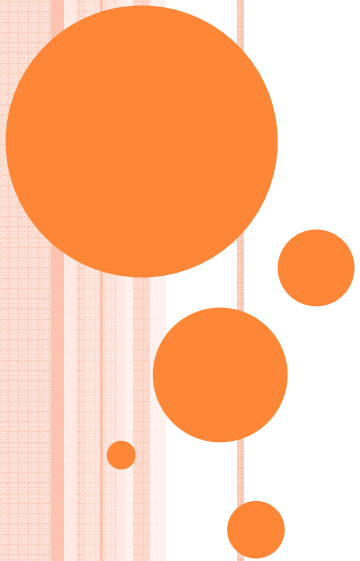


Tipos de datos recursivos TADs

Introducción excepciones

RECURSION

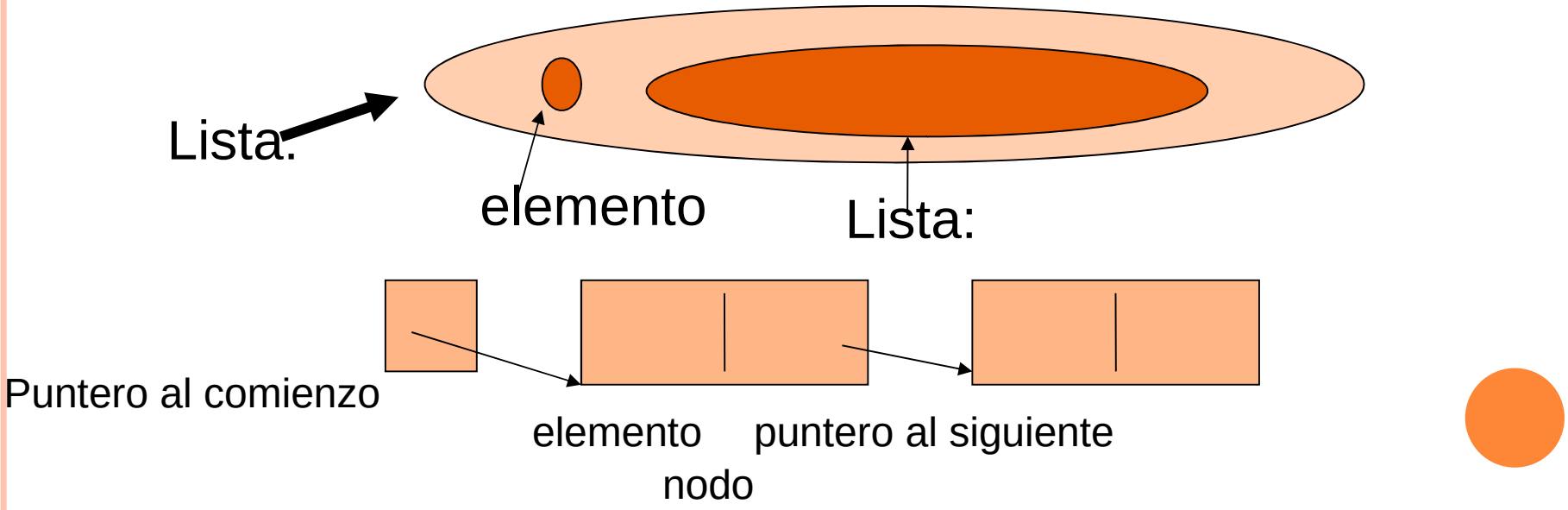
Un tipo de dato recursivo se define como una estructura que puede contener componentes del mismo tipo.

La recursión es un mecanismo de estructuración para definir datos agrupados:

- cuyo tamaño puede crecer arbitrariamente
- cuya estructura puede ser arbitrariamente compleja.

Recursion - Implementación

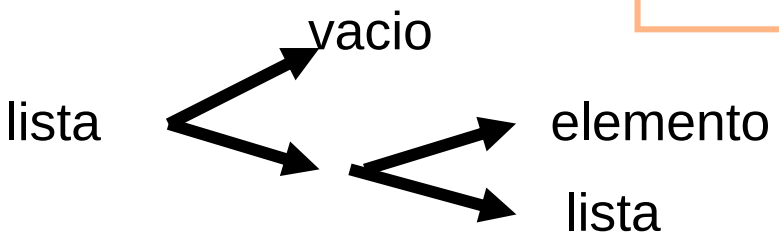
- Los lenguajes de programación convencionales soportan la implementación de los tipos de datos recursivos a través de los **punteros**.
- Los lenguajes funcionales proveen mecanismos más abstractos que enmascaran a los punteros



EJEMPLOS

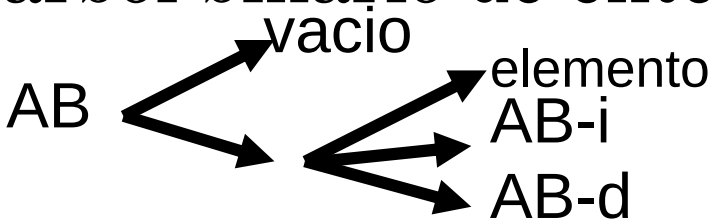
- lista de enteros

Unión entre nil y un producto cartesiano de enteros y lista



int_list = {nil} U (integer X int_list)

- árbol binario de enteros

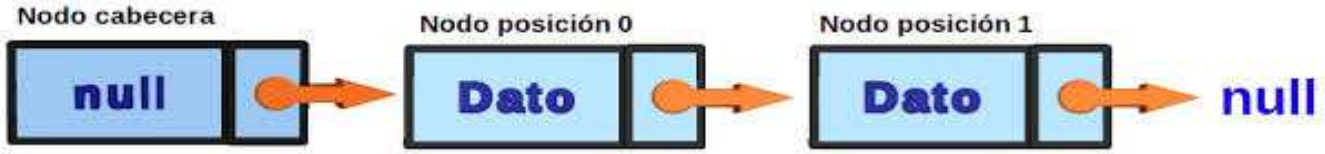


Unión entre nil y un producto cartesiano de enteros y árboles

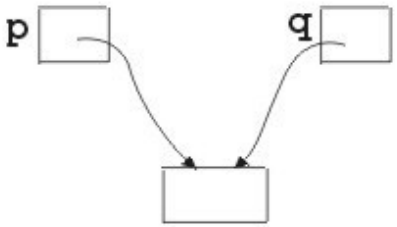
int_bin_tree = {nil} U (integer X int_bin_tree X int_bin_tree)

PUNTEROS

- Estructuras de tamaño arbitrario, **número de items no determinado** (datos anónimos)



- **Relaciones múltiples** entre los items: varias estructuras sin necesidad de duplicarlo.



- **Acceso a bajo nivel:** los punteros están cerca de la máquina en su implementación

Inseguridad en los Punteros

- Por acceder a **bajo nivel**, pueden obscurecer o hacer inseguros a los programas que los usan.

- Se compara a los punteros con los goto
  - los goto amplian el rango de sentencias que pueden ejecutar.
  - los punteros amplian el rango de las celdas de memoria que pueden ser referenciadas por una variable y también amplian el tipo de los valores que puede contener un objeto.

Inseguridad en los punteros

1. Violación de tipos
2. Referencias sueltas - referencias dangling
3. Liberación de memoria: objetos perdidos
4. Punteros no inicializados
5. Punteros y uniones discriminadas
6. Alias

3. Liberación de memoria: objetos perdidos

- Las variables puntero se alocan como cualquier otra variable en la pila de registros de activación
- los objetos (apuntados) que se alocan a través de la primitiva new son alocados en la heap
- La memoria disponible (heap) podría rápidamente agotarse a menos que de alguna forma se devuelva el almacenamiento alocado liberado.

3. Liberación de memoria: objetos perdidos

- Si los objetos en el heap dejan de ser accesibles esa memoria podria liberarse

garbage (**objetos perdidos**)

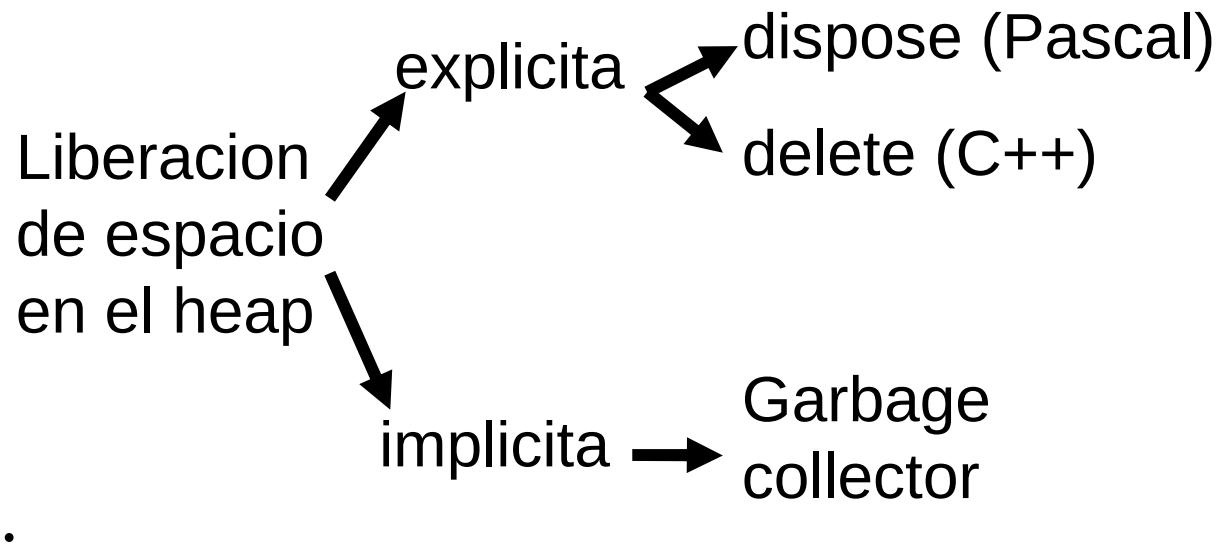
- Un objeto se dice accesible si alguna variable en la pila lo apunta directa o indirectamente. Un objeto es basura si no es accesible.

Mecanismos para desalocar memoria

3. Liberación de memoria: objetos perdidos

reconocimiento de que porción de la memoria es basura

se requiere o no intervención del usuario



Reponsabilidad del programador

Indepediente de la aplicacion

.

LIBERACION DE MEMORIA EXPLICITA

- El reconocimiento de la basura recae en el programador, quien notifica al sistema cuando un objeto ya no se usa.

- No garantiza que no haya otro puntero que apunte a esta dirección definida como basura, este puntero se transforma en dangling (puntero suelto).

  - Este error es difícil de chequear y la mayoría de los lenguajes no lo implementan por que es costoso

EXPLICITA: EJEMPLO EN C

- Función llamada *free* libera memoria reservada de manera dinámica
- Puede generar referencias sueltas
- Para evitarlo se necesitaría una verificación dinámica para garantizar el uso correcto

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {

int *i;
int *p;
i = malloc(sizeof(int));
p=i;
free(i);
x=&p; //a que referencia p?
//Utilizar i despues de esto peligroso, se recomienda :
i = NULL;
return 0;
}
```

libera el espacio de de i

p referencia suelta

IMPLÍCITA: GARBAGE COLLECTOR

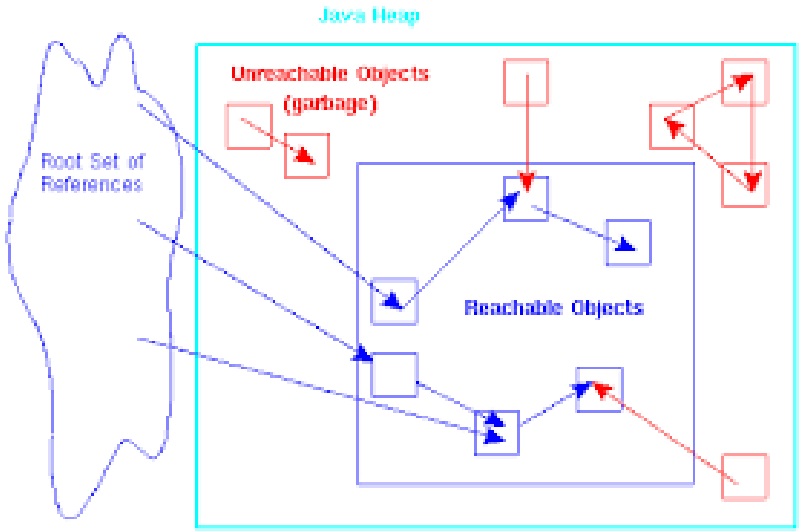
- El sistema, durante la ejecución tomará la decisión de descubrir la basura por medio de una algoritmo de recolección de basura: **garbage collector**.

- muy importante para los lenguajes de programación que hacen un uso intensivo de variables dinámicas.(LISP, Phyton)

IMPLÍCITA: GARBAGE COLLECTOR

- Eiffel y Java que uniformemente tratan a todos los objetos como referenciados por punteros proporcionan un recolector automático

- ADA chequea dinámicamente que el tiempo de vida de los objetos apuntados sea menor igual que el del puntero



GARBAGE COLLECTOR: IMPLEMENTACIONES

- Se ejecuta durante el procesamiento de las aplicaciones

- Sistema interactivo o de tiempo real: no bajar en el rendimiento y evitar los riesgos

Muy eficiente

Ejecución en paralelo

GARBAGE COLLECTOR: IMPLEMENTACIONES

- *"Reference counting"* este esquema supone que cada objeto en la heap tiene un campo descriptor extra que indica la cantidad de variables que lo referencian.

  Si es 0 → garbage

  Existen otras métodos mas complejos, pero se pierde en eficiencia.

Tipos Abstractos de Datos

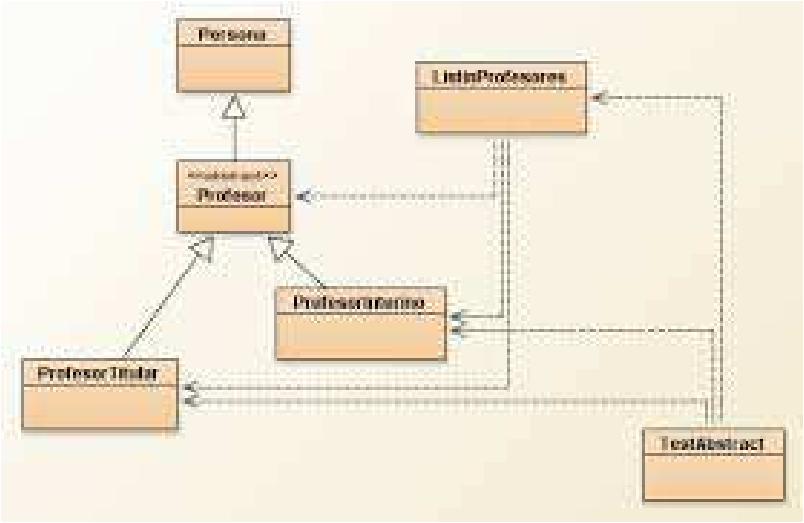
- La abstracción es el mecanismo que tenemos las personas para manejar la complejidad
- Abstraer es representar algo descubriendo sus características esenciales y suprimiendo las que no lo son.
- El principio básico de la abstracción es la **información oculta.**

Tipos Abstractos de Datos

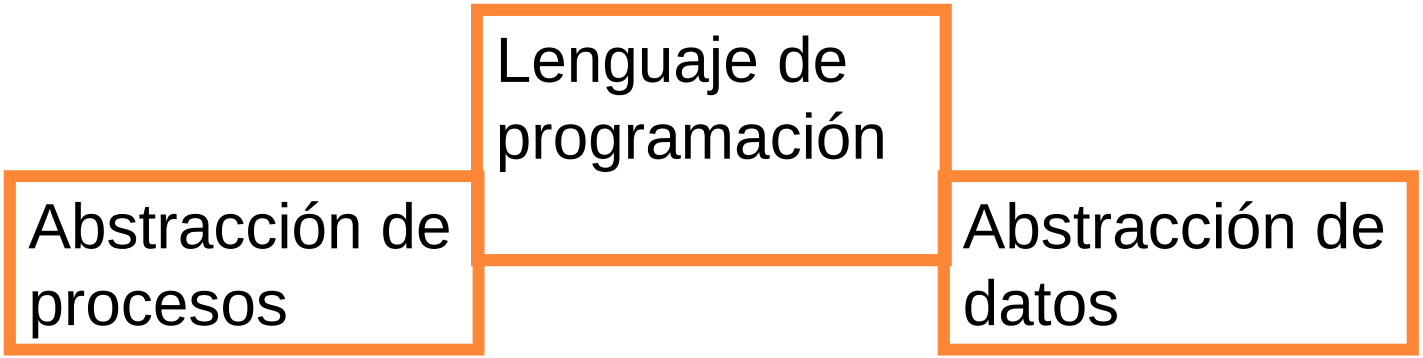
TA D = Representación (datos) + Operaciones (funciones y procedimientos)

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.*



Tipos Abstractos de Datos



Accion abstracta

UNIDADES

TADS

Extensión del lenguaje:una nueva operacion

Descubrir e incorporar nuevos tipos al lenguaje

SIMULA 67

POO

```
void INTER (x,y:int)
cuerpo
end
```

INTER (a,b)

TADs

- **Tipo abstracto de dato** (TAD) es el que satisface:

- **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 la información**: la representación de los objetos y la implementación del tipo permanecen ocultos

Refleja los niveles de abstraccion.
Modificabilidad

TADs

- Las unidades de programación de lenguajes que pueden implementar un TAD reciben distintos nombres:.
  - Modula-2 *módulo*
  - Ada *paquete*
  - C++ *clase*
  - Java *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
- Hay operaciones definidas por sí mismas que se consideran *constructores del TAD.* Normalmente inicializan

Por ejemplo:

TAD nombre del tipo (valores que toma los datos del tipo)

Sintaxis

Operación(Tipo argumento, ...) -> Tipo resultado

....

Semántica

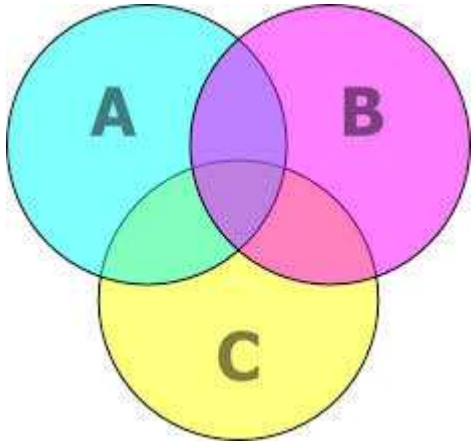
Operación(valores particulares argumentos) ⇒ expresión resultado

EJEMPLO TAD: CONJUNTO

- *TAD Conjunto(colección de elementos sin duplicidades, pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones).*

Sintaxis

- *Conjuntovacio → Conjunto
- *Añadir(Conjunto, Elemento) → Conjunto
- Retirar(Conjunto, Elemento) → Conjunto
- Pertenece(Conjunto, Elemento) → boolean
- Esvacio(Conjunto) → boolean
- Cardinal(Conjunto) → entero
- Union(Conjunto, Conjunto) → Conjunto



Semántica ∀ e1,e2 ∈ Elemento y ∀ C,D ∈ Conjunto

- Añadir(Añadir(C, e1), e1) ⇒ Añadir(C, e1)
- Añadir(Añadir(C, e1), e2) ⇒ Añadir(Añadir(C, e2), e1)
- Retirar(Conjuntovacio, e1) ⇒ Conjuntovacio
- Retirar(Añadir(C, e1), e2) ⇒ si e1 = e2 entonces Retirar(C,e2)
  sino Añadir(Retirar(C,e2),e1)

EJEMPLO TAD: PILA EN ADA

ESPECIFICACION

- PILA de enteros (100)

ENCAPSULA

```
package PILA IS
  type PILA limited private
  MAX: constant := 100
 function EMPTY (P:in PILA) return boolean
 prodedure PUSH (P:inout PILA,ELE:in
INTEGER)
 procedure POP (P: inout PILA)
 procedure TOP (P:inPILA) return INTEGER
```

Implementacion del TAD conjunto en Java u otro lenguaje

OCULTA

```
 private
      type PILA is
      vecpila : array (1..MAX) of INTEGER
      tope: INTEGER range 0..MAX:=0
end PILA
```

IMPLEMENTACION

```
package body PILA is
        function EMPTY (P:in PILA) return boolean
                .....
        end
        prodedure PUSH (P:inout PILA,ELE:in INTEGER)
                .....
        end
        procedure POP (P: inout PILA)
                .....
        end
        procedure TOP (P:inPILA) return INTEGER
                .....
        end
end PILA
```

OCULTA

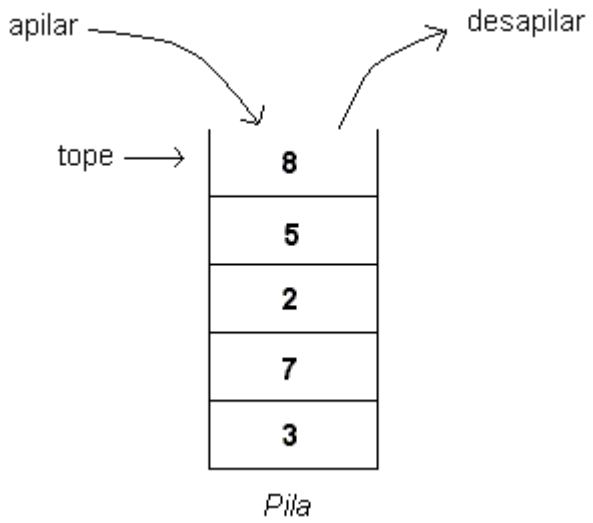
INSTANCIACION DE UNA PILA

```
with PILA
procedure USAR
   pil:PILA
   y: INTEGER
   ..........
   pil.PUSH (pil,y)
End USAR
```

INSTANCIA

ALOCA Y EJECUTA ELCODIGO DE INICIALIACION

APILA



TAD: Clases

- En términos prácticos, una *clase es un tipo definido por el usuario*
- Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce.

Atributos + Métodos

```
class NombreClase
{
        lista_de_miembros
}
```

Formato para definir una referencia
NombreClase varReferencia;

Formato para crear un objeto

varReferencia = new NombreClase(args);

EJEMPLO: CLASE EN JAVA

```
class Punto
{ private int x; // coordenada x
   private int y; // coordenada y
public Punto(int x _ , int y _ ) // constructor
{ x = x _ ;    y = y _ ; }
public Punto() // constructor sin argumentos
{ x = y = 0; }
public int leerX() // devuelve el valor de x
{ return x; }
public int leerY() // devuelve el valor de y
{ return y; }
void fijarX(int valorX) // establece el valor de x
{ x = valorX; }
void fijarY(int valorY) // establece el valor de y
{  y = valorY; }
}
```

Visibilidad de los miembros de la clase

```
Punto p;
p = new Punto();
p.fijarX (100);
System.out.println("
Coordenada x es " + p.leerX());
```

Excepciones

¿Qué es una excepción?

Condición inesperada o inusual, que surge durante la ejecución del programa y no puede ser manejada en el contexto local.

¿Qué hacemos ante la presencia de una excepción?

- El programador tiene tres opciones:
  - Inventar un valor que el llamador recibe en lugar de un valor válido
  - Retornar un valor de estado al llamador, que debe verificarlo
  - Pasar una clausura para una rutina que maneje errores
- El manejo de excepciones por parte de los lenguajes resuelve el problema...
  - El caso normal se expresa de manera simple y directa.
  - El flujo de control se enruta a un *manejador de excepciones sólo* cuando es necesario.

EXCEPCIONES

- Originalmente, se disponía de ejecución condicionada (PL/I)

  ON condición

  instrucciones

- Los lenguajes modernos ofrecen bloques léxicos.
  - El bloque inicial de código para el caso normal.
  - Un bloque contigüo con el manejador de excepciones que *reemplaza la* ejecución del *resto del bloque inicial en caso de error.*

¿Qué debemos tener en cuenta sobre un lenguaje que provee manejo de excepciones?

- ¿Cómo se **maneja** una excepción y cuál es su **ámbito**?
- ¿Cómo se **alcanza** una excepción?
- ¿Cómo **especificar** la unidades (manejadores de excepciones) que se han de ejecutar cuando se alcanza las excepciones?
- ¿A dónde se **cede el control** cuando se **termina** de atender las excepciones?
- ¿Cómo se **propagan** las excepciones?
- ¿Hay excepciones **predefinidas**?

¿Cómo se maneja una excepción y cuál es su ámbito?

- Ocurrida una excepción, que es lo que hace el lenguaje, generalmente busca el bloque de excepciones que corresponde a esa porción de código y deriva allí la ejecución.

- Debemos tener presente cual es el alcance, generalmente igual que las variables que posee el lenguaje

¿Cómo se alcanza una excepción?

- Los lenguajes proveen dos formas:
  - **Implícita**, ocurre cuando se produce una excepción que el lenguaje tiene contemplada, como puede ser la división por cero, le lanza una excepción predefinida.
  - **Explicita**, el programador hace la invocación de la excepción a través de la instrucción que provee el lenguaje: raise, throw..

¿Cómo especificar la unidades (manejadores de excepciones) que se han de ejecutar cuando se alcanza las excepciones?

Dependiendo del lenguaje, el bloque se maneja las excepciones se debe colocar en el código en determinado lugar

```
.....
catch (Exception e) {
 System.out.println("bloque de código donde se trata el problema");
}
```

¿A DÓNDE SE CEDE EL CONTROL CUANDO SE TERMINA DE ATENDER LAS EXCEPCIONES?

- Dos posibilidades:

- **Terminación**: Se **termina la ejecución de la unidad que** alcanza la excepción y se transfiere el control al manejador
- **Reasunción**: Se maneja la excepción y se devuelve el control al punto siguiente dónde se invocó a la excepción, permitiendo la **continuación**

¿Cómo se propagan las excepciones?

Debemos tener presente como hace el programa para buscar un manejador en caso que el bloque no lo contenga explicitamente!!!

- Generalmente Dinámicamente
- Puede haber combinaciones de dinámica con estática…

Algunos lenguajes que incorporaron el manejo de excepciones- PLI

- Fue el primer lenguaje. que incorporó el manejo de excepciones.
- Utiliza el criterio de **Reasunción. Cada vez que se produce la excepción,** la maneja el manejador y devuelve el control a la sentencia **siguiente** de dónde se levantó.
- La excepciones son llamadas CONDITIONS
- Los manejadores se declaran con la sentencia ON:

  ON CONDITION(Nombre-excepción) Manejador
- El manejador puede ser una instrucción o un bloque
- Las excepciones se alcanzan explícitamente con la palabra clave **Signal condition(Nombre-excepción)**

EXCEPCIONES – PLI

- Este lenguaje tiene una serie de excepciones ya **predefinidas con su** manejador asociado. Son las **Built-in exceptions.**
- **Por ej. zerodivide, se levanta cuando hay una división por cero.**
- A las built-in
  - Se les puede **redefinir los manejadores de la siguiente** manera: **ON Nombre-Built-in Begin End**
  - se las puede **habilitar y deshabilitar explícitamente**
  - se habilitan por defecto.
  - Se deshabilita anteponiendo **NO Nombre de la built-in al**

  bloque, instrucción o procedimiento al que va a afectar

- Ej: **(ZERODIVIDE)** **(NOZERODIVIDE)**

Excepciones – PLI

- Los manejadores se ligan **dinámicamente con las excepciones.** Una excepción siempre estará ligada con el último manejador definido.
- El alcance de un manejador termina cuando finaliza la ejecución de la unidad donde fue declarado
- El alcance de un manejador de una excepción se **acota cuando se define** otro manejador para esa excepción y se **restaura cuando se desactivan los** manejadores que lo enmascararon.
- No permite que se pasen parámetros a los manejadores
- **Desventajas:** Los dos últimos puntos provocan programas **difícil de escribir y comprender y la necesidad de uso de variables globales**

EXCEPCIONES – PLI