que no hay dependencias entre las partes del mismo.

### Clase:

Una clase describe un conjunto de objetos comunes (mismo tipo). Es como un “molde” para crear objetos. Un objeto es una instancia de una clase.

Consta de:

- La declaración de las variables de instancia que serán el estado interno de los objetos que se instancien a partir de esa clase.
- La codificación de los métodos que serán el comportamiento de los objetos de esa clase.

Gráfico de una Clase:

Triángulo	Nombre de la Clase empieza en Mayúscula
lado1,lado2,lado3 colorLinea, colorRelleno	
double calcularArea() double calcularPerimetro() /* métodos para obtener valores de las v.i.*/ /* métodos para establecer valores de las v.i. */	Encabezado de métodos (nombres en minúscula)

<sup>18</sup> La implementación de un objeto son sus variables de instancia (atributos) y sus métodos.

## Constructor:

Método que se invoca al crear un objeto con la palabra clave **new**, tiene por objetivo inicializar las variables de instancia de un objeto. Un objeto puede tener varios constructores diferentes (sobrecarga) se ejecutará uno u otro dependiendo los parámetros formales que se le pasen en la sentencia *new*.

Si no se implementa ningún constructor, Java ejecuta uno por defecto al que no se le pasa ningún parámetro. Los Strings son objetos que no necesitan la palabra *new* para ser creados.

## Herencia:

Es un mecanismo que permite que una clase herede características y comportamiento (atributos y métodos) de otra clase (clase padre o superClase). A su vez, la clase hija define sus propias características y comportamiento (desconocidas para la superClase). Agrupa atributos y comportamientos en común para no tener que repetir el mismo código en cada clase. Palabra clave **extends**

```
public class Cuadrado extends Figura{...}
```

## Polimorfismo:

Objetos de clases distintas responden al mismo mensaje de maneras diferentes. Ejemplo: objeto de clase Triángulo y objeto de clase Cuadrado responden al mensaje *calcularArea* realizando cada uno una operación matemática diferente. Ambas clases son hijas de la super clase Figura.

## Clase abstracta:

Una clase abstracta es una clase que no puede ser instanciada (no se pueden crear objetos de esta clase). Define características y comportamiento común para un conjunto de subclases. Puede definir métodos abstractos (sin implementación) que deben ser implementados por las subclases. Palabra clave **abstract**. Permite reutilizar código y no repetir.

Una clase abstracta que nunca se va a ser instanciada, ¿puede tener un constructor?

- Sí, una clase abstracta puede tener un constructor que reúna la inicialización de las variables de instancia que son comunes a todas sus clases hijas y así no repetir el mismo código en cada subclase. Las subclases llamarán a este método dentro de su propio constructor con la palabra clave *super*: **super (...)** ;

## Métodos abstractos:

Cuando se necesita implementar un método en común pero su implementación es diferente en cada subClase.

Su encabezado se define en una clase abstracta, pero no se implementa. Las clases que hereden de esta superclase abstracta estarán obligadas a implementar esos métodos abstractos de la forma que sea adecuada para cada una. Ejemplo:

```
public abstract double calcularArea();
```



## Super

La palabra clave **super** debe utilizarse cuando quiere llamarse a un metodo de una clase superior y existe un método con el mismo nombre implementado en la clase que llama. Llamar con *super* a un método que no esté en la subclase será considerado un error.

## Binding dinámico

Se determina en tiempo de ejecución el método a ejecutar para responder a un mensaje.

## This

Es una referencia que apunta al objeto que está ejecutando el método que contiene al this. Dicho de otra manera, se lo puede considerar un *puntero* generado por el lenguaje al objeto actual. This es el “espejo” que traen todos los objetos para señalarse a sí mismos.

Usos:

- Desambiguación: Distinguir entre atributos del objeto y parámetros de un método cuando se llaman igual:

```
// Constructor
public Persona(String nombre) {
    this.nombre = nombre; // this.nombre refiere a la variable de
    instancia del objeto
}
```

- Reutilización de código aprovechando el polimorfismo y binding dinámico:

toString de Círculo subclase de figura:

```
1. public String toString(){
2.     String aux = super.toString()+ "Radio:" + getRadio();
4.     return aux;
5. }
```

toString de Figura:

```
1. public String toString(){
2.     String aux = "Area:" + this.calcularArea() +
3.         "CR:" + getColorRelleno() +
4.         "CL:" + getColorLinea();
5.     return aux;
6. }
```

Como el objeto desde el que se invocó al método que contiene al `this.calcularArea` es instancia de clase Círculo, el `calcularArea` que se ejecutará será el de esa Clase Círculo. Esto se define en tiempo de ejecución (**binding dinámico**) y permite que el mismo código funcione para todas las subclases de Figura.

A los códigos que pueden ser ampliamente reutilizados se les llama genéricos y son una buena práctica de programación.

## Comparaciones

- `variable1 == variable2` compara si las variables apuntan al mismo objeto
- `variable1.equals(variable2)` compara si el contenido de los objetos es el mismo

## Casting o “casteo”

Convertir un tipo de dato en otro. Java puede hacer esto automáticamente (casting implícito):

```
public static void main (String[] args) {  
    int i = 4/3; {División entera i = 1}  
    double d1 = 4.0/3.0; {División real d1 = 1.3333}  
    double d2 = 4/3; {División entera d2 = 1.0 OJO ACÁ por más que lo  
    guardemos en una variable double, si dividimos enteros, Java guarda la  
    parte entera solamente.  
  
    double d3 = (double) 4/3; {División real d3 = 1.333 Casting explícito  
}
```

A veces no lo notamos y guardamos un promedio con decimales en una variable entera y perdemos información.

## Vectores

Arreglos de una dimensión. En Java los vectores solo pueden ser indexados por números enteros, y comienzan en 0. Por lo que hay que hacer adaptaciones para que la interacción con usuarios sea más legible. Ejemplo: si se pide ingresar por teclado en qué posición del vector se debe guardar algo, se asume que el usuario ingresa números empezando por el 1, por lo que el código debe restar 1 al número ingresado para operar, y suma 1 si debe informar alguna posición al usuario.

```
int [] vector = new int [dimFisica];
```

## Carga de datos en un vector de enteros leídos desde teclado:

```
1. int num; int dimLogica = 0; int dimFisica = 5;  
2. int [] numeros = new int [dimFisica];  
3. num = Lector.leerInt;  
4. while (dimLogica < dimFisica) && (num != 0){  
5.     numeros[dimLogica] = num;  
6.     dimLogica++; //dimLogica se incrementa luego de asignar en el vector  
7.     num = Lector.leerInt;  
8. }
```

---

*Nunca se debe dejar expuesta la modificación de la dimensión lógica de un vector hacia el exterior, eso viola el encapsulamiento y puede romper la lógica del programa, si existiese un `set.dimensionLógica` debe ser `private`, no hacerlo es un grave error*

---

# Matrices

Arreglos de dos dimensiones:

```
int [][] matriz = new int [filas][columnas]; //matriz de enteros
```

Posibles tipos de carga de una matriz:

- Carga completa siguiendo un orden: no requiere dimensión lógica ni inicializarse en null
- Carga siguiendo un orden pero puede no completarse, requiere dimensión lógica<sup>19</sup>
  - Dependiendo del orden de la carga puedo usar una sola dimensión lógica o un arreglo de dimensiones lógicas.
- Se carga sin orden, puede no completarse, requiere inicializarse, no requiere dimensión lógica. (Ejemplo: asientos en un teatro, habitaciones ocupadas por piso)

---

*Nunca se debe pasar una matriz por parámetro a otro objeto, eso rompe el encapsulamiento, se pasa de a un dato, o por ejemplo, se pasa el dato de la suma de una columna, o el promedio de una fila, pero nunca la matriz completa debe ser entregada como un parámetro a otro objeto, es un **error grave**. (Lo mismo con vectores)*

---

Cargar una matriz en orden, que puede no completarse (usando DIV y MOD)<sup>20</sup>

**Carga por columnas:** si se ingresa por teclado 1-2-3-4-5-6-7-8-9-10-11-12

1	4	7	10
2	5	8	11
3	6	9	12

```
1 int fila = 3; int col = 4;
2 int dimL = 0;
3 int dimFis = fila * col;
4 int[][] matriz = new int[fila][col];
5 int num;
6
7 num = Lector.leerInt();
8 while ((dimL < dimFis) && (num != 0) ){
9     matriz[dimL % fila][dimL / fila]= num; // {[dimL MOD fila][dimL DIV fila]}
10    dimL++;
11    num = Lector.leerInt();
12 }
```

---

<sup>19</sup> Si utilizo una dimensión lógica **no** debo inicializar la matriz en null, no voy a consultar si es o no null cuando tenga que agregar un elemento porque esa información ya me la da la dimensión lógica. Se considera un error hacerlo.

<sup>20</sup> Video explicativo: <https://youtu.be/Cmj1t0oOxSTc>

**Carga por filas:** si se ingresa por teclado 1-2-3-4-5-6-7-8-9-10-11-12

1	2	3	4
5	6	7	8
9	10	11	12

```
1.int fila = 3; int col = 4;
2.int dimL = 0;
3.int dimFis = fila * col;
4.int[][] matriz = new int[fila][col];
5.int num;
6.
7.num = Lector.leerInt();
8.while ((dimL < dimFis) && (num != 0) ){
9.    matriz[dimL / col][dimL % col]= num; //[dimL DIV columna][dimL MOD columna]
10.    dimL++;
11.    num = Lector.leerInt();
12.}
```

**Cargar una matriz en orden que puede no completarse (sin usar DIV y MOD)**

**Carga por filas:** si se ingresa por teclado 1-2-3-4-5-6-7-8-9-10-11-12

1	2	3	4
5	6	7	8
9	10	11	12

```
1.    int [][] matriz = new int [3][4];
2.    int fila =0; int columna=0;
3.    int num;
4.
5.    num=Lector.leerInt();
6.    while (fila < 3) && (num!=0){
7.        matriz[fila][columna]=num;
8.        columna++;
9.        if (columna == 4){ //si completó una fila
10.            fila++;        //avanza a la siguiente fila
11.            columna=0;      //reinicia el contador de columna
12.        }
13.        num=Lector.leerInt();
14.    }
```

Si se quisiera cargar por columnas, se deben intercambiar fila por columna, por ejemplo en línea 6 la primer comparación quedaría `while (columna < 4)`

## Cargar matriz llevando dimensiones lógicas por columna (ejemplo)

```
1. public class EjercicioMatriz {
2.     private int dimF = 4;
3.     private int dimC = 5;
4.     private Persona [][] matriz = new Persona [dimF][dimC];
5.     private int vector [] = new int [dimC]; //las dimL de cada columna
6.
7.     public boolean agregarPersona(int columna, String nombre) {
8.         if (vector[columna -1] < dimF) {
9.             matriz[vector[columna-1]][columna-1] = new Persona(nombre);
10.            vector[columna]++; //incrementar vector despues de agregar
11.            return true;
12.        } else
13.            return false;
14.    }
15. }
```

El objeto que tiene la matriz como variable de instancia, tiene un método agregar que recibe a qué columna de la matriz se desea agregar en esta caso a un objeto de Clase Persona, y el nombre. Retorna un true si pudo agregar o un false si no pudo hacerlo por estar completa dicha columna.

En línea 9 la coordenada para fila, es lo que guarda el vector como dimensión lógica para la columna pasada por parámetro. El -1 adapta lo que ingresa el usuario que no sabe que el índice empieza en cero.

## Imprimir una matriz de enteros

```
int i,j;
for (i = 0; i < dimF; i++) { //dimF es cantidad de filas
    for (j = 0; j < dimC; j++) { //dimC es cantidad de columnas
        System.out.print(matriz[i][j] + " "); //print no println
    }
    System.out.println(""); //para que baje cuando terminó una fila
}
```

## Sumar los elementos de la fila 1:

```
1. int suma = 0;
2.     for (int j = 0; j < col; j++) {
3.         suma += matriz[1][j]; //se fija fila 1 y se recorren columnas
4.     }
```

Buscar un elemento en una matriz, si está retornar su posición:

```
1. boolean ok = false;
2. int num;
3. System.out.println("Ingrese un valor a buscar en la matriz:");
4. num = Lector.leerInt();
5. i=0; j=0;
6. while ((!ok) && (i<fila)){
7.     while ((!ok) && (j<col)){
8.         if (matriz[i][j] == num){
9.             ok=true;
10.            System.out.println("Se encuentra en la fila " + ++i + " columna " + ++j);21
11.        }
12.        j++; //incremento j dentro del segundo while (avanzo por columnas)
13.    }
14.    j=0; //vuelvo las columnas a cero al cambiar de fila
15.    i++; //incremento i dentro del primer while (avanzo a la siguiente fila)
16.}
```

---

<sup>21</sup> De esta manera es más claro para el usuario que no sabe que se comienza en 0,0

## Preguntas múltiple choice

### Matrices y vectores

- Dada una matriz de enteros ya cargada. A partir del siguiente fragmento de código, indique cuál de las opciones es verdadera:

```
1. for (int j = 0; j < CANT_A; j++){
2.     int tot = 0;
3.     for (int i = 0; i < CANT_B; i++){
4.         tot = tot + matriz[i][j]; //suma toda la columna
5.     }
6.     System.out.println(tot / CANT_B);
7. }
```

Imprime el promedio para cada columna de la matriz.	Verdadero <b>PERO</b> <sup>22</sup> , tot debería ser de tipo double para guardar el promedio, si es int solo guarda la parte entera y pierde la parte fraccionaria.
Imprime el promedio para cada fila de la matriz.	Falso
Imprime el promedio de todos los elementos de la matriz.	Falso
Ninguna de las opciones anteriores son correctas.	Falso

- De las siguientes sentencias, seleccione la que sea VERDADERA

En Java no hay una estructura de control que sea de iteración post-condicional.	Falso, existe el do-while
En Java los índices de los arreglos son enteros y comienzan desde 0.	Verdadero
La declaración de una matriz y su creación sólo se puede hacer todo junto en una línea de código.	Falso, se puede dividir: int [][] matriz; matriz = new int[filas][col]
En Java, una matriz puede almacenar datos de distintos tipos de datos primitivos a la misma vez.	Falso, las matrices son homogenas.

<sup>22</sup> En una autoevaluación la cátedra la tomó como verdadera, evaluando solo el algoritmo, pero si se considerara el tipo de datos, hay que corregir que tot sea double

- Indique que hace el siguiente código:

```
for (int i=10; i > 0; i=i-2){
    System.out.print(i + " - ");
}
```

El código no es correcto. Se decrementa de manera incorrecta el índice del for.	Falso
Imprime: 10 - 8 - 6 - 4 - 2 - 0 -	Falso, no cumple con la condición i>0
Imprime: 10 - 8 - 6 - 4 - 2 -	Verdadero
Imprime: 8 - 6 - 4 - 2 - 0 -	Falso
Ninguna de las anteriores.	Falso

- Indique qué imprime el siguiente código:

```
1. int[] v = new int[10];
2. for (int i=0;i<6;i++)
3.     v[i]=i/2;
4. for (int i=0;i<6;i++)
5.     System.out.print(" v["+i+"]=" + v[i]);
```

v[0]=0 v[1]=0 v[2]=1 v[3]=1 v[4]=2 v[5]=2	Verdadero
v[0]=0 v[1]=0.5 v[2]=1 v[3]=1.5 v[4]=2 v[5]=2.5	Falso
v[0]=0 v[1]=0 v[2]=1 v[3]=1 v[4]=2 v[5]=2 v[6]=3	Falso no cumple con la condición i < 6
v[0]=0 v[1]=0.5 v[2]=1 v[3]=1.5 v[4]=2 v[5]=2.5 v[6]=3	Falso
Ninguna de las anteriores.	Falso

Alcanza con mirar que en línea 3 se guarda en una variable entera el resultado de una división entre enteros. Por lo que se guarda solo la parte entera, todos los resultados con coma quedan descartados.

- ¿El siguiente código imprime la matriz correctamente?

```
1. int i,j;
2. for (i = 0; i <= dimF; i++) {
3.     for (j = 0; j <= dimC; j++) {
4.         System.out.print(matriz[i-1][j-1] + " ");
5.     }
6.     System.out.println("");
7. }
```

Sí, porque si bien en las líneas 2 y 3 escribe `<=dimF` y `<=dimC` (lo que sería un error), en línea 4 resta 1 en cada coordenada de la matriz, lo que termina corrigiéndolo. No es la forma habitual de realizar este algoritmo.



## Conceptos de POO

- Marque la afirmación verdadera acerca del concepto de **clase e instancia**.

Las clases son instancias	Falso, los objetos son instancias de clases
Una clase es un molde o plantilla a partir de la cual se instancian objetos.	Verdadero
Las acciones del objeto se implementan a través de variables de instancia.	Falso las acciones se implementan a través de métodos
A partir de una clase solo puedo instanciar un objeto.	False, puede haber muchos objetos de de la misma clase
Ninguna es correcta	Falso

- Marque la afirmación verdadera sobre **Clases**

Un método de instancia debe tener al menos un parámetro de entrada.	Falso, contraejemplo los constructores vacíos, también el toString
Un método de instancia no puede retornar un objeto.	Falso, ¡todo es un objeto!
El estado de los objetos se define público para lograr el encapsulamiento.	Falso, se define private para encapsular
Un método de instancia puede no retornar valor alguno.	Verdadero, se usa la palabra void
Ninguna de las afirmaciones anteriores es verdadera.	F

- Marque la afirmación verdadera sobre **constructores**

Un constructor debe recibir obligatoriamente un valor inicial para cada variable de instancia definida para el objeto.	Falso
Una clase solo puede definir un único constructor.	Falso
Un constructor permite inicializar el objeto que está siendo creado.	Verdadero
El programador de una clase debe obligatoriamente implementar un constructor.	Falso

- Marque la afirmación verdadera sobre el **estado de un objeto**

El estado se refiere a los métodos que puede ejecutar.	Falso, lo métodos refieren al comportamiento
El estado se refiere a los valores almacenados en sus atributos o variables de instancia en un momento dado.	Verdadero
El estado se refiere a la referencia en memoria a otros objetos.	Falso
El estado se refiere a la clase a la que pertenece.	Falso

- Dada la variable saludo1 y saludo2 de tipo String, marque la/las afirmación/es correcta/s.

La comparación (saludo1 == saludo2) da error.	Falso
La comparación (saludo1 == saludo2) indica si el contenido es igual.	Falso
La comparación (saludo1 == saludo2) indica si se trata de la misma instancia de objeto.	Verdadero, compara si apuntan al mismo objeto
La comparación saludo1.equals(saludo2) indica si el contenido es igual.	Verdadero
La comparación saludo1.equals(saludo2) indica si se trata de la misma instancia de objeto.	Falso

- Se desea instanciar un vector con información de los autores que escribieron un libro. De cada autor interesa conocer nombre y edad. Indique si el siguiente código es correcto justifique:

```

1. Autor[] autores = new Autor[10];
2. Autor autor = new Autor();
3. for(int i = 0; i < 10; i++){
4.     autor.setNombre(Lector.leerString());
5.     autor.setEdad(Lector.leerInt());
6.     autores[i] = autor;
7. }
```

Es incorrecto. Se crea una sola instancia de autor, por lo que todas las posiciones del vector están apuntando a la misma instancia, que termina por almacenar solo los últimos datos ingresados. Para que funcione, la línea 2 debería estar dentro de for, en línea 4.

Es un ejercicio interesante para entender cómo con la sentencia **new** se crea espacio en memoria para almacenar un objeto y la dirección de ese espacio en memoria se guarda en la variable *autor* en este caso.

## Herencia y polimorfismo

- Dada la siguiente jerarquía de clases, marque la opción correcta:

```
public class Foo extends Doe { . . . }
public class Bar extends Doe { . . . }
public class Doe { . . . }
```

Foo es superclase de Bar y Bar es superclase de Doe.	Falso
Foo y Bar son superclases de Doe.	Falso
Doe es superclase de Bar y Foo.	Verdadero
La jerarquía depende del orden en que se instancian las clases.	Falso

- Dada la siguiente jerarquía de clases, marque la opción correcta:

```
public class Foo extends Bar{           |           public class Bar{
    private int a;                       |       public int x;
    public void m() {                     |       private int y;
        ...                               |       public void o() {
    }                                     |           ...
}                                       |       }
                                     |   }
```

Foo hereda solo la variable y.	Falso
Foo hereda solo la variable x y el método o.	Falso
Foo hereda solo las variables x e y.	Falso
Foo lo hereda todo, las dos variables y el método.	Verdadero

- Dadas las siguientes clases:

```
public class Foo extends Bar {           |           public abstract class Bar {
    private int a;                       |       private int x;
    public void more() {                 |       public void something() {
        ...                               |           ...
    }                                     |       }
}                                       |   }
```

Es posible crear una instancia de Foo.	Verdadero
Es posible crear una instancia de Bar.	Falso, es abstracta
Sea f una instancia de tipo Foo, se puede hacer f.something().	Verdadero, hereda el método, y es público
Sea f una instancia de tipo Foo, se puede hacer f.more().	Verdadero, el método es público

Dadas las siguientes clases:

```
public class Foo{
    private int a;
    public void more() {
        ...
    }
}

public class Bar extends Foo {
    public void something() {
        ...
    }
}
```

Es posible crear una instancia de Foo.	Verdadero
Es posible crear una instancia de Bar.	Verdadero
Sea f una variable que almacena una instancia de la clase Foo, se puede hacer f.something().	Falso, ese método es de una subclase, por lo tanto <b>no</b> lo hereda Foo que es su superclase
Sea b una variable que almacena una instancia de la clase Bar, se puede hacer b.more().	Verdadero, hereda ese método y es público

- Dada la siguiente jerarquía de clases, indique qué imprime el programa:

```
public class Foo extends Bar {
    private int a = 3;

    public int more(int m) {
        return a + m;
    }
    public int something() {
        return super.more(a);
    }
}

public class Bar {
    private int x = 2;

    public int more(int m) {
        return m + x;
    }
    public int something() {
        return more(x);
    }
    public int andMore() {
        return more(x);
    }
}
```

```
public static void main(String[] args) {
    Foo f = new Foo();
    System.out.println(f.something());
    System.out.println(f.andMore());
}
```

5 y 5	Verdadero, cuando el método de la superclase <b>andMore</b> se ejecuta, llama al método <b>more</b> , y por <i>binding dinámico</i> se ejecuta el método de la subclase, es decir ejecuta el método <b>more</b> de la Clase Foo, pero le pasa el parámetro x de la Clase Bar.
6 y 4	Falso
5 y 4	Falso
6 y 5	Falso
No imprime nada el código no compila	Falso

# Programación concurrente

## Definiciones

La concurrencia es la capacidad de ejecutar múltiples tareas simultáneamente.

Un programa concurrente se divide en tareas (2 o más), las cuales se ejecutan al mismo tiempo y realizan acciones para cumplir un objetivo común. Para esto pueden: compartir recursos, coordinarse y cooperar. Cualquier lenguaje que brinde concurrencia debe proveer mecanismos para **comunicar** y **sincronizar** procesos. Estos mecanismos pueden clasificarse en:

- *Comunicación*
  - *Pasaje de Mensajes:*
    - **EnviarMensaje** : El envío de mensajes es *asincrónico*, es decir, el robot que envía el mensaje lo hace y sigue procesando sin esperar que el robot receptor lo reciba.
      - ❖ Puede enviarse un valor o una variable.
      - ❖ El envío debe incluir el nombre de una variable robot (no el tipo).
      - ❖ Solo se puede enviar un valor en cada mensaje.
    - **RecibirMensaje**: La recepción de mensajes es *sincrónica*, es decir, el robot que espera un mensaje NO sigue procesando hasta que recibe el mensaje.
      - ❖ El mensaje de recepción es siempre en una variable.
      - ❖ La recepción debe incluir el nombre de una variable robot (no el tipo).
- *Sincronización*
  - *Memoria Compartida: Bloquear - Liberar*
    - **BloquearEsquina**: Dado un recurso compartido (por 2 o más procesos) que está disponible, se bloquea ese recurso para que otro proceso no pueda accederlo
      - ❖ Hay que bloquear un recurso cuando puede ser accedido por dos o más procesos de un programa.
      - ❖ Puede realizarlo el programador o el sistema operativo.
      - ❖ Solo se bloquea un recurso libre.
      - ❖ Bloquear una esquina impide que otro pueda bloquearla pero no que pueda posicionarse, si un robot bloquea una esquina y el otro se posiciona sin bloquear antes existe riesgo de colisión.
      - ❖ Para maximizar la concurrencia el bloqueo de un recurso debe ser por el mínimo tiempo posible, realizar toda tarea que sea independiente al recurso compartido y solo bloquear antes de acceder a dicho recurso.
      - ❖ No pueden quedar esquinas bloqueadas al finalizar el programa.
    - **LiberarEsquina**: Dado un recurso compartido (por 2 o más procesos) que esté *bloqueado*, el programador libera dicho recurso para que cualquier proceso pueda *bloquearlo*.
      - ❖ Solo se desbloquea un recurso bloqueado.
      - ❖ Puede hacerlo el programador o el sistema operativo.
      - ❖ Hay que liberar un recurso cuando ya no existe peligro de colisión con otro proceso del programa.

## Tipos de áreas

<b>areaC</b>	Área compartida (región a la que pueden acceder todos los robots declarados)
<b>areaPC</b>	Área parcialmente compartida (región a la que pueden acceder más de un robot pero no todos los robots declarados)
<b>areaP</b>	área privada (región a la que puede acceder sólo un robot de los declarados)

Todos los robots declarados deben estar asignados al menos a un área.

Un robot puede estar asignado a una o más áreas del programa.

```
AsignarArea (variableRobot, nombreArea)
```

## Patrones de resolución de problemas

### Sincronización por barrera

Múltiples procesos se ejecutan concurrentemente, hasta que llegan a un punto especial, llamado barrera.

Los procesos que llegan a la barrera deben detenerse y esperar que lleguen todos los procesos.

Cuando todos los procesos alcanzan la barrera, podrán retomar su actividad (hasta finalizar o hasta alcanzar la próxima barrera). Para esto los procesos deben avisar que llegaron

### Passing the baton

Múltiples procesos se ejecutan en concurrente.

Sólo un proceso a la vez, el que posee el testigo (baton), se mantiene activo. Cuando el proceso activo completa su tarea, entrega el baton a otro proceso. El proceso que entregó el baton queda a la espera hasta recibirlo nuevamente. El proceso que recibió el baton completa su ejecución. Al completar su tarea, pasará el baton a otro proceso. Para esto los procesos deben tener una forma de comunicarse con el otro proceso.

¿Dónde está la concurrencia si cada proceso debe esperar a que termine el otro? Mientras los procesos esperan que se les pase la posta pueden estar realizando otras tareas que no dependan de la posta, o puede darse un pipeline donde el proceso1 esté procesando el dato 3, proceso2 esté con el dato 2 y proceso3 esté con el dato 1.

## Productor / Consumidor

Existen dos tipos de procesos:

- Productores: trabajan para generar algún recurso y almacenarlo en un espacio compartido.
- Consumidores: utilizan los recursos generados por los productores para realizar su trabajo.

Para esto los procesos deben coordinar donde almacenan los datos los productores, cuando saben los consumidores que hay datos.

## Cliente / Servidor

Los procesos se agrupan en dos categorías:

- Servidores: permanecen inactivos hasta que un cliente les solicita algo. Cuando reciben una solicitud, realizan su tarea, entregan el resultado y vuelven a “dormir” hasta que otro cliente los despierte.
- Clientes: realizan su trabajo de manera independiente, hasta que requieren algo de un servidor. Entonces realizan una solicitud a un proceso servidor, y esperan hasta que recibir la respuesta. Cuando esto sucede, el cliente continúa su trabajo.

Para esto los procesos cliente deben realizar sus pedidos y el servidor debe administrar como los atiende.

**Ejemplo:** Existe un robot que sirve de flores a tres robots clientes. Cada cliente solicita al servidor que le deposite en su esquina siguiente una cantidad de flores aleatoria, el cliente las junta y las deposita una a una a lo largo de la avenida en la que se encuentra. El programa finaliza cuando todos los robots clientes completan su avenida.

Protocolo **cliente / servidor:**

Cliente:	Servidor
INICIO: calcularRandom flores Enviar ID al servidor Enviar cantFlores al servidor Enviar mi Avenida actua Enviar Calle siguiente Esperar ACK del servidor Ir a la esquina Avenida,Calle JuntarFlores Volver a la esquina Avanzar dejando flores Si llegué a la avenida 100 enviar 0 al servidor sino Volver a INICIO	INICIO: Recibir ID Recibir N Flores de ID si (flores <> 0) recibir avenida de ID recibir calle de ID pos(avenida,calle) depositar N flores volver a (100,100) enviar ACK a robot ID volver a INICIO sino contar un robot terminado si terminaron los 3 robots terminar

(ACK= acknowledgment usado para recibir confirmacion)

## Master / Slave

Los procesos se agrupan en dos categorías:

- Maestro: deriva tareas a otros procesos (trabajadores), determina cuantos trabajadores necesita, cómo les reparte la tarea, cómo recibe los resultados.
- Esclavos: realizan la tarea solicitada y envían el resultado al jefe, quedando a la espera de la siguiente tarea.

## Preguntas múltiple choice

- Qué afirmación es correcta:

Para que un programa sea concurrente, debe existir más de un procesador.	Falso, eso es para el procesamiento en paralelo
Para que un programa sea concurrente, debe existir más de un proceso.	Verdadero
En un programa concurrente todos los procesos están obligados a realizar la misma tarea.	Falso, de hecho concurrencia es poder hacer diferentes tareas a la vez.
En un programa concurrente todos los procesos deberán comunicarse entre ellos al menos una vez.	Falso, depende el problema a resolver.

- Dos procesos deben indicar la cantidad de veces que se encuentra un valor X recibido como parámetro en un vector de tamaño 20. Para solucionar este problema, se propone la siguiente solución en un pseudocódigo similar a Pascal. Marque la o las opciones correctas.

Variable v : vectorDeEnteros; //variable compartida por los procesos	
proceso buscadorCreciente(valor : integer) variables cant, i : integer; begin cant := 0; for i:= 1 to 10 do if (V[i] = valor) then cant := cant + 1; writeln(cant); end;	proceso buscadorDecreciente(valor : integer) variables cant, i : integer; begin cant := 0; for i:= 20 downto 11 do if (V[i] = valor) then cant := cant + 1; writeln(cant); end;

La solución es correcta. Ninguno de los procesos modifica las posiciones del vector.	Verdadero, uno va de 20 a 11 mientras el otro de 1 a 10, no acceden a las mismas posiciones.
La solución es incorrecta, dado que los dos procesos acceden de manera concurrente a las mismas posiciones del vector y no se está protegiendo el acceso.	Falso
La solución es incorrecta, dado que no se está protegiendo el acceso a la variable cant.	Falso, cant no es un recurso compartido
La solución es incorrecta: dado que el vector V es una variable compartida y también debe serlo la variable índice i.	Falso



- Dado el siguiente programa, marque la o las opciones correctas

```

programa ejemplo1
procesos
  proceso juntarFlores
  comenzar
    mientras HayFlorEnLaEsquina
      tomarFlor
    fin
areas
  avenidas : AreaP(1,1,2,11)
robots
  robot florero
  comenzar
    repetir 10
      juntarFlores
    mover
  fin
variables
  robot1, robot2 : florero
comenzar
  AsignarArea(robot1,avenidas)
  AsignarArea(robot2,avenidas)
  Iniciar(robot1,1,1)
  Iniciar(robot2,2,1)
fin

```

El programa es correcto	Falso
El programa es incorrecto: los robots pueden chocar en una esquina	Falso
El programa es incorrecto: los robots no pueden ejecutar el proceso juntarFlores a la vez.	Falso
El programa es incorrecto: el área avenidas debe ser de tipo Compartida.	Verdadero
El programa es incorrecto: el área avenidas debe ser de tipo Parcialmente Compartida.	Falso, para que sea PC debería haber un tercer robot que no ingrese a dicha área.
El programa es incorrecto: los robots se salen del área asignada.	Falso.

```

programa DosRobots
areas
ciudad : AreaC(1,1,100,100)
robots
  robot robot1
  comenzar
    Pos(1,100)
    si (HayFlorEnLaEsquina)
      tomarFlor
    Pos(50,50)
    fin
  robot robot2
  comenzar
    repetir 99
      mientras (HayPapelEnLaEsquina)
        tomarPapel
      mover
    mientras (HayPapelEnLaEsquina)
      tomarPapel
  fin
variables
  r1 : robot1
  r2 : robot2
comenzar
  AsignarArea(r1,ciudad)
  AsignarArea(r2,ciudad)
  Iniciar(r1,2,2)
  Iniciar(r2,1,1)
Fin

```

El programa es correcto y maximiza la concurrencia.	Falso
El programa es correcto pero no maximiza la concurrencia.	Falso
El programa es correcto: si bien los robots van a la misma esquina, no hay riesgo de choque.	Falso, no se puede afirmar.
El programa es incorrecto: los robots podrían chocar.	Verdadero si bien r2 realiza más tareas que r1, el procesador podría darle el control más veces y llegar a la esquina compartida antes de que r1
El programa es incorrecto: la definición de las áreas es incorrecta.	Falso

- Dado los siguientes 3 robots: (r1 de tipo1, r2 de tipo2 y r3 de tipo3), indique cual o cuáles de estas afirmaciones son correctas:

r1:	r2:	r3:
<pre>robot tipo1 variables   ok: numero comenzar   EnviarMensaje(1, r2)   RecibirMensaje(ok, r3) fin</pre>	<pre>robot tipo2 variables   ok: numero comenzar   RecibirMensaje(ok, r1)   EnviarMensaje(1, r3) fin</pre>	<pre>robot tipo3 variables   ok: numero comenzar   RecibirMensaje(ok, r2)   EnviarMensaje(1, r1) fin</pre>

No se puede determinar quién termina primero.	Falso
r2 termina su tarea primero.	Verdadero: r2, r3 esperan recibir mensajes- r1 envía mensaje a r2 y espera mensaje de r3 r2 recibe de r1 y envía mensaje a r3 – Termina r2 r3 recibe de r2 y envía a r1 – Termina r3 r1 recibe de r3 – Termina r1
r1 termina su tarea último.	Verdadero
Ninguno termina.	Falso
Solo termina r2.	Falso

- Dado el siguiente bloque de código indique la o las opciones correctas:

```

areas
  ciudad : AreaC(1,1,10,10)
robots
  robot jefe
  variables
    id : numero
  comenzar
    EnviarMensaje(1,robot1)
    EnviarMensaje(2,robot2)
    repetir 100
      EnviarMensaje(1,robot1)
      RecibirMensaje(id,robot1)
      EnviarMensaje(1,robot2)
      RecibirMensaje(id,robot2)
  fin
  robot juntador
  variables
    yo,aux , miCa, miAv: numero
  comenzar
    RecibirMensaje(yo,robotJefe)
    miCa := PosCa
    miAv := PosAv
    repetir 100
      RecibirMensaje(aux,robotJefe)
      Pos(5,5)
      si (HayFlorEnLaesquina)
        tomarFlor
      EnviarMensaje(yo,robotJefe)
      Pos(miAv,miCa)
  fin
  variables
    robotJefe : jefe
    robot1, robot2 : juntador
  comenzar
    AsignarArea(robotJefe,ciudad)
    AsignarArea(robot1,ciudad)
    AsignarArea(robot2,ciudad)
    Iniciar(robotJefe,3,3)
    Iniciar(robot1,1,1)
    Iniciar(robot2,2,2)
  fin

```

El programa es correcto y no existe riesgo de colisión en la esquina (5,5) dado que el robot Jefe controla el acceso secuencial a la misma.

Falso

El programa es incorrecto, ya que si bien el Jefe controla el acceso a la esquina (5,5), existe riesgo de colisión esa esquina debido a que los robots juntadores vuelven a su esquina después de enviar el mensaje al Jefe, deberían volver antes de enviar el mensaje al jefe.

Verdadero

El programa es incorrecto ya que los robots juntadores podrían quedarse bloqueados esperando mensajes que nunca le enviarán

Falso

El programa es incorrecto ya que el robot Jefe podría quedarse bloqueado esperando mensajes que nunca le enviarán.

Falso, se reciben todos los mensajes que se envían.

El programa es incorrecto ya que los robots juntadores podrían bloquearse intentando enviar un mensaje al Jefe.

Falso

- Tres robots deben juntar flores en la esquina donde se encuentra parado cada uno. Además, un robot coordinador debe informar el robot que finalizó primero y la cantidad de flores que juntó. Indique a continuación la o las opciones correctas

```
robots
robot robotJugador
variables
  miCodigo, flores : numero
comenzar
  flores := 0
  RecibirMensaje(miCodigo,coordinador)
  mientras (HayFlorEnLaEsquina)
    tomarFlor
    flores := flores + 1
  EnviarMensaje(miCodigo,coordinador)
  EnviarMensaje(flores,coordinador)
fin
```

```
robot robotCoordinador
variables
  ganador, flores : numero
comenzar
  EnviarMensaje(1,jugador1)
  EnviarMensaje(2,jugador2)
  EnviarMensaje(3,jugador3)
  RecibirMensaje(ganador,*)
  RecibirMensaje(flores,*)
  Informar(ganador)
  Informar(flores)
fin
```

```
variables
jugador1, jugador2, jugador3: robotJugador
coordinador : robotCoordinador
... {continuación del resto del código}
```

El robot jugador1 siempre finalizará primero porque siempre recibe primero su código.	Falso
Los robots jugadores no podrán comenzar a juntar las flores hasta no haber recibido sus respectivos códigos.	Verdadero, la instrucción recibir los traba hasta que reciban
El robot coordinador siempre informará correctamente el código del robot ganador.	Verdadero, el que termine primero se almacenará en la variable ganador del jefe
El robot coordinador siempre informará correctamente la cantidad de flores que juntó el robot ganador.	Falso, en RecibirMensaje(flores,*) puede recibir el id de alguno de los otros robots
Nunca un robot puede recibir dos mensajes seguidos usando el asterisco.	Falso

- Dado el siguiente bloque de código, en el cual un robot jefe debe informar la cantidad de robots trabajadores que encontraron más de una cierta cantidad de papeles en sus esquinas. Indique la o las opciones correctas:

```
robot robotTrabajador
variables
  misPapeles, max : numero
comenzar
  RecibirMensaje(max, jefe)
  misPapeles := 0
  mientras (HayPapelEnLaEsquina)
    tomarPapel
    misPapeles := misPapeles + 1
  si (misPapeles > max)
    EnviarMensaje(misPapeles, jefe)
fin
```

```
robot robotJefe
variables
  tusPapeles, cant : numero
comenzar
  Random(cant, 1, 10)
  EnviarMensaje(cant, r1)
  EnviarMensaje(cant, r2)
  EnviarMensaje(cant, r3)
  EnviarMensaje(cant, r4)
  cant := 0
  repetir 4
    RecibirMensaje(tusPapeles, *)
    cant := cant + 1;
  Informar(cant)
fin
```

```
variables
r1, r2, r3, r4 : robotTrabajador
jefe : robotJefe
{resto del código}
```

El programa es incorrecto, los robots trabajadores no pueden calcular si superan el valor máximo por sí mismos.	Falso
El programa es incorrecto, el robot jefe podría quedarse esperando mensajes que no van a llegar.	Verdadero, el RecibirMensaje del jefe está dentro de un repetir 4, pero puede ocurrir que no todos junten más flores que max, por lo tanto quedaría esperando un recibir un mensaje que no llegará.
El programa es incorrecto, no se han inicializado las variables cant y max correctamente.	Falso
El programa es incorrecto, el robot jefe debería recibir los mensajes respetando el orden en que los había enviado previamente.	Falso, eso arruinaría la concurrencia

- Dado el siguiente programa indique la o las opciones correctas:

<pre> proceso juntarFlores (ES flores:numero) comenzar   flores := 0   mientras HayFlorEnLaEsquina     tomarFlor     flores := flores + 1   fin  ROBOT JUNTADOR:   robot robotJuntador variables   flores : numero comenzar   repetir 99     juntarFlores(flores)     EnviarMensaje(flores,coordinador)   mover   juntarFlores(flores)   EnviarMensaje(flores,coordinador) fin  ROBOT COORDINADOR:   robot robotCoordinador variables   cant,total : numero comenzar   total := 0   repetir 200     RecibirMensaje(cant,*)     total := total + cant   Informar(total) fin variables   robot1,robot2 : robotJuntador   coordinador : robotCoordinador {Resto del código} </pre>	El programa es correcto y maximiza la concurrencia, ya que el robot coordinador es el encargado de acumular los resultados a medida que se van obteniendo, haciendo que los robots juntadores trabajen más rápido.	Falso
	El programa es correcto pero no maximiza la concurrencia. Se podría mejorar la concurrencia haciendo que los jugadores acumulen las cantidades de cada esquina y envíen un solo mensaje al coordinador con el total para que los sume.	Verdadero
	El programa es correcto pero no maximiza la concurrencia. Se podría mejorar la concurrencia haciendo que el coordinador reciba los mensajes de cada jugador de manera intercalada.	Falso, Si uno es más rápido tendría que esperar al otro
	El programa es incorrecto, ya que el robot coordinador necesita saber qué robot juntador le está enviando el valor cant en cada mensaje.	Falso, solo va a informar el total juntado, no quien juntó más o terminó primero
	El programa es incorrecto, ya que el robot coordinador podría no recibir los 200 mensajes.	Falso
	El programa es incorrecto, ya que los mensajes de los robots juntadores podrían sobrecribirse	Falso, siempre envían el dato cantidad.

- Dado el siguiente bloque de código indique la o las respuestas correctas:

<pre> robot robot1 comenzar   repetir 10     BloquearEsquina (4,20)     Pos (4,20)     Si HayPapelEnLaEsquina       tomarPapel     Pos (1,1)     LiberarEsquina (4,20) fin </pre>	<pre> robot robot2 comenzar   repetir 10     Pos (4,20)     si HayFlorEnLaEsquina       tomarFlor     Pos (2,2) fin </pre>
---	--

Sólo existe riesgo de choque si el robot1 llega primero a la esquina (4,20).	Falso
Sólo existe riesgo de choque si el robot2 llega primero a la esquina (4,20).	Falso
Existe riesgo de choque para ambas situaciones descritas previamente.	Verdadero, esto muestra que bloquear impide que el otro bloquee pero <b>no impide que se posicione</b> en la esquina bloqueada.
No existe riesgo de choque, ya que el robot1 protege el acceso a la esquina (4,20).	Falso

- Dado el siguiente bloque de código indique cuál o cuáles son correctas:

<pre> robot robot1 comenzar   repetir 10     BloquearEsquina (4,20)     Pos (4,20)     si HayPapelEnLaEsquina       tomarPapel     Pos (1,1)     LiberarEsquina (4,20) fin </pre>	<pre> robot robot2 comenzar   repetir 10     BloquearEsquina (4,20)     Pos (4,20)     si HayFlorEnLaEsquina       tomarFlor     LiberarEsquina (4,20)     Pos (2,2) fin </pre>
---	---

El robot2 podría ingresar a la esquina (4,20) mientras el robot1 aún está allí, y generar una colisión	Falso, r2 bloqueaEsquina antes de posicionarse y no puede hacerlo hasta que r1 la libere
El robot1 podría ingresar a la esquina (4,20) mientras que el robot2 aún permanece en dicha esquina y de esa manera provocar una colisión	Verdader, porque r2 la libera antes de irse
No existe riesgo de colisión porque se protege el acceso a la esquina (4,20).	Falso
No existe riesgo de colisión. No es necesario proteger el acceso a la esquina (4,20).	Falso



- Dado el siguiente bloque de código con 3 robots, uno florero de tipo robotFlorero, otro papelero de tipo robotPapelero y un jefe de tipo robotJefe. Indique la o las opciones correctas

<pre> robot robotJefe variables ok : numero comenzar   EnviarMensaje(1, florero)   RecibirMensaje(ok, florero)   EnviarMensaje(1, papelero)   RecibirMensaje(ok, papelero) fin  robot robotPapelero variables   num : numero comenzar   RecibirMensaje(num, jefe)   BloquearEsquina(30, 30)   Pos(30, 30)   mientras (HayPapelEnLaEsquina)     tomarPapel   Pos(1, 1)   LiberarEsquina(30, 30)   EnviarMensaje(1, jefe) fin  robot robotFlorero variables   num : numero comenzar   RecibirMensaje(num, jefe)   Pos(30, 30)   si (HayFlorEnLaEsquina)     tomarFlor   Pos(2, 2)   EnviarMensaje(1, jefe) fin </pre>	No es necesario que el robot papelero proteja la esquina (30,30) porque no hay riesgo de colisión.	Verdadero, porque papelero se posiciona luego de que jefe reciba el ok de florero y el ok de florero se envía cuando éste ya dejó la esquina
	Es necesario que el robot florero proteja la esquina (30,30) antes de posicionarse porque hay riesgo de colisión.	Falso
	El robot papelero siempre terminará antes que el florero.	Falso, siempre arranca una vez que terminó florero.
	El robot florero terminará siempre antes que el papelero.	Verdadero
	No se puede determinar qué robot (papelero o florero) finalizará primero.	Falso, en este caso lo determina la recepción y el envío de mensajes
	El robot jefe siempre terminará primero.	Falso, debe esperar a que termine papelero.

- Tres robots deben juntar flores dentro de un área. Un robot jefe debe determinar qué robot juntó más flores. Indique la o las opciones correctas:

```

programa ganadores
procesos
  proceso juntarFlores (ES flores: numero)
  comenzar
    mientras (HayFlorEnLaEsquina)
      tomarFlor
      flores:= flores + 1
  fin

  proceso ActualizarMax (E ID:numero;E flores:numero;ES maxID:numero;ES max:numero)
  comenzar
    si (max > flores)
      maxID:= ID
      max:= flores
  fin
areas
ciudad: areaC(1,1,100,100)

```

```

robots
robot florero
variables
  miCa,miAv,unaCa,unaAv,flores,id:numero
comenzar
  miCa := PosCa
  miAv := PosAv
  Random(unaCa,2,100)
  Random(unaAv,1,100)
  BloquearEsquina(unaAv,unaCa)
  Pos(unaAv,unaCa)
  juntarFlores(flores)
  Pos(miAv,miCa)
  LiberarEsquina(unaAv,unaCa)
  RecibirMensaje(id,jefe)
  EnviarMensaje(id,jefe)
  EnviarMensaje(flores,jefe)
fin

```

```

robot robotJefe
variables
  max, maxID, ID, flores : numero
comenzar
  EnviarMensaje(1,florero1)
  EnviarMensaje(2,florero2)
  EnviarMensaje(3,florero3)
  max := -1
  { recibo los resultados del robot 1}
  RecibirMensaje(ID,robot1)
  RecibirMensaje(flores,robot1)
  ActualizarMax(ID,flores,maxID,max)
  { recibo los resultados del robot 2}
  RecibirMensaje(ID,robot2)
  RecibirMensaje(flores,robot2)
  ActualizarMax(ID,flores,maxID,max)
  { recibo los resultados del robot 3}
  RecibirMensaje(ID,robot3)
  RecibirMensaje(flores,robot3)
  ActualizarMax(ID,flores,maxID,max)
  Informar(maxID)
fin

```

```

variables
  florero1, florero2, florero3: florero
  jefe: robotJefe
comenzar
  AsignarArea(florero1,ciudad)
  AsignarArea(florero2,ciudad)
  AsignarArea(florero3,ciudad)
  AsignarArea(jefe,ciudad)
  Iniciar(florero1,1,1)
  Iniciar(florero2,2,1)
  Iniciar(florero3,3,1)
  Iniciar(jefe,4,1)
fin

```

Los robots deberían preguntar si la esquina está ocupada antes de ingresar.	Falso
El robot jefe debe inicializar las variables flores e ID	Falso
No es necesario que los robots florero protejan el acceso a la esquina aleatoria, ya que el riesgo de choque es casi nulo	Falso
El jefe podría calcular de manera incorrecta al robot ganador, ya que no puede asegurar que el segundo mensaje que lee proviene del mismo robot que el primero.	Falso
La manera en que el jefe recibe los mensajes no maximiza la concurrencia.	Verdadero, porque recibe los datos desde el robot1 al robot3, y puede haber terminado primero el robot3 o el 2, y tendrían que esperar su turno para entregar su información.
Los robots floreros no pueden realizar su trabajo de recolectar hasta que el jefe les envía su ID.	Falso, reciben su id después de terminar su tarea
Si los robots florero reciben los mensajes con la instrucción RecibirMensaje(id, *) el programa se comporta igual	Verdadero
No es necesario recibir los ID. Cada robot florero ya conoce su número de robot.	Falso

## Instrucciones

Sintaxis	Semántica
AsignarArea(robot, area)	La instrucción asigna al “robot” en un “área” de trabajo. Recibe dos parámetros: el robot y el área que debe asignarse.
Iniciar (robot, avenida, calle)	Instrucción primitiva que posiciona al robot en la esquina indicada orientado hacia el norte.
derecha	Instrucción primitiva que cambia la orientación del robot en 90° en sentido horario respecto de la orientación actual.
mover	Instrucción primitiva que conduce al robot de la esquina en la que se encuentra a la siguiente, respetando la dirección en la que está orientado. Es responsabilidad del programador que esta instrucción sea ejecutada dentro de los límites de la ciudad. En caso contrario se producirá un error y el programa será abortado.
tomarFlor	Instrucción primitiva que le permite al robot recoger una flor de la esquina en la que se encuentra y ponerla en su bolsa. Es responsabilidad del programador que esta instrucción sea ejecutada solo cuando haya al menos una flor en dicha esquina.
tomarPapel	Instrucción primitiva que le permite al robot recoger un papel de la esquina en la que se encuentra y ponerlo en su bolsa. Es responsabilidad del programador que esta instrucción sea ejecutada solo cuando haya al menos un papel en dicha esquina.
depositarFlor	Instrucción primitiva que le permite al robot depositar una flor de su bolsa en la esquina en la que se encuentra. Es responsabilidad del programador que esta instrucción sea ejecutada solo cuando haya al menos una flor en la bolsa.
depositarPapel	Instrucción primitiva que le permite al robot depositar un papel de su bolsa en la esquina en la que se encuentra. Es responsabilidad del programador que esta instrucción sea ejecutada solo cuando haya al menos un papel en la bolsa.
PosAv	Identificador que representa el número de avenida en la que el robot está actualmente posicionado. Su valor es un número entero en el rango 1..100.
PosCa	Identificador que representa el número de calle en la que el robot está actualmente posicionado. Su valor es un número entero en el rango 1..100.
HayFlorEnLaEsquina	Proposición atómica cuyo valor es V si hay al menos una flor en la esquina en la que el robot esta actualmente posicionado, ó F en caso contrario. Su valor no puede ser modificado por el programador.

HayPapelEnLaEsquina	Proposición atómica cuyo valor es V si hay al menos un papel en la esquina en la que el robot está actualmente posicionado, ó F en caso contrario. Su valor no puede ser modificado por el programador.
HayFlorEnLaBolsa	Proposición atómica cuyo valor es V si hay al menos una flor en la bolsa del robot, ó F en caso contrario. Su valor no puede ser modificado por el programador.
HayPapelEnLaBolsa	Proposición atómica cuyo valor es V si hay al menos un papel en la bolsa del robot, ó F en caso contrario. Su valor no puede ser modificado por el programador
Pos(avenida,calle)	Instrucción que requiere dos valores Av y Ca, cada uno de ellos deben estar en el rango 1..100, posiciona al robot en la esquina determinada por el par (Av,Ca) sin modificar la orientación del robot.
Informar(variable)	Instrucción que permite visualizar en pantalla el contenido almacenado en alguna variable. También puede informar una expresión numérica o lógica.
Leer(variable)	Instrucción que permite al programador leer un valor desde el teclado, asignado a “variable”. Los valores posibles dependen del tipo de variable. Pueden ser numéricos o booleanos.
Random(variable,inf,sup)	Instrucción que calcula un número aleatorio entre los límites “inf” y “sup” (límites incluidos), y retorna su valor “variable”.
BloquearEsquina(avenida, calle)	Instrucción para que el robot solicite el acceso exclusivo a la esquina. Compitiendo por el recurso con los demás robots solicitantes. Recibe dos parámetros, el número de avenida y el número de calle a bloquear.
LiberarEsquina(avenida, calle)	Instrucción para que el robot libere el acceso exclusivo a la esquina. Recibe dos parámetros, el número de avenida y el número de calle a liberar.
EnviarMensaje(variable, robot)	Instrucción que permite a un robot enviar un dato a otro. Recibe dos parámetros, el dato o variable a enviar y el robot que va a recibir el mensaje.
RecibirMensaje(variable, robot)	Instrucción que permite a un robot recibir un dato de otro. Recibe dos parámetros, la variable donde va a recibir el dato, y el robot de quien espera el mensaje. Si el parámetro del robot destinatario es *, el mensaje puede ser recibido de cualquier robot por orden de llegada.
RecibirMensaje(variable, *)	Permite a un robot recibir un dato de cualquier otro robot sin saber quién se lo está enviando.