# Resumen Tercera EMT CPLP

Para esta EMT se evalúan los últimos contenidos de la clase 7 junto con los contenidos de las clase 8, 9 y 10 (Sin la parte de Lenguajes Basados en Script)

# Clase 7

## Tipos de Datos Abstractos

### Abstracción

- Es el mecanismo que tenemos las personas para manejar la complejidad.
- Abstraer es representar algo descubriendo sus características esenciales y suprimiendo las que no lo son.
- El principio básico de la abstracción es la **información oculta** (Los programadores no saben cómo está implementado el tipo de dato abstracto internamente)**.**

### TAD

- **TAD = Representación (datos) + Operaciones (funciones y procedimientos)**
- Permiten explorar e incorporar nuevos tipos al lenguaje.
- Los tipos de datos son **abstracciones** y el proceso de construir nuevos tipos se llama **abstracción de datos**.
- Los nuevos tipos de datos definidos por el usuario se llaman **tipos abstractos de datos.**

#### ¿Qué satisface 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 de Información**
    - La representación de los objetos y la implementación del tipo permanecen ocultos.
- **Refleja los niveles de abstracción. Modificabilidad.**

#### Ejemplos de TAD en lenguajes

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

#### Especificación de un TAD

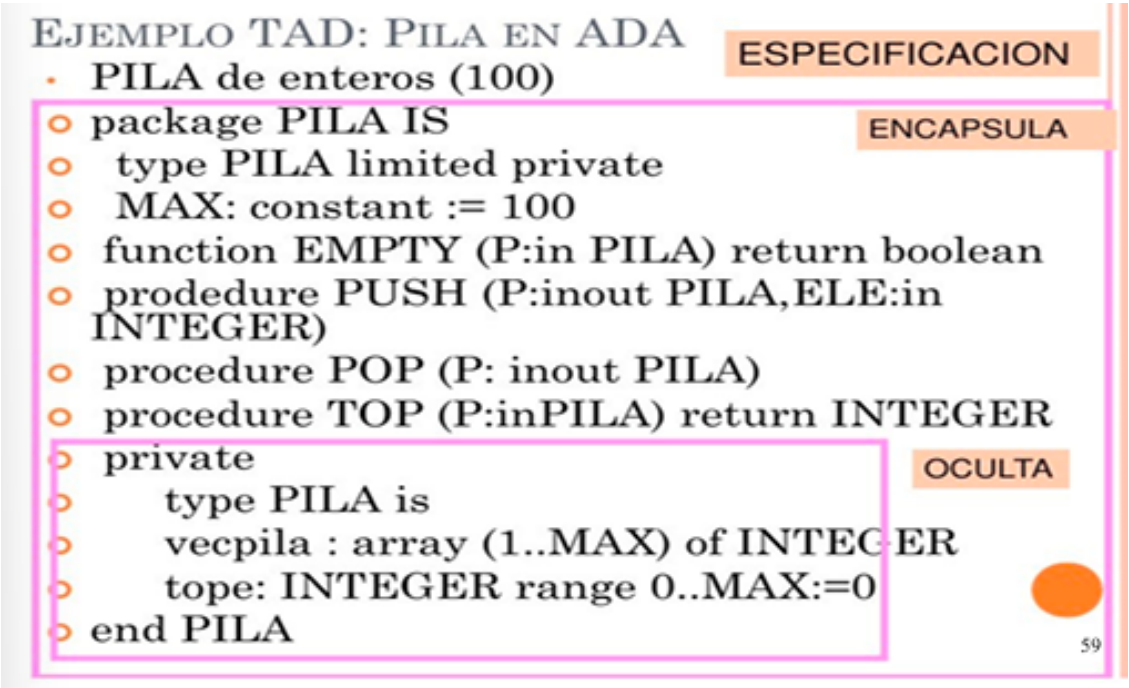
- La **especificación formal** proporciona un conjunto de axiomas que describen el **comportamiento** de todas las **operaciones**. Ha de incluir una parte de **sintaxis** y una parte de **Semántica**:

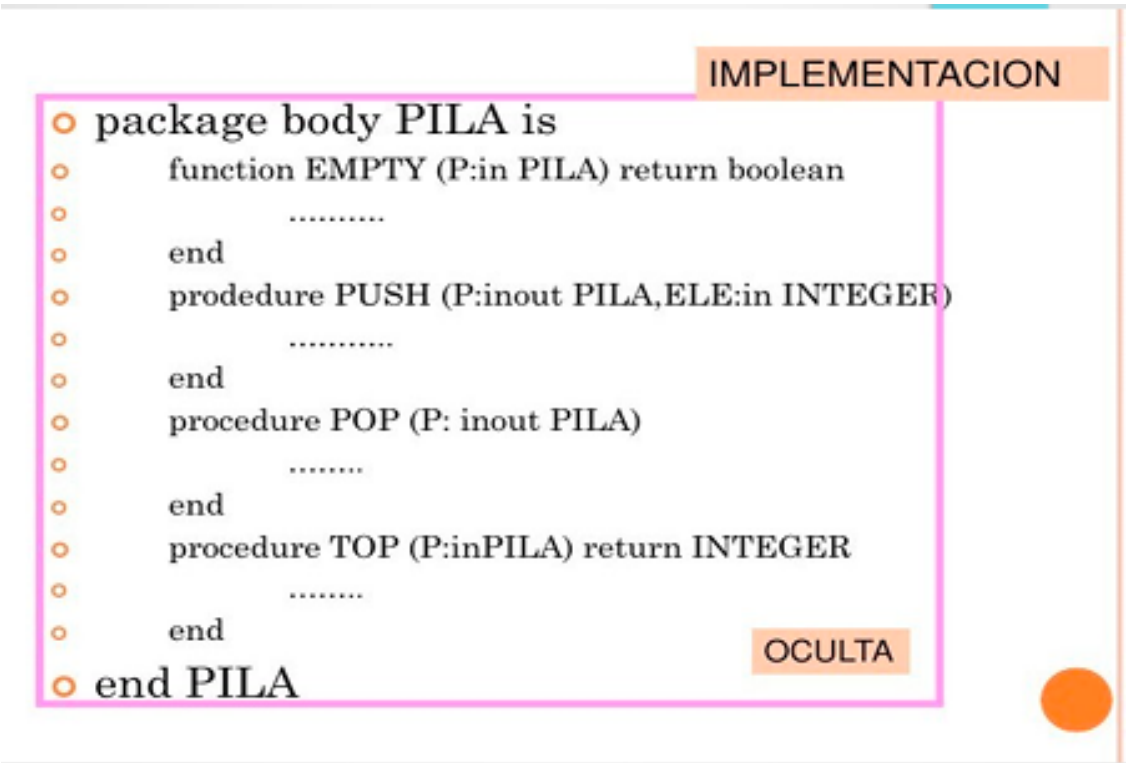
    TAD nombre del tipo (valores que toma los datos del tipo)
    **Sintaxis**
    Operación (Tipo argumento, …) → Tipo resultado
    …

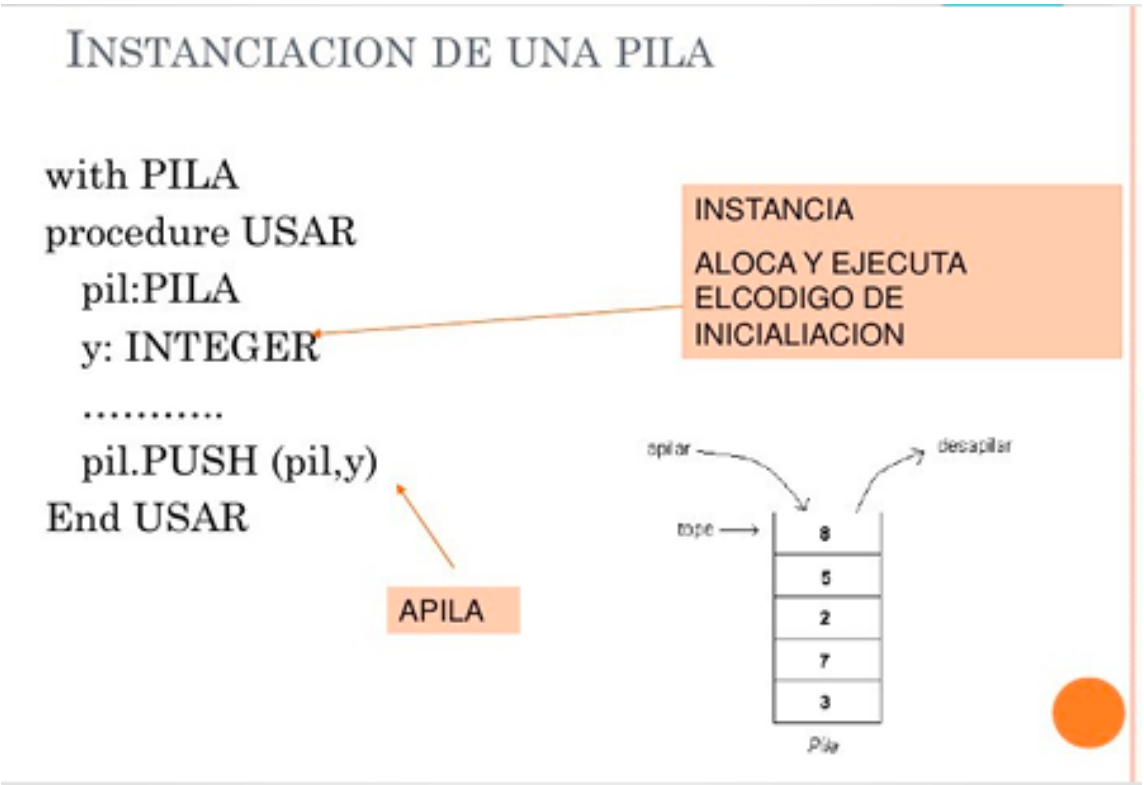
**Semántica**

Operación (Valores particulares argumentos) → Expresión resultado

- Hay operaciones definidas por sí mismas que se consideran **constructores del TAD**. **Normalmente, se elige como constructor la operación que inicializa.**






Clase

- Es un tipo definido por el usuario.
- Contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce. **Atributos + Métodos**
- Agrega un segundo nivel de abstracción que consiste en agrupar las clases en jerarquías de clases. De forma que la clase hereda todas las propiedades de la superclase.

## Sistema de Tipos

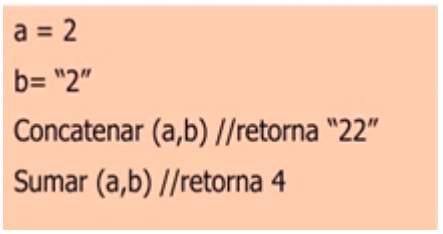
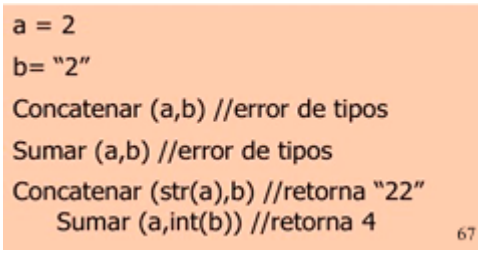
- Conjunto de reglas que usa un lenguaje para estructurar y organizar sus tipos.
- **El objetivo de un sistema de tipos es escribir programas seguros.**
- Conocer estos sistemas nos permite conocer mejor los aspectos semánticos de los diversos lenguajes.

### ¿Qué provee un Sistema de Tipos?

- **Provee mecanismo de expresión:**
  - Expresar tipos predefinidos, definir nuevos tipos y asociarlos con constructores del lenguaje.
- **Define reglas de resolución:**
  - Equivalencia de tipos → ¿Dos valores tienen el mismo tipo?.
  - Compatibilidad de tipos → ¿Puede usarse el tipo en este contexto?.
  - Inferencia de tipos → ¿Cuál tipo se deduce del contexto?

- Mientras más **flexible en el tipado** sea el lenguaje, más **complejo** será su **sistema de tipos.**

## Flexibilidad vs Seguridad

- **Tipado Fuerte - Tipado Débil.**
  - Se dice que el **sistema de tipos** es **fuerte** cuando especifica **restricciones** de forma clara sobre cómo las **operaciones** que involucran valores de **diferentes tipos** pueden **operarse**. Lo contrario establece un **sistema débil de tipos**.
  - Un **sistema débil** es **menos seguro** pero ofrece una **mayor flexibilidad**.
    - 
      
  - Un **sistema fuerte** es **más seguro** pero ofrece una **menor flexibilidad**. El **compilador** asegura la detección de todos los errores de tipos y la ausencia de estos en los programas.
    - 
      
- **Ejemplos en algunos lenguajes**
  - **Python** es Fuertemente Tipado y tiene tipado dinámico.
  - **C** es débilmente tipado y tiene tipado estático.
  - **GOBSTONE** es Fuertemente Tipado y tiene tipado estático.

## Especificación del Sistema de Tipos de un Lenguaje

### Tipo y Tiempo de Chequeo

- **Tipos de Ligadura**
  - **Tipado Estático**: Ligaduras **en compilación**. Puede exigir lo siguiente
    - 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 **tiempo de ejecución**. Que las ligaduras se den en ejecución no vuelve a este tipado un tipado **inseguro**.
  - **Tipado Seguro**: No es **estático**, ni **inseguro**.

### Reglas de Equivalencia y Conversión

- **Tipo Compatible**

    - Reglas semánticas que determinan si el tipo de un objeto es válido en un contexto particular. Por ejemplo, ¿Es compatible la suma entre números enteros y reales?.
    - Un lenguaje debe definir en qué contexto un tipo Q es compatible con un tipo T.
    - Hay que preguntarse si el sistema de tipos define esta compatibilidad.
        - Si no lo hace, esta responsabilidad recae en el compilador que se use.
        - Si lo hace, el sistema es fuerte y el compilador se adapta al mismo.
- **Equivalencia por Nombre**
    - Dos variables son del mismo tipo si y sólo si están declaradas juntas o si están declaradas con el mismo nombre de tipo.
- **Equivalencia por Estructura**
    - Dos variables son del mismo tipo si los componentes de su tipo son iguales.

```
type  t = array [1..100] of integer
var   x,y : array [1..100] of integer
z: array [1..100] of integer
w:t
v:t
```

w, x, y, z,v son del mismo tipo??

- **Llegamos a la conclusión que un tipo es compatible con otro si es equivalente y se puede convertir.**
- **Coerción**
    - Significa convertir un valor de un tipo a otro. Las reglas del lenguajes tienen que estar de acuerdo al tipo de los operandos y a la jerarquía de los mismos.
- **Estrategias para tratar con la Equivalencia y la Conversión de los tipos**
    - **Widening (Ensanchar)**
        - Cada **valor** del **dominio tiene** su correspondiente **valor** en el **rango**, por ejemplo, pasar de entero de a real.
        - Pascal solo tiene Widening de entero a real.
    - **Narrowing (Estrechar)**
        - Cada **valor** del **dominio puede no tener** su correspondiente **valor** en el **rango**, en tal caso, algunos lenguajes producen un mensaje avisando la pérdida de información, por ejemplo, pasar de real a entero.
        - En C esto depende del contexto y se utiliza un sistema de coerción simple.
    - **Cláusula de Casting**
        - Conversiones explícitas, se fuerza a que se convierta.

## Reglas de Inferencia de Tipos

- La **Inferencia de tipos** permite que el tipo de una entidad declarada se “infiera” en lugar de ser declarado. Esta inferencia se puede realizar de acuerdo al tipo de:
    - **Un operador predefinido**
        - **fun f1(n,m) = (n mod m=0)** → La operación “mod” toma 2 valores necesariamente enteros, por lo tanto podemos inferir que “n” y “m” son enteros.

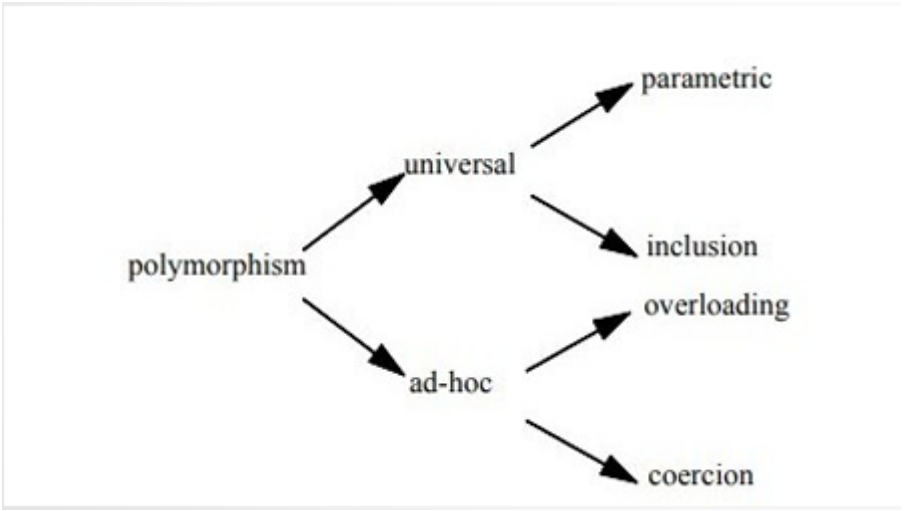
- **Un operando**
  - **fun f2(n) = (n*2)** → Al ser “2” un número podemos inferir que “n” es un número.
- **Un argumento**
  - **fun f3(n:int) = n*n** → Al ser “n” un argumento de tipo entero podemos inferir que lo que devuelve la función es de tipo entero.
- **El tipo del resultado**
  - **fun f4(n):int = (n*n)** → Al ser el resultado que devuelve la función de tipo entero, podemos inferir que “n” es de tipo entero.

## Nivel de Polimorfismo de un Lenguaje

write(e,f(x)+1)

fun disjuntos(s1,s2:Conjunto):boolean;

• La función disjuntos deberá implementarse para cada tipo particular de conjunto?
•Las funciones read y write son “polimórficas”, pero no de forma pura (ya que el compilador infiere el tipo)

- **Lenguaje Mono-Mórfico**
  - Un lenguaje es de este tipo si cada entidad se liga a un único tipo (estáticos).
  - En un lenguaje de este tipo la función *disjuntos* deberá implementarse para cada tipo de conjunto.
- **Lenguaje Polimórfico**
  - Un lenguajes es de este tipo si las entidades pueden estar ligadas a más de un tipo.
  - Las variables de un lenguaje de este tipo pueden tomar valores de diferentes tipos.
  - Las operaciones polimórficas son funciones que aceptan operandos de varios tipos.
  - Los tipos polimórficos tienen operaciones polimórficas.
- **Todo lenguaje tiene cierto grado y profundidad de polimorfismo, estos son**
  - 

- **Polimorfismo Ad-Hoc**
    - Permite que una función se aplique a distintos tipos con un comportamiento sustancialmente diferente en cada caso.
    - **Sobrecarga**
        - Se utiliza para referirse a conjuntos de abstracciones diferentes que están ligadas al mismo símbolo o identificador. Por ejemplo, **“a” + “b” = “ab”** pero **2 + 3 = 5**. Podemos decir que el identificador **“+”** tiene una sobrecarga. **“El mismo operador (+) no trabaja con tipos diferentes, sino que son operadores distintos cuya Sintaxis es la misma”.**
    - **Coerción**
        - Permite que un operador que espera un operando de un determinado tipo T pueda aplicarse de manera segura sobre un operando de un tipo diferente al esperado.
- **Polimorfismo Universal**
    - Permite que una única operación, método o entidad se aplique uniformemente sobre un conjunto de tipos relacionados.
    - Si este **polimorfismo** está dado a través de **parámetros**, hablamos de **polimorfismo paramétrico.**
        - Un tipo parametrizado es un tipo que tiene otros tipos como parámetros
          lista(T) = T*
    - El **polimorfismo por inclusión** es otra forma de **polimorfismo universal** que permite modelar **subtipos** y **herencia**.
        - Si un **tipo** se define como un **conjunto de valores y un conjunto de operaciones**. Un **subtipo** T’ de T puede definirse como un **subconjunto de los valores de T y el mismo conjunto de operaciones.**
        - ```
          subtype TDíaDelMes is Integer range 1..31;
          subtype TDíaFebrero is TDíaDelMes range 1..29;
          subtype TLaborable is TDíaDeSemana range Lunes..Viernes;
          ```
        - El mecanismo de **herencia** permite definir una **nueva clase derivada a partir de una clase base ya existente**. Podría agregar **atributos** y **comportamiento**.

# Clase 8

## Estructuras de Control

- **Son el medio por el cual los programadores pueden determinar el flujo de ejecución entre los componentes de un programa.**
- Están divididas en 2 niveles, **Estructuras de Control a nivel de Sentencia** y a **nivel de Unidad.**
          ```

# Estructuras de Control a Nivel de Unidad

- Cuando el **flujo de control** se pasa entre **unidades** (rutinas, funciones, procedimientos, etc) también es necesario **estructurar** el **flujo** entre ellas. Las formas de controlar esto son:
    - **Pasajes de Parámetros.**
    - Call-Return.
    - Excepciones.
    - Etc.

# Estructuras de Control a Nivel de Sentencia

- Estas se dividen en tres grupos: **Secuencia, Selección e Iteración.**

## Secuencia

- Estructuras de control que **permiten ejecutar una serie de instrucciones en un orden específico**, de arriba hacia abajo, sin ningún tipo de desviación.
- Es el **flujo** de control **más simple**.
- Se basa en la ejecución de **una sentencia a continuación de otra.**
- El **delimitador** más **general** y **usado** como delimitador de **sentencia** es el **";"**
- Existen **lenguajes** que:
    - **No tienen o no usan delimitador** y definen que por línea sólo debe haber **1 instrucción**. A estos se los llama **orientados a línea (Fortran, Basic, Ruby, Python, etc).**
    - Por línea puede haber **más de una instrucción**
    - Permiten estructurar con **Sentencias Compuestas**, es decir, se puede **agrupar varias sentencias** en una con el uso de **delimitadores**:
        - **Begin y End en Ada, Pascal, etc.**
        - **{ } en C, C++, Java, etc.**

### Sentencia de Asignación

- Sentencia que **produce cambios** en los **datos de la memoria**. Ejemplo **"x = a + b".**
- **En general, asigna al l-valor de un objeto dato "x" el r-valor de una expresión "a + b".**
- Sintaxis en diferentes lenguajes
    - 

| | |
|---|---|
| **A := B** | Ej: Pascal, Ada, etc. |
| **A = B** | Ej: Fortran, C, Prolog, Python, Ruby, etc. |
| **MOVE B TO A** | COBOL |
| **A← B** | APL |
| **(SETQ A B)** | LISP |

- **Distinción entre sentencia de asignación y expresión**
    - La **sentencia de asignación devuelve el valor** de la expresión y **modifica el valor** de la posición de **memoria.**

- La **mayoría de los lenguajes** requieren que sobre el **lado izquierdo de la asignación aparezca un l-valor y no un r-valor.**
- En otros lenguajes, **como en C**, se define la **sentencia de asignación como una expresión con efectos colaterales.**
  - Las sentencias de asignación **devuelven valores**.
  - En C **se evalúa la asignación de derecha a izquierda** (a=b=c=0 → 0 se asigna a "c", "c" a "b" y "b" a "a").
  - Si se usa **asignaciones en condiciones primero asigna y luego evalúa**, además, **toma como verdadero todo resultado distinto que 0.**

## Iteración o Bucles

- Estructuras de control que **permiten repetir un bloque de código múltiples veces hasta que se cumpla una condición de salida (loop)**. Esto permite la ejecución repetida de una serie de instrucciones sin tener que escribir las mismas instrucciones una y otra vez. Ejemplos **for, while, do while, etc.**
- La mayoría de los lenguajes de programación proporcionan **diferentes tipos de construcciones de bucle** para definir la **iteración de acciones (llamado el cuerpo del bucle).**
- **Comúnmente son agrupados como:**
  - **Bucle For**: Bucle en el que se conoce el número de repeticiones al inicio del bucle.
  - **Bucle While**: Bucle en el que el cuerpo se ejecuta repetidamente siempre que se cumpla una condición. Hay 2 tipos.

### Sentencia del tipo For Loop

- **Sentencia Do de Fortran**

```
Do  label var-de-control= valorIni, valorFin
.....
label continue
```

  - La **var-de-control solo** puede tomar **valores enteros** y se **incrementa 1 en 1 en la secuencia** (si no se especifica otra cosa según lo permita el lenguaje).
  - **ValorIni** es el valor inicial.
  - **ValorFin** es el valor final.
  - La **declaración continue** junto con la **etiqueta label** se usa como la **última declaración de un DO** (depende versión de FORTRAN).
  - El **Fortran original evaluaba** si la **variable de control** había **llegado al límite al final del bucle**, entonces **por lo menos** se **ejecutaba una vez**. Esto cambió a partir de Fortran 77.
  - La **Variable de control "nunca" deberá ser modificada por otras sentencias dentro del ciclo**, ya que puede generar errores de lógica.

- Ejemplos:

```
DO 1 I = 1,10
   SUM = SUM A(I)
1  CONTINUE

DO 1 I = 10,1
   SUM = SUM A(I)
1  CONTINUE
```

El **2do no ejecuta** porque **valorIni *es mayor que* valorFin** 42

- **Sentencia For de Pascal, ADA, C y C++**
  - `for loop_ctr_var := lower_bound to upper_bound do statement`
  - La **variable de control** puede tomar **cualquier valor ordinal (enumerativos) que indiquen secuencia**, **no sólo enteros**, podrían ser chars por ejemplo.
  - **Pascal estándar "no permite" que se modifiquen los valores** del **límite inferior**, **límite superior**, ni del valor de la **variable de control**.
  - **El valor de la variable de control fuera del bloque se asume no definida.**

```
1
2 Program HelloWorld(output);
3 Uses sysutils;
4 var i: Integer;
5 Procedure VerI();
6 Begin
7   writeln(Concat('valor de i es ',IntToStr(i)));
8   // es inseguro si tocamos el iterador en  otra unidad.
9   //i:=i+20;
10 end;
11 BEGIN
12   writeln('Hello, world!');
13   for i:=1 to 20 do begin
14     writeln(Concat('valor de i es ',IntToStr(i)));
15     IF (i=5) THEN begin
16       VerI();
17     end
18     ELSE begin
19       //no se permite alterar el iterador en la misma unidad.
20       //i:=i+1;
21     end;
22   end
23 END.
```

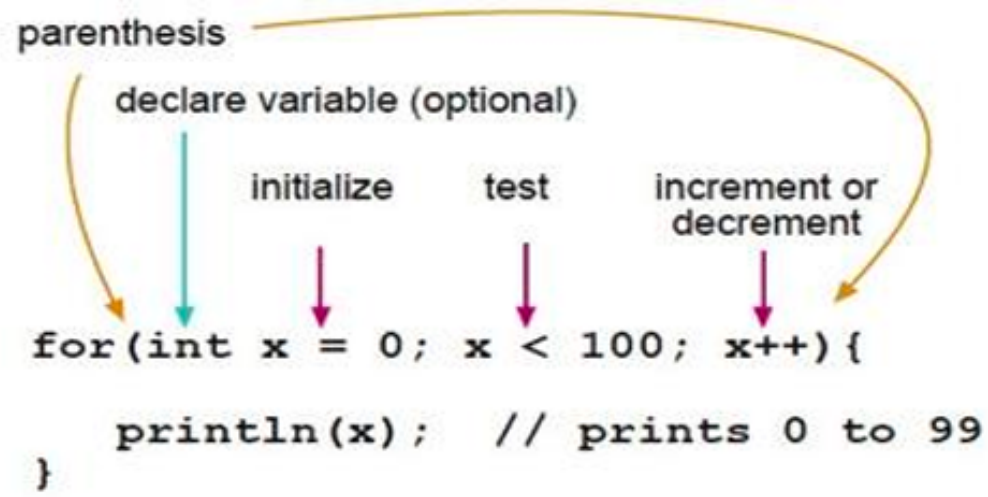
**Salida del programa**
Free Pascal Compiler version 3.2.0
valor de i es 1.....20

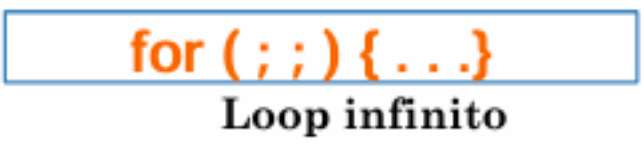
si **descomento i=i+1** da este **error**
Compiling main.p, main.p(22,8)
**Error: Illegal assignment to for-loop variable "i"** main.p(25,4)
Fatal: There were 1 errors compiling module, stopping
**Fatal: Compilation aborted**
Error: /usr/bin/ppcx64 returned an error exitcode **(no permite modificar la variable en el loop)**

si **descomento i=1+20** da
Hello, world!
valor de i es 1 2 3 4 5 5
**Puedo modificar fuera la variable hay efecto colateral**

- **Sentencia For específica para ADA**
  - **Encierra todo** proceso iterativo **entre las cláusulas loop** y **end loop.**
  - Permite el uso de la sentencia **Exit** para **salir del loop**.
  - La **variable de control** (iterador) es de **tipo enumerativa**.
  - La **variable de control NO necesita declararse** (se declara implícitamente al entrar al bucle y desaparece al salir).
  - El **in indica incremento**, permite **decrementar con in reverse**.

- 
```
for i in 1..N loop
V(i) := 0;
end loop
```
```
for i in reverse 1..N loop
V(i) := 0;
end loop
```
45

- **Sentencia For específica para C y C++**
  - Se componen de **3 partes**: **1 inicialización** y **2 expresiones**.
  - **Inicialización**
    - Da el estado inicial para la ejecución del bucle.
  - **1ra expresión**
    - Especifica el test que es realizado antes de cada iteración. Sale del ciclo si la expresión no se cumple (sale del rango, i.e false)
  - **2da expresión**
    - Específica el incremento que se realiza después de cada iteración.
  - 


- **Específicamente para C++**
  - En **C++** se puede realizar **la declaración** de una **variable** dentro del **for**, por ejemplo
    - **for (int p=0,int i=1; i<n; i++)**
      **{**
      **p+= a * b;**
      **b = p * 8;**
      **}**
  - El **alcance** de **dicha variable** se **extiende** hasta el **final del bloque** que encierra la instrucción **for {….}.**
  - Si se **omiten una o ambas expresiones** en un bucle for se puede **crear un bucle sin fin**, del que **solo se puede salir con una instrucción break, goto o return**.
    - 

Loop infinito

- **Sentencia For de Python**

- **Existen 2 tipos de estructuras For en Python:**
  - Para **iterar** sobre una **secuencia de estructuras de datos** de tipo: **lista, tupla, conjunto, diccionario, etc.**
  - Para **iterar** sobre un **rango de valores basado en una secuencia numérica**: usando la **función Range().**
- **For para iterar en una secuencia de estructuras**
  - 
```
for <elem> in <iterable>:
<código>
```
  - **elem** variable que toma el **valor del elemento dentro/in del** iterador **iterable** en **cada paso** del **bucle**.
  - **En cada paso de la iteración se tiene en cuenta a un único elemento del objeto iterable**, sobre el cuál se pueden aplicar una serie de operaciones.
  - **Finaliza su ejecución cuando se recorren todos los elementos.**
- **For para iterar sobre un rango de valores**
  - **Uso de la función/clase range(max)** que **devuelve valores desde 0 hasta max-1.**
  - El For actúa como los demás lenguajes.
  - 
```
1. for i in range(11):
2.     print(i)
```
  - El tipo de datos `range` se puede invocar con uno, dos e incluso tres parámetros:
    - `range(max)`: Un iterable de números enteros consecutivos que empieza en `0` y acaba en `max - 1`
    - `range(min, max)`: Un iterable de números enteros consecutivos que empieza en `min` y acaba en `max - 1`
    - `range(min, max, step)`: Un iterable de números enteros consecutivos que empieza en `min` acaba en `max - 1` y los valores se van incrementando de `step` en `step`. Este último caso simula el bucle for con variable de control.

Sentencia del tipo While Loop (Chequeo al Inicio)

- El **while** es una **estructura** que permite **repetir un proceso mientras se cumpla una condición**.
- La **condición** se **evalúa antes de que se entre al proceso**.

| | |
|---|---|
| PASCAL | while condición do sentencia<br>begin<br>....<br>end |
| C, C++ | while (condición) sentencia;<br>{....} |
| ADA | while condición sentencia<br>....<br>end loop; |
| PYTHON | while condición :<br>Sentencia 1<br>Sentencia 2<br>.....<br>sentencia n |

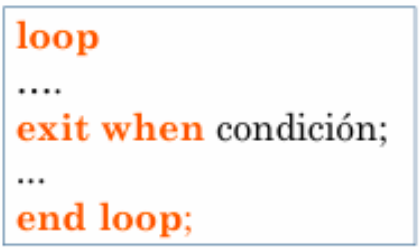
- **break o exit** se pueden usar para **salir del bloque** según el lenguaje.
- **While de Python**
  - 


Sentencia del tipo While Loop (Chequeo al Final)

- El **Until-Repeat y el Do-While** son estructuras que permiten **repetir un proceso "hasta"que se cumpla una condición.** En definitiva, **permite ejecutar un bloque de instrucciones mientras no se cumpla una condición dada (o sea falso).**
- La **condición/expresión se evalúa al final del proceso, por lo que por lo menos 1 vez el proceso se realiza.**

| | |
|---|---|
| PASCAL | repeat<br>Sentencia<br>.....<br>until condición; |
| C, C++ | do sentencia;<br>.....<br>while (condición); |

- **Específicamente en ADA**
  - **Se usa una estructura iterativa.**




```
loop
   Alimentar_Caldera;
   Monitorizar_Sensor;
   exit when Temperatura_Ideal;
end loop;
```

  - Del **bucle** se **sale** normalmente mediante una **sentencia "exit when condition".** O con una **alternativa** que **contenga una cláusula "exit".**

## Selección o Decisiones

- Estructuras de control que **permiten tomar decisiones basadas en ciertas condiciones.** Estas estructuras dirigen el flujo del programa hacia diferentes caminos según el resultado de la evaluación de las condiciones especificadas. Ejemplos: **if-else, switch-case** (o **select-case** en algunos lenguajes), **etc.**

### Sentencia If

- Estructura de control que **permite expresar una elección entre un cierto número posible de sentencias alternativas** (ejecución condicional)
- Entre lenguajes la **semántica** es **similar**, la **sintaxis** es la que más **varía**.
- **Evolución de la sentencia**
  - **If lógico de Fortran** → if (condicion) sentencia
    - Si la **condición** es **verdadera ejecuta** la **sentencia**.
    - **No había else.**
  - **If then else de ALGOL** → if (condicion) then sentencia1 else sentencia2.
    - **Permite 2 caminos posibles.**
    - **Problema de Ambigüedad:** El lenguaje **no establecía cómo se asociaban los else con los if abiertos**, por lo tanto si había más de un if y un solo else, no se podía determinar a cuál if estaba asociado.
      - **Solución de Ambigüedad:** if then else de P1/1, Pascal y C
        - Como **REGLA** cada rama else se empareja con la instrucción if solitaria más próxima, es decir, **cada else se empareja para cerrar al último if abierto.**
        - **Elimina Ambigüedad** pero las **instrucciones anidadas** pueden ser **difíciles** de **leer**, más si el programa se escribió **sin respetar sangrías**.

- **Solución Sintáctica para dar claridad**: Usar **sentencia de cierre del bloque condicional if**. Por ejemplo: fi en Algol 68, end if en Ada o end en Modula-2.
    - **Desventaja**: Programas con muchos **if anidados** pueden ser **ilegibles**, **ambiguos** y **difíciles** de **mantener**.
- **Solución** → Hacer uso de **sentencias compuestas begin** y **end** (estos identificadores pueden variar según el lenguaje).

- **if then else en C**
    - **Sentencia en línea**

```
If (condición)  Instrucción 1;
else Instrucción A;
```

- Usa ()
- No usa then

    - **Bloque de Instrucciones**

```
if (condición) {
        Instrucción 1;
        Instrucción 2;
        .
        .
        .
        Instrucción n;
}
else {
        Instrucción A;
        Instrucción B;
        .
        .
        .
        Instrucción Z;
}
```

- Usa {}
- No usa begin y end

    - **Recomendable el uso de llaves** para un **código** más **legible** y **mantenible** que deje bien clara **la intención del programador**.
    - C no lleva palabra clave then.
- **if corto en C**

Se puede representar con expresión condicional:

**cond ? b : c;**

(i<j?z:y)

- ? *operador condicional* (operador ternario)
- cond es una *expresión booleana*
- b y c pueden ser *expresiones o sentencias.*
- Si el valor de cond es **verdadero** se **devuelve el valor de b**
- Si el valor de cond es **falso** se **devuelve el valor de c**

- **if condición then …. else de Python**
    - **Sin ambigüedad y legible**.
    - Incorpora **elif** si hay más de 2 opciones y **sangría**.
    - el símbolo **":"** es **obligatorio al final** del **if, else y del elif.**
    - La **indentación** es **obligatoria** al colocar las sentencias correspondientes tanto al **if, elif y del else**.
    - El uso de **( )** en la **condición** es **opcional**.
- **[on_true] if [expresión] else [on_false] de Python**
    - Se conoce como **operador ternario o condicional.**

- Es la construcción **equivalente al if corto de C.**

```
>>> altura = 1.79
>>> estatura = "Alto" if altura > 1.65 else "Bajo"
>>> estatura
'Alto'
>>>
```

- **Python evalúa condiciones con circuito corto y de izquierda a derecha.**

### Circuito Corto

- En un lenguaje que utiliza el 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 (“y” /”and” / “&&”)**
  - Da como resultado verdadero únicamente cuando ambos términos son verdaderos. Si el primer término es falso, no es necesario evaluar el segundo ya que el resultado será falso.
- **Operador (“o” / “or” / “||”)**
  - Da como resultado falso únicamente cuando ambos términos son falsos. Si el primer término es verdadero, no es necesario evaluar el segundo ya que el resultado será verdadero.
- **Permite evitar errores y optimizar el rendimiento.**
- Permite **evitar errores** al evaluar **expresiones indefinidas**, es decir, si existe una condición “A and B” en la que “B” es nulo, se puede dar un error al evaluarla. Entonces podemos hacer que la expresión “A” garantice que “B” no sea nulo.

### 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 (“y” /”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 (“o” / “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.

### Selección Múltiple

- La sentencia **Select** de PL/1 (Programming Language 1) es la que introduce la selección entre dos o más opciones de forma que se reemplacen if anidados muy largos.
- **Selección Múltiple en Pascal**

- Usa palabra reservada **case** seguida de una **variable de tipo ordinal o una expresión** y la palabra reservada **of**.
- La **variable-expresión** a evaluar es llamada **"selector".**
- **Lista las sentencias** de acuerdo con diferentes valores que puede adoptar la variable **(los "casos"). Llevan etiquetas.**
- **No importa el orden en que aparecen los casos.**
- Puede existir un **bloque else** para el caso que la **variable** adopte un **valor** que **no coincida** con ninguna de las sentencias de la **lista de casos**.
- Para finalizar se coloca un **"end;"** (no se corresponde con ningún **"begin"** que exista).
- Es **inseguro** si no se establece un **bloque else** y sucede que la **variable** toma un **valor** que **no está dentro de la lista de casos**.
- El formato es el siguiente:

```
case variable_ordinal of
    valor1: sentencia 1;
    valor2: sentencia2;
    valor3: sentencia3;
else
    sentencia4;
end;
```

**Ordinal:** puede obtenerse un predecesor y un sucesor (a excepción del primer y el último (expresa la idea de orden o sucesión)

- **Selección Múltiple en Ada**
  - Las **expresiones** pueden ser **solamente** de **tipo entero o enumerativas.**
  - En **las selecciones** del **case** se **deben estipular "todos" los valores posibles** que puede tomar la expresión.
  - El **when** se acompaña con **=>** para indicar **la acción a ejecutar si se cumple la condición.**
  - Tiene la cláusula **Others** que se puede utilizar para **representar a aquellos valores que no se especificaron explícitamente.**
  - **Others "debe" ser la última opción antes del end;**
  - Después que una **rama** es **ejecutada** el **Case entero finaliza**. (no pasa por otras ramas)
  - **No pasa la compilación si**
    - **NO** se coloca la rama para un posible valor y/o **NO** aparece la opción Others en esos casos.

**Ejemplo 1**

```
case Operador is
        when  '+' => result:= a + b;
        when  '-' => result:= a - b;
        when  others => result:= a * b;
end case;
```

**Importante para el programador:**

La cláusula **others** se **debe colocar** porque las etiquetas de las ramas NO abarcan todos los posibles valores de Operador

**Debe ser la última**

**Ejemplo 2**

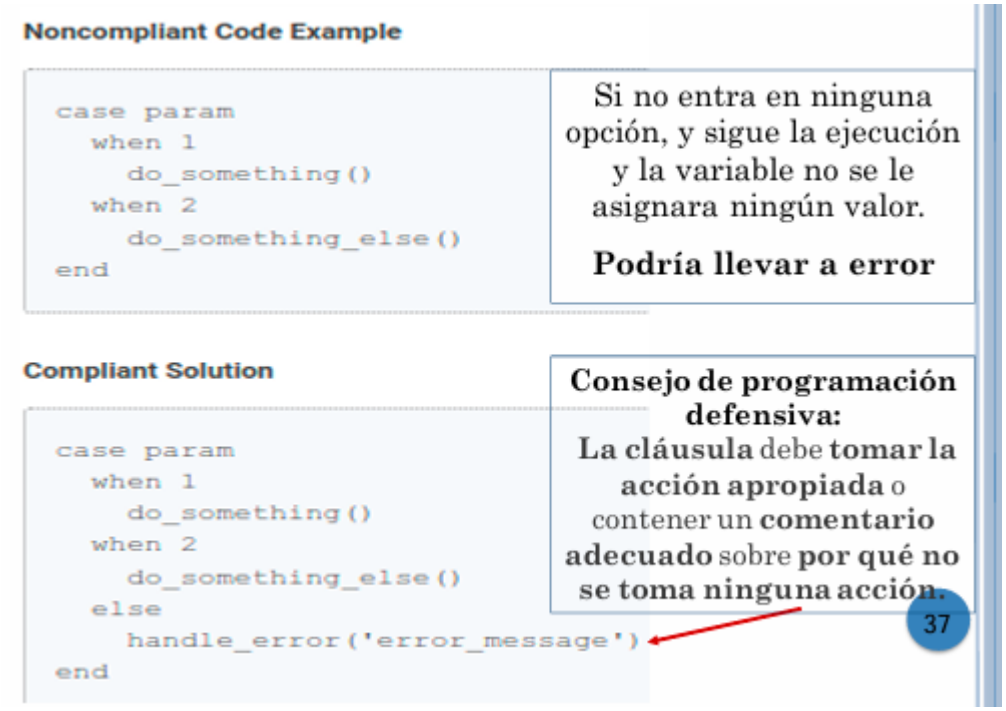
```
case Hoy is
  when MIE..VIE => Entrenar_duro; -- Se puede especificar Rango con ..
  when MAR | SAB => Entrenar_poco; -- Se puede especificar varias elecciones |
  when DOM => Competir; -- Única elección.
  when others => Descansar; -- Debe ser única y la última alternativa. (LUN)
end case;
```

- **Selección Múltiple en C o C++**
  - Constructor **Switch** seguido de **(expresión).**
  - Cada **rama Case** es "**etiquetada**" por uno o más **valores constantes (enteros o char).**
  - 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**).
  - Existe la sentencia **break**, que provoca la **salida de cada rama** (sino continúa)
  - 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**.
  - **El orden en que aparecen las ramas no tiene importancia.**

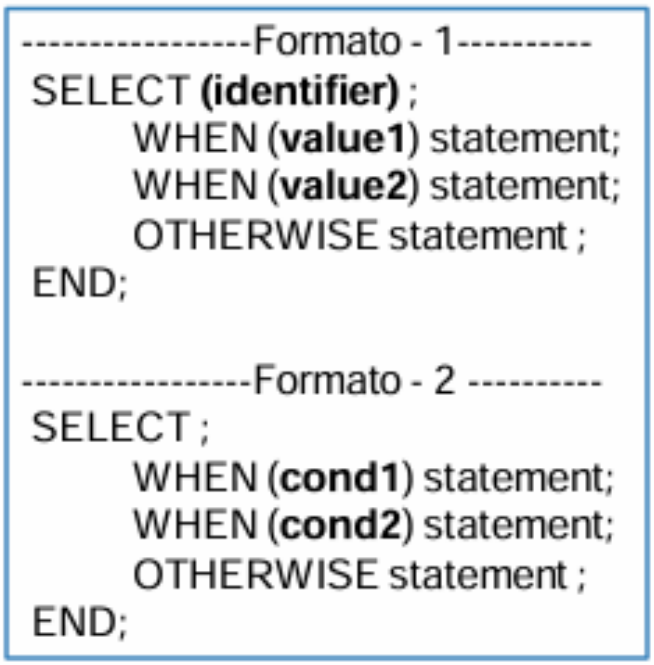
```
Switch (Operador) {
      case  '+' :
            result:= a + b; break;
      case  '-':
            result:= a - b; break;
      default : //Opcional
            result:= a * b;
      }
```

Debe ponerse la sentencia **break** para saltar las siguientes ramas, **si no pasa por todas**

```
switch( i )
{
   case -1:
      n++;
      break;
   case 0 :
      z++;
      break;
   case 1 :
      p++;
      break;
}
```

- **Selección Múltiple en Ruby**
  - Constructor **case expresión, seguido de when y end**
  - **La expresión es cualquier valor** (cualquier cadena, numérico o expresión que proporcione algunos resultados como ("a", 1 == 1, etc.).
  - Dentro de los **bloques when seguirá buscando la expresión, hasta que coincida con la condición, ingresará en ese bloque de código.**
  - **Si no coincide con ninguna expresión irá al bloque else igual que por defecto.**
  - El **else** es **opcional**, esto puede traer **efectos colaterales.**
  - 

- **Selección Múltiple en PL/1**
  - Sentencia **SELECT** de PL/1 incorpora el uso de **WHEN / OTHERWISE / END**
  - **Tiene 2 tipos de formatos: Identificador o Condición**

- 


# Clase 9

## Excepciones

- **Condición inesperada o inusual, que ocurre durante la ejecución del programa y no puede ser manejada en el contexto local.**
- **Denota** un **comportamiento anómalo e indeseable** que es **necesario controlar**.
- La excepción **interrumpe el flujo normal de ejecución** y ejecuta un **controlador de excepciones** registrado previamente.
- Para que un **lenguaje trate excepciones** debe **proveer mínimamente**
    - Un modo de **definirlas**.
    - Una forma de **reconocerlas**.
    - Una forma de **lanzarlas y capturarlas**.
    - Una forma de **manejarlas** especificando el **código y las respuestas**.
    - Un criterio de **continuación**.

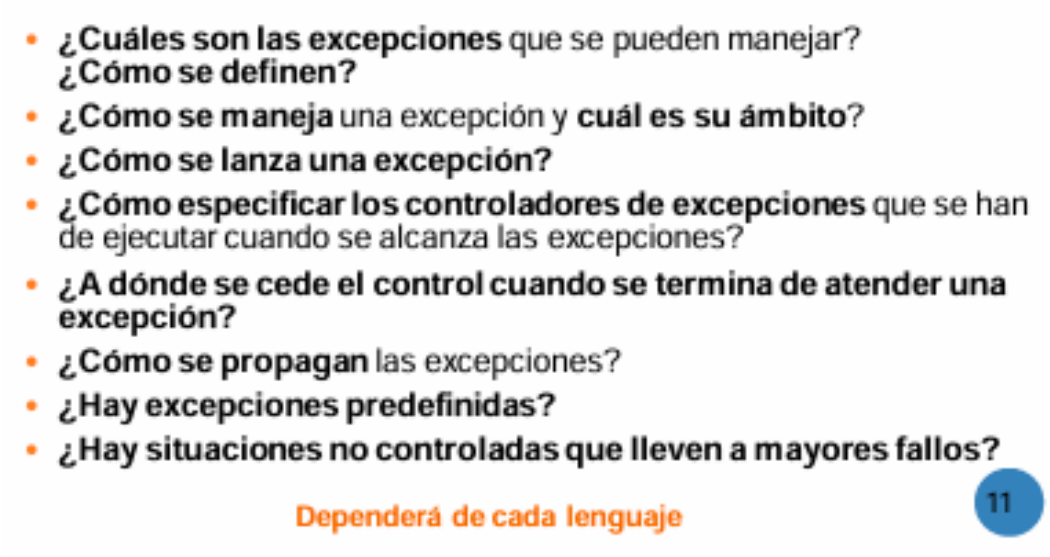
### Controlador de Excepciones

- **Encargado de manejar la excepción.**
- Puede **tomar distintas acciones** según la **situación**
    - Imprimir un mensaje de error.
    - Realizar acciones correctivas.
    - Lanzar otra excepción.
    - Finalizar la ejecución del programa.
- **Debe tomar la solución menos perjudicial.**

## Tipos de Excepciones

- **Implícitas**
  - Definidas por el lenguaje (built-in).
- **Explícitas**
  - Definidas por el programador.

## ¿Qué se debe tener en cuenta de un lenguaje que provee manejo de excepciones?




## ¿Qué instrucciones deben proveer los lenguajes?

- Los lenguajes deben proveer instrucciones para
  - **Definición** de la **excepción**.
  - **Levantamiento** de una **excepción**.
  - **Manejador** de la **excepción**.

## Punto de Retorno

- **Después de atender** a una **excepción**, el **punto de retorno dependerá** del **flujo de ejecución** del programa y de **cómo se haya diseñado el manejo de excepciones** en el código. También va a depender **del lenguaje**.

## Cómo continuar después de una Excepción

### Continuar la ejecución normal del programa

- **Si después de manejar una excepción** el programa **puede continuar la ejecución** del código restante sin problemas.
- El **punto de retorno** será definido por el **lenguaje** (por ejemplo, **el siguiente bloque de código después del bloque de manejo de excepciones o siguiente instrucción**)

### Retornar a un estado anterior

- Cuando el **manejo de excepciones** puede requerir que el programa **regrese a un estado anterior o deshaga acciones realizadas antes de que se produjera la excepción.**

### Propagar la excepción

- En el c**ontrolador de excepciones no puede manejar completamente la excepción, puede optar por propagarla a un nivel superior en la jerarquía de llamadas.**
- El **punto de retorno** sería el **controlador de excepciones en el nivel superior** que pueda manejar la excepción o decidir cómo manejarla.

### Terminar la ejecución del programa

- En **situaciones excepcionales o críticas**, es posible que el **controlador de excepciones determine que no se puede continuar ejecutando el programa de manera segura.**
- El **punto de retorno** puede ser la **finalización del programa o alguna acción específica de cierre antes de la finalización.**

## Modelos de Manejo de Excepciones

### Reasunción

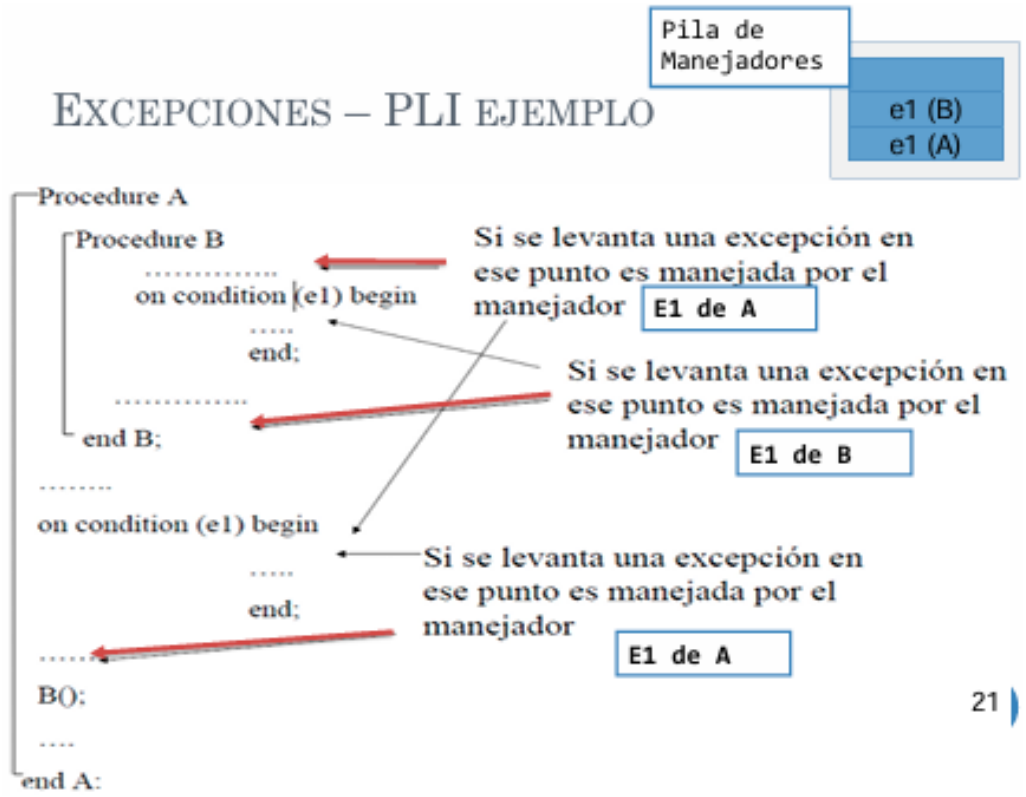
- Hace referencia a la **posibilidad de retomar la ejecución normal del programa después de manejar una excepción.**
- El **controlador de excepciones** realiza las acciones necesarias para manejar la excepción (medidas correctivas) y luego **el programa continúa su ejecución a partir del punto donde se produjo la excepción.**
- **Maneja la excepción y continua la ejecución.**
- **Lenguajes que usan este modelo**
    - PL/1.

### Terminación

- El **controlador de excepciones** realiza las acciones necesarias para manejar la excepción, pero no se retorna al punto donde se produjo la excepción (invocador), **continúa su ejecución a partir de la finalización del manejador.**
- **Terminar la ejecución y manejar la excepción.**
- Es el modelo **más utilizado actualmente**.
- **Lenguajes que usan este modelo**
    - ADA.
    - CLU.
    - C++.
    - Java.
    - Python.
    - PHP.

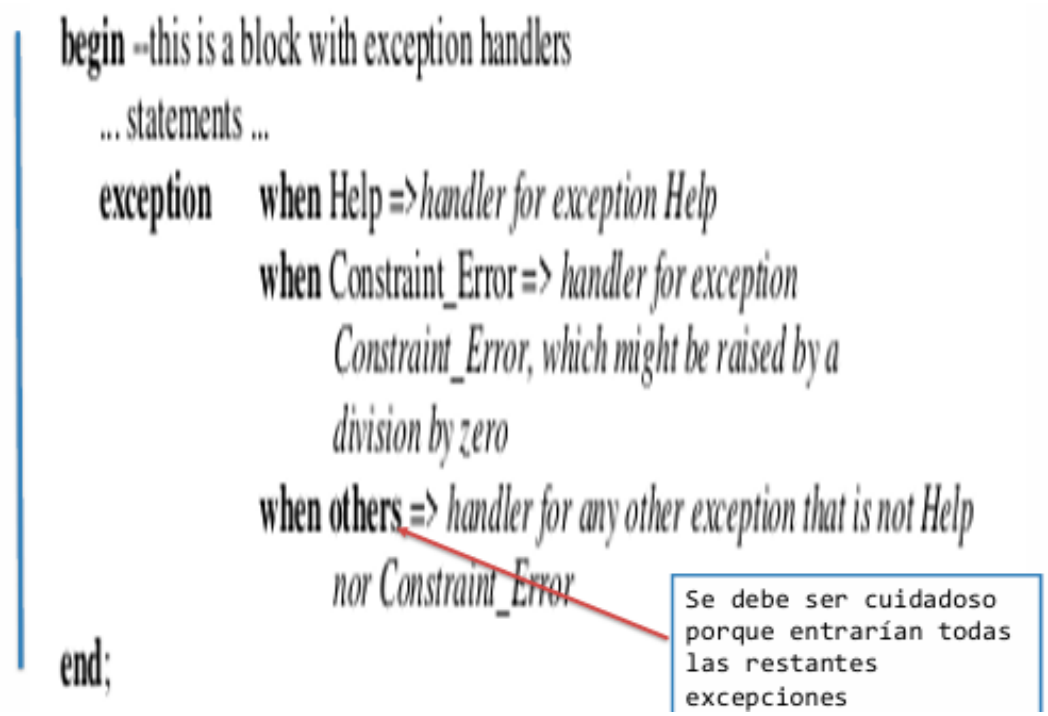
## Excepciones en PL/1

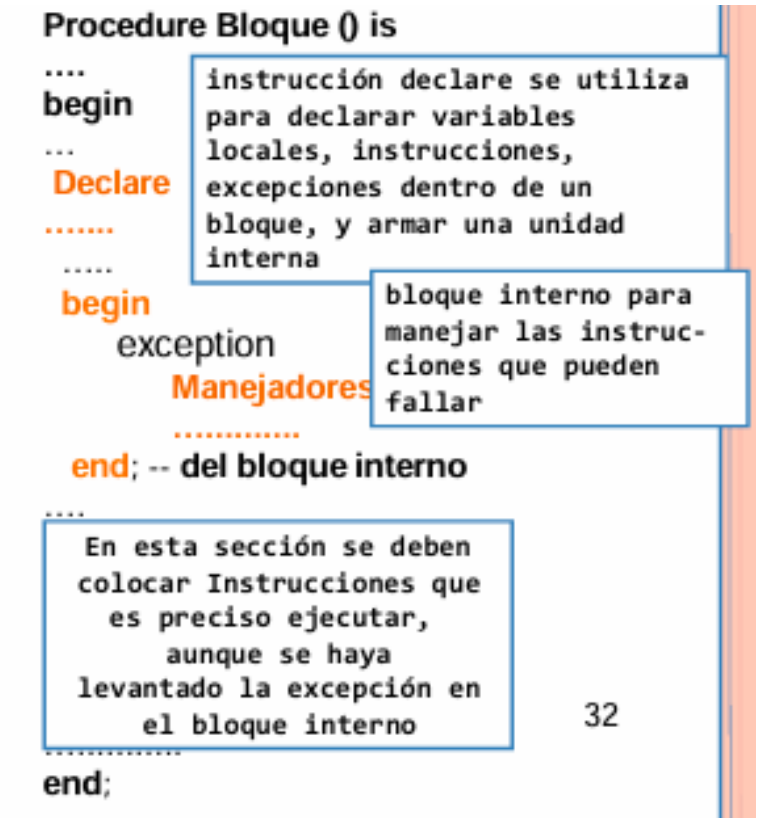
- Fue el primer lenguaje que incorporó el manejo de excepciones..
- Utiliza el criterio de **Reasunción**.
- En este lenguaje las **excepciones** se llaman **CONDITIONS**.
- Los **Manejadores** se **declaran** con la sentencia **ON**
  - ON CONDITION(Nombre-excepción) Manejador
- El **Manejador** puede ser una **instrucción** o un **bloque** (entre **begin** y **end**).
- Las **excepciones se lanzan explícitamente** con la palabra clave **SIGNAL**
  - SIGNAL CONDITION(Nombre-excepción)
- **Ejemplo**
  - ```
    Prog Main
        PROC UNO
        Begin
        ...
        ON CONDITION PEPE begin ... end;     ***DECLARACION (ON....)
                                                 y MANEJADOR3 (BEGIN...END)
        ..
        If (condError) then
              SIGNAL CONDITION PEPE          *** INVOCACION (SIGNAL....)
        end
        ....
    Begin
    ...
    ON CONDITION PIPO begin ... end;         ***DECLARACION y MANEJADOR1
    ..
    ON CONDITION PEPE begin ... end;         ***DECLARACION y MANEJADOR2
    ...
    UNO;
    ...
    If (condError) then
              SIGNAL CONDITION PIPO          *** INVOCACION (SIGNAL....)
    end
    ```
- Este lenguaje tiene una serie de **excepciones ya predefinidas con su manejador asociado.** Son las **Built-in exceptions**. Por ejemplo **zerodivide**.
- **Los manejadores se ligan dinámicamente con las excepciones**. Una **excepción siempre estará ligada con el último manejador definido.** (Manejo de pila de manejadores de excepciones).
- **El alcance de un manejador termina cuando finaliza la ejecución de la unidad donde fue declarado.**
    ```

- 


## Excepciones en ADA

- Aplica criterio de **Terminación**.
- Las **excepciones se definen/declaran** en la **zona de definición de variables** y tienen el **mismo alcance en la unidad.**
- **Su formato para declararlas**
  - `MiExcepcion: exception;`
- La **lista de controladores de excepciones** lleva el prefijo de la **palabra clave exception.**
- **Cada controlador** lleva el prefijo de la **palabra clave when** (con un formato específico), seguido de las **acciones**.
- Se puede utilizar **when others** para capturar **cualquier excepción no especificada**
  - Debe colocarse al **final** del **bloque** de manejo de excepciones.
  - Posee **efectos colaterales**.
- Los manejadores pueden agregarse y encontrarse al final de diferentes unidades de programa.
- Ejemplo

- 


- **La asociación entre excepción y manejador se da por nombre.**
- Si se quiere continuar ejecutando las instrucciones de un bloque que lanzó una excepción, es necesario crear un bloque interno que contenga las instrucciones que pueden fallar junto al manejador de la excepción. **(De la siguiente forma se puede generar una similitud con el modelo de Reasunción)**
- 

### Excepciones predefinidas built-in que posee

- **Constraint_Error**
    - **Surge cuando se intenta viola una restricción impuesta en una declaración.** Por ejemplo, indexar más allá de los límites de un array, zerodivision,  etc.
- **Program_Error**
    - **Surge cuando se intenta violar la estructura de control o regla del lenguaje.** Por ejemplo, una función que termina sin devolver un valor.
- **Storage_Error**
    - **Surge cuando se produce una violación de memoria.** Por ejemplo, que se requiera más memoria que la disponible.
- **Tasking_Error**
    - **Surge cuando hay errores en la comunicación y manejo de tareas del sistema.** Por ejemplo, en concurrencia y la programación de tareas/threads.
- **Name_Error**
    - **Surge cuando hay un error de nombre.** Por ejemplo, se produce cuando se intenta abrir un fichero que no existe.

### Propagación

- **Si la unidad que genera la excepción proporciona un manejador para la misma, el control se transfiere inmediatamente a ese manejador**
    - Se omiten las acciones que siguen al punto en el que se generó la excepción.
    - Se ejecuta el código del manejador.
    - El programa continúa su ejecución desde la instrucción siguiente al manejador.
- **Si la unidad no proporciona un manejador se busca por propagación dinámicamente**
    - Se termina la unidad (bloque, paquete, subprograma o tarea) dónde se produce la excepción.
    - Si el manejador no se encuentra en ese lugar la excepción se propaga dinámicamente(quién lo llamó). Esto significa que se vuelve a levantar en otro ámbito.
    - Siempre hay que tener en cuenta el alcance, puede convertirse en anónima. Al propagarse a otras unidades la variable excepción declarada ya no está en el alcance y quedará sin nombre y entrará por when others.

### Uso del Raise

- La utilidad de **raise** es poder **lanzar excepciones que pueden ser definidas por el programador.**
- Podemos **levantar la misma excepción** que se generó usando únicamente la palabra **raise**.
- Otro sentido para el uso del raise es que el **manejador** podría **realizar algunas acciones de recuperación** y luego utilizar raise para volver a lanzar la excepción y permitir que se **propague más arriba en la jerarquía de manejo de excepciones.**

- ```
  with Ada.Text_IO;

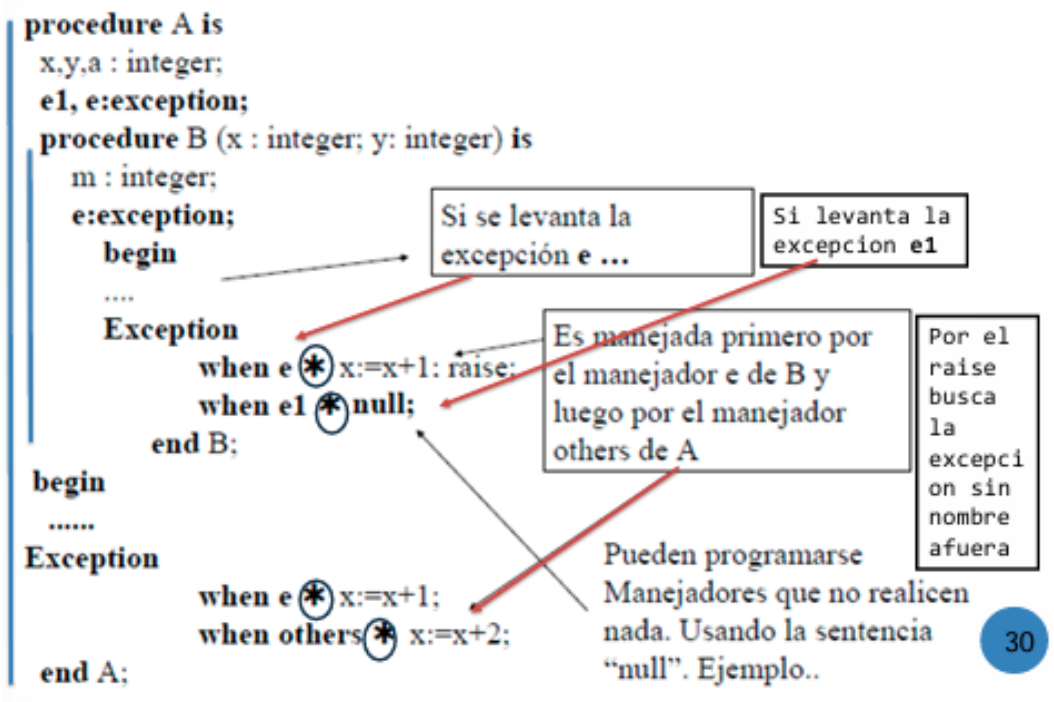
  procedure Excepciones_1 is
     I: Integer;
  begin
     I := 0;
     I := 4 / I;  -- elevará una excepción Constraint_Error
     Ada.Text_IO.Put_Line ("Resultado: " & Integer'Image (I));
  end Excepciones_1;
  ```
  excepción built-in (predefinidas)

  ```
  Resultado: raised CONSTRAINT_ERROR : excepciones.adb:7 divide by zero
  ```

  ```
  with Ada.Text_IO;

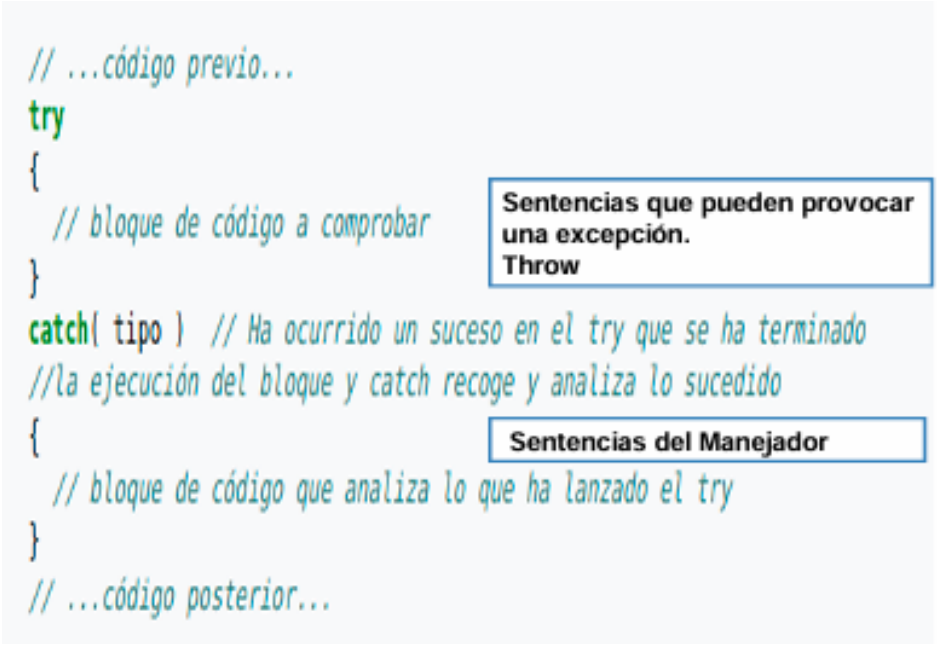
  procedure Excepciones_2 is
     I: Integer;
  begin
     I := 0;
     I := 4 / I;  -- elevará una excepción Constraint_Error
     Ada.Text_IO.Put_Line ("Resultado: " & Integer'Image (I));
  exception
     when Constraint_Error =>
        Ada.Text_IO.Put_Line ("Intento de dividir por 0");
  end Excepciones_2;
  ```
  excepción del programador

  ```
  Resultado: Intento de dividir por 0
  ```

- 

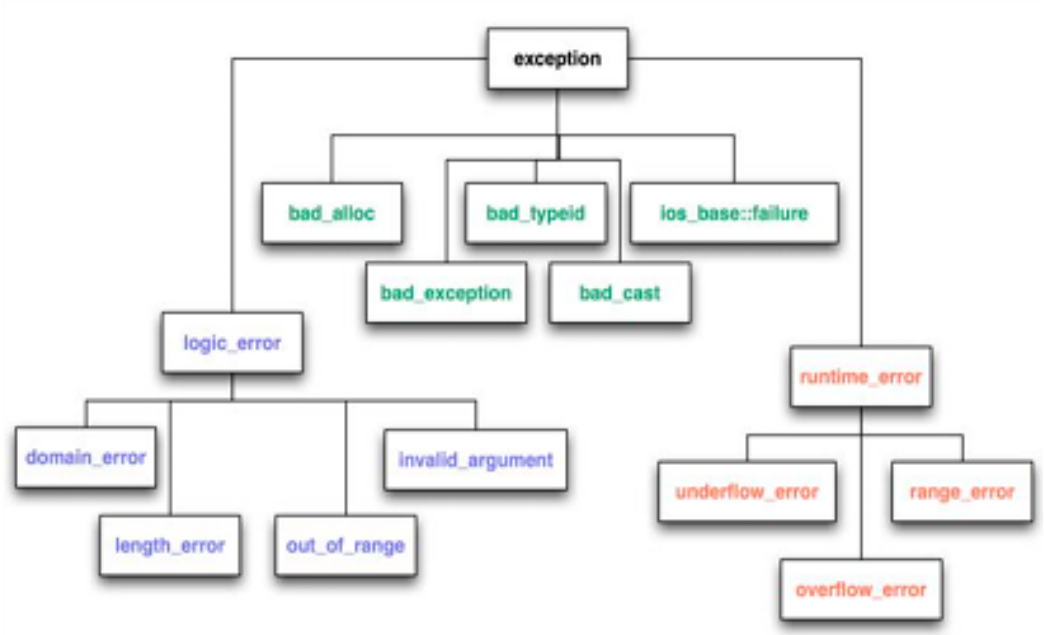

## Excepciones en C++

- Utiliza el criterio de **Terminación**.
- Se usa la **palabra clave Try** para indicar los **bloques de código donde pueden llegar a levantarse excepciones**.
- Se usa la **palabra clave Catch** para **especificar los manejadores de excepciones.** Tienen la siguiente declaración:
  - *Catch(NombreDeLaExcepción)*
  ```

    - **Las cláusulas catch deben estar después del bloque try y antes de cualquier código que esté fuera del bloque try.**
    - Los manejadores van asociados a bloques { }.
- Se usa la **palabra clave Throw** para **lanzar explícitamente una excepción.**
- Ejemplo:
    - 

- El lenguaje permite **pasar parámetros al levantar una excepción**
    - `throw MiExcepcion(parametro1, , parametroN);`
- El lenguaje permite que las **rutinas puedan listar las excepciones que pueden alcanzar.**
    - **Void rutina() throw (Ayuda,ZeroDivide);**
    - ¿Qué sucede si la rutina…?
        - **Alcanzó otra excepción no contemplada en el listado.**
            - No se propaga la excepción y la función especial unexpected() es ejecutada automáticamente, dicha función normalmente causa abort() que provoca el final del programa. Unexpected puede ser redefinida por el programador.
        - **Colocó en su interfaz el listado de posibles excepciones alcanzables.**
            - Se puede propagar la excepción. Pero si una excepción se propaga repetidamente y no encuentra manejador, entonces la función terminate() se ejecuta de forma automática.
        - **Colocó en su interfaz una lista vacía → throw().**
            - Ninguna excepción será propagada.

### Excepciones predefinidas en el lenguaje




### Funcionamiento

- El **bloque try** que contiene código que **puede lanzar una excepción.**
- **Si se lanza** una excepción el **control se transfiere** inmediatamente a la cláusula **catch**.
- Si la **excepción coincide** con el **tipo especificado** en la cláusula **catch**, se ejecuta el **bloque de código** de esa cláusula catch.
- Si la **excepción se maneja exitosamente**, la **ejecución continúa después** del **bloque try-catch**.
- Si **no se encuentra** una **catch correspondiente** o **no se maneja la excepción**, la excepción puede **propagarse hacia bloques try-catch externos**.
- Sinó **puede resultar** en una **finalización abrupta del programa**.

## Excepciones en CLU

- Utiliza el criterio de **Terminación**.
- **Solamente pueden ser lanzadas por los procedimientos.**
    - Si una instrucción genera una excepción, el procedimiento que contiene la instrucción retorna anormalmente al generar la excepción.
    - **Un procedimiento no puede manejar una excepción generada por su ejecución, quien llama al procedimiento debe encargarse de manejarla.**
- Las **excepciones** que puede **lanzar** un **procedimiento** se deben **declarar en su encabezado.**
- Se **lanzan explícitamente** con la palabra clave **signal**.
- **Los manejadores se colocan al lado de una sentencia simple o compleja y llevan la palabra clave when**
    - `<instrucción> except <lista_de_controladores> end`

    - <instrucción> puede ser cualquier instrucción (compuesta) del lenguaje. Si la ejecución de una invocación de procedimiento dentro de <instrucción> genera una excepción, el control se transfiere a <lista_de_controladores>.
- Posee **excepciones predefinidas** con su **manejador asociado**. Por ejemplo, **failure**.
- Se puede **pasar parámetros** a los **manejadores**.
- Se puede **levantar nuevamente** una **excepción** usando **resignal**.
- Las **excepciones** se pueden **levantar** en **cualquier lugar del código**.

### Propagación

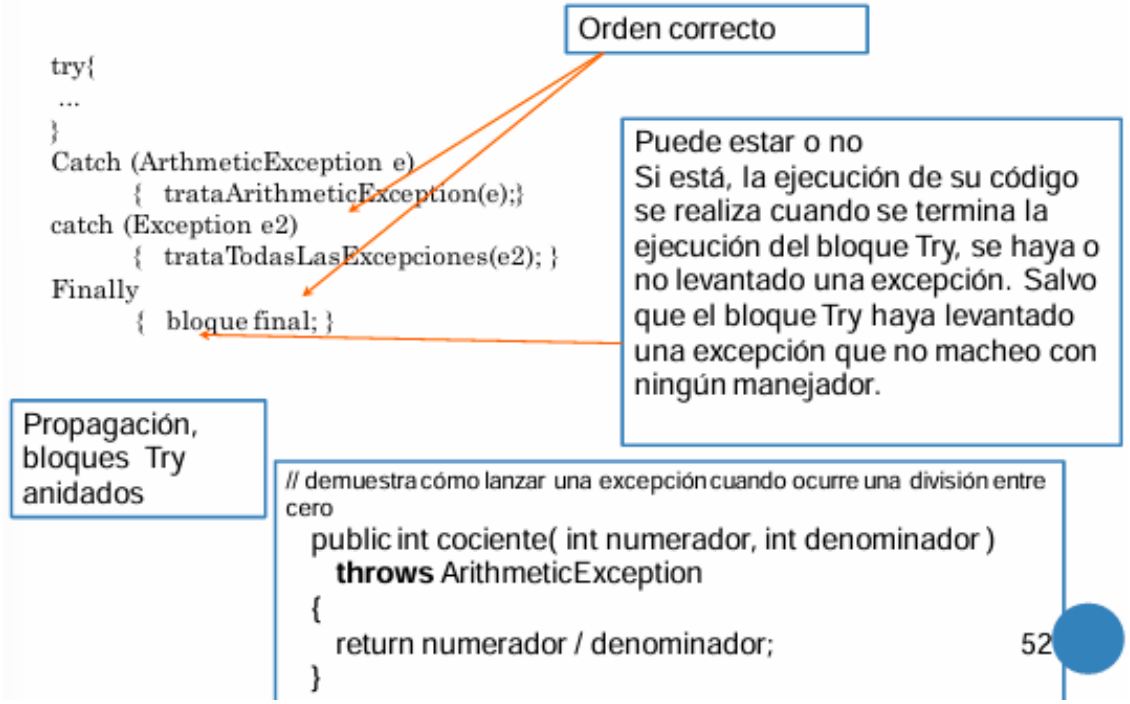
- **Se termina el procedimiento donde se levantó la excepción** y devuelve el control al llamante, **justo en la sentencia donde se hizo el llamado** al proceso que levantó la excepción donde **se debe encontrar al manejador.**
    - **Por esto asociamos manejadores con sentencias.**
- Si el **manejador se encuentra** en ese ámbito, se **ejecuta** y luego se **pasa el control a la sentencia siguiente a la que está ligado dicho manejador.**
- Si el **manejador no se encuentra** en ese lugar, **se propaga estáticamente la excepción.** Es decir, **se busca hacia abajo si hay algún manejador que tome la excepción.**
    - **Ocurre una sola propagación dinámica:** cuando terminamos el procedimiento al principio.
- En caso de **no encontrar ningún manejador en el procedimiento que hizo la llamada se levanta una excepción failure** y devuelve el control, **terminando todo el programa.**

## Excepciones en Java

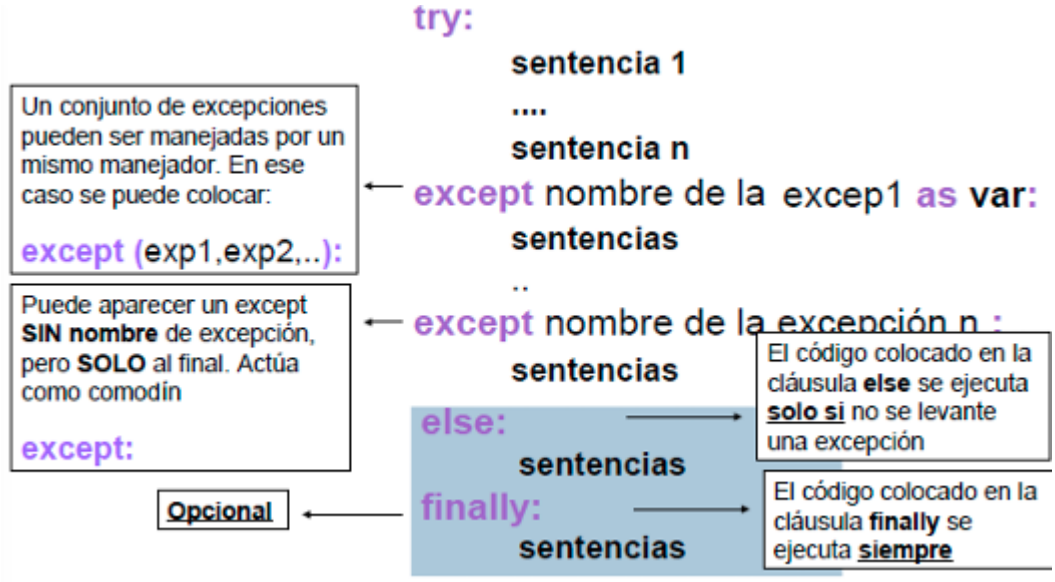
- Al igual que **C++** las **excepciones** son **objetos** que pueden ser **alcanzados** y **manejados** por **manejadores** adicionados al bloque donde se produjo la excepción.
- Cada **excepción** está **representada** por una **instancia** de la clase **Throwable** o de una de sus **subclases** (**Error** y **Exception**).
- La gestión de excepciones se lleva a cabo mediante **cinco palabras clave**:
    - **try**
        - Después de esta palabra especificamos las instrucciones que pueden generar excepciones.
    - **catch**
        - Al usar esta palabra especificamos que excepción vamos a capturar.
    - **throw**
        - Se usa para lanzar una excepción.
    - **throws**
        - Se usa para especificar cualquier excepción que se envía desde un método (posibles excepciones que puede lanzar).
    - **finally**
        - Bloque de código que siempre se va a ejecutar.

### Fases del tratamiento de Excepciones

- **Detectar e informar del error**

    - **Lanzamiento de Excepciones → throw.**
    - Un método detecta una condición anormal que le impide continuar con su ejecución y finaliza "lanzando" un objeto Excepción.
- **Recoger el error y tratarlo**
    - **Captura de Excepciones → bloque try-catch.**
    - Un método recibe un objeto Excepción que le indica que otro método no ha terminado correctamente su ejecución y decide actuar en función del tipo de error.
- 


## Excepciones en Python

- Se manejan a través de **bloques try except**.
- **Presenta la siguiente estructura**
    - 

### Funcionamiento del Try

- Se ejecuta el bloque try.
- Si no ocurre ninguna excepción, el bloque except es salteado y se termina la ejecución del bloque try.
- Si ocurre una excepción en el bloque try, el resto del bloque try es salteado. Luego, si el tipo de excepción coincide con alguna excepción nombrada luego de la palabra reservada except, se ejecuta el bloque except que corresponda, y la ejecución del programa continúa.
- Si ocurre una excepción que no coincide con ninguna excepción nombrada en el except, ésta se pasa a declaraciones try de más afuera, si no se encuentra un manejador, significa que es una excepción no manejada, por lo que la ejecución se frena con un mensaje de error.

### ¿Qué ocurre cuando una excepción no encuentra un manejador en su bloque “try-except”?

- **Busca estáticamente**
    - Analiza si ese try está contenido dentro de otro y si ese otro tiene un manejador para esa excepción. Sino…
- **Busca dinámicamente**
    - Analiza quién lo llamó y busca allí.
- **Si no se encuentra un manejador, se corta el proceso** y larga el mensaje standard de error
- **Levanta excepciones explícitamente con “raise”.**

## Excepciones en PHP

- Modelo de **terminación**.
- Una **excepción** puede ser **lanzada** (**thrown**) y **atrapada** (**catched**).
- **El código está dentro de un bloque try.**
- Cada **bloque try** debe tener **al menos** un **bloque catch** correspondiente.
- Las **excepciones** pueden ser **lanzadas** o **relanzadas dentro** de un **bloque catch**.
- **Se puede utilizar un bloque finally después de los bloques catch.**
- El **objeto lanzado** debe ser una **instancia** de la clase **Exception** o de una **subclase** de Exception. **Intentar lanzar un objeto que no lo es resultará en un Error Fatal de PHP.**
- **Cuando una excepción es lanzada**, **el código siguiente a la declaración no será ejecutado, y PHP intentará encontrar el primer bloque catch coincidente.**
    - Si una excepción no es capturada, se emitirá un Error Fatal de PHP con un mensaje "Uncaught Exception ..." ("Excepción No Capturada"), **a menos que se haya definido un gestor con set_exception_handler().**

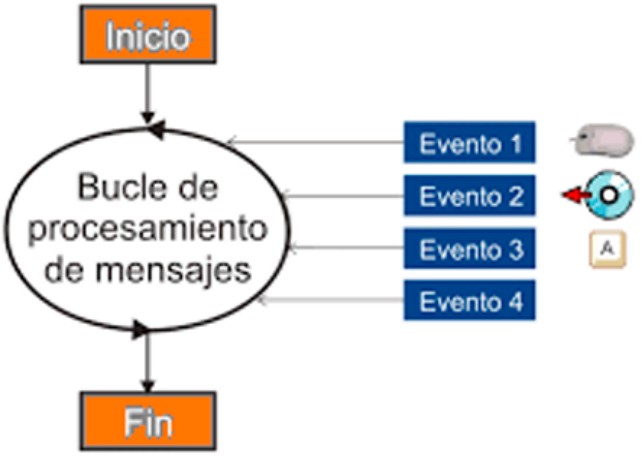

### Lenguajes que no tienen manejo de excepciones

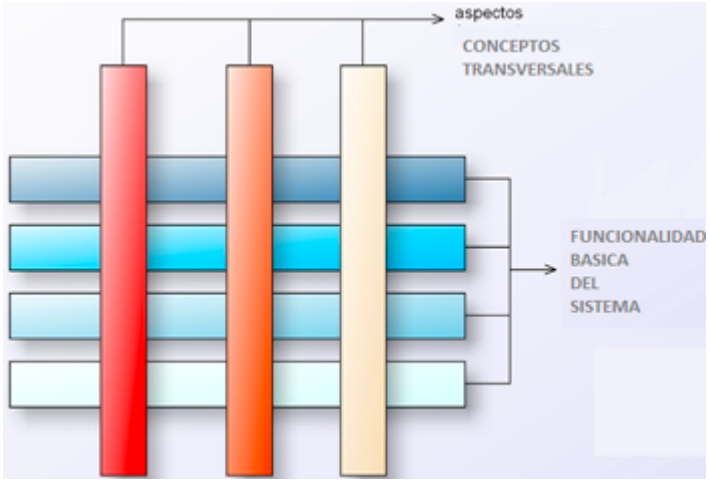
- **Los lenguajes pueden simular este manejo mediante el uso de If's y llamados a procedimientos que manejan los respectivos errores.**

# Clase 10

## Paradigmas

- **Un paradigma de programación es un estilo de desarrollo de programas**, un modelo para resolver problemas computacionales. Los **lenguajes de programación,** necesariamente, se encuadran en uno o varios paradigmas a la vez, a partir del tipo de órdenes que permiten implementar, tiene una **relación directa con su sintaxis.**
- **Principales Paradigmas**
  - **Imperativo**
    - Sentencias + Secuencias de comandos.
  - **Declarativo**
    - Los programas describen los resultados esperados sin listar explícitamente los pasos a llevar a cabo para alcanzarlos.
    - Uno declarativo puede ser a su vez lógico, que se basa en aserciones lógicas.
  - **Funcional**
    - Los programas se componen de funciones, estas son su componente principal.
  - **Orientado a Objetos**

        - Métodos + mensajes.
- **Otra forma de clasificación**
    - **Dirigido por eventos**
        - El flujo del programa está determinado por sucesos externos (por ejemplo, una acción del usuario).
        - 

    - **Orientado a Aspectos**
        - Apunta a dividir el programa en módulos independientes, cada uno con un comportamiento y responsabilidad bien definido.
        - 


# Programación Lógica

- Es un tipo de **paradigma** de programación dentro del **paradigma de programación declarativa**.
- Los **programas** son una serie de **aserciones lógicas (cláusulas)** que pueden ser **reglas** o **hechos** que proveen una **especificación declarativa de que es lo que se conoce.**
- Una **Query** es el **objetivo que queremos alcanzar con la ejecución de un programa**.
- El **conocimiento** se **representa** a través de **reglas** y **hechos**.
- Los **objetos** son **representados** por **términos**, los cuales **contienen constantes** (**determinado**) y **variables** (**indeterminado**).
- PROLOG es el lenguaje lógico más utilizado.

## Elementos de la Programación Lógica

### Variables

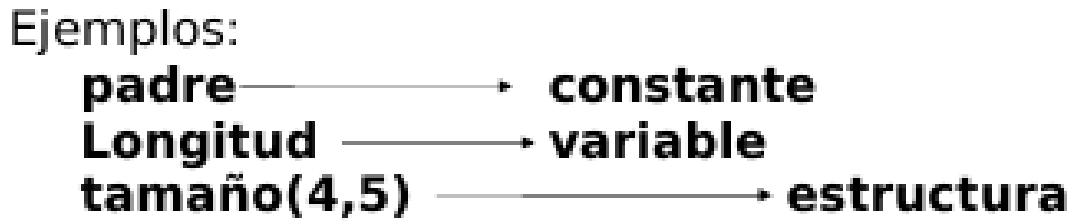
- Se refieren a **elementos indeterminados** que pueden **sustituirse** por cualquier otro.
    - “humano(X)” → La X puede ser sustituida por constantes como juan, pepe, etc.
- Los **nombres** de las variables **comienzan** con **mayúsculas** y pueden **incluir números**.

### Constantes

- A diferencia de las **variables** son **elementos determinados**.
    - “humano(juan)”
- Las **constantes** son **string** de **letras** en **minúsculas** (representan objetos atómicos) o **string** de **dígitos** (representan números).

### Término compuesto

- Consisten en un ”**functor**” seguido de un **número fijo** de **argumentos encerrados** entre **paréntesis**, los cuales son a su vez **términos**.
- Se denomina “**aridad**” al **número de argumentos**.
- Se denomina “**estructura**” (ground term) a un **término compuesto cuyos argumentos no son variables**.

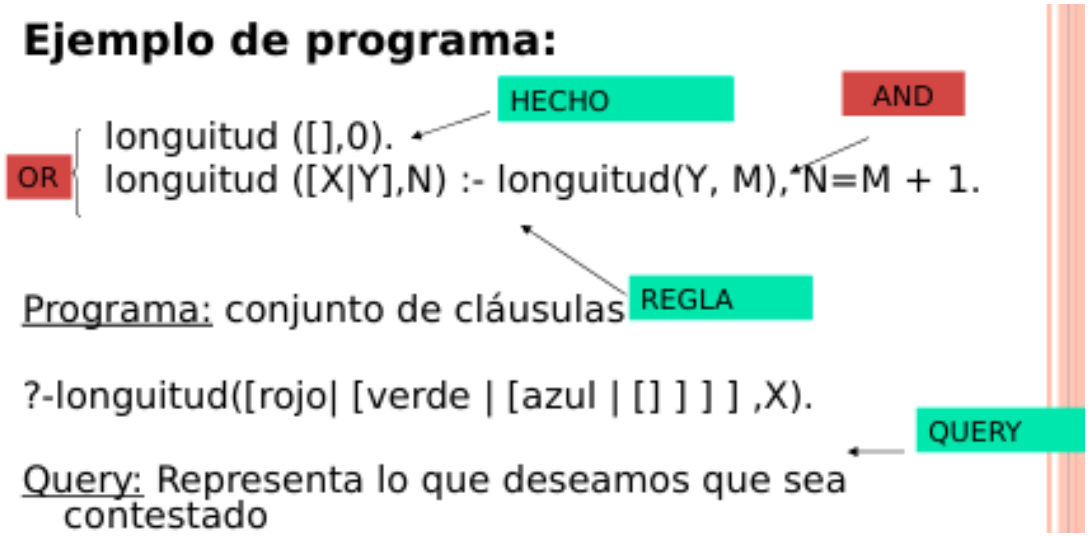



### Listas

- La **constante []** representa una **lista vacía**.
- El functor **“.”** construye una **lista de un elemento y una lista**.
    - Ejemplo: **.(alpha,[])**, representa una lista que contiene un único elemento que es alpha.
- Otra manera de representar la lista es **usando [] en lugar de .().**
    - Ejemplo anterior la lista quedaría: **[alpha,[]]**
- También se puede representar **utilizando el símbolo |**
    - Ejemplo anterior: **[alpha|[]]**
- La **notación general** para denotar **lista** es : **[X|Y]**
    - **X** es el **elemento cabeza** de la lista.
    - **Y** es una **lista**, que **representa la cola** de la **lista** que **se está modelando.**

### Cláusulas de Horn

- **Son los elementos utilizados para escribir programas.**

- Se dividen en 2:
    - **Hecho**
        - Expresan **relaciones** entre **objetos**.
        - Expresan **verdades**.
        - Son **expresiones** del **tipo** → **p(t1, t2, …, tn)**
            - **tiene(coche,ruedas)** → representa el hecho que un coche tiene ruedas.
            - **longitud([], 0)** → representa el hecho de que una lista vacía tiene longitud 0.
            - **moneda(peso)** → representa el hecho que peso es una moneda.
    - **Regla**
        - Tiene la forma → **conclusión :- condición** (Sintaxis de Prolog).
            - **:-** Indica **"Si"/"if".**
            - **conclusión** es un **simple predicado.**
            - **condición** es una **conjunción de predicados separados por comas. Estos representan un AND lógico.**
        - En un **lenguaje procedural** una **regla** la podríamos representar como: **if condición else conclusión.**
- 

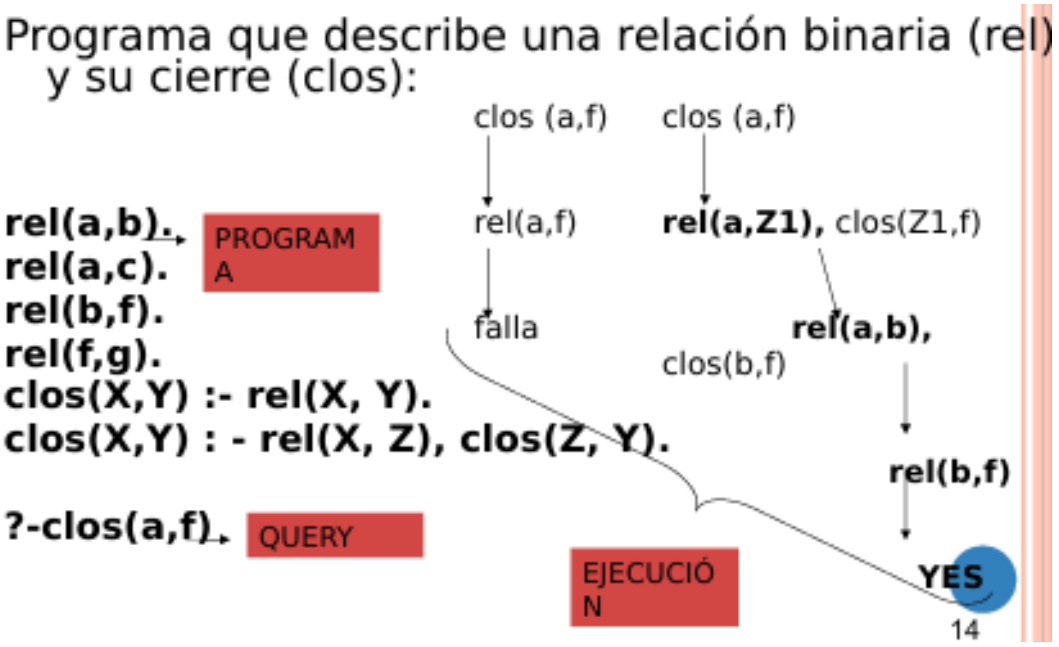
**Programa:**

```
longuitud ([],0).
longuitud ([X|Y],N) :- longuitud(Y, M), N=M + 1.

?-longuitud([rojo| [verde | [azul | [] ] ] ] ,X).
```

longuitud([verde | [azul | [] ] ],M) y X=M+1
longuitud([azul | [] ] ,Z) y M=Z+1
longuitud([],T) Z=T+1  T=0 => Z=1
M=2
X=3

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




## Programación Orientada a Objetos

- Un **programa** escrito con un lenguaje orientado a objetos es un **conjunto de objetos que interactúan mandándose mensajes.**

### Elementos y Conceptos de la Programación Orientada a Objetos

#### Mensajes

- Es una **petición** de un **objeto** a **otro** para que este se **comporte** de una **determinada manera**, **ejecutando** los **métodos correspondientes**.

### Métodos

- Es un **programa** que está **asociado** a un **objeto determinado** y cuya **ejecución** solo puede **desencadenarse** a través de un **mensaje recibido** por **éste** o por sus **descendientes**.

### Clases

- Es un **tipo definido** por el **usuario** que **determina** las **estructuras** de **datos** y las **operaciones asociadas** con ese **tipo**.
- **Primer nivel de abstracción de datos**: definimos estructura, comportamiento y tenemos ocultamiento.
- Cada **objeto pertenece** a una **clase** y **recibe** de ella su **funcionalidad**.
- La **información contenida** en el **objeto sólo puede ser accedida** por la **ejecución** de los **métodos correspondientes**.

### Instancia de Clase

- Cada vez que se **construye** un **objeto** se está **creando** una **INSTANCIA** de esa **clase**.
- Es un **objeto individualizado** por los **valores** que tomen sus **atributos**.

### Herencia

- Concepto que nos dice que **si una clase A tiene una superclase B, entonces A hereda todas las propiedades de su superclase B.**
- **El segundo nivel de abstracción** consiste en agrupar las clases en jerarquías de clases (definiendo SUB y SUPER clases).

### Polimorfismo

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

## Paradigma Funcional o Aplicativo

- **Basado en el uso de funciones (componente principal).**
- Muy popular en la **resolución** de **problemas** de **inteligencia artificial**, **matemática**, **lógica**, **procesamiento paralelo**.

### Ventajas

- **Vista uniforme de programa y función.**
- **Tratamiento de funciones como datos.**
- **Liberación de efectos colaterales.**
- **Manejo automático de memoria.**

### Desventaja

- **Ineficiencia de ejecución.**

## Características

- Provee un **conjunto** de **funciones primitivas**.
- Provee un **conjunto** de **formas funcionales**.
- **Semántica basada** en **valores**.
- **Transparencia referencial** (Aspecto **MUY importante** para la **seguridad**).
- **Regla** de **mapeo** basada en **combinación o composición**.
- **Las funciones que utiliza el lenguaje son de primer orden.**

## Funciones

### Valor de una Función

- El **VALOR** más **importante** en la **programación funcional** es el de una **FUNCIÓN**. **Matemáticamente** una función es una **correspondencia A → B**. Las **funciones** son **tratadas** como **valores**, pueden ser **pasadas** como **parámetros**, **retornar resultados**, etc. **Básicamente una función es el valor que retorna.**

### Definición de una Función

- Existen muchas maneras de definir una misma función, **pero siempre dará el mismo valor.** Ejemplos:
    - DOBLE X = X + X
    - DOBLE' X= 2 * X

### Tipo de una Función

- Puede estar **definido explícitamente** dentro del **SCRIPT**. Ejemplos:
    - cuadrado::num→ num **(definición del tipo)**
    - cuadrado x= x * x **(definición de la función)**
- O puede **deducirse/inferirse** el tipo de una función.
- **Toda función tiene asociado un tipo y no debería haber errores de tipo.**

## Expresiones y Valores

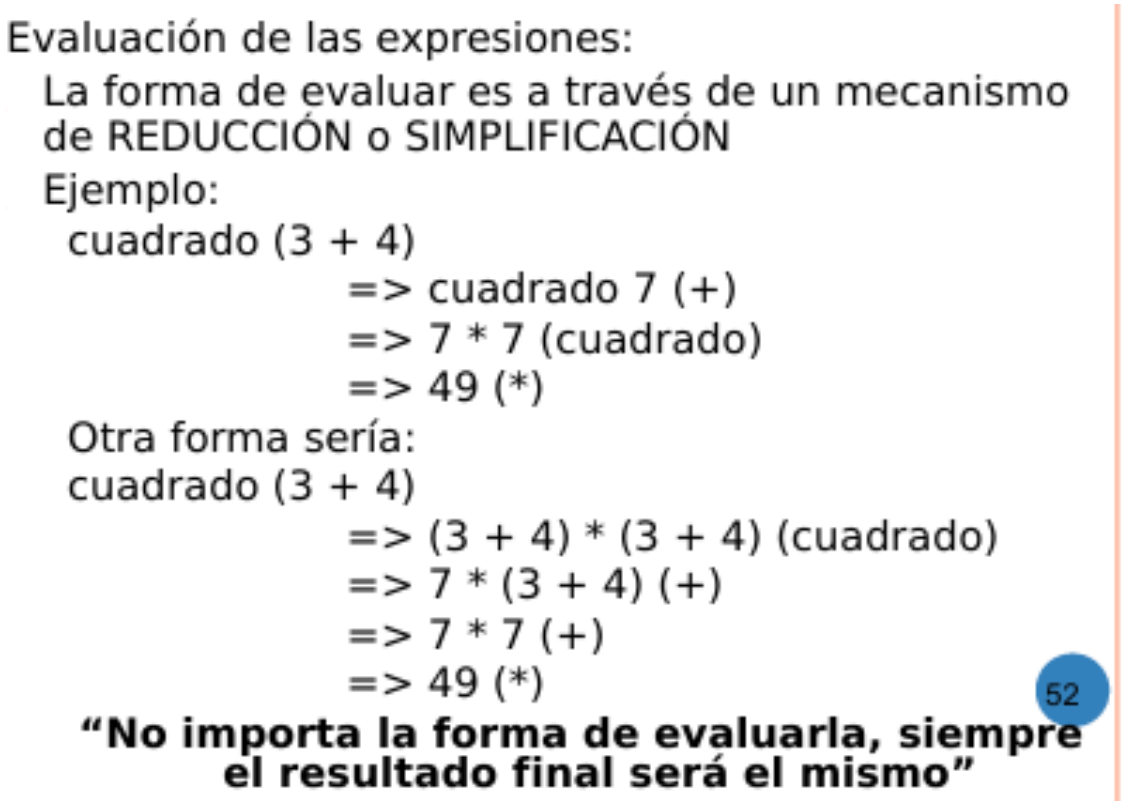
### Expresión

- Es la **noción central** de la **programación funcional**.
- **Una expresión es su VALOR**.
- El valor de una **expresión depende ÚNICAMENTE** de los **valores** de las **sub expresiones** que la **componen**.
- **Pueden contener VARIABLES (variables matemáticas)** que son **valores desconocidos**.
- Cumplen con la siguiente propiedad:
    - **TRANSPARENCIA REFERENCIAL**
        - **Dos expresiones sintácticamente iguales darán el mismo valor.**
        - **NO** existen efectos colaterales.

- Algunas **expresiones** pueden **NO llegar** a **reducirse** del **todo**, a estas se las denomina **CANÓNICAS**, pero se les asigna un **VALOR INDEFINIDO** y **corresponde** al **símbolo bottom(^)**
- **Toda EXPRESIÓN siempre denota un VALOR.**

### Script

- Un **script** es una **lista de definiciones** y pueden **someterse a evaluación**.




## Formas de Reducción

### Orden Aplicativo

- **Aunque no lo necesite SIEMPRE evalúa los argumentos.**

### Orden Normal (Lazy Evaluation)

- **No calcula más de lo necesario**.
- La **expresión NO** es **evaluada** hasta que su **valor se necesite**.
- Una **expresión compartida NO es evaluada más de una vez.**

### Haskell

- **Se mantiene todo en memoria.**
- Las funciones se evalúan por una única vez y el valor se guarda en memoria.

## Tipos

### Básicos

- Son los **primitivos**, ejemplo:
    - **NUM (INT y FLOAT) → (Números).**
    - **BOOL → (Valores de verdad).**
    - **CHAR → (Caracteres).**

### Derivados

- Se **construyen** de otros **tipos** (parecidos a los **definidos por el usuario**), ejemplo:
    - (num,char) → **Tipo de pares de valores**.
    - (num→[]char) → **Tipo de una función.**

### Expresiones de tipo polimórficas

- **Se consideran expresiones de tipo polimórficas a las funciones que no es fácil deducir su tipo, se utilizan letras griegas para tipos polimórficos.**

## Currificación

- **Mecanismo** que **reemplaza argumentos estructurados** por **argumentos más simples**.

## Cálculo Lambda

- Es un **modelo** de **computación** para **definir funciones** que se **independiza** de la **sintaxis** del **lenguaje de programación.**
- Se **utiliza** para **entender** los **elementos** de la **programación funcional** y la **semántica subyacente**, independientemente de los **detalles sintácticos** de un **lenguaje de programación en particular**.