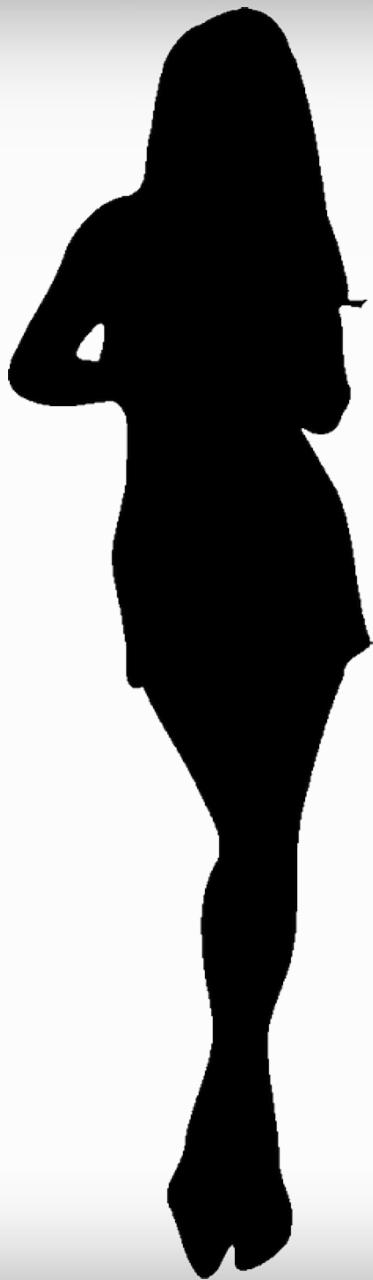


Estructuras de Datos

**Prof. Jaime Dávila
Ingeniería de Sistemas
Universidad de Nariño**

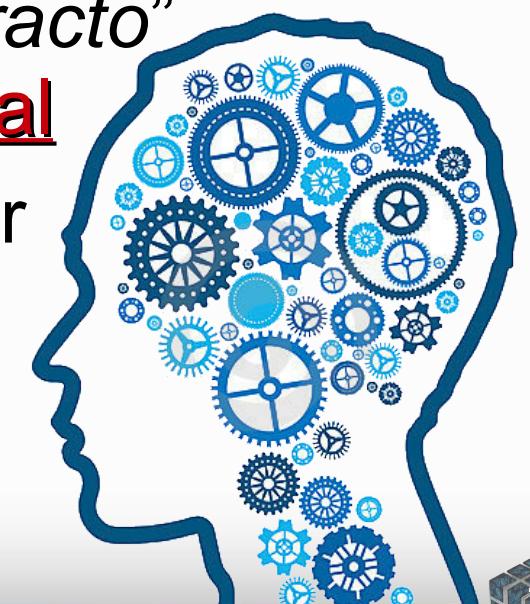
La Abstracción

¿Qué es la Abstracción?



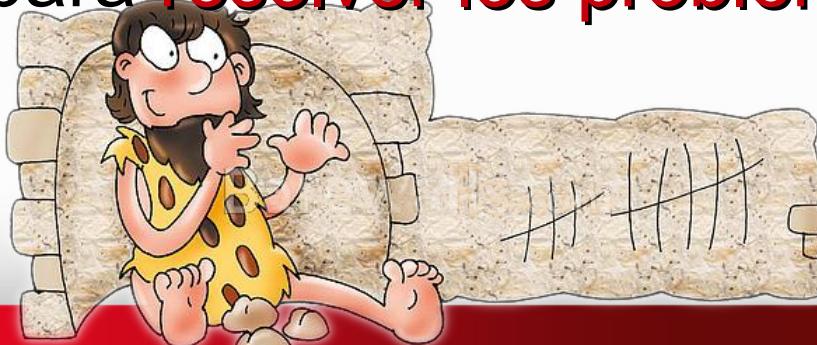
¿Qué es la Abstracción?

- Es la capacidad para **encapsular** y **aislar** la **información**, del **diseño** y **ejecución**
- Es la **clave** para resolver **problemas** e implica:
 - **Identificar** elementos importantes del problema
 - **Nombrarlos**
 - **Plantear** y **resolver** el problema “*abstracto*” y **adaptar la solución** al **problema real**
- La **abstracción** es la clave para diseñar buen SW

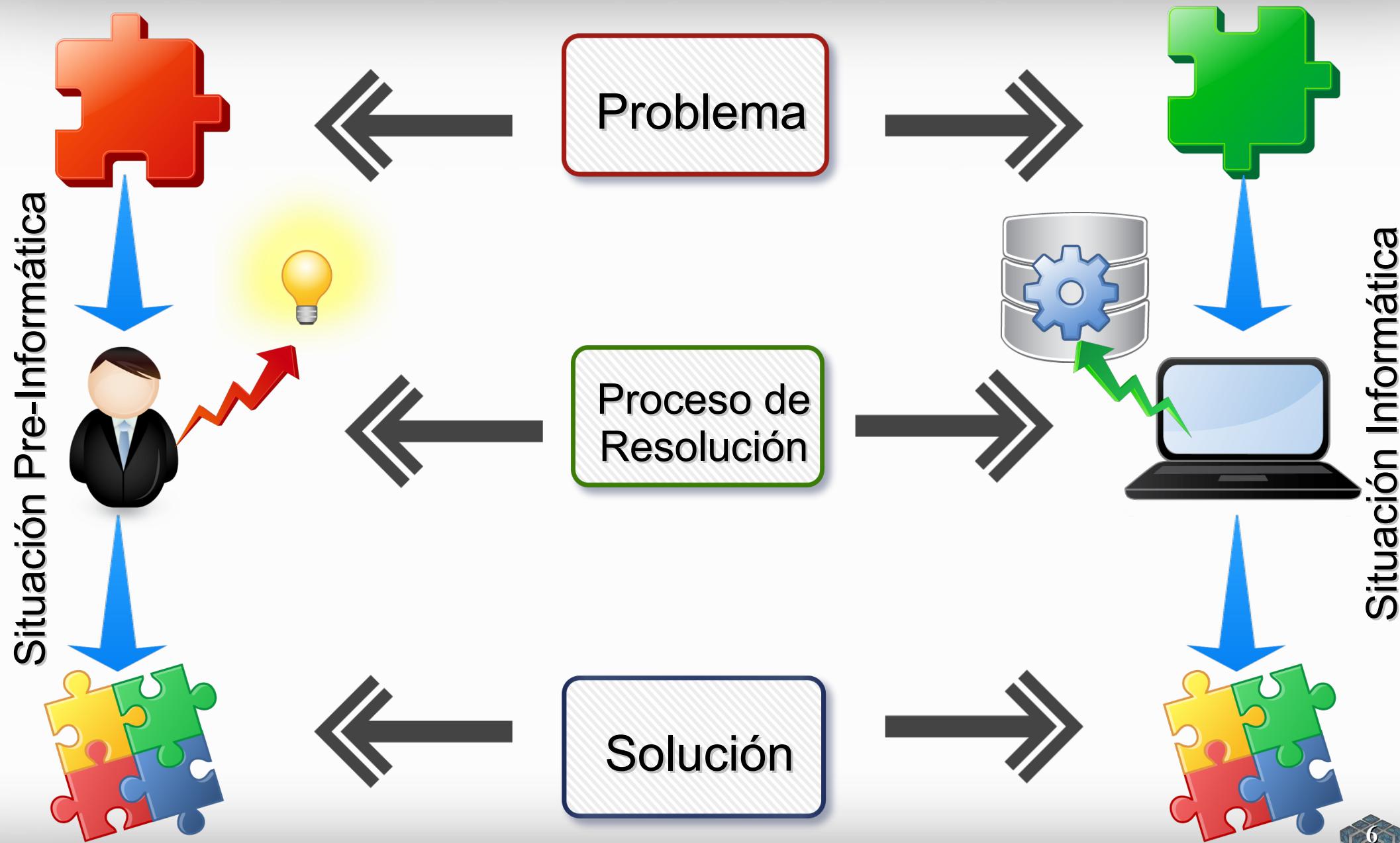


El papel de la Abstracción

- “Los humanos hemos desarrollado una técnica excepcionalmente potente para tratar la complejidad: abstraernos de ella. Incapaces de dominar en su totalidad los objetos complejos, se ignora los detalles no esenciales, tratando en su lugar con el modelo ideal de objeto y centrándonos en los aspectos esenciales” (Wulft)
- Las personas construimos **modelos** (abstracciones) mentales para **comprender el mundo** y nos servimos de ellos para **resolver los problemas**.



Resolución de Problemas



Diseño basado en Abstracciones

- Programas de cierto **tamaño** y **complejidad** pueden ser difíciles de manejar
- **Modularización** y **diseño descendente** (o “**top-down**”) aportan una solución y está asociado al proceso de refinamiento
- Un programa lo **dividimos** en tareas, que a su vez se dividen en sub-tareas,...



Esfuerzo(t1,t2) > Esfuerzo(t1) + Esfuerzo(t2)

Tipos de Abstracciones

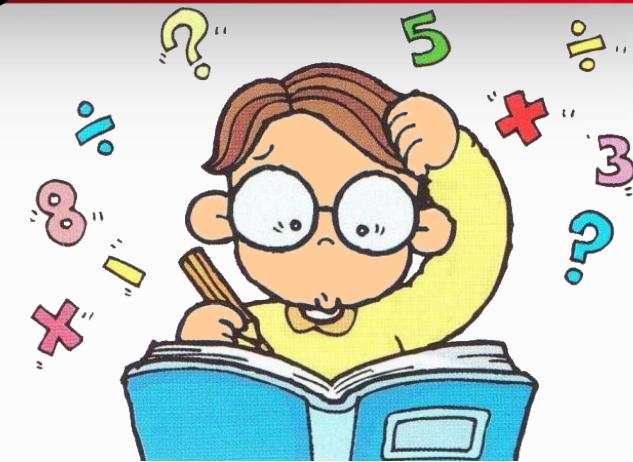
- Abstracción **funcional** o **procedimental**: usuario sólo necesita la especificación (**qué hace**) para utilizarlo, ignorando los detalles (**cómo está hecho** o **cómo se Hace**).
- Abstracción de **datos**:
 - ▶ Tipos de datos: representación irrelevante para el usuario
 - ▶ Tipos definidos por el programador: definición de valores cercanos al problema (typedef)
 - ▶ Tipos abstractos de datos (TAD's): definición y representación de tipos de datos (valores y operaciones)
 - ▶ Clases de Objetos: Extensión de los TAD's para soportar herencia, polimorfismo.



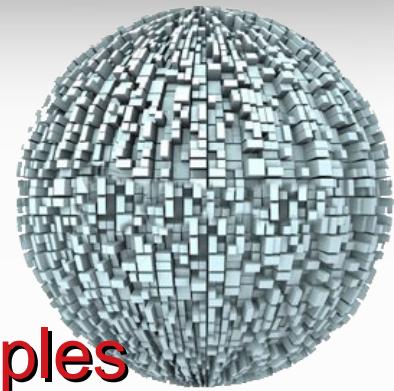
Tipos de Datos y Estructuras de Datos

Tipos de Datos

- Es un **conjunto de valores** y un **conjunto de operaciones** definidas sobre esos valores.
- Cada Lenguaje de Programación incorpora:
 - ◆ **Tipos de datos fundamentales** (primitivos, atómicos): No se construyen a partir de otros tipos y son **entidades únicas**, que no se pueden descomponer en otros. **Ejm:** enteros, decimales (*flotantes*), carácter, booleano.
 - ◆ **Tipos compuestos**: Son tipos de datos cuyos valores constan de **colecciones de elementos** de datos previamente definidos.
 - ◆ **Ejm:** arreglos, cadenas, secuencias, estructuras, **clases**.



¿Estructuras de Datos?



- **Agrupación** de **datos** (atómicos y compuestos) que se trata como una **unidad** en su conjunto.
- Se **construyen** a partir de los tipos de **datos simples**
- **Clasificación:**
 - Por su **naturaleza**:
 - ◆ Homogénea: Todos del mismo tipo (ej: vectores, tablas, matrices n dimensionales).
 - ◆ Heterogénea: De diferente tipo (ej: registros).
 - Por su forma de **almacenamiento**:
 - ◆ Memoria central:
 - Estática. Tamaño conocido a priori
 - Dinámica. El tamaño varía durante la ejecución del programa (ej: listas, árboles, grafos)
 - ◆ Memoria externa (archivos y bases de datos)

Estructuras de Datos Estáticas

- Homogéneas:

- **Unidimensionales** (vectores)

- Modelo:

indices	0	1	2	3	4
valores	50	25	39	8	16

- Sintaxis:

C y C++	Java
<code><tipoDato> <nombreVector> [] = {d1, d2, d3,..., dn};</code>	<code><tipoDato> []<nombreVector>;</code>
<code>char vocales [] = {'a', 'e', 'i', 'o', 'u'};</code>	<code>char [] nombre;</code>
<code><tipoDato> <nombreVector> [<tamaño>]</code>	<code><tipoDato> [] <nombreVector> = new <tipoDato> [<tamaño>];</code>
<code>int edades [5];</code>	<code>char [] nombre = new char [20];</code>
Acceso:	
<ul style="list-style-type: none"> • Al conjunto (referencia): <code><nombreVector></code> • A un elemento: <code><nombreVector> [<indice>]</code> 	

Estructuras de Datos Estáticas

- Homogéneas:

- **Bidimensionales** (matrices o tablas)

- Modelo:
- Sintaxis:

		Indice Columna				
		Indice Columna	0	1	2	3
Indice Fila	0	1,8	2,0	1,7	1,4	
	1	1,5	1,8	1,7	1,9	
	2	1,9	1,6	1,4	1,6	

C y C++

```
<tipoDato> <nombreMatriz> [ ][ ] =  
{ {d1,d2,d3}, {d4,d5,d6} };
```

```
int mt [ ][ ] = {{10, 20},{30, 40},{50, 60}};
```

```
<tipoDato> <nombreMatriz> [<tam_fil>]  
[<tam_col>]
```

```
float estaturas [3][4];
```

Java

```
<tipoDato> [ ][ ]<nombreMatriz>;
```

```
char [ ][ ] nombres;
```

```
<tipoDato> [ ][ ]<nombreMatriz> =  
new <tipoDato> [<tam_fil>]  
[<tam_col>];
```

```
char [ ][ ] nombres = new char [3][20];
```

Acceso:

- Al conjunto (referencia): <nombreMatriz>
- A un elemento: <nombreMatriz> [<ind_fil>] [<ind_col>]

Estructuras de Datos Estáticas

- **Heterogéneas** (Registro *(Es el tipo de dato más próximo a la idea de Clase/Objeto)*):
 - Contiene elementos de datos **fundamentales** y **compuestos**, llamados campos.

- Modelo:

codigo → nombre → año_nac → carrera

Registro Estudiante:	81256	Juan Pérez	2003	Ing. Sistemas
----------------------	-------	------------	------	---------------

- Sintaxis:

C y C++

```
struct <nombreRegistro>{
    <tipoDato> campo1;
    <tipoDato> campo1;
    ...
    <tipoDato> campoN;
};
```

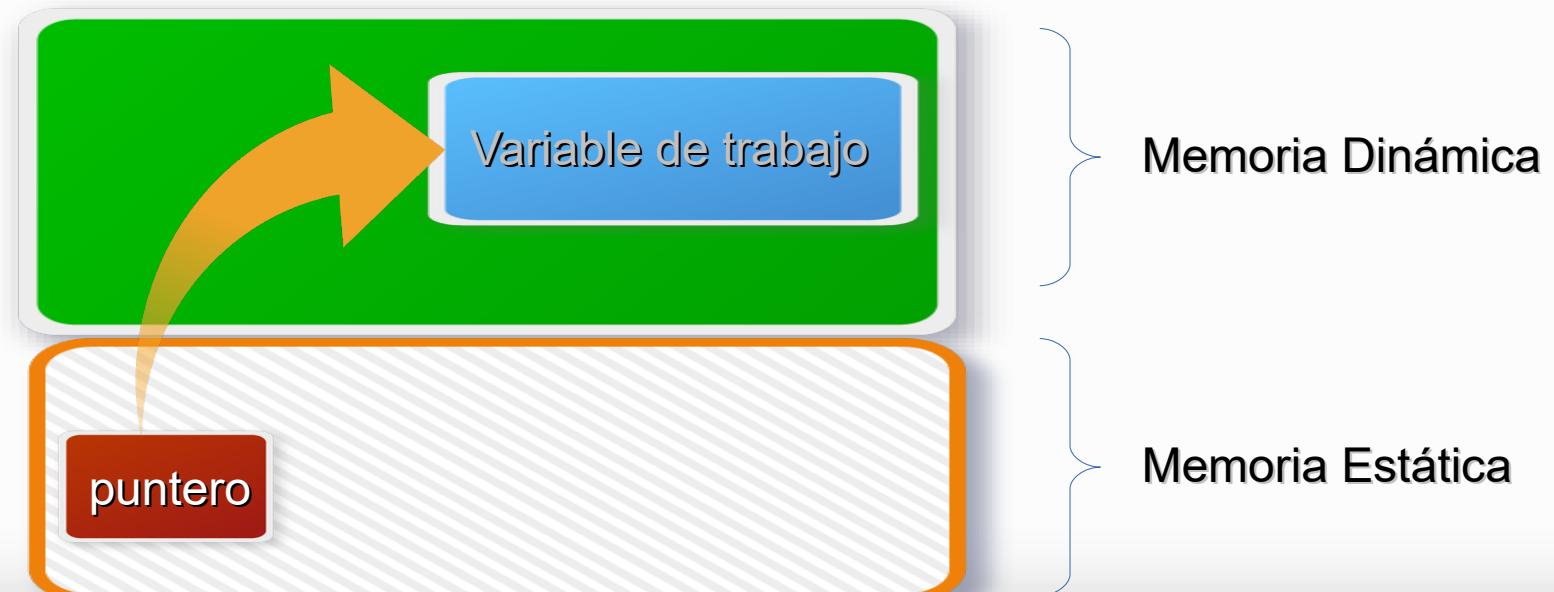
```
struct estudiante {
    char codigo[5];
    char nombre[15];
    int año_nac;
    char carrera [20];
};
```

Acceso:

- Al conjunto (referencia): <nombreRegistro>
- A un elemento: <nombreRegistro>.<nombreCampo>

Estructuras de Datos Dinámicas

- Se basa en el concepto de **puntero** (pointer)
- **Puntero**: **variable estática** cuyo **contenido** (*referencia*) es la **dirección de una región de la memoria dinámica** (*nodo*) que contiene el **dato**.
- Mediante un **puntero** se puede (además de acceder al *contenido de su nodo*) **reservar** y **liberar espacio**.



Estructuras de Datos Dinámicas

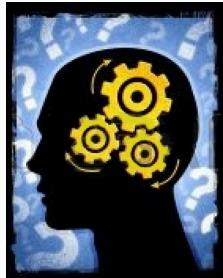
- En **Python** los punteros apuntan (*referencian*) a **objetos**. En otros lenguajes (C o Pascal) pueden apuntar a estructuras de datos (struct).
- Sintaxis en **Python**:
 - **Crear** una nueva referencia (puntero):
`<variableRef1> = ClaseDA([argumentos])`
 - **Eliminar** una variable con referencia: **innecesario** en
 - **Asignación**: `<variableRef2> = <variableRef1>`
La referencia `<variableRef1>` se copia en `<variableRef2>`
 - **Comparación**: `<variableRef1> is <variableRef2>`
Devuelve un valor booleano (True, False) en función de que ambas referencias sean iguales o no.
 - Referencia “especial”: **None**
No apunta a ningún sitio (no hay reserva de memoria)



Etapas en la selección de una ED

Los pasos a seguir para **seleccionar** una **estructura de datos** que resuelva un **problema** son:

 1) **Analizar** el **problema** para determinar las **restricciones de recursos** que debe cumplir cada posible solución.

 2) **Determinar** las **operaciones básicas** que se deben **soportar** y **cuantificar** las restricciones de recursos para cada una. Ejemplos de operaciones básicas son la *inserción* de un dato en la estructura de datos, *suprimir* un dato de la estructura o *encontrar* un dato determinado en dicha estructura.

 3) **Seleccionar** la **estructura de datos** que cumple mejor los **requisitos o requerimientos**.

Tipos Abstractos de Datos

¿Tipos Abstractos de Datos?

- Concepto propuesto hacia 1974 por **John Guttag** y otros investigadores del **MIT**.
- Un **tipo abstracto de datos** es una **colección** de **valores** y de **operaciones** que se definen mediante una **especificación** que es independiente de cualquier representación.
- Lo llamaremos abreviadamente **TAD**, aunque es común verlo como ADT (del inglés) o TDA



Tipos Abstractos de Datos

- Los LP tienen características que permiten **ampliar** el lenguaje, añadiendo sus **propios tipos de datos**.
- Un **tipo de dato definido** por el **programador** se denomina **Tipo Abstracto de Datos (TAD)**. Por ejemplo, el TAD Punto en el LP C o C++ (aunque sí lo podemos encontrar en Java: *java.awt.Point*)

TAD = **Representación (datos) +**
Operaciones (procedimientos, funciones, métodos)

- La estructura de un **TAD** (clase) de forma global se compone de:
 - **Interfaz Pública** {Método 1, Método 2, Método 3, Método 4}
 - **Implementación Privada** {Representación [datos, atributos]; Implementación de métodos 1, 2, 3 y 4}

Tipos Abstractos de Datos

Ventajas:

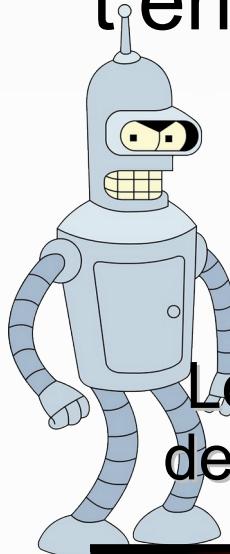
- 
1. Mejor **Conceptualización** y **modelización** del mundo real.
 2. Mejora la **robustez** del sistema.
 3. Mejora el **rendimiento** (prestaciones).
 4. Separa la **implementación** de la **especificación**.
 5. Permite la **extensibilidad** del sistema.
 6. Recoge mejor la **semántica del tipo** (localizan las operaciones y la representación de atributos)

Implementación de los TAD

Lenguaje de Programación	Implementación TAD
Pascal	Unidad
Modula-2	Módulo
Ada	Paquete
C	Estructura
C++	Clase
Java	Clase
Python	Clase

Implementación de los TAD

- La evolución de los lenguajes de programación tiende a introducir **más abstracciones**.



Lenguajes
de bajo nivel

Lenguajes
estructurados

Lenguajes
orientados a objetos



(Ensamblador)

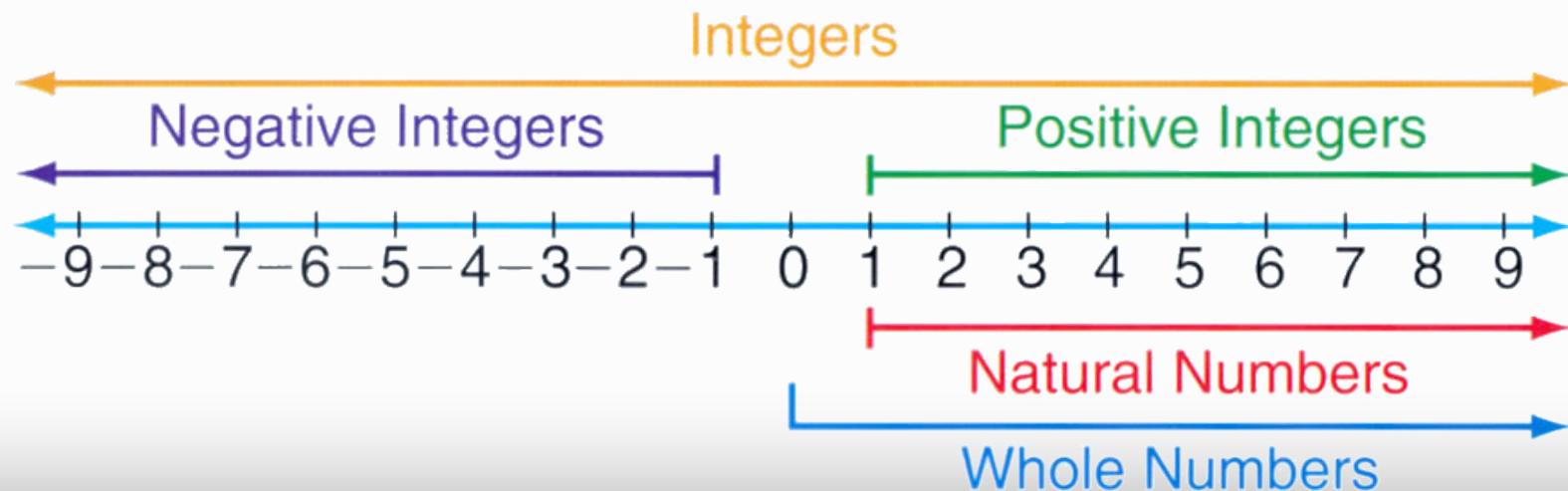
(Pascal, C,
Modula, ADA, ...)

(Eiffel, Smalltalk, C++,
Java, **Python**, ...)



Ejemplo: Especificación del TAD

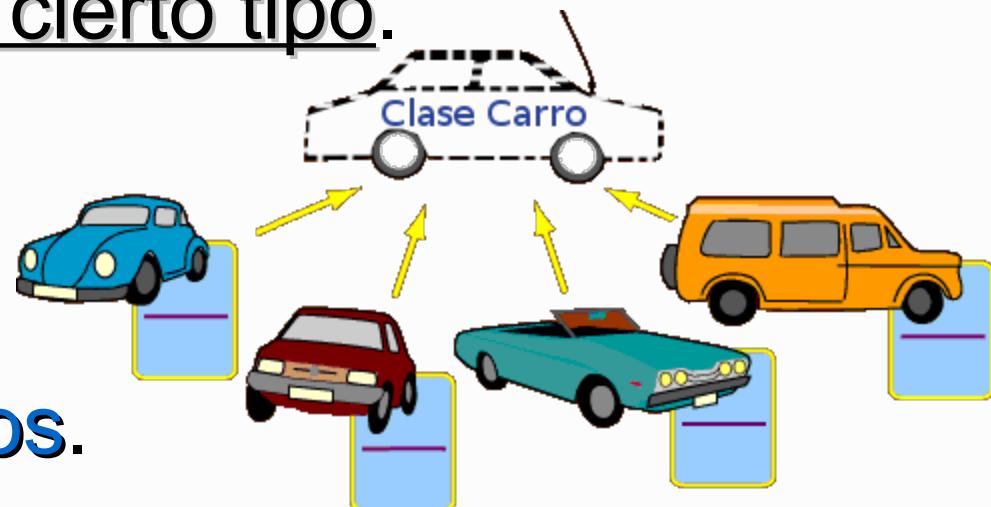
- **TAD:** Entero
- **Tipo de Datos:** Tipo **int** de C, C++ o Java, tipo Integer de Pascal
- **Estructura de Datos:** **Representación** mediante enteros de 16 bits, 32 bits, listas de dígitos (enteros cortos, enteros largos), etc



Tipos Abstractos en Python

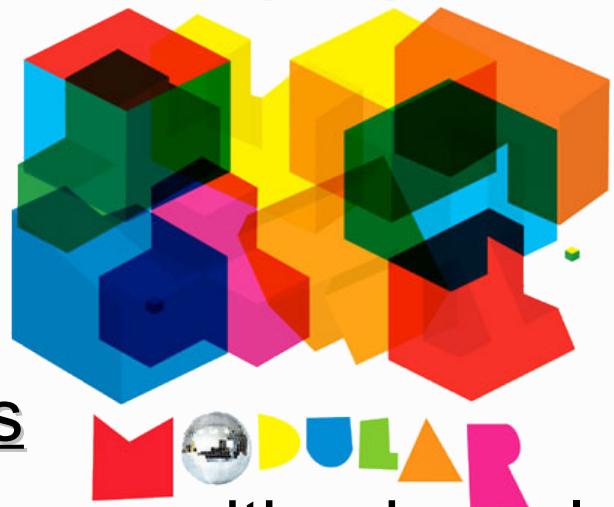
¿Qué es una Clase?

- Es algo **abstracto**, un **modelo** o **prototipo** que define las características (**atributos**) y comportamiento (**métodos**) **comunes** a todos los objetos de un cierto tipo.
- Una **clase** es un **tipo de dato** que contiene **código** (**métodos**) y **datos**.
- Una **clase** permite **encapsular** todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa.



Programación Modular

- Es una unidad lógica que realiza un trabajo específico. Su tamaño se mantiene pequeño llamando a otros módulos.
- Ventajas:
 - 1) **Reduce** la complejidad
 - 2) **Facilidad** de depuración de errores
 - 3) **Facilidad** de implementación, permitiendo el desarrollo paralelo de las diferentes partes de un programa modular
 - 4) **Facilidad** de cambios (debido a que determinadas dependencias están sólo en una rutina)



Ejercicio

- Crear el TAD **Cadena**, definiendo un tamaño máximo de la cadena y considerando las operaciones públicas:
 - **asignar(nueva_cadena)**: Cambia el contenido de cadena.
 - **concat(otra_cadena)**: Concatena la cadena con otra cadena, retornando una nueva cadena.
 - **a_may()***: Convierte todos los caracteres alfabéticos a mayúsculas.
 - **a_min()***: Convierte todos los caracteres alfabéticos a minúsculas.
 - **tamaño()****: Devuelve un valor entero con el número total de caracteres que tiene la cadena.

* sin utilizar los métodos **upper()** y **lower()** de la clase **String**

** sin utilizar la función **len()**



Fin de la Presentación