

# Programación con Tipos Abstractos de Datos

---

Tema I

# Bibliografía

- Capítulos 1 y 2 del libro “X. Franch: *Estructuras de datos. Especificación, diseño e implementación*, 3ª edición, Ediciones UPC, 2001”.
- Capítulos 1 y 2 del libro “Z.J. Hernandez, etc: *Fundamentos de Estructuras de Datos. Soluciones en Ada, Java y C++*. Thomson, 2005”
- Capítulos 1 y 2 del libro “Joyanes, L., Zahonero, I., Fernández, M. y Sánchez, L.: *Estructuras de datos. Libro de problemas*. McGraw Hill, 1999.”,
- Capítulos 1 y 2 del libro “Martí Oliet, N., Ortega Mallén, Y., Verdejo López, J.A.: *Estructuras de datos y métodos algorítmicos*. 2ª edición: 213 ejercicios resueltos. Garceta, 2013”,
- Tema I del libro de apuntes “Campos Laclaustra, J.: *Apuntes de Estructuras de Datos y Algoritmos, segunda edición (versión 4)*, 2022.
- Capítulos 1 y 3 del libro “Weiss, M.A.: *Data Structures and Algorithm Analysis in C++*, 4th Edition, Pearson/Addison Wesley, 2013.”
- Capítulo 1 del libro “Shaffer, Clifford A.: *Data Structures and Algorithm Analysis, C++ version of Edition 3.2. (Last updated: 03/28/2013)*”  
<http://people.cs.vt.edu/shaffer/Book/C++3elatest.pdf>
- *Y otros....*



# Lecciones

1. Tipos Abstractos de Datos (TADs)
2. Especificación de TADs
3. Implementación de TADs
4. TADs genéricos
5. TADs fundamentales

# Tipos Abstractos de Datos (TADs)

---

## Lección 1

# Esquema

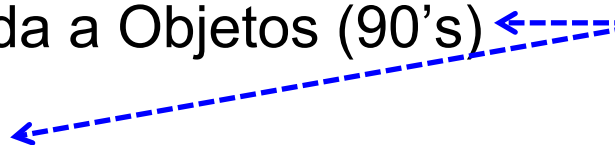
- Paradigmas de programación
- Definición de TAD
- Programación con TADs
- Ventajas de la programación con TADs

# Paradigma de programación



- *Conjunto de teorías, estándares y métodos que en conjunto representan un medio de organización del conocimiento*
- Múltiples paradigmas y clasificaciones de paradigmas
- Evolución histórica del **paradigma imperativo**:
  - Programación Imperativa (época “caótica” hasta los 60’s)
  - Programación Estructurada<sup>(1)</sup> (auge 70’s)
  - Programación Modular (80’s)
  - Programación Orientada a Objetos (90’s)
  - ....
- Paradigma declarativo (*Prog. Lógica, Prog. Funcional, ...*) tienen su propia evolución pasando por modulares, orientados a objetos,...

*Más en la  
asignatura:  
“Tecnología de  
Programación”*



# Diseño descendente

*{Abstracción: método de resolución de problemas consistente en destacar los detalles importantes, y evitar (o retrasar) fijarse en los irrelevantes}*



- Diseño mediante abstracción y refinamientos sucesivos:
  - Se basa en la abstracción de acciones
  - Ineficiente para programación a gran escala
- Inconvenientes:
  - Acciones abstractas (proced. y funciones) de alto nivel, que manipulan datos concretos o de bajo nivel
  - Fuerza a seleccionar la representación de los datos desde el principio → mejor retrasarlo hasta identificar qué operaciones serán necesarias

**PROGRAMAS = DATOS + ALGORITMOS**

---

**PROGRAMAS = DATOS + (ALG. DATOS + ALG. CONTROL)**

# Diseño modular



- Ampliación/evolución del diseño descendente
- División en módulos:
  - Cada uno con una misión bien determinada, interacción mínima y clara con el resto, que puedan ser desarrollados de forma independiente, reutilizables, etc.
  - Todos juntos solucionan el problema inicial
- Requisitos de la descomposición en módulos:
  - Cada módulo debe tener una conexión (dependencia) mínima con el resto → *interfaz*
    - Garantizar la independencia en el desarrollo
    - Reutilización
  - Cambios y mejoras deben afectar sólo a un número pequeño de módulos
  - Tamaño adecuado (ni muy grande, ni muy pequeño)

# Tipos Abstractos de Datos

- Con los lenguajes de programación estructurados (años 60) surge el concepto de **tipo de datos** (*conj de valores que sirve de dominio de ciertas operaciones*)<sup>(1)</sup> .
- Ese concepto es insuficiente para software a gran escala: sólo el compilador restringe el uso de los datos.
- En los 70 aparece el concepto de **TAD**: un tipo de datos no sólo es el conjunto de valores, sino también sus operaciones con sus propiedades  
→ determinan inequívocamente su comportamiento.

TAD = valores + operaciones

---

(1) Ver transparencias sobre vectores, registros, etc., al final de la lección

# Tipos Abstractos de Datos

- El concepto de TAD ya existe en los lenguajes de programación estructurados: **los tipos predefinidos**.
  - No hace falta saber nada sobre su implementación para usarlos

## Ejemplo:

Definición del tipo de datos *int* de los enteros en C++:

- valores: los del intervalo [mínimo valor entero, máximo valor entero]  
En C++ [std::numeric\_limits<int>::min(), std::numeric\_limits<int>::max() ]
- operaciones: +, -, \*, /, %, ... abs(), sqrt()...
- propiedades de las operaciones:  $a+b=b+a$ ,  $a+0=a$ ,  $a*0=0$ ...

Nota:

- Definición del tipo de datos de los enteros en JAVA:
  - valores: los del intervalo [Integer.MIN\_VALUE, Integer.MAX\_VALUE ]
  - operaciones: +, -, \*, /, resto, módulo, valor absoluto, exp...
  - propiedades de las operaciones:  $a+b=b+a$ ,  $a+0=a$ ,  $a*0=0$ ...
- Definición similar en el lenguaje ADA pero con valores en el intervalo [INTEGER'FIRST, INTEGER'LAST]

# Definición de TAD

- Un **Tipo Abstracto de Datos** es un conjunto de valores y de operaciones definidos mediante una **especificación independiente de cualquier representación**.

TAD = valores + operaciones

- El **uso** de un TAD sólo depende de su especificación (conjunto de valores, operaciones y sus propiedades), **nunca** de su implementación
  - Ej: para manipular los enteros nos olvidamos de cómo se representan los valores y de cómo están implementadas las operaciones.
- Construiremos nuevos tipos de datos, robustos, reutilizables, y eficientes, para ser usados de forma similar a los predefinidos en un lenguaje de programación, pero de naturaleza mucho más compleja que los tipos predefinidos

# Tipos Abstractos de Datos (TADs)

- Ejemplos de TADs:
  - Los números complejos con las operaciones de suma, producto, parte real y parte imaginaria
  - Las fechas tal como las usamos habitualmente...

¿Cómo los **especificamos**?

**TAD = valores + operaciones**

Garantizar separación:

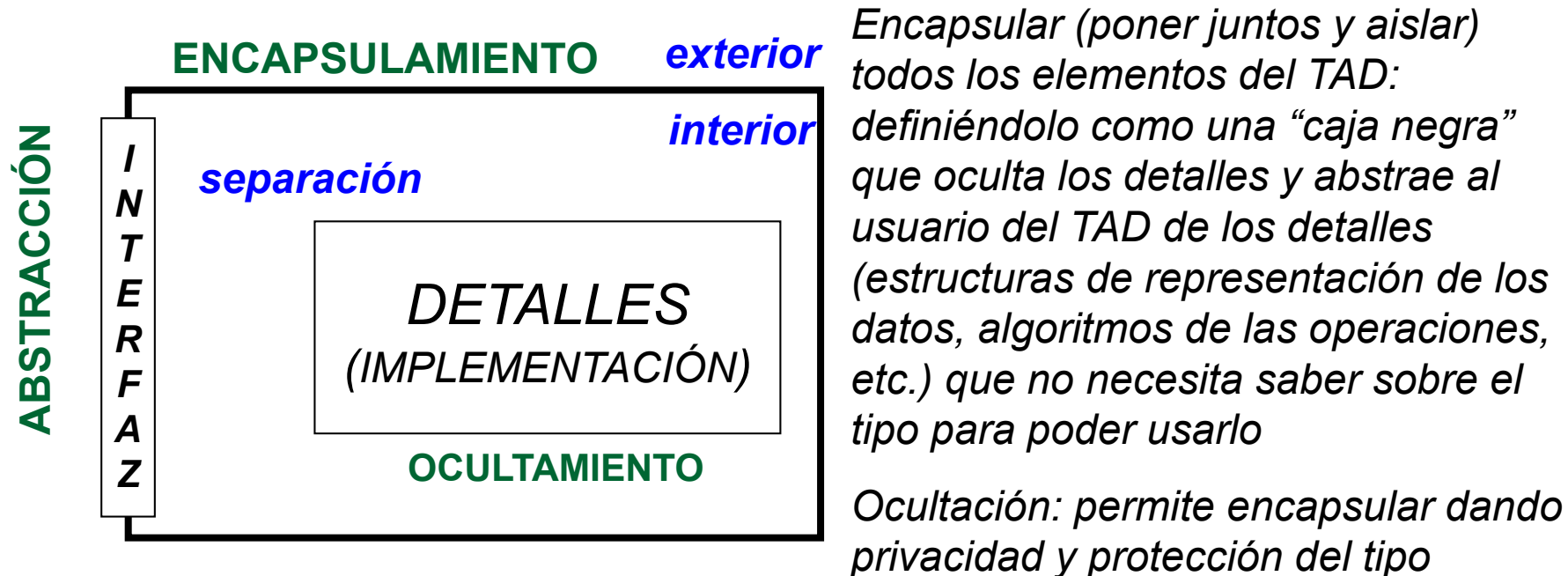
**uso** de un TAD **versus** **representación** de un TAD

*... Evolución de los lenguajes de programación...*

---

# Tipos Abstractos de Datos (TADs)

- El uso de un TAD sólo depende de su especificación (conjunto de valores, operaciones y sus propiedades), **nunca** de su representación
- Separar estrictamente la representación de un tipo de dato, del uso del tipo de datos → **abstracción, encapsulación y ocultación**

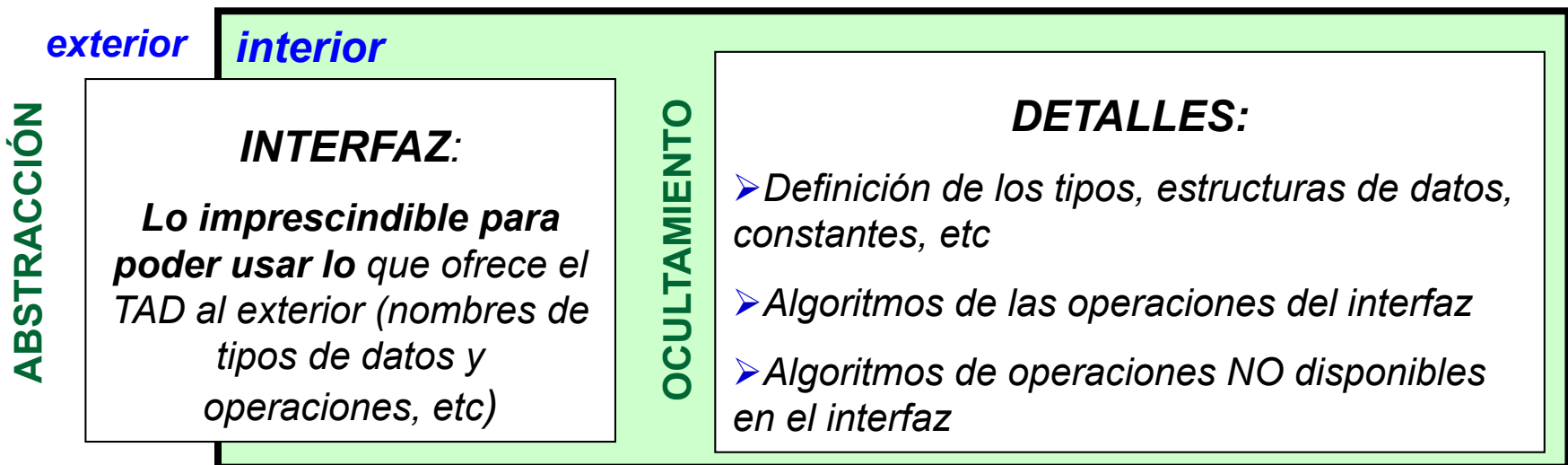


- **Encapsular** adecuadamente facilita:
  - la reutilización del TAD en distintos programas
  - la modificación de la implementación, sin afectar a los programas que usen el TAD

# Tipos Abstractos de Datos (TADs)

- Para el **usuario** de un TAD:
  - el tipo consta del conjunto de valores y del conjunto de operaciones para la creación, modificación y manipulación de los valores del tipo → **especificación e interfaz**
- La única forma de **usar** los datos del tipo abstracto será invocando las operaciones de la **interfaz**.
  - Para usar un tipo no hace falta saber el detalle de su implementación sino sólo su especificación e interfaz
  - No se podrá acceder a la representación interna del tipo de dato

## ENCAPSULAMIENTO



# Programación con TADs

PROGRAMAS = DATOS + ALG. DATOS + ALG. CONTROL

Diseño  
Modular

-----> PROGRAMAS = TADs + ALG. CONTROL

## Pasos en la programación con TADs:

1. Especificación del tipo → resultante de lo identificado en la **fase de diseño**

Lección 2

- Las operaciones necesarias para el TAD se conocerán tras haber diseñado el programa/s usuarios del TAD

2. Implementación del tipo

Lección 3

- Al implementar el TAD se buscará la máxima eficiencia (se conocen las operaciones a ofrecer)

3. Uso del tipo (*objetivo: reutilización*)

- uso concreto y/o usuarios pueden no ser conocidos en los pasos anteriores...

## Programación con TADs:

En programación modular el TAD se **encapsula** en un **módulo**

En programación Orientada a Objetos (OO) el TAD se **encapsula** en una **clase**

**En esta asignatura no vamos a utilizar OO, se estudiará en “Tecnología de programación”**

# Programación Orientada a Objetos

- La Programación OO incorpora los conceptos de **abstracción**, **encapsulación** y **ocultamiento**,....
- Basada en un nuevo concepto:  
objeto → entidad de un tipo abstracto de datos con estado (atributos o campos) y comportamiento (operaciones o métodos) propios
- Clase:
  - categoría de objetos con propiedades y operaciones comunes
  - encapsula el tipo de datos junto con sus operaciones, ocultando los detalles internos
- **Programación Orientada a Objetos=**
  - Soporte sintáctico explícito para abstracción de datos (TADs) +
  - Cambio de perspectiva (los programas son apéndices de los datos) +
  - Prestaciones asociadas a la jerarquía de clases (herencia, polimorfismo, etc.)

# Programación con TAD's

- **Especificación e implementación de TADs:**
  - Dada una *especificación* de un TAD, podremos tener diferentes *implementaciones* del mismo TAD
    - Siempre que respeten la misma *especificación*, las *implementaciones* serán intercambiables de forma transparente para los usuarios (programas) del TAD
  - Dada una *especificación* de un TAD, se podrán *implementar* de forma independiente, simultáneamente y por separado<sup>(1)</sup>:
    - a. Los algoritmos que implementan las operaciones del TAD
    - b. los algoritmos que usan el TAD, conociendo únicamente su **interfaz** y la descripción del comportamiento esperado de las operaciones

---

<sup>(1)</sup> En los lenguajes de programación modular se incorpora la compilación separada de los módulos

# Criterios de calidad del software

**Aplicables al desarrollo de software a todos los niveles** (desde trozos pequeños de código, bibliotecas de código, grandes proyectos... con matices en su significado):

- **Corrección** → debe realizar exactamente las tareas definidas por su especificación
- **Legibilidad** → debe ser fácil de leer y lo más sencillo posible. Contribuyen la abstracción y la codificación con comentarios, sangrados, etc.
- Extensibilidad → facilidad para adaptarse a cambios en su especificación
- **Robustez** → capacidad de funcionar correctamente incluso en situaciones anormales
- **Eficiencia** → hacer un buen uso de los recursos, tales como el tiempo, espacio (memoria y disco), etc.
- Facilidad de uso → la utilidad de un sistema está relacionado con su facilidad de uso
- Portabilidad → facilidad con la que puede ser transportado a diferentes sistemas físicos o lógicos
- Verificabilidad → facilidad de verificación de un software, su capacidad para soportar los procedimientos de validación, juegos de test, ensayo o pruebas
- **Reutilización** → capacidad de ser reutilizados en nuevas aplicaciones o desarrollos
- **Integridad** → capacidad de proteger sus propios componentes frente a accesos o usos indebidos
- Compatibilidad → facilidad para ser combinados con otros y usados en diferentes plataformas hardware o software

*Normalmente el objetivo será alcanzar un equilibrio entre ellos*

# Ventajas Programación con TADs

Ventajas en el diseño, implementación y uso

- Abstracción:
  - la complejidad del problema se diluye. Los módulos en los que se descompone el problema serán de menor complejidad
  - Se pueden implementar los TAD's sólo a partir de la especificación, sin saber para qué se van a usar → *Reusabilidad*
- Corrección:
  - los TADs pueden ser desarrollados y probados de forma independiente
  - Se pueden utilizar los TAD's sólo conociendo la especificación. Facilita la integración de módulos.
- Eficiencia:
  - La implementación puede retrasarse hasta conocer las restricciones de eficiencia sobre sus operaciones
  - para un TAD podemos contar con diferentes implementaciones válidas y optar por la más eficiente y adecuada a las restricciones a cumplir
- Legibilidad:
  - La especificación de un TAD es suficiente para entender su significado y comportamiento
  - Un TAD tiene un tamaño y complejidad acotados que facilita su legibilidad

# Ventajas Programación con TADs

Ventajas en el diseño, implementación y uso

- Modificabilidad y mantenimiento:
  - Tanto durante el periodo de desarrollo y pruebas, como durante en el periodo de mantenimiento y de vida del software
  - Cambios localizados y acotados
  - Cambios que no afecten a la especificación no afectarán a los programas que usen el TAD
- Organización:
  - Facilita el reparto de tareas y la comunicación en un grupo de programadores
  - El equipo desarrolla en paralelo las múltiples partes o módulos del sistema
- Reusabilidad:
  - TADs reutilizables en otros contextos con pocos o ningún cambio (¡escoger bien el conjunto de operaciones!)
- Seguridad:
  - imposibilidad de manipular directamente la representación interna de los datos u objetos del tipo
  - Impide el mal uso y la generación de valores incorrectos



# Vectores y registros

- En los lenguajes de **programación estructurada** el programador tiene la posibilidad de definir nuevos tipos de datos:
  - Tipos simples
    - ejemplo de tipo enumerado:                      tipo semáforo= (rojo, amarillo, verde)
  - Tipos compuestos: sirven para agrupar información compuesta por partes o datos de naturaleza más sencilla, pero que tienen sentido como un todo
    - VECTORES:
      - Agrupan múltiples datos o información, pero todos ellos del mismo tipo
      - Cada parte suele recibir el nombre de *componente* y se identifica mediante un valor de índice
      - El índice deberá ser un tipo enumerable
      - Se manipulan como un todo, o cada parte individualmente utilizando un valor de índice para identificar la componente a utilizar
    - REGISTROS:
      - Agrupan múltiples datos o información, que puede ser de diferentes tipos
      - Cada parte suele recibir el nombre de *campo* y se identifica mediante un nombre dado al campo en la definición del tipo
      - Se manipulan como un todo o cada parte individualmente utilizando el nombre del campo

# Vectores y registros - Ejemplos

- Definición de tipos:

## tipos

*{Vector de enteros indexado por letras mayúsculas:}*

vEnteros= **vector** ['A'..'Z'] de entero;

*{Registro de persona:}*

persona = **registro**

    nombre,apellidos: cadena;

    edad: entero;

    altura: real

**freg;**

*{Vector de personas:}*

vPersonas= **vector** [1..100] de persona;

- Declaración de variables:

## variables

v:vEnteros;            p1,p2:persona;    miClase:vPersonas;

*Disponible en el material de  
clase:*

*documento resumen (chuleta)  
de la notación algorítmica o  
seudocódigo que se utilizará  
en clase*

# Vectores y registros - Ejemplos

- Declaración de variables:

**variables** v:vEnteros; p1,p2:persona; miClase:vPersonas;

- Uso de variables:

p1:=("Jorge"; "Perez", 18, 1.80);

p2:=p1;

p2.nombre:="Antonio"; *{Acceso a campos de un registro con notación de punto}*

p2.altura:=p2.altura+0.10;

Escribir(p1.nombre," ",p1.apellidos," de ", p1.edad," años y ", p1.altura, " cm de altura"); saltarLinea; *{escribirá en pantalla: Jorge Perez de 18 años y 1.80 cm de altura}*

Escribir(p2.nombre," ",p2.apellidos," de ", p2.edad," años y ", p2.altura, " cm de altura"); saltarLinea; *{escribirá en pantalla: Antonio Perez de 18 años y 1.90 cm de altura}*

miClase[1]:=p2; MiClase[2]:=p1;

**para** i:=1 **hasta** 2 **hacer**

    escribir(miClase[i].nombre," ", miClase[i].apellidos," de ");

    escribir(miClase[i].edad," años y ", miClase[i].altura, " cm de altura"); saltarLinea;

**fpara**

*{escribirá en pantalla:*

*Antonio Perez de 18 años y 1.90 cm de altura*

*Jorge Perez de 18 años y 1.80 cm de altura*

*}*

- volver