

# Práctica 6

## Ejercicio 1

- **Parámetro:** Forma de comunicación entre rutinas. Proporciona legibilidad, seguridad, flexibilidad, etc.
- **Parámetro real:** Parámetro enviado desde el llamante. Puede ser un valor, entidad, expresión, etc., que pueden ser locales, no locales o globales.
- **Parámetro formal:** Los parámetros que aparecen en la definición de la rutina. Se considera una variable local a su entorno. Parámetro que se recibe, son como variables locales.
- **Ligadura posicional:** Le ligan *uno a uno*, es decir, en el orden posicional en el que fueron declarados y luego invocados.
- **Ligadura por nombre:** Se ligan por el nombre, o sea, cuando se invoca la rutina, se debe especificar a qué parámetro corresponde el parámetro real que le estamos dando.

## Ejercicio 2

- Modo IN:
  - Por valor
  - Por valor constante
- Modo OUT:
  - Por resultado
  - Por resultado funciones
- Modo IN/OUT:
  - Por valor/resultado
  - Por referencia
  - Por nombre

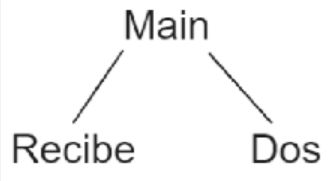
## Ejercicio 3

Tipo de pasaje de parámetros	Lenguaje
Copia <b>IN</b> , Resultado <b>OUT</b> , <b>IN-OUT</b> : Valor resultado para los tipos primitivos, por	ADA

Referencia para los no primitivos	
Por valor (si se necesita por referencia se usan punteros) y permite el pasaje por valor constante (agregándole el const)	C
Por valor, pero al igual que Python si se pasa es un objeto mutable. No se pasa una copia se trabaja sobre él	Ruby
El único mecanismo es el paso por <b>copia de valor</b> . Pero como las variables de tipos NO primitivos son todas referencias a variables anónimas en el Heap. El paso por valor de una de estas variables constituye en realidad un paso por <b>referencia</b> de la variable	Java
Envía objetos que pueden ser <b>"inmutables"</b> o <b>"mutables"</b> . Si es inmutable actuará como por valor, y si es <b>mutable</b> , ejemplo: lista, no se hace una copia sino que se trabaja sobre él	Python

b. ADA es más seguro ya que el pasaje de parámetros en funciones se realiza a través de IN por valor, lo que evita que desde la función pueda modificarse. En pascal no está esta restricción, lo que lo hace menos seguro. Las funciones deberían únicamente retornar un resultado y no alterar nada.

#### Ejercicio 4

Árbol de anidamiento sintáctico	RA Main	RA Recibe	RA Dos
 <pre> graph TD     Main --&gt; Recibe     Main --&gt; Dos </pre>	Registro de Activación Main Pto. Retorno EE ED j m i Procedure Recibe Procedure Dos Valor retorno...	Registro de activación Recibe Pto. Retorno EE ED Valor retorno...	Registro de activación Dos Pro Retorno EE ED m Valor retorno..

i) Modo IN/OUT - Referencia

Código	Cadena Estática	Cadena Dinámica
<pre> <b>Procedure Main;</b>   var j, m, i: integer;    <b>Procedure Recibe</b> (x:integer; y:integer);   begin     m:= m+1+y;     x:=i + x + j;     y:=m- 1;     write (x, y, i, j, m);   end;    <b>Procedure Dos;</b>   var m:integer;   begin     m:= 5;     Recibe(i, j);     write (i, j, m);   end;    begin     m:= 2;     i:=1; j:=3;     Dos;     write (i, j, m);   end. </pre>	<p><b>1* Registro de activación Main</b>  Pto. Retorno  EE()  ED()  j -&gt; 3 -&gt; 5  m -&gt; 2 -&gt; 6  i -&gt; 1 -&gt; 5  Procedure Recibe  Procedure Dos  Valor retorno...  <b>Imprime: 5, 5, 6</b></p> <p><b>2* Registro de activación Dos</b>  Pto. Retorno  EE(Main)  ED(Main)  m -&gt; 5  Valor retorno...  <b>Imprime: 5, 5, 5</b></p> <p><b>3* Registro de activación Recibe</b>  Pto. retorno  EE(Main)  ED(Dos)  x (flecha apunta a i)  y (flecha apunta a j)  Valor retorno..  <b>Imprime: 5, 5, 5, 5, 6</b></p>	<p><b>1* Registro de activación Main</b>  Pto. Retorno  EE  ED  j -&gt; 3 -&gt; 8  m -&gt; 2  i -&gt; 1 -&gt; 5  Procedure Recibe  Procedure Dos  VR...  <b>Imprime: 5,8,2</b></p> <p><b>2* Registro de activación Dos</b>  Pto. Retorno  EE 1  ED 1  m -&gt; 5 -&gt; 9  VR...  <b>Imprime: 5,8,9</b></p> <p><b>3* Registro de activación Recibe</b>  Pto. Retorno  EE 1  ED 2  x (flecha apunta a i)  y (flecha apunta a j)  VR...  <b>Imprime: 5,8,5,8,9</b></p>

ii) Modo IN - Valor

Código	Cadena Estática	Cadena Dinámica
<pre> <b>Procedure Main;</b>   var j, m, i: integer;    <b>Procedure Recibe</b> (x:integer; y:integer);   begin     m:= m+1+y;     x:=i + x + j;     y:=m- 1;     write (x, y, i, j, m);   end;    <b>Procedure Dos;</b>   var m:integer;   begin     m:= 5;     Recibe(i, j);     write (i, j, m);   end;    begin     m:= 2;     i:=1; j:=3;     Dos;     write (i, j, m);   end. </pre>	<p><b>1* Registro de Activación</b></p> <p>Main  Pto. Retorno  EE()  ED()  j = 3  m = 2 -&gt; 6  i = 1  Procedure Recibe  Procedure Dos  Valor retorno...  <b>Imprime: 1,3,6</b></p> <p><b>2* Registro de Activación Dos</b></p> <p>Pto. Retorno  EE 1  ED 1  m = 5  Valor retorno...  <b>Imprime: 1,3,6</b></p> <p><b>3* Registro de Activación</b></p> <p>Recibe  Pto. Retorno  EE 1  ED 2  x = 1 -&gt; 5  y = 3 -&gt; 7  Valor retorno...  <b>Imprime: 5,7,1,3,6</b></p>	<p><b>1* Registro de activacion Main</b></p> <p>Pto. retorno  EE()  ED()  j -&gt; 3  m -&gt; 2  i -&gt; 1  Procedure Recibe  Procedure Dos  Valor retorno...  <b>Imprime: 1, 3, 2</b></p> <p><b>2* Registro de activacion Dos</b></p> <p>Pto. Retorno  EE(Main)  ED(Main)  m -&gt; 5 -&gt; 9  Valor retorno...  <b>Imprime: 1, 3, 9</b></p> <p><b><u>3* Registro de activacion Recibe</u></b></p> <p>Pto. Retorno  EE(Main)  ED(Dos)  x = 1 -&gt; 5  y = 3 -&gt; 8  Valor retorno..  <b>Imprime: 5, 8, 1, 3, 9</b></p>

iii) Modo IN/OUT - Valor Resultado

Código	Cadena Estática	Cadena Dinámica
<pre> <b>Procedure</b> Main;   var j, m, i: integer;    <b>Procedure</b> Recibe (x:integer; y:integer);   begin     m:= m+1+y;     x:=i + x + j;     y:=m- 1;     write (x, y, i, j, m);   end;    <b>Procedure</b> Dos;   var m:integer;   begin     m:= 5;     Recibe(i, j);     write (i, j, m);   end;  begin   m:= 2;   i:=1; j:=3;   Dos;   write (i, j, m); end. </pre>	<p><b>1* Registro de activacion Main</b>  Pto. Retorno  EE()  ED()  j -&gt; 3 -&gt; 5  m -&gt; 2 -&gt; 6  i -&gt; 1 -&gt; 5  Procedure Recibe  Procedure Dos  Valor retorno..  <b>Imprime: 5, 5, 6</b></p> <p><b>2* Registro de activacion Dos</b>  Pto. Retorno  EE(Main)  ED(Main)  m -&gt; 5  Valor retorno..  <b>Imprime: 5, 5, 5</b></p> <p><b>3* Registro de activacion Recibe</b>  Pto. Retorno  EE(Main)  ED(Dos)  x -&gt; 1 -&gt; 5 (referencia a i)  y -&gt; 3 -&gt; 5 (referencia a j)  Valor retorno..  <b>Imprime: 5, 5, 1, 3, 6</b></p>	<p><b>1* Registro de Activación</b>  Main  Pto. Retorno  EE()  ED()  j = 3 -&gt; 8  m = 2  i = 1 -&gt; 5  Procedure Recibe  Procedure Dos  Valor retorno..  <b>Imprime: 5,8,2</b></p> <p><b>2* Registro de activación Dos</b>  Pro Retorno  EE (Main)  ED (Main)  m = 5 -&gt; 9  Valor retorno..  <b>Imprime: 5,8,9</b></p> <p><b>3* Registro de activación Recibe</b>  Pto. Retorno  EE (Main)  ED (Dos)  x = 1 -&gt; 5 (Se referencia a i)  y = 3 -&gt; 8 (se referencia a j)  Valor retorno..  <b>Imprime: 5,8,1,3,9</b></p>

Código	Cadena Estática	Cadena Dinámica
<pre> <b>Procedure</b> <b>Main</b>;   var j, m, i: integer;    <b>Procedure</b> <b>Recibe</b> (x:integer; y:integer);   begin     m:= m+1+y;     x:=i + x + j;     y:=m- 1;     write (x, y, i, j, m);   end;  <b>Procedure</b> <b>Dos</b>;   var m:integer;   begin     m:= 5;     Recibe(i, j);     write (i, j, m);   end;  begin   m:= 2;   i:=1; j:=3;   Dos;   write (i, j, m); end. </pre>	<p><b>1* Registro de activación Main</b>  Pto. Retorno  EE()  ED()  j -&gt; 3 -&gt; 5  m -&gt; 2 -&gt; 6  i -&gt; 1 -&gt; 5  Procedure Recibe  Procedure Dos  Valor retorno...  <b>Imprime: 5, 5, 6</b></p> <p><b>2* Registro de activación Dos</b>  Pto. Retorno  EE(Main)  ED(Main)  m -&gt; 5  Valor retorno...  <b>Imprime: 5, 5, 5</b></p> <p><b>3* Registro de activación Recibe</b>  Pto. retorno  EE(Main)  ED(Dos)  x ↑ i  y ↑ j  Valor retorno..  <b>Imprime: 5, 5, 5, 5, 6</b></p>	<p><b>1* Registro de activación Main</b>  Pto. Retorno  EE  ED  j -&gt; 3 -&gt; 8  m -&gt; 2  i -&gt; 1 -&gt; 5  Procedure Recibe  Procedure Dos  VR...  <b>Imprime: 5,8,2</b></p> <p><b>2* Registro de activación Dos</b>  Pto. Retorno  EE 1  ED 1  m -&gt; 5 -&gt; 9  VR...  <b>Imprime: 5,8,9</b></p> <p><b>3* Registro de activación Recibe</b>  Pto. Retorno  EE 1  ED 2  x ↑ i  y ↑ j  VR...  <b>Imprime: 5,8,5,8,9</b></p>

v) Modo OUT - Resultado

No es posible realizarlo. Se genera un error al intentar modificar los parámetros dentro de los procedimientos, ya que no están inicializados y el pasaje es OUT. Es decir, no reciben nada (tienen basura) y no son inicializados.

- c. Sí, en el caso de Modo OUT por Resultado.
- d. Respondida en b)

Ejercicio 5

- a. (4,6,7), (4,6,7), 2, 2 - Tipo de parámetro: IN/OUT Referencia en los 3 parámetros.
- b. (3,5,6),(4,6,7), 2, 2 - Tipo de parámetro: IN por valor vector, IN/OUT referencia para iteración y vit.
- c. (3,5,6),(5,5,6), 0, -1 - Tipo de parámetro: Por IN Valor los 3.

Ejercicio 6

**Parámetro por nombre:**

- Un valor entero: Se comporta como IN/OUT por referencia.
- Una constante: Se comporta como un IN por Valor.
- Un elemento de un arreglo: Puede cambiar el suscripto entre las distintas referencias. Se evalúa cada vez.
- Una expresión: Se evalúa cada vez.

Ejercicio 7

Código	Cadena Estática	Cadena Dinámica
--------	-----------------	-----------------

```

Procedure Uno;
  y, z: integer;
  r1:array[1..6] of integer;
  r2:array[1..5] of integer;

  Procedure Dos( nombre x, t:integer;
var io:integer; valor-resultado
y:integer);

    Procedure Dos( nombre
t1:integer);

        Procedure Tres;
        begin
            y:= y + 1;
            z:= z + 1;
        end;

    begin
        t1:= t1 + 1;
        t:= t + 1;
        Tres;
        t1:= t1 + 2;
        t:= t + 2;
    end;

begin
  x:= x + 1;

```

#### **\*1 Registro de Activación Main**

Pto Retorno  
 EE()  
 ED()  
 y = 1..6 -> 1..5 -> 1 -> 2 -> 1..6 -> 1..6  
 z = 2 -> 3 -> 3  
 r1[1] = 2  
 r1[2] = 2 -> 3  
 r1[3] = 2  
 r1[4] = 2 -> 4  
 r1[5] = 2  
 r1[6] = 2

r2[1] = 1  
 r2[2] = 1 -> 2 -> 3 -> 4  
 r2[3] = 1 -> 3 -> 5  
 r2[4] = 1  
 r2[5] = 1  
 Procedure Dos  
 Valor retorno..

#### **\*2 Registro de Activacion Dos**

Pto. Retorno  
 EE(\*1)  
 ED(\*1)  
 x -> (nombre) ↑ r1[y + r2[y]]  
 t -> (nombre) ↑ r2(z)  
 io -> (flecha a y)  
 y -> 2 -> 3 (valor resultado z)  
 Procedure Dos  
 Valor Retorno..

#### **\*3 Registro de Activacion Dos**

Pto retorno  
 EE(\*2)  
 ED(\*2)  
 t1 -> (nombre) ↑ t  
 Procedure Tres  
 Valor retorno..

#### **\*4 Registro de Activacion Tres**

Pto. Retorno

#### **\*1 Registro de Activacion Main**

Pto. Retorno  
 EE()  
 ED()  
 y -> 1..6 -> 1..5 -> 1 -> 2 -> 1..6 -> 1..5  
 z -> 2 -> 3 -> 3  
 r1[1] = 2  
 r1[2] = 2 -> 3  
 r1[3] = 2  
 r1[4] = 2 -> 4  
 r1[5] = 2  
 r1[6] = 2

r2[1] = 1  
 r2[2] = 1 -> 2 -> 3 -> 4  
 r2[3] = 1 -> 3 -> 5  
 r2[4] = 1  
 r2[5] = 1  
 Procedure Dos  
 Valor retorno..

#### **\*2 Registro de Activación Dos**

Pto. Retorno  
 EE(\*1)  
 ED(\*1)  
 x -> ↑ r1(y+r2(y))--r1(y+r2(1))--r1(y+1)--r1(2)  
           r1(y+r2(y))--r1(y+r2(2))--r1(y+2)--r1(4)  
 t -> ↑ r2(z)--r2(2)  
 io -> (flecha a y \*1)  
 y -> 2 -> 3 (valor-resultado z)  
 Procedure dos  
 Valor retorno..

#### **\*3 Registro de Activación Dos**

EE(\*2)  
 ED(\*2)  
 t1 -> ↑ t ↑ r2(z)--r2(2)  
           r2(z)--r2(3)  
 Procedure Tres  
 Valor retorno..

#### **\*4 Registro de Activacion Tres**



<pre> t:= t + 1; io:= io + 1; x:= x + 2; if z =2 then Dos ( t ); end;  begin for y:= 1 to 6 do r1(y):= 2; for y:= 1 to 5 do r2(y):= 1; z:= 2; y:= 1; Dos( r1( y + r2( y )), r2( z ), y, z); for y:= 1 to 6 do write (r1(y)); for y:= 1 to 5 do write (r2(y)); end. </pre>	EE(*3) ED(*3) Valor retorno..  <b>Imprime:</b> 2, 3, 2, 4, 2, 2 1, 4, 5, 1, 1	EE(*3) ED(*3) Valor retorno..  <b>Imprime:</b> 2, 3, 2, 4, 2, 2 1, 4, 5, 1, 1
---	---	---

### Ejercicio 8

- a. **Ligadura Shallow o Superficial:** El ambiente de referencia donde voy a buscar esa variable que no encuentro es el del subprograma que tiene el parámetro formal. “Dinámica”
- Ligadura Deep o Profunda:** El ambiente de referencia donde voy a buscar es donde esté declarado el subprograma utilizado como parámetro real. “Estática”

b.

Ligadura Shallow	
Código	Cadena Estática y Dinámica

Program A

Var x:integer;

Var y: char;

Procedure B;

Var h:integer;

Begin

h:=1+x;

Write (y);

C(D);

Write (y);

End;

Procedure C (Subrutina S);

Var x:integer;

Var y: char;

Begin

x:=3;

y:= "b";

x:=S(x,y)

y:= "j";

Write (x,y);

End;

Function D (j:integer, k:char);

Begin

j:=j+x;

k:=y;

Write (k);

Return j;

End;

BEGIN

x:=0;

y:="a";

B();

Write (x,y);

END.

### **\*1 Registro de activación A**

Pto. Retorno

EE()

ED()

x -> 0

y -> a

Procedure B

Procedure C

Procedure D

Valor retorno..

### **\*2 Registro de Activación B**

Pto. Retorno

EE(\*1)

ED(\*1)

h -> 1

Valor retorno..

### **\*3 Registro de Activacion C**

Pto. Retorno

EE(\*1)

ED(\*2)

S -> Procedure D

x -> 3 -> 6

y -> b -> j

Valor retorno..6

### **\*4 Registro de Activacion D**

Pto. Retorno

EE(\*1)

ED(\*3)

j -> 3 -> 6

k -> b -> b

Valor retorno..

**Imprime:**

**a**

**b**

**6, j**

**a**

**0, a**

## Ligadura Deep

Código	Cadena Estática y Dinámica
<pre>Program A   Var x:integer;   Var y: char;    Procedure B;</pre>	<p><b>*1 Registro de activación A</b> Pto. Retorno EE() ED() x -&gt; 0 y -&gt; a Procedure B Procedure C Procedure D Valor retorno..</p> <p><b>*2 Registro de Activacion B</b> Pto. Retorno EE(*1) ED(*1) h -&gt; 1 Valor retorno..</p> <p><b>*3 Registro de Activacion C</b> Pto. Retorno EE(*1) ED(*2) S -&gt; Procedure D x -&gt; 3 -&gt; 3 y -&gt; b -&gt; j Valor retorno..3</p> <p><b><u>*4 Registro de Activacion D</u></b> Pto. Retorno EE(*1) ED(*3) j -&gt; 3 -&gt; 3 k -&gt; b -&gt; a Valor retorno..</p> <p><b>Imprime:</b> <b>a</b> <b>a</b></p>

<pre> Var h:integer; Begin   h:=1+x;   Write (y);   C(D);   Write (y); End;  Procedure C (Subrutina S); Var x:integer; Var y: char; Begin   x:=3;   y:= "b";   x:=S(x,y)   y:= "j";   Write (x,y); End;  Function D (j:integer, k:char); Begin   j:=j+x;   k:=y;   Write (k);   Return j; End;  BEGIN   x:=0;   y:="a";   B();   Write (x,y); END. </pre>	<pre> 3, j a 0, a </pre>
---	--------------------------

## Ejercicio 9

a)

**Nombre:** El parámetro formal es sustituido textualmente por una expresión del parámetro real más un puntero al entorno del parámetro real. (se maneja una estructura aparte que resuelve esto).

Se establece la ligadura entre parámetro formal y parámetro real en el momento de la invocación, pero la "ligadura de valor" se difiere hasta el momento en que se lo utiliza (la dirección se resuelve en ejecución). Distinto a por referencia. Es decir, no apunta a una dirección fija, puede ir cambiando (pero el nombre tiene que ser el mismo).

```
a(1) -> 1 -> 3
a(2) -> 1 -> 2
a(3) -> 1 -> 0
a(4) -> 1
a(5) -> 1
x -> 3 -> 0 -> 1 -> 2
i -> 1..5
Uno (m=a(x))
Imprime: 3, 2, 0, 1, 1
```

**Referencia:** El parámetro formal será una variable local que contiene la dirección al parámetro real de la unidad llamadora que estará entonces en un ambiente no local. Cualquier cambio que se realice en el parámetro formal dentro del cuerpo del subprograma quedará registrado en el parámetro real.

```
a(1) -> 1
a(2) -> 1
a(3) -> 1 -> 3 -> 2 -> 3
a(4) -> 1
a(5) -> 1
x -> 3 -> 0 -> 1 -> 2
i -> 1..5
Uno (m=a(3))
Imprime: 1, 1, 3, 1, 1
```

**Valor/Resultado:** El parámetro formal es una variable local que recibe una copia (a la entrada) del contenido del parámetro real y el parámetro real (a la salida) recibe una copia de lo que tiene el parámetro formal. Básicamente lo que hace la rutina lo copia en la variable. Cada referencia al parámetro formal es una referencia local.

```
a(1) -> 1
a(2) -> 1
```

a(3) -> 1 -> 0 -> 4  
a(4) -> 1  
a(5) -> 1  
x -> 3 -> 0 -> 1 -> 2  
i -> 1..5  
Uno (m=a(3))  
m -> a(3) -> 1 -> 3 -> 4 ↑  
Imprime: 1, 1, 4, 1, 1

b) Mismas impresiones:

**Nombre:**

a(1) -> 1 -> 3  
a(2) -> 1 -> 2  
a(3) -> 1 -> 0  
a(4) -> 1  
a(5) -> 1  
x -> 3  
i -> 1..5  
Uno (m=a(x))  
x -> 0 -> 1 -> 2  
Imprime: 3, 2, 0, 1, 1

**Referencia:**

a(1) -> 1  
a(2) -> 1  
a(3) -> 1 -> 3 -> 2 -> 3  
a(4) -> 1  
a(5) -> 1  
x -> 3  
i -> 1..5  
Uno (m=a(3))  
x -> 0 -> 1 -> 2  
Imprime: 1, 1, 3, 1, 1

<pre> Procedure main   a: array(1..5) of integer;   x: integer;   i;integer;   Procedure Uno (tipo_pasaje   m:integer)     Begin       x:=0;       x:=x+1;       m:=m+x + a(3);       x:=x*2;       a(3):=a(3) - 1;       m:=m+1;     End; </pre>	<pre> Begin   For i:=1 to 5 a(i):=1;   x:=3;   Uno(a(x));   For i:=1 to 5 write (a(i)); End. </pre>
---	---

#### Valor/Resultado:

a(1) -> 1  
 a(2) -> 1  
 a(3) -> 1 -> 0 -> 4  
 a(4) -> 1  
 a(5) -> 1  
 x -> 3  
 i -> 1..5  
 Uno (m=a(3))  
 x -> 0 -> 1 -> 2  
 m -> a(3) -> 1 -> 3 -> 4 ↑  
Imprime: 1, 1, 4, 1, 1

## Práctica 7

### Ejercicio 1

**1. Sistema de Tipos:** Conjunto de reglas usadas por un lenguaje para **estructurar y organizar** sus tipos. **El objetivo de un sistema de tipos es prevenir errores y escribir programas más seguros.**

- Conocer el sistema de tipos de un lenguaje nos permite conocer de una forma los aspectos semánticos del lenguaje.

Provee mecanismos de expresión:

- Expresar tipos predefinidos, definir nuevos tipos y asociarlos con constructores del lenguaje.

#### Define reglas de resolución:

- Equivalencia de tipos, saber si dos valores tienen el mismo tipo.
- Compatibilidad de tipos, saber si se puede usar un tipo en un contexto determinado.
- Interferencia de tipos, saber el tipo a partir del contexto.

Mientras más flexible sea el lenguaje, más complejo el sistema de tipos. Seguridad vs Flexibilidad.

## 2. Tipado Fuerte y Débil:

### **Lenguaje Fuertemente Tipado:**

- El compilador puede garantizar la ausencia de errores de tipo en los programas.
- El sistema de tipos impone restricciones que aseguran que no se producirán errores en ejecución.
- Un lenguaje se dice fuertemente tipado si todos los errores de tipo se detectan. Más seguro.

### **Sistema Débil:**

- Menos seguro pero ofrece una mayor flexibilidad.

Ejemplos:

### **Python - Fuertemente tipado**

- No se permiten operaciones entre tipos incompatibles sin una conversión explícita. Por ejemplo, intentar sumar un número con una cadena (`3 + "4"`) lanza un error en tiempo de ejecución.

### **C - Débilmente tipado**

- En C, el compilador permite conversiones implícitas entre tipos, incluso si pueden causar errores en tiempo de ejecución. Por ejemplo, se puede asignar una dirección de memoria (`int*`) a una variable `int` usando cast sin ninguna verificación. También es común que se mezclen `char`, `int` y `float` en operaciones sin advertencias. Esto muestra que el control de tipos es laxo.

## 3. Tipado Estático y Dinámico:

**Tipado Estático:** Ligadura en compilación. Para esto puede exigir:

- Se pueden utilizar tipos de datos predefinidos.
- Todas las variables se declaran con un tipo asociado.



- Todas las operaciones se especifican indicando los tipos de los operandos requeridos y el tipo del resultado.

**Tipado Dinámico:** Ligaduras en ejecución, provoca más comprobaciones en tiempo de ejecución. **Que las ligaduras se den en ejecución no vuelve a este tipado un tipado inseguro.**

Ejemplos:

#### **Python - Tipado dinámico**

- Las variables no tienen un tipo fijo. El tipo se asocia al valor, no al nombre de la variable, y se determina en tiempo de ejecución.

#### **C - Tipado Estático**

- El tipo de cada variable debe declararse explícitamente y no puede cambiar en tiempo de ejecución.

## Ejercicio 2

**1. Tipo de Dato:** Un tipo de datos es un conjunto de valores y un conjunto de operaciones asociadas al tipo para manejar esos valores.

**2. Los tipos de datos predefinidos** (primitivos) elementales los brinda el lenguaje. Reflejan el comportamiento subyacente del hardware, son abstracciones de él. Tiene varias ventajas, como la **invisibilidad** de la representación, verificación estática, operadores no ambiguos, o el control de precisión. Ejemplos:

- Número
  - Enteros
  - Reales
- Caracteres
- Booleano

Que un conjunto de valores de un tipo sea definido por la implementación del lenguaje significa que será seleccionado por el compilador, mientras que si el tipo es definido por el lenguaje será definido en su definición.

**3. Los lenguajes de programación permiten al programador especificar agrupaciones de objetos de datos elementales y de forma **recursiva**, agregaciones de agregados. Se logra mediante **constructores**. A estas agrupaciones se las conoce como tipos de datos definidos por el usuario.**

- Enumerados.

- Arreglos.
- Registros.
- Listas.

Separan la **especificación de la implementación** y se definen los tipos que el problema necesita. También permite:

- Instanciar objetos de las agregaciones.
- Definir nuevos tipo de dichas agregaciones.
- Chequeos de consistencia.

Tienen ventajas como la *legibilidad* (elección apropiada de nombres), *factorización* (uso cuantas veces sea necesario) y *modificabilidad* (sólo se cambia en la definición).

## Ejercicio 3

### 1. Constructores:

- **Producto Cartesiano:** El Producto cartesiano de n conjuntos de tipos variados. Se conforma por las distintas n-tuplas que se generan de combinaciones de elementos de los diferentes n tipos de datos. Permite producir registros (*Pascal*) o struct (C).
  - Combina varios tipos de datos en una sola estructura.
- **Correspondencia Finita:** Función de un conjunto finito de valores de un tipo de **dominio DT** (tipo del dominio, es decir, un tipo de variable) en valores de un tipo del **dominio RT** (resultado del dominio, acceso a través de un índice). Se accede a través de un **índice**. Listas, vectores, matrices, etc.
  - Mapea un dominio finito de índices a valores. Acceso por posición (índice).
- **Unión y Unión Discriminada:** La unión/unión discriminada de dos o más tipos define un tipo como la disyunción de los tipos dados. Permite manipular diferentes tipos en distintos momentos de la ejecución. Hay un chequeo dinámico. La declaración es muy similar a la del producto cartesiano. La diferencia es que sus campos son mutuamente excluyentes.
  - Unión: Una misma región de memoria puede contener distintos tipos, pero solo uno a la vez.
  - Unión discriminada: Unión con discriminador que indica el tipo activo. Control de tipo en tiempo de ejecución.
  - La Unión Discriminada agrega un descriptor (enumerativo) que me permite saber con quien estoy trabajando y acceder correctamente a lo que tengo que acceder, ya que nos dice cual de los campos posee valor. Básicamente

manipulo el elemento según el valor del discriminante, es una mejora de la unión que brinda una mayor seguridad. Este discriminante igualmente puede omitirse.

- **Recursión:** Un tipo de dato recursivo T se define como una estructura que puede contener componentes de tipo T. Define datos agrupados; cuyo tamaño puede crecer arbitrariamente y cuya estructura puede ser arbitrariamente compleja. Los lenguajes suelen implementar tipos de datos recursivos a través de punteros.
  - Ejemplos: árboles o listas de Pascal.

2.

<b>Java</b> <pre>class Persona {     String nombre;     String apellido;     int edad; }</pre> <p>Producto cartesiano</p>	<b>C</b> <pre>typedef struct _nodoLista {     void *dato;     struct _nodoLista *siguiente } nodoLista;  typedef struct _lista {     int cantidad;     nodoLista *primero } Lista;</pre> <p>Producto cartesiano y Recursión</p>	<b>C</b> <pre>union codigo {     int numero;     char id; };</pre> <p>Unión</p>
<b>Ruby</b> <pre>hash = {   uno: 1,   dos: 2,   tres: 3,   cuatro: 4 }</pre> <p>Correspondencia finita Producto cartesiano?</p>	<b>PHP</b> <pre>function doble(\$x) {     return 2 * \$x; }</pre> <p>Correspondencia finita? Recursión?</p>	<b>Python</b> <pre>tuple = ('physics', 'chemistry', 1997, 2000)</pre> <p>Correspondencia finita</p>

<p><b>Haskell</b></p> <pre>data ArbolBinarioInt =   Nil     Nodo int     (ArbolBinarioInt dato)     (ArbolBinarioInt dato)</pre> <p><b>Ayuda para interpretar:</b>        'ArbolBinarioInt' es un tipo de dato que puede ser Nil ("vacío") o un Nodo con un dato número entero (int) junto a un árbol como hijo izquierdo y otro árbol como hijo derecho</p>	<p><b>Haskell</b></p> <pre>data Color =   Rojo     Verde     Azul</pre> <p><b>Ayuda para interpretar:</b>        'Color' es un tipo de dato que puede ser Rojo, Verde o Azul.</p> <p style="text-align: center;">Unión</p>
--	--

Recursión

## Ejercicio 4

**1. Mutabilidad e Inmutabilidad:** Son términos que se refieren a la capacidad o no de un dato de ser modificado después de su creación. Un **dato mutable** es aquel que se puede cambiar después de su creación, mientras que uno **inmutable** es aquel que no se puede cambiar después de haber sido creado.

- En **Python**, algunos datos inmutables son los enteros, las cadenas y las tuplas. Mientras que los datos mutables incluyen listas, conjuntos y diccionarios, ya que se pueden modificar después de haber sido creados.
- En **Ruby**, la mayoría de los objetos son mutables por defecto, pero se puede hacer que un objeto sea inmutable utilizando el método *freeze*. Después de que un objeto es congelado, no se puede modificar.

## 2.

```
a = Dato.new(1)
```

```
a = Dato.new(2)
```

No se puede brindar una respuesta completamente certera con el fragmento de código proporcionado, el hecho de que se modifique el valor de "a" a partir de la creación de un

nuevo objeto podría dar indicios de que el objeto "Dato.new(1)" es inmutable pero no se puede hablar con certeza ya que no se tiene acceso a la implementación de *Dato*.

La mutabilidad se refiere a si el estado interno de un objeto puede cambiar sin crear uno nuevo. Para saber si *Dato.new(1)* es mutable, habría que ver si ese objeto permite modificar su contenido una vez creado (por ejemplo, con setters u operaciones internas que alteren su estado).

## Ejercicio 5

1. Si, en C se puede tomar el l-valor de una variable (manipulando direcciones y controlando la memoria) utilizando el operador "&".

```
#include <stdio.h>

int main() {
    int x = 10;
    int *p = &x; // Tomamos el l-valor de 'x' con &x

    printf("Valor de x: %d\n", x);          // 10
    printf("Dirección de x: %p\n", &x);    // Dirección (l-valor)
    printf("Contenido de p: %p\n", p);     // Misma dirección
    printf("Valor apuntado por p: %d\n", *p); // 10

    *p = 20; // Modificamos x a través de su dirección
    printf("Nuevo valor de x: %d\n", x);   // 20

    return 0;
}
```

2. Al trabajar con punteros, pueden haber casos donde suceden distintos tipos de inseguridades:

- **Violación de tipos:** Se puede operar con el valor al que apunta la variable sin chequear el tipo.
- **Referencias sueltas/dangling:**
  - Es un puntero que contiene una dirección de una variable dinámica que fue desalocada, si luego se usa el puntero producirá error.
- **Punteros no inicializados:**

- Peligro de acceso descontrolado a posiciones de memoria. Verificación dinámica de la inicialización.
- Solución -> valor especial nulo: *nil* en Pascal, *void* en C/C++, *null* en ADA, Python.
- **Punteros y uniones discriminadas:**
  - Puede permitir accesos a cosas indebidas.
  - Java lo soluciona eliminando la noción de puntero explícito completamente.
- **Alias:**
  - Si 2 o más punteros comparten alias, la modificación que haga uno también se verá reflejado en los demás.
- **Liberación de memoria (objetos perdidos):**
  - Si los objetos en la heap dejan de ser accesibles, esa memoria podría liberarse.
  - Un objeto es accesible si alguna variable en la pila lo apunta, directa o indirectamente.
  - Un objeto es basura si no es accesible.

## Ejercicio 6

### 1. Característica que debe cumplir una unidad para que sea un TAD:

- **Encapsulamiento:**
  - La representación del tipo y las operaciones permitidas para los objetos del tipo se describen en una única unidad sintáctica.
  - Refleja las abstracciones descubiertas en el diseño.
- **Ocultamiento:**
  - La representación de los objetos y la implementación del tipo permanecen ocultos.
  - Refleja los niveles de abstracción. Modificabilidad.

Se debe definir **qué datos representa**.

Se debe definir **qué operaciones se pueden** realizar sobre los datos, sin implementarlas.

### 2. Ejemplos de TAD:

- En ADA se conocen como **Paquete**.
- En Modula-2 se conoce como **módulo**.
- En C++ y Java se conocen como **clase**.

# Práctica 8

## Ejercicio 1

### Sentencia simple:

- Es una expresión o combinación de expresiones que realiza una acción específica, como asignar un valor a una variable, realizar una operación aritmética, llamar a una función o controlar el flujo del programa. Las sentencias suelen terminar con “;” en muchos lenguajes de programación.

### Sentencia compuesta:

- Agrupa varias sentencias simples o incluso otras compuestas, formando una **unidad lógica**. Dependiendo de las reglas de cada lenguaje, puede requerir usar *delimitadores*, como *begin-end* o *{}*.
- Las sentencias compuestas permiten agrupar varias sentencias como si fueran una sola, lo cual es útil en estructuras de control como *if*, *for*, *while*, y en la definición de funciones.

## Ejercicio 2

### Sentencia de Asignación en C:

- Las sentencias de asignación devuelven valores. Devuelve el valor que es asignado.
- En C se evalúa la asignación de derecha a izquierda. “a=b=c=0” se asigna 0 -> c -> b -> a. Todos quedan en 0.
- **Si se usa asignaciones en condiciones (if) primero asigna y luego retorna el lo que asignó. Y en C cualquier valor distinto a 0 es true.** Ejemplo:

```
int x;
if (x = 0) {
    // Esto no se ejecuta porque x = 0 devuelve 0 → falso
}

if (x = 42) {
    // Esto sí se ejecuta porque x = 42 devuelve 42 → verdadero
}
```

## Ejercicio 3

Cuando hay una asignación con un incremento de variable o con una llamada de función, dependiendo **cuál se evalúe primero el resultado puede ser distinto**. En C, **el estándar no especifica el orden exacto** en que se evalúan los operandos de muchas expresiones, incluyendo los argumentos de funciones y los lados de una asignación compleja. Esto da lugar a comportamientos indefinidos o dependientes del compilador.

```
int x = 1;
int y = (x = 5) + x; // ¿Qué valor tiene y?
```

Al no saberse el orden de ejecución (por izquierda o por derecha por ej), **Y** podría tomar el valor 10 o el valor 6.

## Ejercicio 4

### Circuito Corto:

- En un lenguaje que utiliza circuito corto, la evaluación de una expresión lógica **se detiene tan pronto como se determina el resultado** final sin necesidad de evaluar el resto de la expresión.
- También conocida como evaluación perezosa o evaluación de cortocircuito.
- **Operador and:**
  - Da como resultado verdadero únicamente cuando ambos términos son verdaderos. Si el primer término es falso, no es necesario evaluar el segundo.
- **Operador (or):**
  - Da como resultado falso únicamente cuando ambos términos son falsos. Si el primer término es verdadero, no es necesario evaluar el segundo.
- Permite evitar errores y optimizar el rendimiento.
- Ejemplos: C, Python, Java, etc.

### Circuito Largo:

- En un lenguaje que utiliza el circuito largo, la evaluación de una expresión lógica **continúa evaluando todas las partes** de la expresión, incluso si el resultado final ya se ha determinado.
- **Operador and:**



- En un circuito largo, ambas partes de la expresión se evalúan independientemente del valor de la primera parte. Esto asegura que se realicen todas las comprobaciones necesarias.
- **Operador or:**
  - En un circuito largo, ambas partes de la expresión se evalúan independientemente del valor de la primera parte. Esto asegura que se realicen todas las comprobaciones necesarias.
- Ejemplos: Pascal, ADA.

Un ejemplo en el que un circuito dé error y otro no: *if( a != 0 && (b/a) > 1)* . En este caso, el circuito corto no daría error, mientras que el circuito largo podría dar error al dividir por 0.

## Ejercicio 5

### Lenguaje C:

- Usa la regla del if más cercano.
- No importa la indentación.
- Se recomienda usar llaves para evitar ambigüedades.

### Lenguaje Delphi (basado en Pascal):

- También asocia el else con el if más cercano.
- Para evitar la ambigüedad se usa begin..end en lugar de {} para delimitar bloques.

### Lenguaje ADA:

- También sigue la regla del if más cercano.
- Pero no hay ambigüedad porque se usa **end of** siempre. El bloque siempre es explícito.

### Lenguaje Python:

- No tiene este problema porque la **indentación es obligatoria**, lo que elimina ambigüedad.
- El else siempre se asocia al if a partir de la indentación.

## Ejercicio 6

La construcción que usa C para expresar la selección múltiple es el **switch**:

```
C Copiar  
  
switch (expresión) {  
    case constante1:  
        // acciones  
        break;  
    case constante2:  
        // acciones  
        break;  
    default:  
        // acciones por defecto  
}
```

- Solo acepta **valores constantes enteros** o **enumerados**.
- Si coincide con una etiqueta del Switch se ejecutan las sentencias asociadas, y se continúa con las sentencias de las otras entradas. (chequea todas salvo que exista un **break**).
  - **Fall-through**: En un **switch**, cuando el flujo entra en un caso, si no hay un **break** o alguna otra instrucción que termine el bloque, el programa **sigue ejecutando las instrucciones de los siguientes casos** hasta encontrar un **break** o llegar al final.
- Existe opción **default** que sirve para los casos que el valor no coincida con ninguna de las opciones establecidas, es opcional.
- Si un valor no cae dentro de alguno de los casos de la lista de casos y no existe un default no se provocará un error por esta acción, pero se podrían generar efectos colaterales dentro del código. Por eso es importante chequear el valor si puede tomar valores fuera del rango de la lista de casos.

## Ejercicio 7

- En el caso específico de Pascal el código no genera error aunque conceptualmente esté mal ya que modifica manualmente la variable iteradora. Lo que hace que no se genere error es que la modificación de la variable no se da dentro de la misma unidad que el bloque for.
- En el caso de Ada el código si generaría error. No compila porque la variable de control `i` es de solo lectura. Esto obliga al programador a no modificar la variable de control dentro del ciclo.

## Ejercicio 8

Ese código en **C** sí tiraría error ya que **no se permiten los rangos** en los case.

- Para solucionarlo habría que evaluar cada expresión por separado.

En **Pascal no genera error** ya si una variable no cae dentro de alguno de los casos de la lista de casos del case, se sigue la ejecución y la variable no se verá modificada.

En **ADA sí generaría un error** en compilación, ya que no se están considerando todos los casos posibles. Esto ADA no lo permite.

- Para solucionarlo habría que agregar el bloque *others*, para que tome los casos no contemplados.

## Ejercicio 9

### YIELD

- Cuando se encuentra una declaración **yield**, la función se detiene temporalmente y devuelve el valor especificado. Sin embargo, la función no se termina por completo; en su lugar, mantiene su estado actual y puede continuar ejecutándose cuando se solicita el próximo valor.
- El generador recuerda su estado y puede reanudar la ejecución desde el punto donde se detuvo anteriormente.

### RETURN

- Return se utiliza para devolver un valor desde una función y terminar la ejecución de la función en ese punto.
- Cuando se encuentra una declaración **return**, la función se detiene y devuelve el valor especificado, si lo hay, al lugar donde fue llamada.
- Después de que se ejecuta una declaración return, cualquier código que siga a esa declaración dentro de la función no se ejecutará.

**Yield se vuelve útil cuando se necesita producir una secuencia de valores uno a la vez y mantener el estado de la función entre las llamadas.** Por ejemplo al leer archivos de texto que por cada línea leída se hace un yield 1. De esta forma se va a ir leyendo de a una línea a la vez y cada vez que se vuelve a la unidad que itera al archivo, no se pierde el estado.

## Ejercicio 10

La función **map()** de JavaScript recorre cada elemento de un array, aplica una función para transformarlo, y **lo retorna en un nuevo array**, para no modificar el original.

Una alternativa puede ser **forEach()**, el cual recorre cada elemento de un array, pero **no devuelve nada**. También puede usarse el **for** tradicional, o el **reduce()**.

Todos, salvo **forEach()**, que no retorna nada, **retornan un array nuevo**.

## Ejercicio 11

Un **espacio de nombres** es un contexto que asocia **nombres (identificadores)** con **objetos (valores, funciones, clases, etc.)**. Permite **organizar y evitar colisiones** entre nombres definidos en distintas partes del programa.

- Es **una tabla interna donde se guardan las asociaciones entre nombres y objetos**. Es como una "agenda" donde Python anota a qué objeto apunta cada nombre.
- Sirven por ejemplo para programas grandes, ya que se pueden tener muchos nombres iguales. Entonces, se necesita **separar los contextos**. Un mismo nombre puede existir en diferentes partes del programa sin interferirse, si están en distintos espacios de nombres.

Python implementa espacios de nombres de forma jerárquica y dinámica. Se utilizan estructuras como:

- **import**: para traer módulos (espacios de nombres externos)
  - Ejemplo: `import math` → accedés con `math.sqrt()`
- **from ... import ...**: para importar símbolos específicos
  - Ejemplo: `from math import sqrt` → accedés con `sqrt()`
- **globals()** y **locals()**: funciones que permiten inspeccionar o manipular el espacio de nombres actual.
- Las **funciones, clases y módulos** también definen espacios de nombres propios

Características importantes de Python:

- **Jerarquía de espacios de nombre** - Python organiza los espacios en niveles:
  - Local (dentro de funciones)
  - Enclosing (funciones anidadas)
  - Global (archivo actual)

- Built-in (definiciones del lenguaje)
- **Evita colisiones:** Los módulos y funciones pueden tener nombres iguales sin interferir entre sí si están en distintos namespaces.
- **Flexibilidad dinámica:** Se pueden inspeccionar y modificar namespaces en tiempo de ejecución.
- **Modularidad y claridad:** Forma la organización del código mediante módulos y paquetes, lo cual mejora el mantenimiento y escalabilidad.

## Práctica 9

### Ejercicio 1

**Excepción:** Es una situación anómala que se da en la ejecución de un programa y que se supone que ocurre con poca frecuencia.

- Es una condición inesperada que ocurre durante la ejecución del programa y que no puede ser manejada en el contexto local. Es necesario controlarlo. Interrumpe el flujo normal de ejecución y ejecuta un *controlador de excepciones* registrado previamente (ya sea por el programador o el lenguaje).

### Ejercicio 2

Para que un lenguaje trate excepciones debe proveer:

- Un modo de **definirlas**.
- Una forma de alcanzarlas, **invocarlas**.
- Una forma de **manejarlas** (catch).
- Un criterio de **continuación** (terminación o reasunción).

No todos los lenguajes implementan un manejo estructurado de excepciones, ya que se apoyan en **códigos de error**.

Un lenguaje moderno debería ofrecer mecanismos explícitos, estructurados y seguros para manejar excepciones. No todos lo hacen: algunos lo simulan, otros lo resuelven con patrones de diseño o estructuras funcionales.

## Ejercicio 3

Cuando un lenguaje no provee un manejo de excepciones, no se puede interrumpir el flujo normal del programa de forma estructurada. Esto significa que el programa podría *crashear* instantáneamente, **no se puede propagar un error a niveles superiores**, y el programador debe *manualmente* verificar cada operación que piense que puede levantar una excepción.

Para simular un manejo de excepciones, se puede, por ejemplo, hacer que una función retorne un valor especial ante una situación anómala, o registrar errores globales en C con la variable **errno**, o directamente usar estructuras **if-else** para simular el uso de bloques **try-catch**.

## Ejercicio 4

1) Los modelos que existen son:

- **Por terminación:** El bloque que genera la excepción busca manejarla y **finaliza**.
- **Por reasunción:** Ningún proceso termina cuando se levanta una excepción, simplemente se detecta la excepción, se la atiende y **continúa el flujo de ejecución** en el punto siguiente a donde se levantó la excepción.

2)

- **Java**
  - Maneja el modelo por **terminación**.
  - Las excepciones que maneja son **clases**, las cuales extienden la clase *Exception*. Se instancian como cualquier otra clase.
  - Posee propagación **dinámica**: una vez lanzadas, en caso de no ser tratadas por la unidad que las generó, se propagan dinámicamente.
    - Si un método puede generar excepciones pero decide no manejarlas localmente, debe enunciar en su encabezado **throws**.
- **Python**
  - Maneja el modelo por **terminación**.
  - Los bloques que manejan excepciones se identifican con **try**, y los manejadores con **except**.
  - En primer lugar tiene propagación **estática**. Cuando la excepción ocurre dentro del bloque try, busca inmediatamente el manejador en los except siguientes. Si no lo encuentra, lo propaga **dinámicamente** a quién lo invocó. Si llega al main sin encontrar el manejador termina en error.

- Se pueden lanzar excepciones explícitamente con ***raise*** **<nombre\_excepcion>**. Si no aparece el nombre se levanta anónimamente: se levanta la última excepción que estaba activa en el entorno, y si no había ninguna, se levanta RuntimeError.
- **PL/1**
  - Maneja el modelo de **reasunción**.
  - Se definen con la instrucción **ON CONDITION nombre Manejador**.
  - Se levantan con la instrucción **SIGNAL CONDITION nombre**.
  - A medida que se van encontrando los ON CONDITION se van aplicando en una pila de manejadores, y cuando se levanta una excepción se busca el nombre de la misma desde el tope de la pila hacia abajo.
    - El primer manejador que se encuentra es el que se ejecuta.
    - Cuando una unidad termina se desapilan sus manejadores.

3) El modelo más inseguro es el de PL/1, ya que los manejadores **se activan automáticamente** al entrar en una unidad y **se ejecutan en cualquier punto del código** cuando ocurre la excepción, sin garantizar que el contexto del programa esté preparado para manejarla. Puede provocar:

- Inconsistencia de estado.
- Menos legible.
- Menor encapsulamiento.

Python y Java son más seguros porque su modelo por terminación **garantiza un control explícito y ordenado** de las excepciones. Implica:

- Flujo de ejecución claro.
- Encapsulamiento del manejo de errores.
- Propagación dinámica controlada.
- Declaración explícita.

## Ejercicio 5

### Java

- Propagación: **Dinámica** (la excepción sube por la pila de llamadas hasta encontrar un catch).
- Si un método no maneja una excepción, debe declararla con throws. Si nadie la atrapa, el programa crashea.

### Python

- Propagación: Primero **estática** (busca en los except del try actual), luego **dinámica** (si no la maneja, sube al caller).
- Si no se atrapa, llega al programa principal y muestra un error. Se pueden relanzar con raise.

#### PL/1

- Propagación: Basada en pila (busca manejadores ON CONDITION desde el más reciente).
- Si no hay manejador, no hay propagación automática. Esta situación puede ser manejada en la implementación, pero por lo general el programa termina.

### Ejercicio 6

Este programa en Pascal simula el manejo de excepciones por **reasunción**, ya que en caso de haber una excepción en el procedimiento P, la misma es manejada (con el if llamando al procedimiento manejador "X") y una vez que termina de manejarse continúa con la ejecución del bloque.

- Para que encuadre con los lenguajes que no utilizan este modelo, en el mismo if luego de llamar al procedimiento "X" habría que tener una instrucción para forzar la salida del procedimiento. Como el exit en free pascal.

### Ejercicio 7

a) b)

<https://detailed-cathedral-9a9.notion.site/Pr-ctica-7-C-digos-20e8804632d580229a59e67dfcd3116b>

### Ejercicio 8

- a) Este programa en Java ejecutaría el manejador **Exception a**, ya que al ejecutarse el método **acceso\_por\_indice()** este levanta una excepción genérica (encabezado que dice *throws Exeption*) y la propaga dinámicamente al main.
- Esta situación se da cuando el segundo for tenga los valores 500, 525 y 575 e imprimiría en pantalla "el índice (índices de arriba) no es una posición válida".
  - Nunca ejecutaría el manejador "ArrayIndexOutOfBoundsException", porque el if del método controla que no se pueda acceder a una posición inválida.
  - Y en cada iteración también se ejecutará la sentencia finally.



- b) La excepción no es manejada localmente en el método **acceso\_por\_indice()** y es propagada dinámicamente al main, al indicar **throws** en su encabezado, para ambas excepciones.
- Indica que el método puede lanzar esas excepciones y que cualquier llamada a ese método debe manejar o propagar esas excepciones. En este caso, el método "main" captura y maneja las excepciones utilizando bloques try-catch.
- c) Para que maneje él mismo las excepciones habría que agregar un try-catch directamente en el método **acceso\_por\_indice()**.

## Ejercicio 9

Manejo de excepciones en Java y Python:

**Similitudes:** La semántica es prácticamente la misma.

- Ambos lenguajes manejan el modelo por terminación.
- Estructura try: Ambos lenguajes utilizan una estructura try para capturar y manejar excepciones.
- Tipos de excepciones: Utilizan una jerarquía de clases de excepciones predefinidas.
- Bloque finally: Ambos permiten el uso de un bloque finally (opcional), que se ejecuta siempre se haya producido una excepción o no.

**Diferencias:**

- Declaración de excepciones: En Java, se requiere que los métodos declaren las excepciones específicas que pueden lanzar utilizando la palabra clave **throws** en su firma. En Python, no se requiere una declaración explícita de excepciones en la firma de un método.
- Bloque except: En Python, se utiliza la palabra clave **except** para capturar excepciones específicas. En Java, se utiliza la palabra clave **catch** para capturar y manejar excepciones.
- Python define el bloque opcional **else**, el cual se ejecuta en caso de que no se levante ninguna excepción.

## Ejercicio 10

**Ruby** proporciona una serie de instrucciones específicas para el manejo de excepciones que permiten capturar, lanzar y manejar errores en el código. Enfoque por **terminación**.

- **begin-rescue-end:** Esta es la estructura básica utilizada para capturar y manejar excepciones en Ruby. El código problemático se coloca dentro del bloque begin, y las excepciones se capturan y manejan en el bloque rescue.

```
begin
  # Código problemático
rescue ExceptionType
  # Manejo de la excepción
end
```

- StandardError para cualquier cualquier excepción estándar.
- **raise:** Esta instrucción se utiliza para lanzar una excepción explícitamente en un punto determinado del código.
- **rescue:** Esta instrucción se utiliza dentro de un bloque begin-rescue para capturar excepciones específicas. Puede capturar excepciones individuales o agrupar múltiples excepciones. Similar al catch o except.
- **ensure:** Esta instrucción se utiliza dentro de un bloque begin-rescue para ejecutar código que se debe ejecutar siempre, independientemente de si se produce o no una excepción. Similar al finally.

## Ejercicio 11

**Javascript** utiliza un mecanismo de excepciones similar a otros lenguajes de programación. Proporciona una estructura de control try-catch-finally para capturar y manejar excepciones.

- **try-catch-finally:** Se utiliza el **try** para el código problemática. Las excepciones se capturan y manejan en el bloque **catch**. El bloque finally es opcional y se utiliza para ejecutar código que siempre debe ejecutarse, independientemente de si se produce una excepción o no.

```
try {
  // Código problemático
} catch (error) {
  // Manejo de la excepción
} finally {
  // Código que se ejecuta siempre
}
```

- **throw:** Esta instrucción se utiliza para lanzar una excepción explícitamente en un punto determinado del código. Puede lanzar una excepción predefinida o una excepción personalizada.
- Javascript tiene excepciones predefinidas y todas las excepciones son objetos que heredan de la clase Error.

También proporciona otros mecanismos adicionales para manejar excepciones, como el uso de bloques catch anidados para capturar excepciones específicas y el acceso a la traza de la pila a través de la propiedad stack del objeto Error. Las excepciones pueden propagarse a través de las llamadas de función hasta que se capturen o lleguen al ámbito global, donde pueden generar un error no capturado que detiene la ejecución del programa.

## Ejercicio 12

a) Caminos de ejecución:

- a y b dos números, y  $b \neq 0$  realiza la división correctamente. E imprime "Vuelve a probar"
  - El camino sería: try, ejecuta el procedimiento, imprime, imprime el finally y termina. No se levanta ninguna excepción.
- a y b dos números, y  $b = 0$  se levanta la excepción ZeroDivisionError, se imprime "No se permite dividir por cero" y "Vuelve a probar".
  - El camino sería: try, ejecuta al procedimiento, al intentar dividir por 0 levanta la excepción y es manejada por el except, ejecuta el try y vuelve a solicitar dos números.
- a y b no son dos números: se produce un ValueError al intentar convertir el valor ingresado a entero, el programa lanzará la excepción sin capturarla y mostrará el rastreo de la pila (traceback) junto con un mensaje de error estándar. En este caso, el bloque finally no se ejecutará y el programa se detendrá.

b) Notion

## Ejercicio 13

### Ejecución

- Se instancia e inicializan todos los objetos/variables necesarias.
- Se inicia un bloque **try** para manejar posibles excepciones:
  - Se inicializa el FileInputStream con un txt.

- Mientras la lectura del txt != -1 se lee.
  - Si el valor leído es negativo, se llama a el método de "lanzaSiNegativo" que lanza una instancia de la excepción "ExcepcionUno" con el mensaje "Numero negativo".
  - Si el valor leído es mayor que 100, se verifica si es menor que 125. Si se cumple esta condición, se llama a el método de "lanzaSiMayor100" que lanza una instancia de la excepción "ExcepcionDos" con el mensaje "Numero mayor100".
  - Si no ocurre ninguna excepción en la lectura, se cierra el archivo txt y se imprime "Todo fue bien".
- Si ocurre alguna excepción de tipo ExcepcionUno, se captura en el bloque **catch (ExcepcionUno e)** y se muestra el mensaje "Excepcion: " + mensaje de la excepción, que sería "Numero negativo".
- Si ocurre alguna excepción de tipo ExcepcionDos, se captura en el bloque **catch (ExcepcionDos e)** y se muestra el mensaje "Excepcion: " + mensaje de la excepción, que sería "Numero mayor100".
- Si ocurre alguna excepción de tipo ExcepcionTres, se captura en el bloque **catch (ExcepcionTres e)** y se muestra el mensaje "Excepcion: " + mensaje de la excepción, que sería "Numero mayor125".
- En el bloque finally:
  - Si la entrada no es null, dentro del try intenta cerrar el archivo (esto es por si se lanza una excepción, el archivo puede no cerrarse). A su vez se define el manejador de la excepción si falla el cierre.
  - Imprime "Fichero cerrado".

### Posibles caminos

1. No ocurre ninguna excepción en la lectura del archivo, imprimiendo "Todo fue bien" y seguido de "Fichero cerrado" del finally.
  - a. En el finally no entra al if.
2. Ocurre una excepción en la lectura del archivo debido a que el valor leído es negativo.
  - a. En el bloque try, se lanza la excepción de tipo **ExcepcionUno**, y su manejador imprime "Excepción: Numero negativo".
  - b. El archivo se cierra a través del try del finally e imprime "Fichero cerrado".
3. Ocurre una excepción en la lectura del archivo por un valor leído entre 100 y 125.
  - a. En el bloque try, se lanza la excepción de tipo **ExcepcionDos**, y su manejador imprime "Excepcion: Numero mayor100".

- b. El archivo se cierra a través del try del finally e imprime "Fichero cerrado".
- 4. Ocurre una excepción de entrada/salida, por algún problema al abrir, leer o cerrar el archivo.
  - a. En el bloque try, se lanza una excepción de tipo **IOException**, y su manejador imprime "Excepcion:" + el mensaje en sí de la excepción IOException.
  - b. El archivo se cierra a través del try del finally e imprime "Fichero cerrado".
- 5. Ocurre una excepción que interrumpe el bloque try que itera sobre el txt, o falla al cerrar el archivo (lo mencionado en los puntos anterior). Y al intentar cerrarlo en el **try** del **finally**, también se lanza una excepción.
  - a. Dicha excepción es manejada por el catch (Exception e) y se imprime "Excepcion: " + el mensaje de la excepción generada.
  - b. También se imprime "Fichero cerrado".

## Ejercicio 14

### Posibles caminos de ejecución:

- No se produce ninguna excepción en la apertura del archivo y en la lectura del mismo.
  - Se imprime "Es persona adulta, Todo fue bien", en todas las edades del archivo.
  - El finally no entra al if e imprime "Fichero cerrado".
- Se produce una excepción al intentar abrir el archivo, debido a que no es encontrado.
  - Se lanza la excepción en el try de más afuera y es manejada por el **catch(FileNotFoundException e1)** que imprime "No se encontró el archivo".
  - Al entrar al finally se detecta que el archivo es null (no se encontró) y se imprime "Fichero cerrado".
- Se produce una excepción al intentar leer el archivo.
  - Se lanza la excepción en el try de más afuera y es manejada por **catch(IOException e)** que imprime "Problema al leer los datos".
  - Al entrar al finally se detecta que el archivo es distinto de null y lo cierra e imprime "Fichero cerrado".
- La apertura y lectura del archivo funciona correctamente. Al leer un dato se detecta que el mismo es negativo lo que lanza una excepción.

- Se lanza la excepción en el try anidado que está dentro del while, y la misma es manejada por el **catch (ExcepcionE1 e)** e imprime "Excepcion: Edad inválida".
  - El while continúa su ejecución.
  - El finally no entra al if e imprime "Fichero cerrado".
- La apertura y lectura del archivo funciona correctamente. Al leer un dato se detecta que el mismo es mayor a 120 lo que lanza una excepción.
  - Se lanza la excepción en el try anidado que está dentro del while, y la misma es manejada por el **catch (ExcepcionE1 e)** e imprime "Excepcion: Edad inválida".
  - El while continúa su ejecución.
  - El finally no entra al if e imprime "Fichero cerrado".
- La apertura y lectura del archivo funciona correctamente. Al leer un dato el método Evalua() lanza una excepción por edad menor a 18.
  - Se lanza la **ExcepcionE1** en el método Evalua(), y la misma es propagada dinámicamente al main que la captura con el **catch (ExcepcionE1 e)** e imprime "Excepcion: Es una persona menor de edad".
  - El while continúa su ejecución.
  - El finally no entra al if e imprime "Fichero cerrado".
- La apertura y lectura del archivo funciona correctamente. Al leer un dato el método Evalua() lanza una excepción por edad mayor a 70.
  - Se lanza la **ExcepcionE2** en el método Evalua(), y la misma es propagada dinámicamente al main que la captura con el **catch (ExcepcionE2 e)** e imprime "Excepcion: Es persona mayor de edad".
  - El while continúa su ejecución.
  - El finally no entra al if e imprime "Fichero cerrado".
- La apertura y lectura del archivo funciona correctamente. Al leer un dato el método Segmenta() lanza una excepción por edad menor a 35.
  - Se lanza la **ExcepcionE1** en el método Segmenta(), y la misma es propagada dinámicamente al main que la captura con el **catch (ExcepcionE1 e)** e imprime "Excepcion: Es una persona joven".
  - El while continúa su ejecución.
  - El finally no entra al if e imprime "Fichero cerrado".
- Ocurre una excepción al intentar cerrarlo en el **try** del **finally**.
  - Dicha excepción es manejada por el **catch (Exception e)** y se imprime "Excepcion: " + el mensaje de la excepción generada.
  - También se imprime "Fichero cerrado".

# Práctica 10

## Ejercicio 1

Un programa escrito en un lenguaje funcional es un programa que tiene como componente principal a las **funciones** que *transforman* datos de forma **inmutable**.

- En vez de secuencias de instrucciones (como en C), se usan funciones matemáticas puras, que siempre dan el mismo resultado para el mismo input y no cambian el estado del programa (no tienen efectos colaterales).

La computadora cumple un rol de **evaluador de expresiones**, comportándose como un *intérprete* de fórmulas matemáticas resolviendo funciones.

## Ejercicio 2

El lugar donde se definen las funciones en un lenguaje funcional se define como **entorno léxico**. Este entorno es una estructura que asocia identificadores con valores o funciones. Los entornos son inmutables, es decir, cuando se define un identificador, no puede cambiar.

El lugar donde se definen las funciones puede variar dependiendo del lenguaje y del estilo de programación funcional utilizado.

- Definición en módulos o archivos.
- Definición en el ámbito global.
- Definición local dentro de una función.
- Definición mediante expresiones lambda.

### Ejercicio 3

La noción de **variable** es la de “variable matemática”, no la de celda de memoria. Las variables son inmutables y no pueden ser modificadas una vez asignado un valor. Esto promueve la seguridad, la concurrencia y facilita el razonamiento y la comprensión del código.

### Ejercicio 4

Una **expresión** es una combinación de valores, variables y funciones que se evalúa para producir un resultado. El valor de una expresión depende únicamente de los valores de las subexpresiones que la componen.

- Una expresión siempre produce el mismo resultado cuando se evalúa con los mismos valores de entrada, sin importar cuándo o dónde se evalúe.

### Ejercicio 5

- **Orden aplicativo:** Aunque no lo necesite siempre evalúa los argumentos. Evalúa las expresiones de inmediato, es decir, tan pronto como las expresiones se encuentran en el flujo de control del programa. Calcula los valores de las expresiones antes de utilizarlos en el programa. Esto significa que los argumentos de las funciones se evalúan antes de que se llame a la función y los resultados se pasan a la función.
- **Orden normal (Lazy):** No calcula más de lo necesario. En lugar de evaluar una expresión inmediatamente, retrasa la evaluación hasta que el resultado sea necesario en otra parte del programa. Esto permite evitar la evaluación innecesaria de expresiones y mejora la eficiencia en ciertos casos. Una expresión compartida NO es evaluada más de una vez

### Ejercicio 6

Sí, en general, **los lenguajes funcionales son considerados fuertemente tipados.**



Esto significa que los lenguajes funcionales imponen restricciones más estrictas en las operaciones y conversiones entre diferentes tipos de datos, y requieren una correspondencia precisa de tipos para realizar operaciones.

En los lenguajes funcionales, los tipos de datos son importantes y se verifican en tiempo de compilación para garantizar la coherencia y la seguridad del programa. Los tipos ayudan a prevenir errores comunes, como la aplicación de operaciones incorrectas o la asignación de valores incompatibles. Los tipos comunes que se encuentran en los lenguajes funcionales:

- Básicos (primitivos): NUM (INT y FLOAT), BOOL, CHAR.
- Derivador: Se construyen a partir de otros tipos.

## Ejercicio 7

Un programa escrito en el paradigma orientado a objetos se compone por una serie de objetos que se mandan **mensajes** entre sí para ir cambiando sus respectivos estados internos.

## Ejercicio 8

Los elementos más importantes de la programación orientada a objetos son:

- **Mensajes:** son **peticiones** de un objeto a otro para que éste se *comporte* de una determinada manera, que se ejecuta por medio de los métodos correspondientes.
- **Métodos:** es un programa asociado a un objeto determinado y cuya ejecución sólo puede desencadenarse a través de un **mensaje recibido**
- **Clases:** es un tipo (normalmente definido por el usuario) que determina las estructuras de datos que éste utiliza y las operaciones asociadas. Se define la estructura, comportamiento y **se ocultan los datos**. La información del objeto sólo debe ser accedida por la ejecución de sus *métodos*.
- **Instancias:** cada vez que se construye un objeto se *instancia* su clase definida. Es un **objeto individualizado** por los valores que sus **atributos** toman.
- **Herencia:** es una propiedad de las clases que enuncia que si una clase es subclase de otra, entonces *hereda* jerárquicamente sus **propiedades**.

- **Polimorfismo:** capacidad que tienen los objetos de distintas clases de **responder a mensajes con el mismo nombre**.

## Ejercicio 9

El segundo nivel de abstracción en la programación orientada a objetos después del ocultamiento y encapsulamiento es la **herencia**. Permite crear nuevas clases basadas en clases existentes, heredando sus atributos y métodos, y estableciendo relaciones jerárquicas entre las clases.

- La herencia promueve la reutilización de código, la modularidad y la extensibilidad en el diseño de programas orientados a objetos.

## Ejercicio 10

Existen diferentes tipos de herencia en la programación orientada a objetos. Los dos tipos más comunes son la **herencia simple** (single inheritance) y la **herencia múltiple** (multiple inheritance).

- **Herencia simple:** Una clase puede heredar de una única clase base. Esto significa que una clase derivada puede extender y especializar los atributos y métodos de una única clase padre. La herencia simple promueve una relación jerárquica simple entre las clases y se utiliza ampliamente en muchos lenguajes orientados a objetos.
  - Un ejemplo es Java.
- **Herencia múltiple:** Una clase puede heredar de múltiples clases bases. Esto permite que una clase derivada herede atributos y métodos de varias clases padres diferentes.
  - La herencia múltiple ofrece una mayor flexibilidad y capacidad de reutilización de código, pero también puede ser más compleja de manejar y puede dar lugar a problemas de ambigüedad si existen métodos con el mismo nombre en diferentes clases padres.

Es importante tener en cuenta que algunos lenguajes de programación no admiten la herencia múltiple debido a la complejidad que puede generar. En su lugar, pueden proporcionar alternativas, como interfaces.

## Ejercicio 11

En el paradigma lógico, una **variable** representan elementos indeterminados que pueden sustituirse por cualquier otro. Los nombres de las variables comienzan con mayúsculas y pueden incluir números.

- "humano(X)"
- X sería una variable, que puede sustituirse por cualquier valor (juan, pepe, etc).

Las **constantes** son **elementos determinados**. Pueden ser una cadena de caracteres (objetos atómicos) o de dígitos (números)

## Ejercicio 12

Un programa en el paradigma lógico se escribe mediante **aserciones lógicas** (reglas o hechos) que proveen una **especificación declarativa** de lo que se conoce. Estas aserciones se evalúan por las **queries**, el cual sería el *objetivo que se quiere alcanzar*.

### Hecho

Son relaciones entre objetos y siempre son verdaderas:

- tiene(coche,ruedas): representa el hecho que un coche tiene ruedas
- longitud([],0): representa el hecho que una lista vacía tiene longitud cero
- moneda(peso): representa el hecho que peso es una moneda.

### Regla

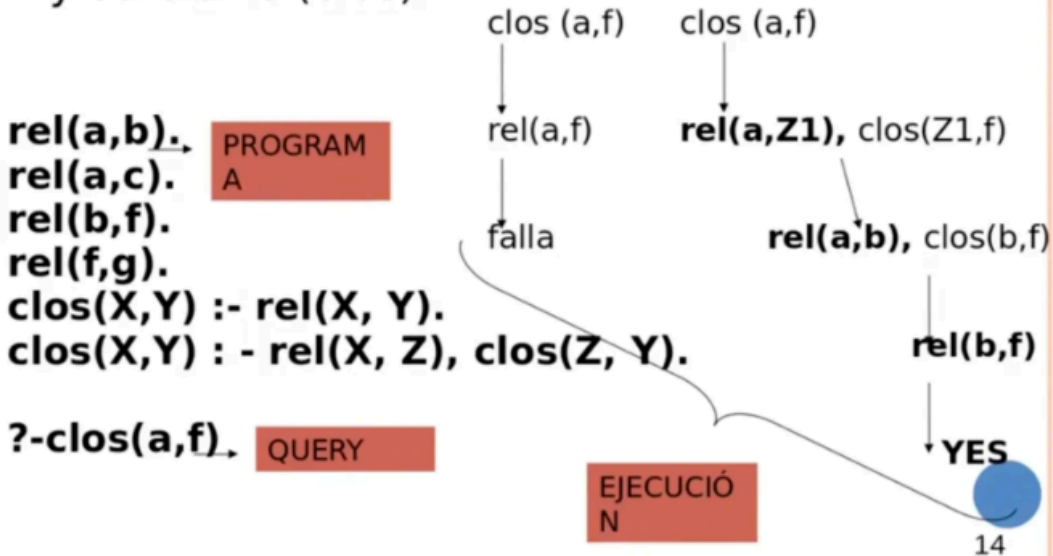
Cláusulas de Horn La cláusula tiene la forma conclusión :- condición

- Donde :- es If
- conclusión es un **predicado**
- **condición** es una conjunción de predicados separados por comas, representando un AND lógico.

Sería como decir if condición else conclusión.

## EJECUCIÓN DE PROGRAMAS: EJEMPLO

Programa que describe una relación binaria (rel) y su cierre (clos):



### Ejercicio 13

- Orientado a objetos

```
public class ParesEImpares {  
    private Scanner input;  
  
    public void evaluacion() {  
        input = new Scanner(System.in);  
        int valor = input.nextInt();  
        if (valor % 2 == 0) System.out.println("El valor ingresado es PAR");  
        else System.out.println("El valor ingresado es impar");  
    }  
}
```

- Paradigma imperativo

```

program paresEImpares;
var
  valor: integer;
begin
  read(valor);
  if (valor MOD 2 = 0) then
    write("El valor ingresado es PAR");
  else
    write("El valor ingresado es impar");
  end.

```

- Paradigma funcional

```

h

main = do
  putStrLn "Ingrese un número:"
  input <- getLine
  let numero = read input :: Int
  putStrLn (paridad numero)

paridad n
  | even n    = "El valor ingresado es PAR"
  | otherwise = "El valor ingresado es IMPAR"

```

- **main**: Es la función principal que maneja la entrada/salida.
- **getLine**: Lee una línea desde el teclado.
- **read input :: Int**: Convierte el texto ingresado a entero.
- **paridad**: Es una función pura que devuelve un string según si el número es par o impar.
- **even n**: Es una función de Haskell que devuelve true si el número es par.

- Paradigma lógico

prolog

```
par_o_impar :-  
    write('Ingrese un número: '),  
    read(X),  
    verificar_paridad(X).  
  
verificar_paridad(X) :-  
    0 is X mod 2,  
    write('El valor ingresado es PAR'), !.  
  
verificar_paridad(_) :-  
    write('El valor ingresado es IMPAR').
```

- **par\_o\_impar**: Predicado principal, pide un número y se lo pasa a *verificar\_paridad*.
- **verificar\_paridad(x)**: Verifica si el número es divisible por 2 (*0 is X mod 2*).
- **!**: Operador de corte, evita que Prolog siga buscando soluciones si ya encontró una.
- Si no es par, Prolog pasa al siguiente predicado **verificar\_paridad(\_)**, que asume que es impar.

## Ejercicio 14

Los lenguajes basados en scripts combinan **programas escritos en otros lenguajes (scripts)** para *extenderlos*, para poder armar un nuevo programa por medio de sus uniones.

Las características más importantes son:

- **Uso de scripts** para la combinación de los programas: promueven el uso como lenguaje de extensión, ya que permiten al usuario adaptar o extender funcionalidades de scripts.
- **Desarrollo y evolución rápida**: algunos scripts se escriben y ejecutan una única vez como una secuencia de comandos, y en otros casos se utilizan más frecuentemente,

por lo que deben ser adaptados a nuevos requerimientos. Son **fáciles de escribir** y tienen una **sintaxis concisa**.

- Uso de **editores livianos**: pueden ser escritos en procesadores de texto.
- Normalmente **interpretados**: la eficiencia no es un requisito esencial para los scripts, pero debe ser considerado al combinar.
- **Alto nivel de funcionalidad** en aplicaciones para áreas específicas.

Normalmente los lenguajes basados en scripts soportan un **tipado dinámico**, ya que puede necesitarse intercambiar datos de distinto tipo entre distintos subsistemas y éstos pueden ser *incompatibles*, por lo que se debe ofrecer una flexibilidad. Las declaraciones también suelen ser poco frecuentes.

## Ejercicio 15

Si, existen muchísimos paradigmas más. Algunos de ellos:

1. **Paradigma Orientado a Aspectos (AOP)**: Este paradigma se centra en la modularidad y separación de preocupaciones. Permite la encapsulación y reutilización de código relacionado con aspectos transversales, como el registro, la seguridad o el rendimiento, mediante la aplicación de aspectos a través de anotaciones o configuraciones.
2. **Paradigma Orientado a Eventos**: En este paradigma, el flujo de ejecución se basa en la ocurrencia de eventos y la respuesta a ellos. Los programas están compuestos por manipuladores de eventos que se activan cuando ocurre un evento específico. Es comúnmente utilizado en interfaces gráficas de usuario y sistemas distribuidos.
3. **Paradigma Funcional Reactivo**: Se centra en la programación de sistemas y aplicaciones reactivas, donde la interacción con eventos externos y la propagación de cambios son fundamentales. Utiliza flujos de datos inmutables y funciones puras para describir las transformaciones de datos a lo largo del tiempo.