

# Introducción a la programación.

---

# Contenidos:

---

- 1) Los paradigmas de programación
- 2) Tipos de datos
- 3) Representación de algoritmos
- 4) PSeInt

# Los paradigmas de programación

Los estilos de programación han evolucionado con el tiempo, pasando por diferentes períodos:

- Programación desestructurada (entre 1950 y 1970)
- Programación estructurada clásica (entre 1970 y 1990)
- Programación modular (entre 1970 y 1990)
- Programación orientada a objetos (desde 1990)

¿Qué es un algoritmo?

Un **algoritmo** es una secuencia ordenada de pasos que resuelven un problema en tiempo finito. Principales características de los algoritmos:

- Contienen instrucciones concretas, sin ninguna ambigüedad.
- Deben terminar, es decir, son finitos.
- Todos sus pasos son simples y están ordenados.

Un **programa** es la traducción de un algoritmo a un lenguaje de programación capaz de ser entendido y procesado por el ordenador. Generalmente, el orden de las instrucciones de un programa es el mismo en que se escriben, pero muchas veces es necesario repetir conjuntos de instrucciones (bucles) o realizar saltos de una instrucción a otra.

# Los paradigmas de programación

---

En la programación **desestructurada** clásica se usan bucles y saltos muy frecuentemente, dando lugar a un código farragoso, a veces confuso, que genera gran probabilidad de incurrir en errores y hace muy difícil el mantenimiento o evolución del programa. Los lenguajes más antiguos, como Fortran, Cobol, Basic, etc... se consideran desestructurados.

La programación **estructurada** consiste en la utilización de estructuras que optimizan los recursos lógicos y físicos del ordenador. Los lenguajes estructurados también se conocen como imperativos o de tercera generación. Por ejemplo, C, Pascal o Modula-2 son estructurados. Algunos lenguajes antiguos se adaptaron con el tiempo a este paradigma, aunque permitían seguir haciendo programación desestructurada si el programador lo deseaba.

La programación **modular** convive con la estructurada y con frecuencia se usa conjuntamente. Consiste en dividir el programa complejo en varios programas sencillos que interactúan entre ellos. Cada programa sencillo se llama "módulo". Los módulos deben ser independientes entre sí, no interferir unos con otros. El lenguaje modular clásico es "Modula-2".

La programación **orientada a objetos** (OOP / POO) es una evolución de la anterior. Es programación estructurada y modular al mismo tiempo, en la que las instrucciones y los datos se encapsulan en entidades denominadas "clases", de las que luego se crean los "objetos", que tienen una serie de propiedades y capacidades para realizar acciones o "métodos". Lo veremos más adelante.

# Evolución histórica de los lenguajes de programación

Con el paso del tiempo, se va incrementando el nivel de abstracción, pero en la práctica, los de una generación no terminan de sustituir a los de la anterior:

**Lenguajes de primera generación (1GL):** Código máquina.

**Lenguajes de segunda generación (2GL):** Lenguajes ensamblador, por ejemplo, COBOL 1959.

**Lenguajes de tercera generación (3GL):** La mayoría de los lenguajes modernos, diseñados para facilitar la programación a los humanos. Ejemplos: C, Java.

**Lenguajes de cuarta generación (4GL):** se ha dado este nombre a ciertas herramientas que permiten construir aplicaciones sencillas combinando piezas prefabricadas. Hoy se piensa que estas herramientas no son, propiamente hablando, lenguajes. Algunos proponen reservar el nombre de cuarta generación para la programación orientada a objetos.

**Lenguajes de quinta generación (5GL):** La intención es que el programador establezca qué problema ha de ser resuelto y las condiciones a reunir, y la máquina lo resuelve. Se usan en inteligencia artificial. Ejemplo: Prolog.

# Tipos de datos

---

Los programas manejan **datos**. Un tipo de datos es la propiedad de un valor que determina su dominio (qué valores puede tomar), qué operaciones se le pueden aplicar y cómo es representado internamente por el ordenador. Es decir, los datos se caracterizan por:

- **Dominio de posibles valores:** qué valores puede tomar.
- **Cómo se representan:** cuál es la representación interna y cómo se representan en el lenguaje de programación.
- **Operadores asociados:** qué operaciones o cálculos se pueden realizar con esos datos.

Todos los valores que aparecen en un programa tienen un **tipo**. Los tipos elementales o primitivos de un lenguaje de programación son:

- Entero (byte, short, int, long)
- Booleano (boolean)
- Carácter (char)
- Real (float, double)

# Tipos de datos

---

## Números enteros

El tipo `int` (del inglés `integer`, que significa «entero») permite representar números enteros. Los valores que puede tomar un `int` son todos los números enteros: ... -3, -2, -1, 0, 1, 2, 3, ...

Los números enteros literales se escriben con un signo opcional seguido por una secuencia de dígitos:

1570

+4591

-12

# Tipos de datos

---

## Números reales

El tipo float permite representar números reales. El nombre float viene del término "punto flotante", que es la manera en que el computador representa internamente los números reales. Hay que tener mucho cuidado, porque los números reales no se pueden representar de manera exacta en un computador. Por ejemplo, el número decimal 0.7 es representado internamente por el computador mediante la aproximación 0.69999999999999996. Todas las operaciones entre valores float son aproximaciones.

Los números reales literales se escriben separando la parte entera de la decimal con un punto. Las partes entera y decimal pueden ser omitidas si alguna de ellas es cero:

>>> 881.9843000	>>> -3.14159	>>> 1024.	>>> .22
881.9843	-3.14159	1024.0	0.22

Otra representación es la notación científica, en la que se escribe un factor y una potencia de diez separados por una letra e. Por ejemplo:

```
>>> -2.45E4
-24500.0
>>> 7e-2
```



# Tipos de datos

---

## Valores lógicos

Los valores lógicos True y False (verdadero y falso) son de tipo bool, que representa valores lógicos. El nombre bool viene del matemático George Boole, quien creó un sistema algebraico para la lógica binaria. Por lo mismo, a True y False también se les llama valores booleanos.

# Tipos de datos

---

## Texto

A los valores que representan texto se les llama strings, y tienen el tipo str. Los strings literales pueden ser representados con texto entre comillas simples o comillas dobles:

"ejemplo 1"

'ejemplo 2'

La ventaja de tener dos tipos de comillas es que se puede usar uno de ellos cuando el otro aparece como parte del texto:

"Let's go!"

'Ella dijo "hola"'

# Tipos de datos

---

Es importante entender que los strings no son lo mismo que los valores que en él pueden estar representados:

```
>>> 5 == '5' => False
```

```
>>> True == 'True' => False
```

Los strings que difieren en mayúsculas y minúsculas, o en espacios también son distintos:

```
>>> 'mesa' == 'Mesa' => False
```

```
>>> ' mesa' == 'mesa ' => False
```

## Nulo

Existe un valor llamado **None** (en inglés, «ninguno») que es utilizado para representar casos en que ningún valor es válido, o para indicar que una variable todavía no tiene un valor que tenga sentido.

# Representación de algoritmos: El pseudocódigo

Para la creación de programas, como paso previo a la utilización de lenguajes de programación, usamos lo que se conoce como "pseudocódigo", es decir, un lenguaje de especificación de algoritmos que usa una notación en lenguaje natural, permitiendo representar la programación estructurada y haciendo que el paso final a la codificación de un programa sea relativamente fácil.

El pseudocódigo debe cumplir estas reglas:

- Debe ser **preciso** e indicar el orden de realización de cada paso.
- Debe estar **definido**, es decir, debe dar un mismo resultado (si ponemos los mismos datos no puede dar un resultado distinto).
- Debe ser **finito**, es decir, debe de acabar en algún punto.
- Debe ser **independiente** de un lenguaje de programación


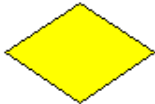








# Representación de algoritmos: Los diagramas de flujo

Un diagrama de flujo es la representación gráfica de un algoritmo o proceso. Estos diagramas utilizan símbolos con significados definidos que representan los pasos del algoritmo, y representan el flujo de ejecución mediante flechas que conectan los puntos de inicio y de fin del proceso.

Los diagramas de flujo utilizan una simbología y normas determinadas. Algunos de los símbolos más utilizados son:

- **Óvalo** o **Elipse**: **Inicio y Final** (Abre y cierra el diagrama).
- **Rectángulo**: **Actividad** (Representa la ejecución de una o más actividades o procedimientos).
- **Rombo**: **Decisión** (Formula una pregunta o cuestión).
- **Círculo**: **Conector** (Representa el enlace de actividades con otra dentro de un procedimiento).
- **Triángulo boca abajo**: **Archivo definitivo** (Guarda un documento en forma permanente).
- **Triángulo boca arriba**: **Archivo temporal** (Proporciona un tiempo para el almacenamiento del documento).

# Representación de algoritmos: Los diagramas de flujo

	<b>Inicio/Final</b> Se utiliza para indicar el inicio y el final de un diagrama; de inicio sólo puede salir una línea de flujo y al final sólo debe llegar una línea		<b>Decisión</b> Indica la comparación de dos datos y dependiendo del resultado lógico (falso o verdadero) se toma la decisión de seguir un camino del diagrama u otro
	<b>Entrada/Salida</b> Entrada/Salida de datos por cualquier dispositivo (scanner, lector de código de barras, micrófono, parlantes, etc.)		<b>Impresora/Documento.</b> Indica la presentación de uno o varios resultados en forma impresa
	<b>Entrada por teclado.</b> Entrada de datos por teclado. Indica que el computador debe esperar a que el usuario teclee un dato que se guardará en una variable o constante		<b>Pantalla</b> Instrucción de presentación de mensajes o resultados en pantalla
	<b>Acción/Proceso</b> Indica una acción o instrucción general que debe realizarse (operaciones aritméticas, asignaciones, etc.)		<b>Conector Interno</b> Indica el enlace de dos partes de un diagrama dentro de la misma página
	<b>Flujo/Flechas de Dirección</b> Indica el seguimiento lógico del diagrama. También indica el sentido de ejecución de las operaciones		<b>Conector Externo</b> Indica el enlace de dos partes de un diagrama en páginas diferentes

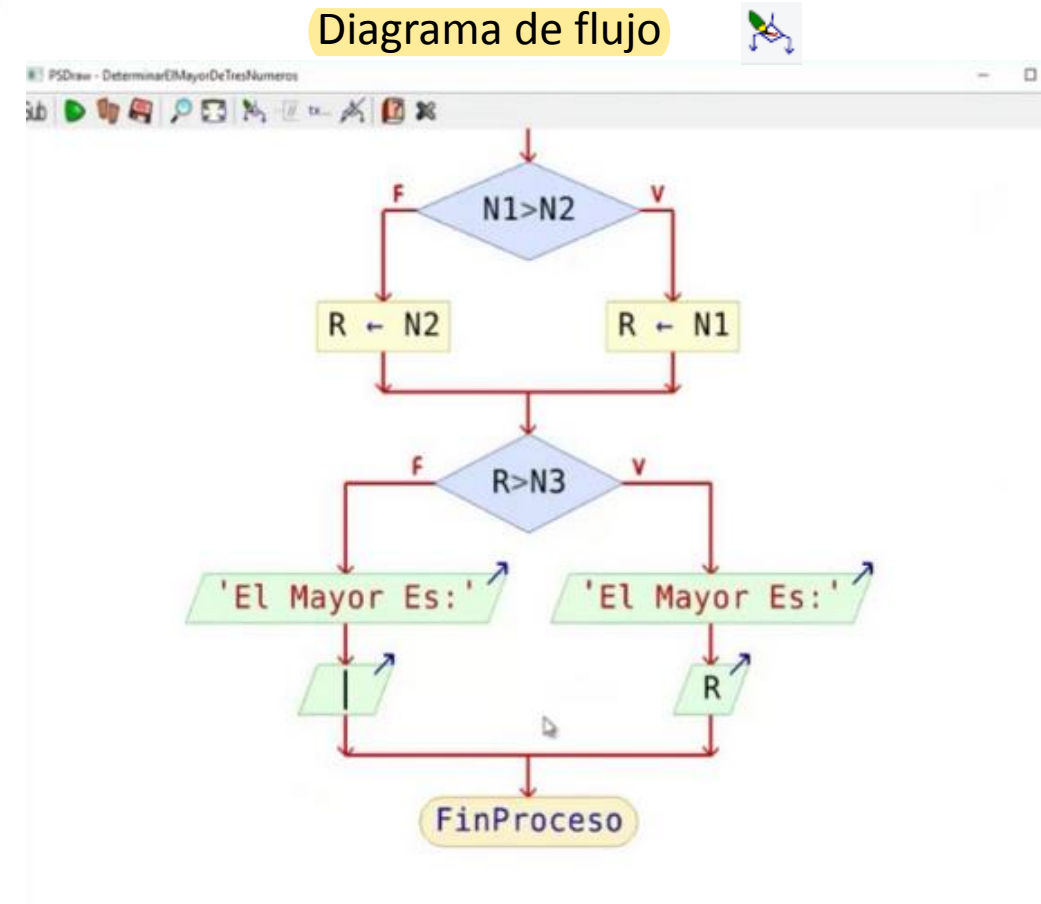
## Pseudocódigo

```
1 Proceso sin_titulo
2
3   Escribir "Ingrese la cantidad: "; Leer "N";
4
5   Dimension a[n];
6   para i<-1 hasta 10 hacer
7     Escribir "Ingrese el dato" i;
8     Leer a[i];
9   FinPara
10
11 FinProceso
```

The screenshot shows the PSeInt application window. The main editor contains the pseudocode above. A tooltip is visible over line 5, stating: "(1) Las dimensiones deben ser constantes." The right sidebar shows a "Comandos" (Commands) palette with icons for various operations like "Escribir" (Write), "Leer" (Read), "Asignar" (Assign), "Si-Entonces" (If-Then), "Segun" (Case), "Mientras" (While), "Repetir" (Repeat), and "Para" (For). The status bar at the bottom right displays "v20120614".



## Diagrama de flujo



---

# Funciones y procedimientos.

---



# Contenidos:

---

- 1)Justificación
- 2)Tipos de datos
- 3)Rutina y tipos de rutinas.

# Justificación

---

- Cuando un programa comienza a ser largo y complejo no es apropiado tener un único texto con sentencias una tras otra. => La mayoría de los problemas son de este tipo.
- El principal motivo de su uso es que no se comprende bien qué hace el programa debido a que se intenta abarcar toda la solución a la vez. => El programa se vuelve difícil de leer, de modificar y de mantener.
- Además suelen aparecer trozos de código muy similares entre sí repetidos a lo largo de todo el programa.

# Justificación

---

## ¿Solución?

- Los lenguajes de alto nivel suelen disponer de una herramienta que permite estructurar el programa principal en pequeños subprogramas o **rutinas**, las cuales, resuelven problemas parciales del problema principal.
- A su vez, cada uno de estos subprogramas puede estar resuelto por otro subconjunto de programas o problemas parciales.
- Los **procedimientos y las funciones** son mecanismos de estructuración que permiten ocultar los detalles de la solución de un problema y resolver una parte de dicho problema en otro lugar del código.

# Rutina

---

PseInt permite definir "subrutinas" (o "funciones") dentro del pseudocódigo, desde la versión del 10 de octubre de 2012.

En su caso, se llaman "subprocesos" o "subalgoritmos".

**SubProceso**   ejemploRutina

**FinSubProceso**

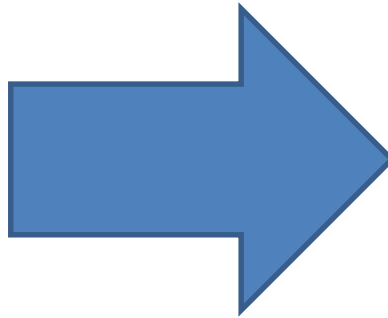
**SubAlgoritmo**   ejemploRutina2

**FinSubAlgoritmo**

# Rutina: **Parámetros**

En la mayoría de lenguajes de programación se puede indicar detalles adicionales ("**parámetros**") para que se puedan utilizar desde dentro de esa subrutina.

```
SubProceso  Asteriscos (cantidad)
  Escribir  "Esto es una rutina";
  Mientras cantidad > 0 Hacer
    Escribir '*';
    cantidad <- cantidad - 1;
  FinMientras
FinSubProceso
```



Un **parámetro**, generalmente es un valor o una variable de un tipo dado que se define en nivel superior o anterior.

Pueden ser utilizados dentro de la rutina, sin verse afectado su valor externamente.

# Rutina: Retorno

---

Podemos crear subprocesos que realicen ciertas operaciones aritméticas y **devuelvan un resultado**. En PseInt se puede hacer con la misma palabra "subproceso" que hemos empleado hasta ahora, pero muchos lenguajes de programación distinguen entre un "procedimiento" o "subrutina".

Cuando se da una serie de pasos pero no se devuelve ningún valor => **procedimiento**  
Y en caso contrario => **función**

---

# Introducción a la POO.

---

# Contenidos:

---

- Introducción.
- Abstracción
- Clases.
- Objetos.



# Introducción

---

**¿Qué es la programación orientada a objetos? ¿Y un lenguaje orientado a objetos?**

- ✓ La programación orientada a objetos es una “filosofía”, un modelo de programación, con su teoría y su metodología.
- ✓ Un lenguaje orientado a objetos es un lenguaje de programación que permite el diseño de aplicaciones orientadas a objetos.

# ¿Qué es la abstracción?

Supresión intencionada (u ocultación) de algunos detalles de un proceso o artefacto, con el fin de destacar más claramente otros aspectos, detalles o estructuras.

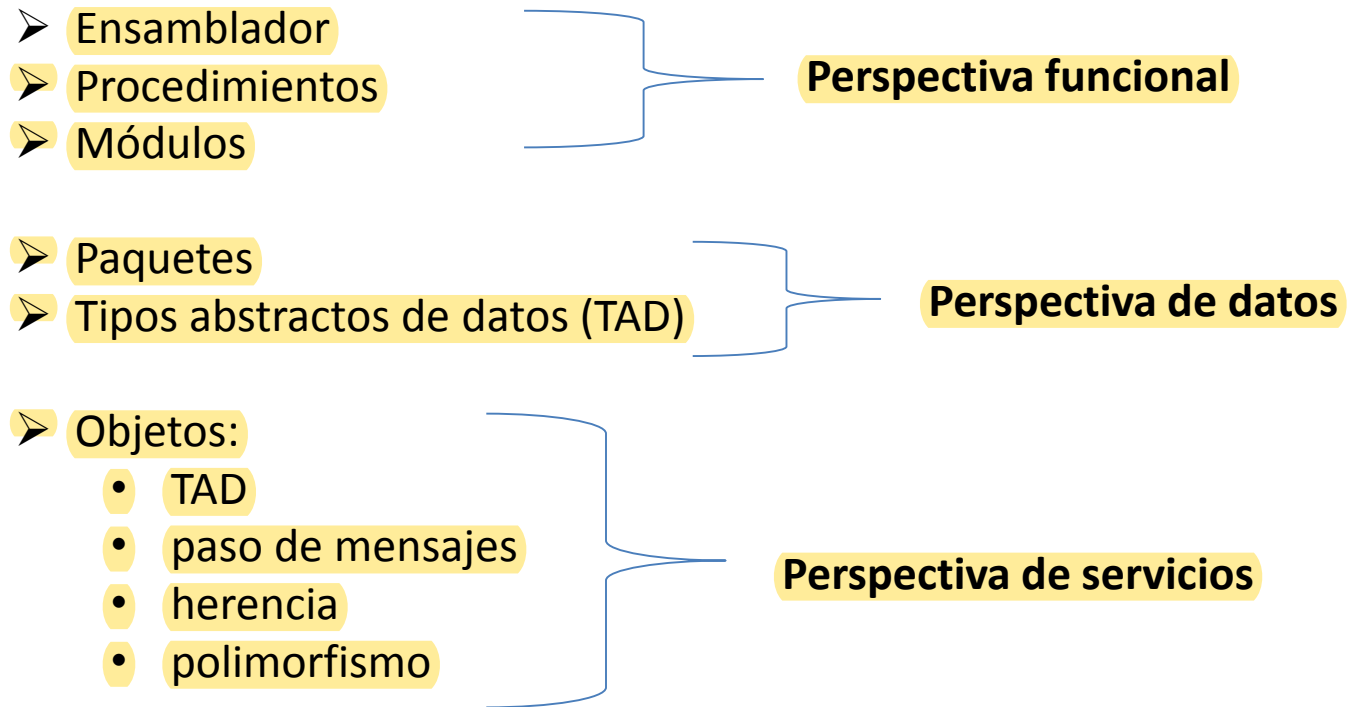
En cada nivel de detalle cierta información se muestra y cierta información se omite => Mapas de Google

Mediante la abstracción creamos **MODELOS** de la realidad.



# Niveles de abstracción.

El nivel de abstracción depende del lenguaje de programación utilizado, ya que cada uno proporciona unos mecanismos propios.



# Encapsulación

Consiste en la omisión intencionada de detalles en un **desarrollo**.

Cuando esta omisión es interna (objeto) y externa => **ENCAPSULACIÓN**.

**Vista externa**

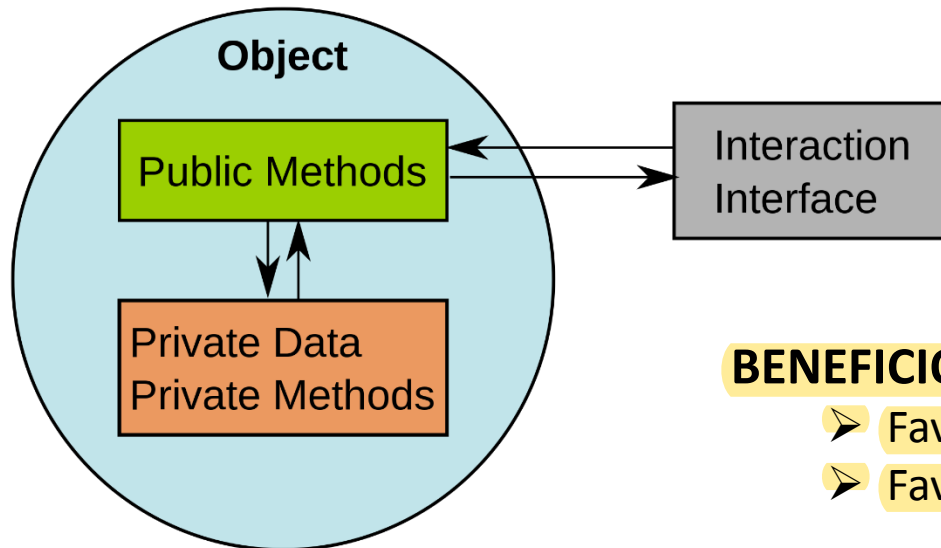
**¿QUÉ** sabe hacer el objeto?

**INTERFAZ**

**Vista interna**

**¿CÓMO** lo hace?

**IMPLEMENTACIÓN**



## **BENEFICIOS:**

- Favorece la intercambiabilidad.
- Favorece la comunicación entre miembros del equipo de desarrollo.

# Programación Orientada a Objetos

Es una metodología de desarrollo de aplicaciones en la cual éstas se organizan como colecciones cooperativas de **objetos**, cada uno de los cuales representan una instancia de alguna **clase**, y cuyas clases son miembros de **jerarquías de clases** unidas mediante relaciones de **herencia**. (Grady Booch).



# Clases

---

Son una descripción de un conjunto de objetos del mundo real que comparten una estructura y un comportamiento común.

Las **clases** constan de datos y métodos (rutinas) que resumen las características comunes de un conjunto de objetos.

Un programa informático está compuesto por un *conjunto de clases*, a partir de las cuales se crean objetos que interactúan entre sí.

Podemos definir una clase como una plantilla o prototipo en el que se definen:

- Los **atributos** comunes a todos los objetos de la clase.
- Los **métodos** definen el comportamiento de la clase, es decir, las acciones que pueden realizar.

# Objetos

---

Una **Clase**, como hemos visto, no es más que una especificación que define las características y el comportamiento de un determinado tipo de objetos. Piensa en ella como si se tratara de una plantilla, molde o esquema a partir del cual podremos construir **objetos** concretos.

Los objetos tienen un **estado** y un **comportamiento**.

Los objetos tienen unas características fundamentales que los distinguen:

- **Identidad.** Es la característica que permite diferenciar un objeto de otro. De esta manera, aunque dos objetos sean exactamente iguales en sus atributos, son distintos entre sí. Puede ser una dirección de memoria, el nombre del objeto o cualquier otro elemento que utilice el lenguaje para distinguirlos. Por ejemplo, dos vehículos que hayan salido de la misma cadena de fabricación y sean iguales aparentemente, son distintos porque tienen un código que los identifica.
- **Estado:** El estado de un objeto viene determinado por parámetros o atributos que lo describen y los valores de éstos. Por ejemplo, si tenemos un objeto Coche, el estado estaría definido por atributos como Marca, Modelo, Color, Cilindrada, etc.
- **Comportamiento:** Son las acciones que se pueden realizar sobre el objeto. Con respecto al ejemplo del objeto Coche, el comportamiento serían acciones como: arrancar(), parar(), acelerar(), frenar(), etc.

---

# Estructuras de control (II).

---



# Contenidos:

---

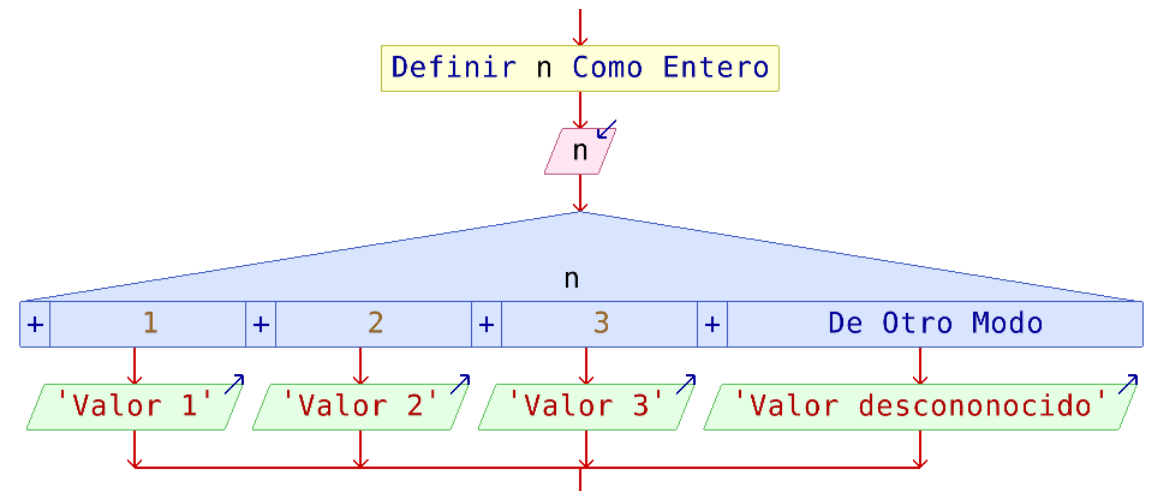
- 1)Según
- 2)Repetir
- 3)Para.

# Según

Es una alternativa a la instrucción si, y se caracteriza por ofrecer la posibilidad de elegir entre más de dos opciones

Al igual que en la instrucción si, el camino a seguir depender del valor que se evalúe

```
Segun variable_numerica Hacer
opcion_1:
    secuencia_de_acciones_1
opcion_2:
    secuencia_de_acciones_2
opcion_3:
    secuencia_de_acciones_3
De Otro Modo:
    secuencia_de_acciones_dom
FinSegun
```



# Repetir

Cumple la misma función que la estructura mientras.

La principal diferencia está en que la estructura mientras comprueba la condición al inicio y repetir lo hace al final.

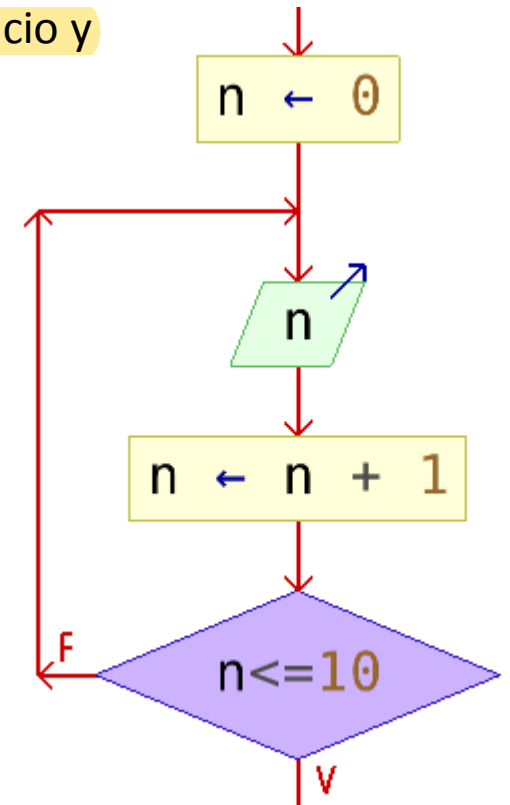
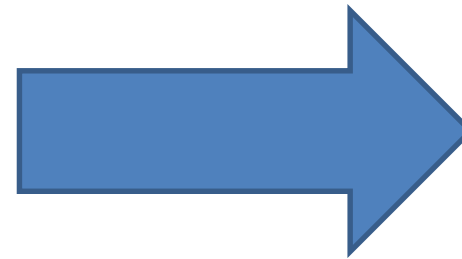
Por lo tanto, la estructura repetir se ejecuta por lo menos una vez.

**Repetir**

...

secuencia de acciones

**Hasta Que** expresion logica



# Para

- 1) Se evalúa la expresión de inicialización
- 2) Se evalúa el valor final. Si es falso, salimos del Para
- 3) Se ejecuta la secuencia de acciones
- 4) Se ejecuta Paso y se vuelve al 2)

```
Para variable_numerica <- valor_inicial Hasta valor_final Con Paso paso Hacer  
..... secuencia_de_acciones  
FinPara
```

