
Programación II

# Introducción a los Tipos Abstractos de Datos (TAD)

"Los estudiantes deben convencerse de que la programación *no es un arte* misterioso, sino una *disciplina ingenieril*...

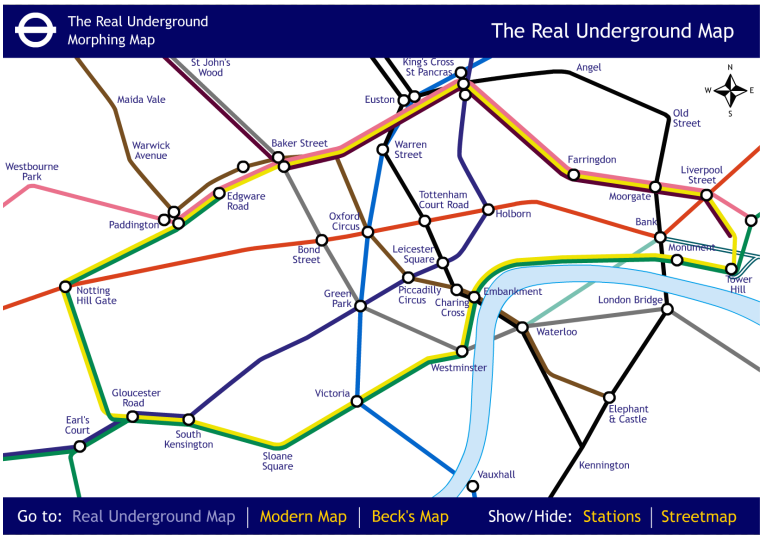
...la *abstracción* y la *especificación* deben ser las piezas clave para conseguir una *programación efectiva*"

Barbara Liskov y John Guttag

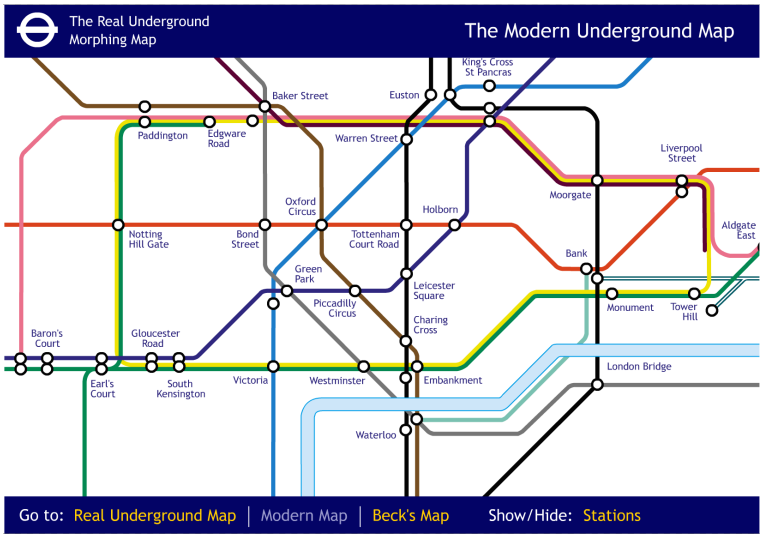

**Del lat. abstrahĕre.**

**1. tr.** Separar por medio de una operación intelectual un rasgo o una cualidad de algo para analizarlos aisladamente o considerarlos en su pura esencia o noción.

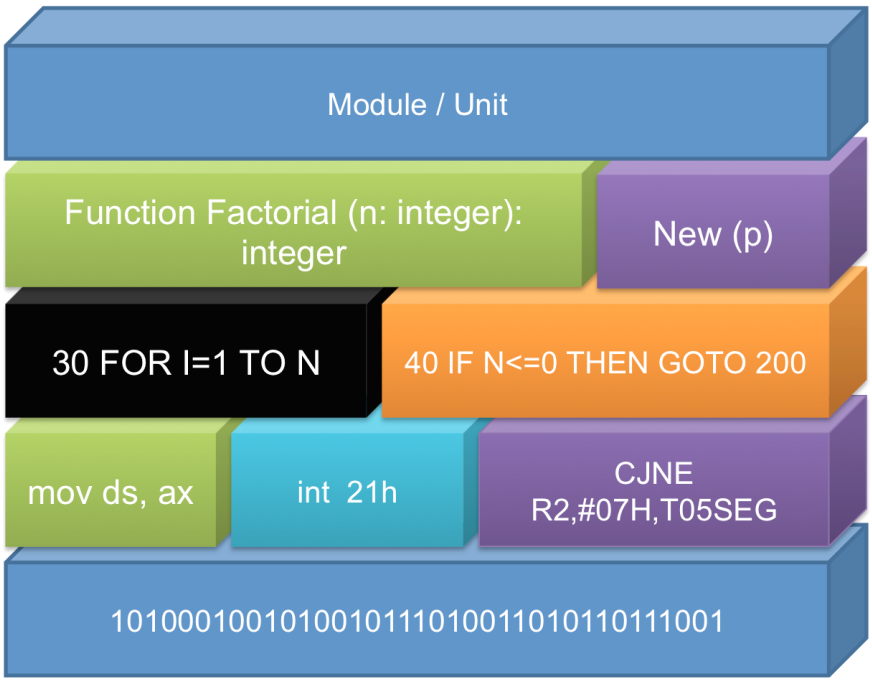
# ¿Con cuál te quedas?





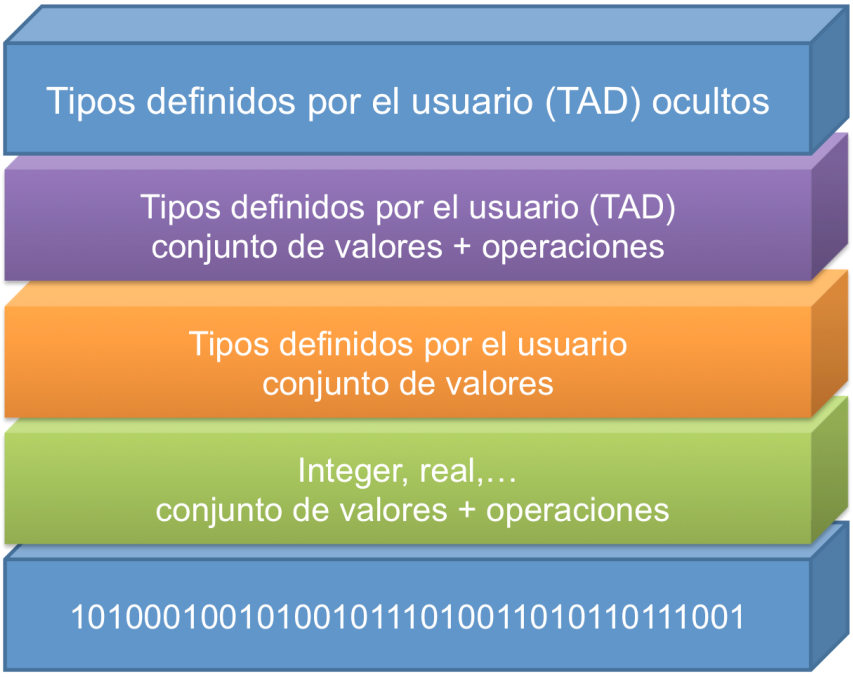


## Niveles de abstracción funcional




## Niveles de abstracción de datos

**Objetivo:** Conseguir que los tipos definidos por el programador se manejen como los incluidos en el propio lenguaje



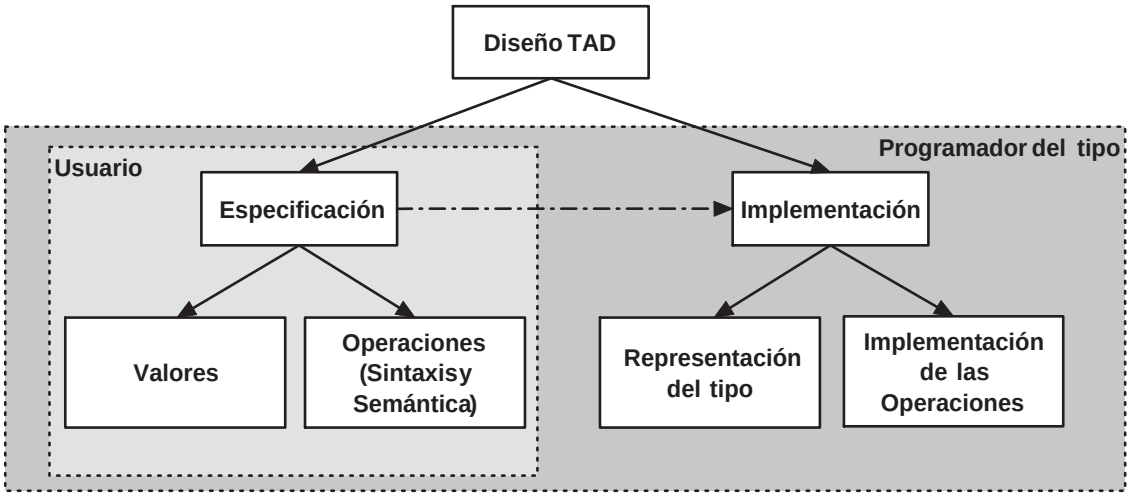
# Definición de Tipo Abstracto de Datos (TAD)

**John Guttag (1974)**:

Un tipo abstracto de datos es aquél **definido por el programador** que puede ser manipulado de forma **similar** a los **definidos por el sistema**.

Al igual que estos últimos, un tipo abstracto de datos corresponde a un **conjunto de valores** lícitos y de **operaciones** asociadas a los mismos, operaciones que se definen mediante una **especificación** que es **independiente de la implementación** de esos datos.

# Niveles de especificación, implementación y uso de un TAD



## Especificación formal de un TAD

**Tipo** *Nombre del TAD*
**Sintaxis**
*Operaciones, tipo argumentos y resultado*
**Semántica**
*Comportamiento de las operaciones*

Hay que tener en cuenta que:

- Las reglas de comportamiento se intentarán en el orden indicado.
- Ciertos operadores no se definen. Pueden considerarse axiomas o *constructores* de los valores del tipo.
- La expresión del resultado puede ser recursiva, conteniendo referencias al mismo operador o a otros del TAD.
- Las expresiones pueden contener referencias a otros tipos ya definidos.

Ejemplo: definición formal del TAD correspondiente a los números naturales

**Tipo** *Natural*
**Sintaxis**

| | |
|---|---|
| cero: | $\rightarrow$ *Natural* |
| sucesor: | *Natural* $\rightarrow$ *Natural* |
| escero: | *Natural* $\rightarrow$ *Boolean* |
| igual: | *Natural* $\times$ *Natural* $\rightarrow$ *Boolean* |
| suma: | *Natural* $\times$ *Natural* $\rightarrow$ *Natural* |

**Semántica**

escero(cero) $\Rightarrow$ true
escero(sucesor($n$)) $\Rightarrow$ false
igual(cero, $n$) $\Rightarrow$ escero($n$)
igual(sucesor($n$), cero) $\Rightarrow$ false
igual(sucesor($n$), sucesor($m$)) $\Rightarrow$ igual($n$,$m$)
suma(cero, $n$) $\Rightarrow$ $n$
suma(sucesor($n$), $m$) $\Rightarrow$ sucesor(suma($n$,$m$))

donde "cero" y "sucesor" son los *contructores* del TAD.

# Especificación informal de un TAD

**TAD Nombre_del_tipo**

**VALORES:** valores que pueden tomar los datos del tipo

**OPERACIONES:** nombre de las operaciones que los manipulan

Para cada operación (SINTAXIS y SEMÁNTICA):

**Nombre_de_operación (tipo_de_argumento) → tipo_de_resultado**

{**Objetivo:** Descripción de la operación

**Entrada:** Descripción de los datos de entrada

**Salida:** Indica qué es lo que retorna la operación al invocarla

**Precondiciones:** Posibles excepciones. Características que tendrán que reunir los datos de entrada para que se realice bien la operación.

**Poscondiciones:** Indica un efecto lateral en la invocación a una función. Afirmaciones que podemos hacer sobre los datos después de que se ejecute la operación y que complete la información del objetivo y las salidas}

# Pasos en la especificación de un TAD T

1. Seleccionar las operaciones, teniendo en cuenta para qué se va a utilizar dicho TAD $T$.
2. Clasificarlas:
   - *Constructoras*: Su resultado es de tipo $T$
     - *Generadoras*: Sólo con ellas es posible generar cualquier valor del tipo $T$ y excluyendo cualquiera de ellas hay valores que no pueden ser generados.
     - *Modificadoras*: El resto
   - *Observadoras/Acceso*: Su resultado no es de tipo $T$.
   - *Destructoras*: Su resultado es de tipo $T$.

## Ejemplo: Especificación del Tipo Abstracto de Datos *Racional*

### VALORES

- Concepto matemático de números racionales, es decir, un par de números enteros tal que el primero es el numerador y el segundo el denominador.

### OPERACIONES (Sintáxis y Semántica)

- Constructoras Generadoras
  - `CreaRacional (n,d:entero)`→ `Racional`
    {*Objetivo*: Crea un número racional
    *Entrada*:
    n: Numerador del nuevo racional
    d: Denominador del nuevo racional
    *Salida*: El número racional creado}

- Constructoras Modificadoras
  - `Suma (r1,r2:Racional)` → `Racional`
    {*Objetivo*: Calcula la suma de dos número racionales
    *Entrada*: r1, r2: Números racionales a sumar
    *Salida*:Un nuevo racional suma de los números a la entrada}

- Observadoras
  - `Numerador (Racional)`→ `Entero`
    {*Objetivo*: Obtiene el numerador de un número racional
    *Entrada*:
    Número racional del que obtener el numerador
    *Salida*: Numerador del número a la entrada}
  - `Denominador (Racional)`→ `Entero`
    {*Objetivo*: Obtiene el denominador de un número racional
    *Entrada*:
    Número racional del que obtener el denominador
    *Salida*: Denominador del número a la entrada}

# Definición Ampliada de Tipo Abstracto de Datos (TAD)

**Ghezzi (1987)**:

Un nuevo tipo de dato se considera un TAD sólo si:

- El lenguaje proporciona algún método para permitir **asociar** la **representación** de los datos con las **operaciones** que los manipulan.
- La representación del nuevo tipo de dato así como la implementación de las operaciones pueden permanecer **ocultas** al resto de los módulos que los utilizan.

Es decir, para construir un tipo abstracto de datos, debemos ser capaces de:

- Exportar una definición de tipo.
- Proporcionar un conjunto de operaciones que puedan usarse para manipular los ejemplares de tipo.
- Proteger los datos asociados con el tipo de tal manera que se pueda operar con ellos sólo mediante las operaciones provistas.

# Estructura de una unit en Pascal

```
unit <nombre_unidad>;

interface

    <clausula uses>

    <constantes, tipos y variables publicas>

    <cabeceras de procedimientos y funciones>


implementation

    <clausula uses>

    <constantes, tipos y variables privadas>

    <procedimientos/funciones privadas>

    <cuerpos procedimientos/funciones publicas>


begin

    <secuencia de inicializacion>

end.
```

## Ejemplo de `Tipo de dato` como `conjunto de valores`

```
program ejemplo1;

type
     Racional = record
        num, den: integer
     end;

var
     r1,r2,r3,r4,s: Racional;

begin
     r1.num:= 2; r1.den:= 3;
     r2.num:= 5; r2.den:= 7;

     r3.num:= 7; r3.den:= 8;
     r4.num:= 5; r4.den:= 0;

     (* s suma de r1 y r2 *)
     s.num:= r1.num * r2.den + r2.num * r1.den;
     s.den:= r1.den * r2.den;
     writeln('la suma es ', s.num,'/',s.den);

     (* s suma de r3 y r4 *)
     s.num:= r3.num * r4.den + r4.num * r3.den;
     s.den:= r3.den * r4.den;
     writeln('la suma es ', s.num,'/',s.den);
end.
```

## Ejemplo de Tipo de dato como conjunto de valores y operaciones

```
program ejemplo2;
type
    Racional = record
        num, den:integer
    end;
var
    r1,r2,r3, r4, s:Racional;

    function CreaRacional (n,d:integer): Racional;
    (* Operacion para crear un racional *)
    var
        temp: Racional;
    begin
        temp.num:= n; temp.den:= d;
        CreaRacional:= temp;
    end;

    function Numerador (r:Racional): integer;
    (* Operacion que retorna el numerador de un racional *)
    begin
        Numerador:= r.num
    end;

    function Denominador (r:Racional): integer;
    (* Operacion que retorna el denominador de un racional *)
    begin
        Denominador:= r.den
    end;

    function Suma (r1,r2:Racional): Racional;
    (* Operacion que retorna la suma de dos racionales *)
    var
        s: Racional;
    begin
        s.num:= r1.num * r2.den + r2.num * r1.den;
        s.den:= r1.den * r2.den;
        Suma:= s;
    end;

begin
    r1:= CreaRacional(2,3); r2:= CreaRacional(5,7);
    r3:= CreaRacional(7,3); r4:= CreaRacional(5,4);

    s:= Suma(r1,r2); writeln('La suma es ', Numerador(s),'/',Denominador(s));
    s:= Suma(r3,r4); writeln('La suma es ', Numerador(s),'/',Denominador(s));
end.
```

## Encapsulando y compilando de forma independiente...utilizando como estructura de datos un registro...

```
Unit UnitRacional;
Interface
type
    Racional = record
        num,den: integer
    end;

    function CreaRacional (n,d: integer): Racional;
    function Numerador (r: Racional): integer;
    function Denominador (r: Racional): integer;
    function Suma (r1,r2: Racional): Racional;

Implementation
    function CreaRacional (n,d: integer): Racional;
    (* Operacion para crear un racional *)
    var
        temp: Racional;
    begin
        temp.num:= n; temp.den:= d;
        CreaRacional:= temp;
    end;

    function Numerador (r: Racional): integer;
    (* Operacion que retorna el numerador de un racional *)
    begin
        Numerador:= r.num
    end;

    function Denominador (r: Racional): integer;
    (* Operacion que retorna el denominador de un racional *)
    begin
        Denominador:= r.den
    end;

    function Suma (r1,r2: Racional): Racional;
     (* Operacion que retorna la suma de dos racionales *)
     var
        s: Racional;
     begin
        s.num:= r1.num * r2.den + r2.num * r1.den;
        s.den:= r1.den * r2.den;
        Suma:= s;
     end;
end.
```

## ...utilizando como estructura de datos un puntero a un registro...

```
Unit UnitRacional;

Interface
    type
        Racional= ^datos;
        datos= record
            num, den:integer
        end;

    function CreaRacional (n,d: integer): Racional;
    function Numerador (r: Racional): integer;
    function Denominador (r: Racional): integer;
    function Suma (r1,r2: Racional): Racional;

Implementation
    function CreaRacional (n,d: integer): Racional;
    var
        temp: Racional;
    begin
        new(temp);
        temp^.num:= n;
        temp^.den:= d;
        CreaRacional:= temp;
    end;

    function Numerador (r:Racional): integer;
    begin
        Numerador:= r^.num
    end;

    function Denominador (r: Racional): integer;
    begin
        Denominador:= r^.den
    end;

    function Suma (r1,r2: Racional): Racional;
    var
        s: Racional;
    begin
        new(s);
        s^.num:= r1^.num * r2^.den + r2^.num * r1^.den;
        s^.den:= r1^.den * r2^.den;
        Suma:= s;
    end;
end.
```

## ...el programa que utiliza el tipo de datos es independiente de la representación del tipo de dato

```
program ejemplo3;

uses UnitRacional;
(*uso del tipo de dato Racional*)

var
   r1,r2,s: Racional;

begin
   r1:= CreaRacional(2,3); r2:= CreaRacional(5,7);

   (* s es la suma de r1 y r2 *)
    s:= Suma(r1,r2);
    writeln('La suma es ', Numerador(s),'/',Denominador(s));
end.
```

# Ventajas del uso de un Tipo Abstracto de Datos (TAD)

1. Recogen mejor la semántica de los tipos. Al agrupar la representación junto a las operaciones que definen su comportamiento, y forzar a utilizar el TAD a través de estas operaciones se evitan errores en el manejo del tipo de datos (Por ejemplo, la división por 0 en el caso del TAD *Racional*).
2. Abstracción: Separa la especificación (**qué hace**) de la implementación (**cómo lo hace**). Los usuarios de un TAD no necesitan conocer sus detalles de implementación. Como consecuencia:
   *a*) Se favorece la extensibilidad del código: Es posible *modificar* y *mejorar* la implementación del TAD sin afectar a los demás módulos que lo utilizan.
   *b*) Aumenta la facilidad de uso.
   *c*) Aumenta la legibilidad del código que usa el TAD.
3. Produce código reutilizable.
4. Favorece la ausencia de errores, al reutilizar código ya probado y forzar a utilizar la estructura de datos correctamente.