# PARADIGMAS Y TIPOS DE LENGUAJES INFORMÁTICOS

## Un enfoque practico

Desde las horas extras

Realizado por
**José Luis Bautista Martín**

Diciembre de 2019

Esta hoja está en blanco para facilitar la impresión de este documento a doble página.

# ACERCA DEL INSTRUCTOR

José Luis Bautista Martín, Ingeniero de Sistemas, con maestría en "investigación en ingeniería de Software".

Mi experiencia laboral en cuanto a desarrollo de software abarca desde tecnologías "legacy", hasta tecnologías de vanguardia, poniendo siempre una especial atención en la construcción de software escalable, modular y sostenible.

Igualmente estoy especializado en la interconexión de diversos sistemas y plataformas para conseguir una solución coherente entre la tecnología actual en producción y nuevas tecnologías del mercado.

Una de mis inquietudes actuales es simplificar el desarrollo de software, permitiendo mediante herramientas generadoras de código, patrones de software, programación orientada a aspectos o simplemente interfaces sencillas y claras que el programador se concentre en resolver los problemas propios de la solución a implementar (esto es, los requisitos de negocio a representar en forma de software) y no se tenga que preocupar de tareas repetitivas, generalidades de los sistemas, o problemas técnicos, que no hacen más que distraerle de sus verdaderos objetivos.

# INTRODUCCIÓN

En el 2019 decidí comenzar una serie de artículos sencillos para mi blog (Desde las Horas Extras) sobre tipos y paradigmas de programación, la intención era crear máximo unos tres artículos, ya la naturaleza del blog es tratar temas relativos a la programación y al desarrollo de software en la empresa.

Sin darme cuenta el número de artículos y su complejidad creció, y parecía que los artículos planeados se quedaban cada vez más cortos, al final me ocupo prácticamente todo el año, donde no hubo espacio para otros temas.

Sin embargo se intentó que todos los temas tratados, tuvieran impacto real en el desarrollo de software y pudieran usarse tanto para una capacitación académica, como para instruir y apoyar a alguien que ya desarrolla software de forma profesional.

El conjunto de los lenguajes que se ha seleccionado para los ejemplos son variados, y solo corresponde a un gusto personal, son mis lenguajes favoritos, o por lo menos con los que más he tenido que trabajar profesionalmente.

Las clasificaciones de los lenguajes han sido las siguientes:

- Según el nivel.
- Según la generación.

- Lenguajes interpretados o compilados.
- Estáticos o dinámicos
- Según el "Tipado".
- Según el Paradigma.

En el anexo 1, se incluye un mapa de todo los tipos, que creo es bastante instructivo.

En los datos históricos se ha intentado ser lo más rigoroso posible, aunque a veces es difícil precisar en qué momento exacto se introdujo una características en un lenguaje, que lo hace ser más de un tipo que de otro.

Sin embargo en todo momento se intenta mostrar los (tipos de) lenguajes de programación, no como una verdad absoluta, si no como algo que ha fluido a lo largo del tiempo, adaptándose a las necesidades (empresariales) de cada momento.

# ACERCA DE ESTE DOCUMENTO



Este documento está (junto el código fuente) en un repositorio de GitHub en:

https://github.com/jbautistamartin/ParadigmasTiposLenguajes

Son bien recibidos las correcciones y nuevos ejemplos y temas, tanto en código, como en el mismo documento.

El documento es un compendio de los artículos publicados en el blog de autor, https://desdelashorasextras.blogspot.com/, sobre tipos y paradigmas en los diferentes lenguajes de programación.

.

# CONTENIDO

**Acerca del instructor** ..... **3**

**Introducción** ..... **4**

**Acerca de este documento** ..... **6**

**Contenido** ..... **7**

**Capítulo I. Paradigmas y tipos de lenguajes informáticos** ..... **9**

**Capítulo II. Clasificación según el nivel** ..... **11**

2.1 Lenguajes de bajo nivel ..... 11

2.2 Lenguajes medios ..... 12

2.3 Lenguajes de alto nivel ..... 13

**Capítulo III. Clasificación según la generación** ..... **15**

3.1 Lenguaje de 1ª generación (años 40 y 50) ..... 16

3.2 Lenguajes de 2ª generación (a partir de los años 50) ..... 16

3.3 Lenguajes de 3ª generación (Desde finales de los 50) ..... 17

3.4 Leguajes de 4ª Generación (a partir de los 70) ..... 19

**Capítulo IV. Lenguajes interpretados o compilados** ..... **22**

4.1 Lenguajes compilados ..... 22

4.2 Lenguajes interpretados ..... 23

4.3 Lenguajes de compilación intermedia ..... 25

**Capítulo V. Lenguajes estáticos y dinámicos** ..... **28**

**Capítulo VI. Clasificado según el "Tipado"** ..... **29**

6.1 Lenguajes de tipado débil ..... 29

6.2 Lenguajes de tipeado fuerte ..... 30

**Capítulo VII. Combinaciones de lenguajes estáticos/dinámicos y débiles/fuertes** ..... **31**

7.1 JavaScript, tipado débil y dinámico ..... 31

7.2 C, tipado débil y estático ..... 33

7.3 C#, tipado fuerte y estático ..... 34

7.4 Ruby, tipado fuerte y dinámico ..... 36

**Capítulo VIII. Clasificación según el paradigma** ..... **38**

8.1 Lenguajes de programación según el paradigma ..... 40

**Capítulo IX. Paradigma Imperativo** ..... **41**

9.1 Programación estructurada ..... 42

9.2 Programación Modular ..... 44

**Capítulo X. Paradigma orientado a objetos** ..... **48**

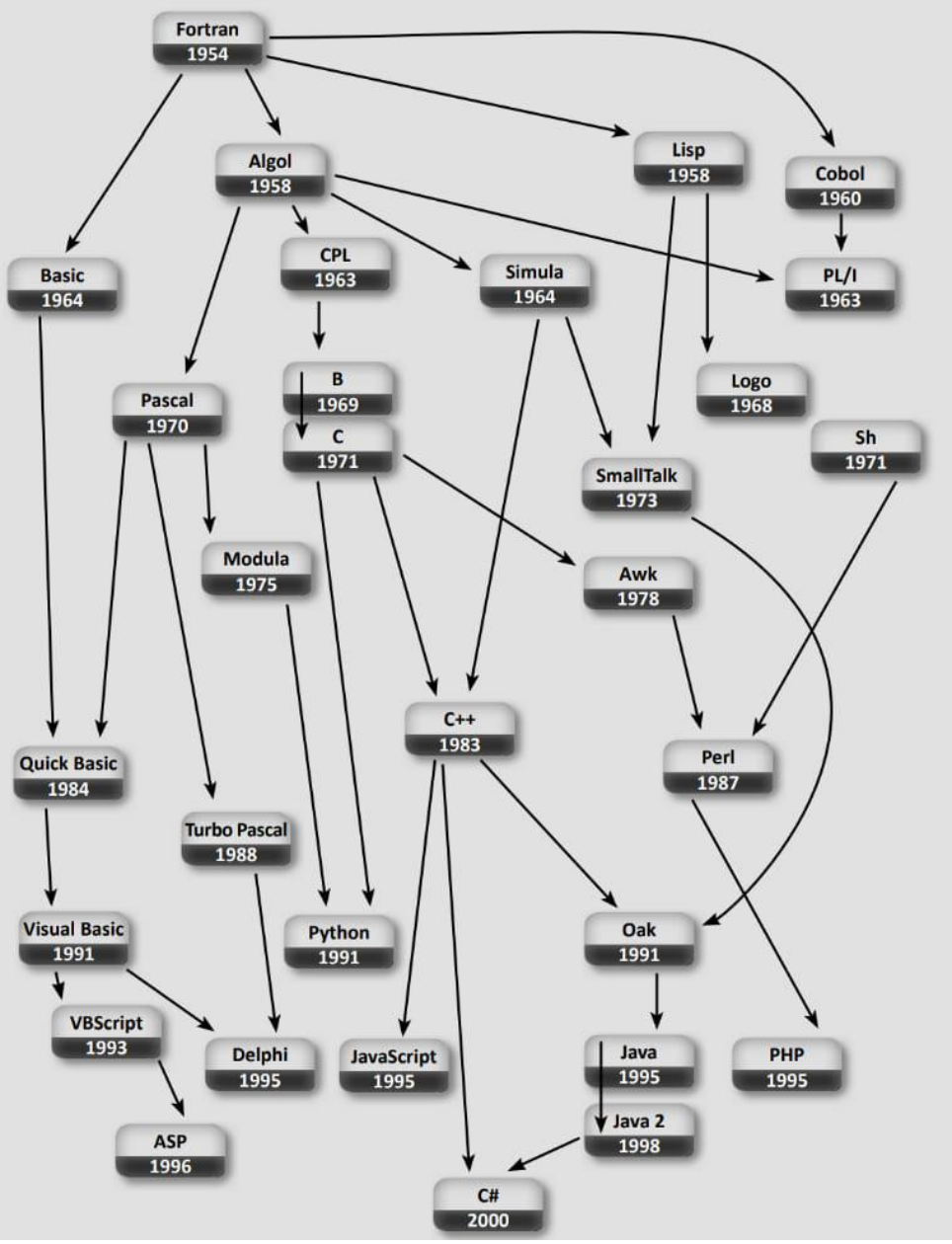
10.1 Elementos que posee una clase ..... 49
10.2 Características de la programación orientada a objetos ..... 49
10.3 Ejemplo en C# de programación orientada a objetos. ..... 50
10.4 ¿Qué fue antes la clase o el objeto? ..... 55
**Capítulo XI. Programación declarativa ..... 56**
11.1 Programación funcional ..... 57
11.2 Programación lógica ..... 60
11.3 Lenguajes específicos del dominio ..... 64
**Capítulo XII. Programación declarativa en lenguajes empresariales ..... 73**
12.1 Ruby ..... 77
12.2 C# ..... 83
**Anexo I: Mapa ..... 96**
**Anexo I: Codigo fuente de ejemplo ..... 98**
**Anexo II: Enlaces online de este documento ..... 99**

# Capítulo I. Paradigmas y tipos de lenguajes informáticos

La informática es una carrera constante entre el software y el hardware pero no es una carrera competitiva, los avances de hardware, hace que queremos tener más servicios, más funcionalidad, mas automatización… provocan un desarrollo de las inquietudes con respecto al desarrollo de nuevas funcionalidades y la velocidad en la que somos capaces de desarrollarlo. Un cambio de paradigma, nuevas herramientas, y métodos en la construcción de software, provocan una obsolescencia en el hardware, Nuestros sistemas vistosos, amigables e interactivos, son cada vez más lentos, los que hace que necesitamos avanzar en nuevos dispositivos cada vez más potentes y versátiles.

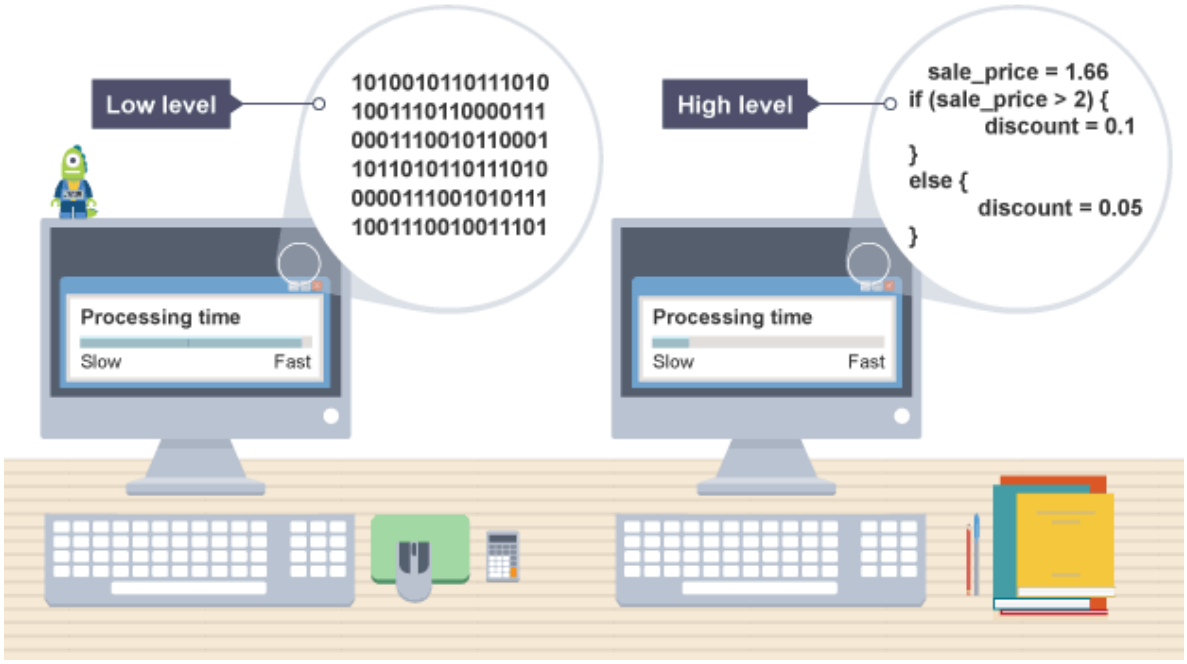
La carrera de los lenguajes de programación no da enormes brincos pero si pasos constantes y rápidos, cada lenguaje es una evolución de los anteriores, una mejora sobre la forma y el proceso, por eso a veces es difícil poder catalogarlos. Cada lenguaje tiene características de los anteriores y se podría situar entre un tipo y otro, solo en casos muy

específicos es posible una clasificación clara, en esta entrada (y las siguientes) a ver distintas formas de clasificar un lenguaje.



# Capítulo II. Clasificación según el nivel

El nivel, hace referencia a la "distancia" que tienen el lenguaje con respecto a al hardware. Un nivel más bajo implica instrucciones directas al hardware, un nivel más alto implica una abstracción (y desconocimiento) mayor con respecto al hardware en particular que corre nuestro hardware.




## 2.1 Lenguajes de bajo nivel

Los lenguajes de bajo nivel ejecutan instrucciones directamente en el hardware destino, y son extremadamente dependientes del mismo hardware, de forma que un lenguaje

para un hardware, suele ser exclusivo para este (o para una familia de hardware en específico si así lo decide el fabricante).

En este punto está el código binario, instrucciones en unos y ceros que ejecuta el procesador directamente, y un poco por encima el lenguaje ensamblado. Hay que recordar que en su forma más primitiva el lenguaje ASM son solos nemotécnicos que son convertidos directamente en código binario. Cada procesador y arquitectura tiene su propio lenguaje ASM.

**Ejemplos:**

- Assembly

## 2.2 Lenguajes medios

Son lenguajes que si tienen estructuras claras de control, y nos alejan de conceptos tales como registros y microprocesadores, pero todavía dependemos mucho de conceptos que tiene que ver más con como organiza la computadora la memoria, que con como diséñanos los algoritmos.

**Estos lenguajes tienen las siguientes características:**

- Delegan la gestión de la memoria completamente al codificador, él debe reservar la memoria, y liberarla.
- Además no existe ninguna restricción en cuanto al acceso de memoria, que generalmente es por punteros y libre (lo cual puede provocar que el programa falle).

- No existe ningún control sobre si estamos asignando un tipo de valor correcto a una variable, ni siquiera si es esta variables se está almacenando en el lugar de memoria adecuado.
- Generalmente los parámetros se pasan por valor (y no por referencia), para modifica parámetros (tener parámetros de salida) sería necesario pasar direcciones de memoria.

**Ejemplos:**

- El ejemplo más representativo es C

## 2.3 Lenguajes de alto nivel

Son lenguajes más independientes de la estructura y características del hardware y más asociados a la forma de pensar del programador o a las necesidades de negocio a resolver.

**Tienen las siguientes características:**

- Una gestión de memoria automática, que suele contar con un recolector de basura.
- Comprobación automática de tipos y de datos, de forma que es casi imposible compilar si se hay un error de asignación de tipos y en el caso que no se puede compilar el sistema generaría un error que haría que se detuviera (en los lenguajes anteriores el programa seguiría ejecutándose sin generar ninguna advertencia y trabajando con datos corruptos).
- Tiene sentencias de control de errores claras y sencillas de usar
- No solo se limita al uso de sentencias condiciones como "if", sino que tiene estructuras complejas como bucles de diversa índole (for, foreach, while, do until).

- Generalmente tienen algún mecanismo de reutilización sencilla de código, ya sea a través de los mismos archivos de código fuente (referenciándolos), algún tipo de librería (dll, jar), componentes de algún tipo (COM, ActiveX), o servicios de alguna índole (Servicios SOAP, REST, etc.).
- Lo más importante. Debido a que están más enfocados a facilitar la resolución de las necesidades operativa, se basan más en la implementación de algoritmos sin tener una relación en particular del hardware, lo que hace que sean sencillos de aprender y además, sin que los conozcamos, el análisis de un programa realizado en estos lenguajes es sencillo de comprender.

**Ejemplos:**

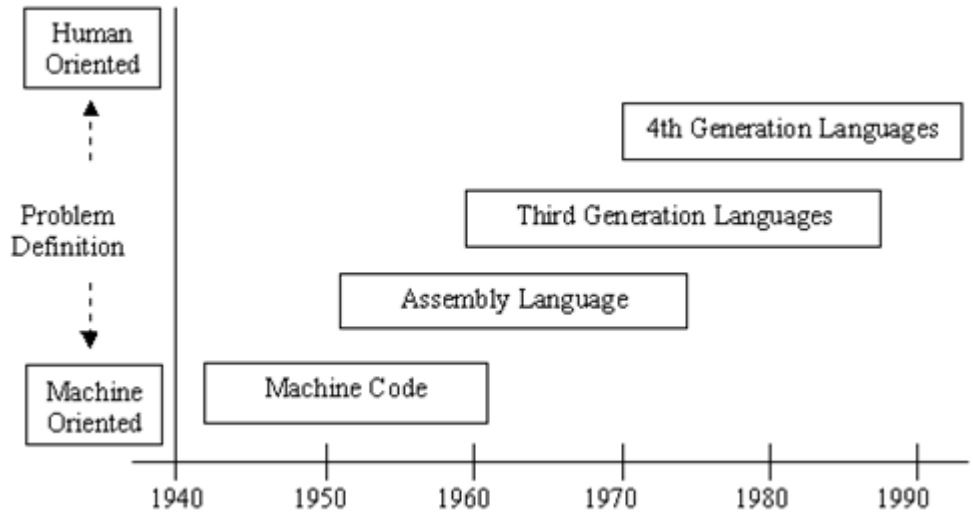
En la actualidad abundan los ejemplos de lenguajes de alto nivel, como los siguientes:

- C#
- Java
- Ruby

## Capítulo III. **Clasificación según la generación**

En esta clasificación se crean generaciones, en la que se aprecia a través del tiempo, la evolución de los lenguajes. Cada generación es más abstracta (con respecto al hardware) y más sencilla de manejar, **con "menos" hacemos "más".**

A quien ocurre un curioso fenómeno, que yo llamo, **el "conservadurismo" del programador**, personas que se afianzaron en una generación (o tecnología) en particular, ensalzando las virtudes de sus lenguajes y rechazando los avances que les sigue porque son "poco óptimos", a saber *"Es mejor programar en ensamblador, porque C es lento"*, *después "Es mejor programar en C, porque C++ es lento"*, *"el mejor lenguaje para programar aplicaciones empresariales es COBOL"*, posteriormente *"Java es el mejor lenguaje de programación empresarial y siempre va a ser vigente"*, **y esto se va repetir para los lenguajes actuales y los futuros**. Hay que comprender que nuestro mercado es cambiante, abarca multitud de actividades humanas y seguramente en el futuro próximo abarque más todavía. Es necesario el cambio y la adaptación, si hay algo mejor, no podemos quedarnos en nuestra zona de confort, necesitamos actualizarnos y adaptarnos, para poder suplir con suficiente rapidez las necesidades de nuestro negocio.



A continuación las diferentes generaciones (las fechas con aproximadas)

## 3.1 Lenguaje de 1ª generación (años 40 y 50)

Aquí tenemos elementos programables, a través de ceros y unos, código máquina. Las primeras computadoras programables (que no se *programaban* desplazando elementos hardware), recibían instrucciones a través de códigos en binario, y se programaba directamente hacia el hardware.

Evidentemente cada computadora tenía su propia secuencia de ceros y unos y cada secuencia hacia una cosa diferente, debido a que realmente estas máquinas no podían hacer muchas cosas (Según nuestra perspectiva actual) dicho conjunto de instrucciones era más o menos manejable y limitada.

First Generation – Machine language

11010100 0011 11001101

01011100 1010 10001111

11001111 1010 11111110

10000111 1011 11000001

## 3.2 Lenguajes de 2ª generación (a partir de los años 50)

La potencia de las computadoras aumento, así como su diversificación, manejar y comprender toda esa secuencia de *unos y ceros*, se convierte en una tarea colosal, y mantener un programa en una odisea inmensa, llena de dificultades.

Se crea un lenguaje basado en mnemotécnicos llamado ensamblador, cada sentencia corresponde directamente a una secuencia de ceros y uno que comprende el procesador. Así tenemos por un lado un lenguaje basando en sentencias *comprensibles* por los programadores, y por otro lado fácilmente traducible a código máquina.

| Address | Machine Language | | | | Assembly Language | | |
|---|---|---|---|---|---|---|---|
| 0000 0000 | 0000 | 0000 | 0000 | 0000 | TOTAL | .BLOCK | 1 |
| 0000 0001 | 0000 | 0000 | 0000 | 0010 | ABC | .WORD | 2 |
| 0000 0010 | 0000 | 0000 | 0000 | 0011 | XYZ | .WORD | 3 |
| 0000 0011 | 0001 | 1101 | 0000 | 0001 | | LOAD | REGD, ABC |
| 0000 0100 | 0001 | 1110 | 0000 | 0010 | | LOAD | REGE, XYZ |
| 0000 0101 | 0101 | 1111 | 1101 | 1110 | | ADD | REGF, REGD, REGE |
| 0000 0110 | 0010 | 1111 | 0000 | 0000 | | STORE | REGF, TOTAL |
| 0000 0111 | 1111 | 0000 | 0000 | 0000 | | HALT | |

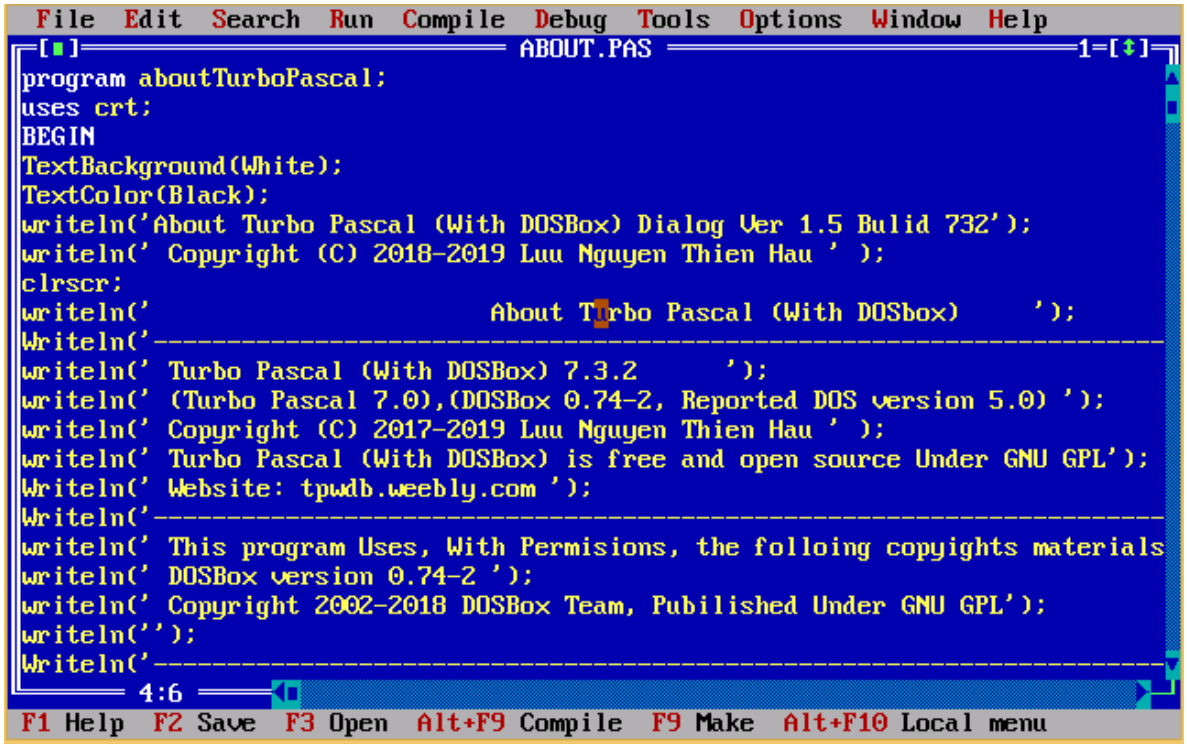
## 3.3 Lenguajes de 3ª generación (Desde finales de los 50)

**Repitamos esto:**

*"La potencia de las computadoras aumento así como su diversificación, manejar y comprender toda esa secuencia enorme de nemotécnicos, se convierte en una tarea colosal, y mantener un programa en una odisea inmensa llena de dificultades."*

Solo cambiamos algunas palabras, pero el enfoque sigue siendo lo mismo, *el hardware puede hacer más, el software lo debe aprovechar.*

Aquí tenemos lenguajes ya más alejados de las sentencias propias de una computadora y mucho más cercanos a una forma de resolver problemas más "humana".




**Las características son:**

- Lenguajes más comprensibles, incluso aunque no se conozca el lenguaje en sí, muy parecidos al inglés.
- Una mayor estructuración del código, se comienza a dividir el código en segmentos que son reusables.
- Los lenguajes suelen tener un propósito general.
- Al ser más alejados de las instrucciones de la CPU, es más sencillo crear lenguajes que son más fácilmente traducibles entre plataformas distintas.

**Algunos ejemplos son:**

- Pascal
- C,
- Fortran,

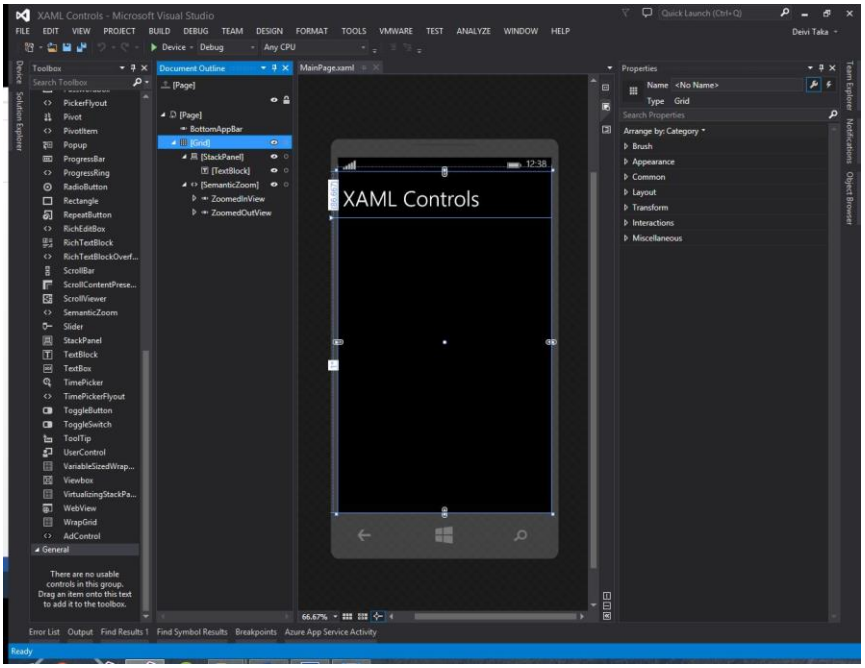
- COBOL
- C++

## 3.4 Leguajes de 4ª Generación (a partir de los 70)

Los lenguajes de 4ª generación se comienzan a desarrollar a partir de los 70, pero tienen un impacto realmente importante a partir de mediados de los 90).

Se acentúa las necesidades comerciales y diarias con respecto al desarrollo de software y la automatización de cualquier mercado imaginable. La informática llega a nuestros hogares y empresas.

Es necesario crear software, crearlo adaptable y funcional pero sobretodo rápido.



**Los lenguajes en esta generación tienen las siguientes características:**

- Frecuentemente, aunque pueden (y suelen) tener la etiqueta *multipropósito*, si tienen un propósito en específico, como la creación de software empresarial. Cuando este propósito es muy específico, tal que solo sirven para atacar un ámbito en particular, se llaman Domain Specific Languages (DSL)

Los DSL son por ejemplo SQL (para la generación de consultas), HTML o CSS (para la generación de páginas de internet)… en su momento JavaScript, podría considerarse un DSL (para agregar dinamismo a una página estática), pero ha evolucionado tanto y se puede usar de formas tan variadas, que sería un error colocarlo como DSL.

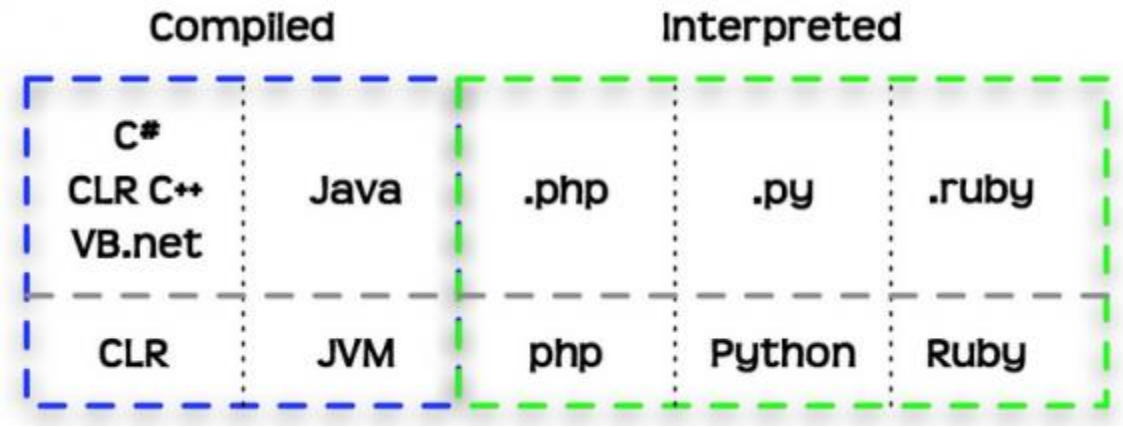
- Hay realmente un énfasis especial en el software empresarial, como tal hay también un gran auge en la conexión y uso de bases de datos.
- El reusó de los componentes es extremadamente común y pueden ser componentes gráficos en muchas ocasiones, complementándose con poderosos IDE. Además suelen estar disponibles por diversos medios y ser fácilmente accesibles (como por MAVEN, gemas de Ruby o NuGet).
- Se delegan en arquitecturas basadas en servicios, en los que hay una aplicación consumidora y un servidor que procesa las peticiones.
- Se cuenta con multitud de herramientas que generan código automáticamente.
- Los lenguajes suelen ser multiplaforma, independientes de la arquitectura de las maquinas donde corren.

**Ejemplos**

- La arquitectura .NET (C#, VB.NET).
- SQL

- HTML, CSS (Si, aunque te parezca extraño son DSL)

# Capítulo IV. **Lenguajes interpretados o compilados**




## 4.1 Lenguajes compilados

Los lenguajes compilados son aquellos en los que el código fuente pasa por una serie de transformaciones hasta que se genera un código máquina (dicho proceso, se llama compilación). El código maquina es el "producto final", el producto que se va a distribuir, y ejecutar.

**Características**

- Están compilados para una arquitectura hardware en particular y para unos sistemas operativos en concreto, esto hace que, si bien su velocidad sea la más óptima posible, sean dependiente de dichas arquitecturas, es decir si queremos ejecutar nuestro software en diversas plataformas, debemos compilarlo varias veces.

Aunque pudiera parecer "sencillo" compilarlo varias veces, una para cada plataforma, descubríamos ciertamente que no lo es. Entre los problemas a encontrar están los siguientes:

- Los procesadores de 32 y 64 bits, tienen distintos tamaños para sus tipos de variables, por ejemplo un entero en el compilador de 32 bites pude medir 2 byte y en el de 64bits, puede medir 4, lo cual provoca caer fácilmente en problemas de memoria si dependemos del tamaño de las variables. ara tomar decisiones. Lo anterior también puede ocurrir entre distintas versiones de un mismo lenguaje en varios compiladores.
- Las librerías y utilerías de una plataforma, no tienen por qué estar en otra, con lo que nos obligaría a reestructurar completamente nuestro código, según tengamos dependencias de estas.

Es ciertamente una forma de no compartir nuestro código fuente sin necesidad, pero debemos considerar que existen herramientas de descompilación, que nos pueden dar una idea aproximada de cómo era nuestro código fuente originalmente.

Quizás el mas típico representante de los lenguajes compilados es C.

## 4.2 Lenguajes interpretados

Ciertamente no se puede ejecutar, en ningún caso, un código fuente directamente, es decir la transformación de código a fuente a código máquina, siempre se tiene que dar, pero en este caso se da cada vez que se ejecuta nuestro sistema, puesto que el producto final es el código fuente. El código fuente es lo que distribuiremos y es lo que se configurara en las maquinas destino.

Los lenguajes interpretados son bastante populares en la actualidad, lo que ha impulsado que haya multitud de librerías y frameworks disponibles en internet, además de mecanismos para consumirlos de forma sencilla y directa.

Debido a que la distribución de código fuente es indispensable, son lenguajes propicios tanto para el software libre, así como para aplicaciones de servidor.

**Las principales características que encontramos son:**

- Homogeneidad ente distintas plataformas: Debido a que el código fuente es portable, este se ejecuta de igual forma (sin modificaciones), entre los distintos escenarios posibles.
- Menor necesidad de versiones concretas de sistemas operativos, o componentes externos en particular. Prácticamente son sistemas que se enlazan dinámicamente a dependencias, y generalmente es necesario usar el nombre del componente a usar, sin tener ningún tipo de liga compleja basado en compatibilidad binaria, o semejantes.
- Debido a que le código, por necesidad, está disponible, siempre es posible hacer un diagnóstico basado en este.
- Facilidad de despliegue, debido a la sencillez para resolver dependencias, casi siempre consiste en copiar los archivos requeridos a la ruta indicada.
- Clásicamente la ejecución es  más lenta que en los sistemas compilados, pero los lenguajes interpretados modernos tienen mecanismos para garantizar que esta lentitud es solo apreciable la primera vez que se ejecuta el sistema, disminuyendo el tiempo drásticamente en ejecuciones posteriores.

**Algunos ejemplos son:**

- PHP
- Ruby
- Python

| Compiled | | Interpreted | |
|---|---|---|---|
| PROS | CONS | PROS | CONS |
| ready to run | **not** cross platform | cross-platform | interpreter required |
| often **faster** | inflexible | simpler to test | often **slower** |
| source code is **private** | extra step | easier to debug | source code is **public** |

## 4.3 Lenguajes de compilación intermedia

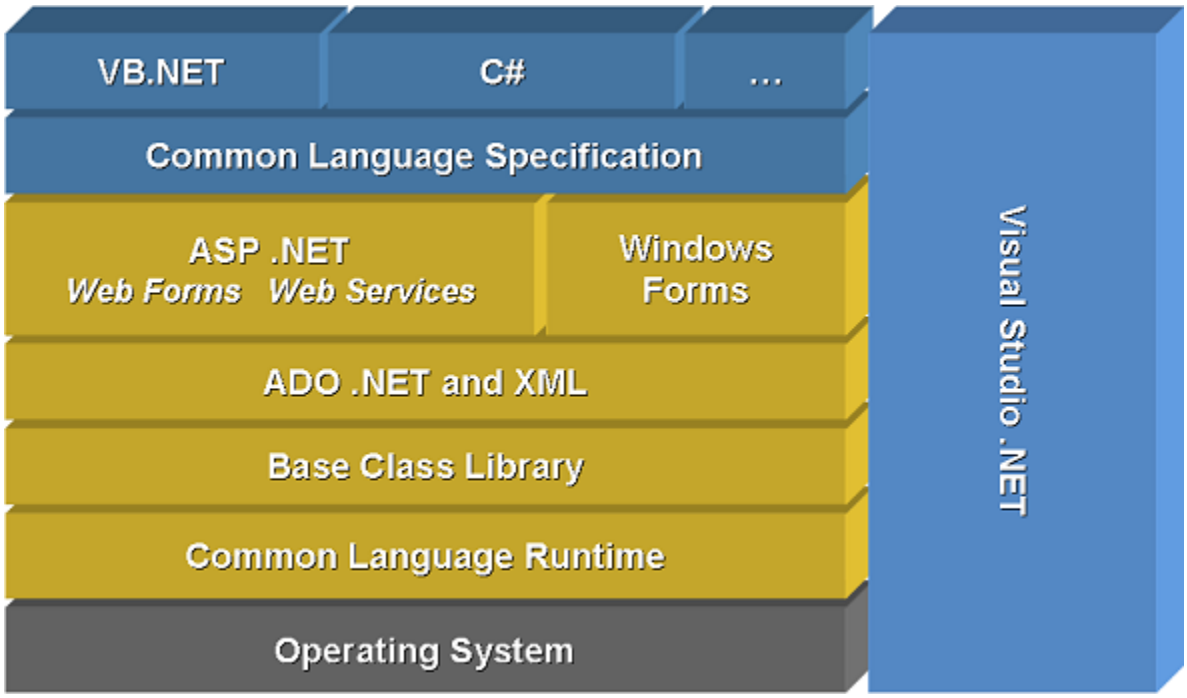
Algunos lenguajes caen en un punto intermedio, son lenguajes compilados, siendo el producto final un ejecutable en código máquina, pero dicho código maquina no es de una plataforma en particular, sino de una máquina que no existe físicamente. Es lo que se conoce como una máquina virtual.

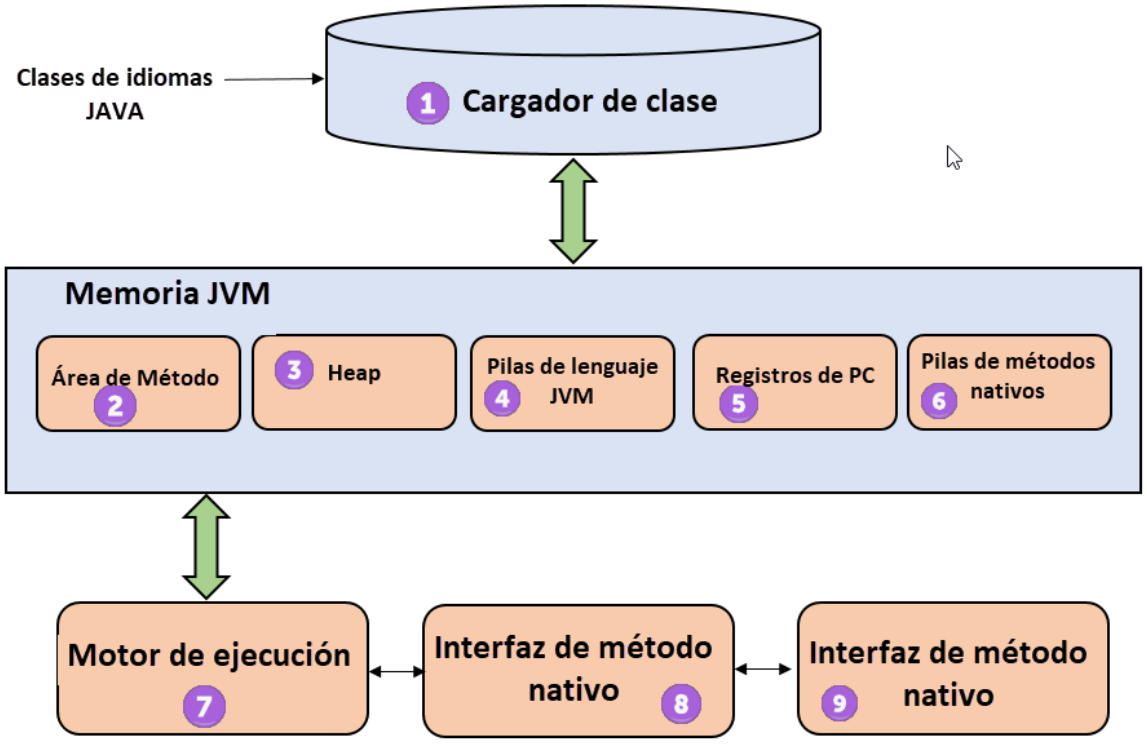
Este código intermedio debe ser interpretado por cada plataforma destino para poder ejecutarse. ¿Cuáles la ventaja que tenemos entonces con este tipo de lenguajes?

- Sistemas Multiplataforma
- Homogeneidad
- Rapidez aceptable

Al programar para una máquina virtual, nuestro código maquina no está ligado a ninguna plataforma en particular, no tiene dependencias particulares de hardware. El tamaño en memoria de las variables básicas, y su estructura esta definida y siempre es la misma (independientemente si el procesado es de 32 o 64 bits). Adicionalmente el código intermedio de la máquina virtual esta tan próxima a los modelos tradicionales de máquinas físicas que la traducción es bastante rápida. Así tenemos los mejor de los lenguajes compilados y los lenguaje interpretados.

**Lenguajes de este tipo son por ejemplo C# y Java**




Clases de idiomas JAVA
1 Cargador de clase
Memoria JVM
Área de Método 2
3 Heap
Pilas de lenguaje 4 JVM
Registros de PC 5
Pilas de métodos 6 nativos
Motor de ejecución 7
Interfaz de método nativo 8
Interfaz de método 9 nativo

## Capítulo V. **Lenguajes estáticos y dinámicos**

- Los lenguajes estático son aquellos en los que una vez definido el tipo de una variable, dicha variable siempre será del mismo tipo, no pueden apuntar en un momento a un entero y al siguiente a un cadena de texto por ejemplo, esto es propio de los lenguajes compilados.

- Los lenguajes dinámicos, son aquellos en que el tipo al que puede apuntar una variable puede cambiar, por ejemplo en este caso en un momento puede apuntar a un entero y al siguiente a una cadena de texto, este comportamiento es típico de los lenguajes interpretados.

Hay que tener en cuenta un punto, algunos lenguajes orientados a objetos (sobre todo los modernos), tienen un clase antecesora común para todas las demás clases, generalmente se llama Object. Un objeto de tipo Object, pudiera apuntar a cualquier variable, en cualquier momento. Esto nos puede hacer pensar que el lenguaje es dinámico, pero no es cierto, simplemente están involucrados mecanismos de herencia pero un objeto de tipo Object, apunta a un Object, para usarlo como un entero, o una cadena de texto, obligatoriamente debemos hacer un cast.

# Capítulo VI. **Clasificado según el "Tipado"**

Todas la variables en memoria de un programa, tiene evidentemente un tipo, es decir son cadenas de texto, numero enteros, decimales, fechas, o cualquier otro tipo de estructura o tipo.

Ahora bien, las variables son espacios de memoria, una secuencia de bytes, lo que le da identidad realmente es lo que "creemos" que hay en ese espacio de memoria. El "creemos", hace referencia al tipo de variable que esta apuntado a ese espacio de memoria, si el tipo de variable que está apuntándolo es un entero, supondremos que el espacio de memoria al que apunto es un entero, si es una cadena supondremos que es una cadena.

El como el lenguaje permite gestionar el "tipo" de las variables es lo que llamaremos su "tipeado", según mas restricciones tenga, mayor y más fuerte será el tipeado, una tipeado mayor nos garantizara que solo variables de un tipo apunten a espacios de memoria donde este tipo, por ejemplo garantiza que si nuestra variable es un entero, solo pueda apuntar a un espacio de memoria que haya un entero.

## 6.1 Lenguajes de tipado débil

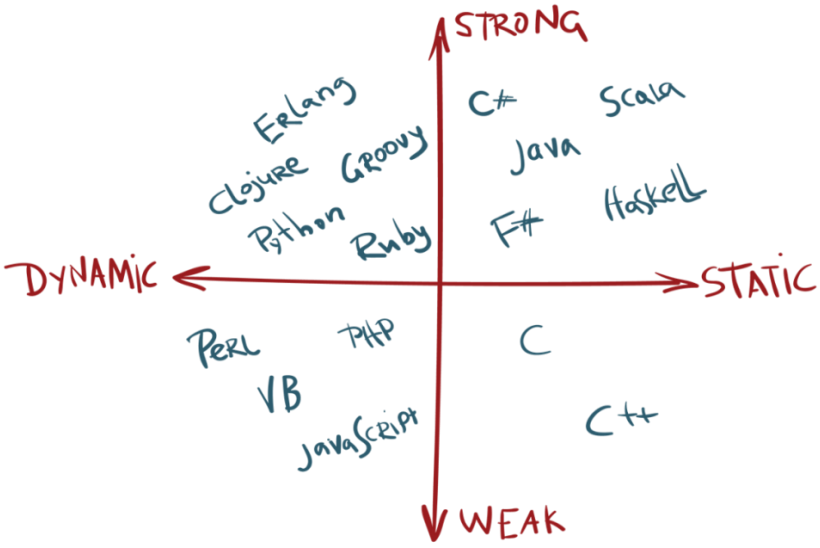
Es cuando se conoce de que tipo es una variable, pero es posible hacer que dicha variable apunte a un espacio de memoria, donde no este un valor de dicho tipo, esto puede generar un error o no en tiempo de ejecución (aunque lo más problema es que por lo menos genere un comportamiento extraño y no deseado).

## 6.2 Lenguajes de tipeado fuerte

Los lenguajes de tipeado fuerte, son aquellos que tienen un estricto control sobre el tipo de variable y el tipo de contenido a la que están apuntando, si están apuntado a un tipo incorrecto el programa no compilaría, si están en tiempo de ejecución se da la circunstancia que una variable apuntara a un tipo incorrecto, el programa de detendría generando un error (en vez de los tipados débiles, que continuaría haciendo cosas "raras", hasta que el fallo fuera demasiado grande).

# Capítulo VII. Combinaciones de lenguajes estáticos/dinámicos y débiles/fuertes

Generalmente se ven en conjunto la características débiles/fuertes y estáticas/dinámicas de un lenguaje, porque en si están ambas clasificaciones relacionadas en como gestiona las variables un lenguaje de programación, cuando más débil y dinámico es un lenguaje más sencillo de comprender es, más permisivo y más propenso a fallos de codificaciones, por el otro lado cuando mas estático y fuerte es un lenguaje más explicito es, menos versátil y menos propenso a fallos. No es una cosa mejor que la otra, simplemente debemos conocerlos y situar nuestra solución en el punto en que se requiera.




## 7.1 JavaScript, tipado débil y dinámico


Analicemos el siguiente código en JavaScript, para ver cómo se comportan las variables (que son de tipado débil y dinámicas). El código es simple suma:

```
// Las variables no tienen un tipo, se declara con var
var a=1;
var b=2;
var c= a + b;
//En este caso debe C debe valer 3
console.log(c)
```

La salida será **3**

Ahora bien, por error podemos asignar el valor "hola" a la variable "a" y volver a realizar la suma, esto no generara ningún error (ni al compilar, ni la ejecutar), pero si una circunstancia inesperada

```
// Las variables no tienen un tipo, se declara con var
var a=1;
var b=2;
var c= a + b;
//En este caso debe C debe valer 3
console.log(c)
//Ahora bien imaginemos que asignamos el valor "hola" a "a"
//Nos permite hacerlo sin problemas
a="hola "
c= a + b;
// En este caso el resultado sera "hola 2"
console.log(c)
```

La salida será **"hola 2"** lo cual posiblemente no tenga ningún sentido.

## 7.2 C, tipado débil y estático

El siguiente código en C, muestras un ejemplo de un lenguaje de tipado débil y declaración de variables estáticas, esto es que forzosamente debo declarar el tipo de las variables, pero estas pueden a apuntar a espacios de memoria, que no tuvieran ese tipo en particular.

El código del siguiente ejemplo es válido, compila y no generar errores al ejecutarse, sin embargo, estoy convirtiendo enteros a floats, y string, sin que posiblemente me dé cuenta de ello, por lo que pudiera tomarse como un error de dedo y los resultados son "extraños" (nótese que nunca se interrumpe el programa).

```
int main()
{
    int a=120;
    //Imprimo a como un numero, es valor es 50;
    printf ("El valor de a es '%d' \n",a);

    //imprimo a como un float se imprime un valor
    //pero no es 50,y no hay ningun fallo
    printf ("El valor de a es '%f'\n",a);

    //Lego el valor de a del teclado como cadena de texto
```

```
    printf ("Introducta el valor de a: ");
    scanf("%s",&a);
    printf ("El valor de a es '%d' \n",a);

}
```

La salida del código es:

```
El valor de a es '120'
El valor de a es '0.000000'
Introduzca el valor de a: hola
El valor de a es '1634496360'
```

A pesar que no tiene sentido, el programa funciona y continua su ejecución, sin percibir nada extraño.

## 7.3 C#, tipado fuerte y estático

C# es un lenguaje de fuerte tipeado y estático, es decir las variables tienen un tipo específico y solo pueden apuntar a un valor de dicho tipo.

Veamos el siguiente ejemplo

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;
```

```
namespace Rextester
{
    public class Program
    {
        public static void Main(string[] args)
        {
            int a=20;
            string b=(string) a;
        }
    }
}
```

Genera el siguiente error:

```
(16:22) Cannot convert type 'int' to 'string'
```

Esto es porque no se puede convertir un entero a una cadena de texto, un variable de tipo string, no puede apuntar a un valor de tipo entero.

Para hacerlo posible, habría que convertir explícitamente el valor entero a una cadena de texto, básicamente esta conversión consiste en crear una nueva dirección de memoria donde este una cadena de texto y no un entero (con lo que tendríamos dos direcciones de memoria, la del entero y la cadena de texto).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;

namespace Rextester
{
    public class Program
    {
        public static void Main(string[] args)
        {
            int a=20;
            string b=a.ToString();
        }
    }
}
```

A veces pensamos que los lenguajes en los que especificamos explícitamente el tipo de la variable son estáticos y los que no dinámicos, pero esto no es siempre cierto, veamos el siguiente código:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;

namespace Rextester
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var a=20;
            a="hola";
        }
    }
}
```

Nos generara el error

```
(16:15) Cannot implicitly convert type 'string' to 'int'
```

Aquí usamos la cláusula var, para crear variable y le asignamos el valor "20" esto lo convierte en una variable de tipo entero, y nunca podrá cambiar su tipo, ni apuntar a otro espacio de memoria donde no haya un entero, las sentencia var a=20, equivale a int a=20, la declaración del tipo es estática y en tiempo de compilación.

## 7.4 Ruby, tipado fuerte y dinámico

Es común suponer que si un lenguaje es interpretado debe ser no de tipado débil, pero esto no siempre es así, por ejemplo veamos el siguiente código de Ruby (lenguaje interpretado):

```
a=10
b="2"
c=a+b
```

Nos generar el error:

```
String can't be coerced into Integer (repl):3:in `+' (repl):3:in `<main>'
```

Porque no es posible convertir explícitamente la cadena de texto en un entero, para corregir el código, hacemos las siguientes modificaciones:

```
a=10
b="2"
c=a+b.to_i
```

Esta vez nos da el resultado **12**.

Como vemos para iniciar una variable no hay ninguna palabra reservada, solo debemos asignarle un valor, pero Ruby si tiene ligado el tipo de la variable a su contenido, de forma que no se pueden convertir directamente, ni usar un tipo como otro, sin sé que se indique.

# Capítulo VIII. **Clasificación según el paradigma**

La palabra "paradigma" tiene en la actualidad connotaciones negativas, muchos cursos y libros de autoayuda de han encargado de eso, tampoco ayuda lo realmente mal que se introducen los paradigmas en las carreras relacionada con las ciencias de la computación, creo que a la mayoría se les introduce mediante el ejemplo del "paradigma de los monos", el cual es bastante popular y mal aplicado en general, básicamente es un estudio (quien sabe si real) donde tiene unos monos en una jaula y les ponen comida a la mano, cuando intentan conseguir la comida, son todos golpeados, al final los mismos monos se golpean entre sí para evitar que nadie se acerca a la comida, los científicos ya no necesitan golpear a los monos, y los van sustituyendo hasta que no queda ninguno de los que están de origen, los monos se siguen golpeando entre sí para evitar que nadie tome la comida, aunque ya no quedaba ningún mono de los que fueron golpeados por los mismos científicos. Esto queda bien en algunos contestos, pero es fatal para la compresión de los paradigmas y su aplicación en la ciencia y la ingeniería.

Un paradigma, consideramos aquí como el equivalente de axioma en lógica o filosofía, es algo que consideramos cierto, bueno y útil para un problema en particular, aunque no tengamos los argumentos necesarios para justificarlo. Si tenemos buenos paradigmas nuestras posibilidades de tener éxito en un problema son elevados, si tenemos malos paradigma evidente pasa lo contrario, que estamos avocados al fracaso. Aquí es donde está la diferencia entre ingeniería y ciencia, la ingeniería es práctica, necesita resolver problemas reales, si la ingeniera se planteara a cada punto si lo que hace es cierto o no, dejaría de ser practica, es más seria inútil, la ingeniería se plantea que pueden hacer las cosas, y las une para obtener un fin, la ciencia se plantea el por qué las cosas hacen lo que hacen. En realidad cada ingeniero, tiene un poco de científico, pero no puede perderse en las minucias de por qué funcionan las cosas, sino que la experiencia (y las investigaciones de los científicos), le llegan a conocer intuitivamente cual es mejor camino a la resolución de un

problema. Esto es importante para poder proveer el software que las necesidades actuales de la sociedad requieren.

Los algoritmos de encriptación, son un ejemplo de esto. Una de las grandes recomendaciones es que no "inventes" tus propios algoritmos, porque seguramente fracasaras creando algún algoritmo frágil y vulnerable. Los algoritmos de encriptación, son funciones matemáticas, creadas por personas con conocimiento extenso y profundo sobre ese tema en particular, y durante investigaciones que duran años, además dichos algoritmos llevan mucho tiempo probando su seguridad y validad en multitud de sistemas. Si uno que hhace programas de aplicación, por ejemplo, e intenta hacer un algoritmo de encriptación, realmente se está desviando de su propósito que es cumplir una necesidad de negocio, con lo que está dedicando tiempo y recursos a algo que ya esta resulto, Al final hará un programa malo y un algoritmo de encriptación malo (y peligroso), con tal de cumplir unas fechas establecidas por el mercado. Lo lógico y correcto es usar algún algoritmo de encriptación, que aunque no entendamos en profundidad su base matemática, haya demostrado su valía. (Hay que precisar que los algoritmos de encriptado de vuelven obsoletos con el tiempo, y es posible que se demuestren vulnerabilidad, es necesario claro, una actualización de los algoritmos, según sea preciso).

Otro ejemplo de paradigmas son los patrones de software, los patrones de software, son caminos demostrados mediante la práctica, que se establecen como la solución idónea para resolver un problema, de esta forma al ver un problema semejante, sabemos que tendremos éxito en resolverlo al seguir el patrón, si nos desviamos del patrón, posiblemente perdamos tiempo y nuestro sistema sea de poca calidad (aquí es curioso la existencia de anti patrones, que llegan a ser "la mejor manera de hacer algo mal"). El desafío aquí es saber identificar claramente el problema, no como resolverlo, es un problema de análisis, no de implementación.

Una vez establecido lo anterior, vamos a ver los tipos de paradigmas de programación existentes.

## 8.1 Lenguajes de programación según el paradigma

Las computadoras en su versión más simple, son maquinas que ejecutan una orden tras otra, comienzan desde la primera línea de un programa, y paso a paso las van ejecutando, este paradigma se llama imperativo, se tiene una orden y se cumple.

Aunque los humanos a veces organizamos nuestros procesos de dichas forma, creando listas que se ejecutan paso a paso, realmente nuestro pensamiento natural no funciona así, nosotros pensamos mas en clasificar los elementos que posee el mundo real, y concederles ciertas propiedades y características, nos interesa más lo que pueden hacer dichas elementos y no él como lo hacen, en ultima instancian nos concentramos en nuestras necesidades y no como estas pueden ser resultas, por ejemplo, si vamos a una cafetería, pedimos un café, y expresamos como lo queremos, pero los mecanismos internos por los cuales el café es hecho, nos da igual, no es objeto de nuestro interés, esto es se llama "declarativo", declaramos lo que queremos, no el proceso para resolverlo.

Todos los paradigmas de programación se encuentran en algún punto que está entre los lenguajes imperativos y los lenguajes declarativos.

# Capítulo IX. **Paradigma Imperativo**

En el paradigma imperativo los programas se ejecutan línea a línea, como si fuera una lista, cuando queremos usa una bifurcación en el flujo de nuestro código, usamos una sentencia condicional que nos lleve a otra parte del código donde seguirá ejecutándose liana a línea de la misma forma.

Se llama lenguaje imperativo, que significa "orden", porque la maquina va cumpliendo las ordenes, según las vaya analizando.

Veamos el siguiente código en BASIC tradicional:

Ejemplo sacado es de la Wikipedia https://es.wikipedia.org/wiki/BASIC

```
10 INPUT "Cuál es su nombre:"; NN$
20 PRINT "Bienvenido al 'asterisquero' ";NN$
25 PRINT
30 INPUT "con cuántos asteriscos inicia [Cero sale]:"; N
40 IF N<=0 THEN GOTO 200
50 AS$=""
60 FOR I=1 TO N
70 AS$=AS$+"*"
80 NEXT I
90 PRINT "AQUI ESTAN:"; AS$
100 INPUT "Desea más asteriscos:";SN$
110 IF SN$="" THEN GOTO 100
120 IF SN$<>"S" AND N$<>"s" THEN GOTO 200
130 INPUT "CUANTAS VECES DESEA REPETIRLOS [Cero sale]:"; VECES
140 IF VECES<=0 THEN GOTO 200
150 FOR I=1 TO VECES
160 PRINT AS$;
170 NEXT I
180 PRINT
185 REM A repetir todo el ciclo (comentario)
190 GOTO 25
200 END
```

Al principio lo que más destaca es la dificultad para leerlo y comprenderlo. Es más hay que leer todo como un bloque y de arriba abajo. Básicamente la salida del programa es la siguiente:

```
Cuál es su nombre: Jose Luis
Bienvenido al 'asterisquero' Jose Luis

con cuántos asteriscos inicia [Cero sale]: 5
AQUI ESTAN:*****
Desea más asteriscos: S
CUANTAS VECES DESEA REPETIRLOS [Cero sale]: 5
**********************
```

Es más relevante como se van ejecutando las ordenes, una a una, porque para BASIC, en su versiones tradicional, es necesario programar escribiendo el numero de línea, en que se van ejecutando las ordenes.

Adicionalmente como podemos ver si queremos ir a otra parte del programa usamos una sentencia if y un comando goto (para cambiar de línea).

## 9.1 Programación estructurada

Los sistemas informativos se hacen más complicados, se hace patente que la programación imperativa tiene características que hacen que los sistemas sean difíciles de mantener. El uso de goto y saltos de líneas sin control hace que muy difícil seguir el flujo de un programa, si el programa es grande, se acaba convirtiendo en una tarea imposible.

La programación estructurada viene a mejorar la calidad y el mantenimiento de los sistemas software. Consiste en la eliminación de la sentencia goto en los programas, debiendo recurrir nuestro programa solo a subrutinas, y a tres estructuras básicas:

- **Sentencias:** básicamente como programación imperativa, se ejecuta una sentencia tras otra.

- **Condicionales:** Estructuras condicionales con secciones muy claras, que no implican un salto de línea. Son las clausulas if, else y swicth.
- **Iteraciones:** repeticiones de código, como fors, y whiles.

Por otro lado la subrutinas son segmentos de código que se ubican en otra parte de nuestro sistema, y que cuando son llamas regresan el control, al punto siguiente de la llamada, son las funciones (cuando devuelve un valor) y los procedimientos (funciones que no devuelven ningún valor).

Veamos el siguiente código en C (adaptándolo al de BASIC)

```
#include <stdio.h>
#include <stdbool.h>

int main() {

    char nombreUsuario[20];
    bool salir = false;

    printf("¿Cuál es su nombre? ");
    scanf("%[^\n]s", nombreUsuario);
    printf("Bievenido al 'asterisquero', %s", nombreUsuario);

    do {
        int nroAsteriscos = 0;
        printf("\n¿Cuántos asteriscos quieres [Cero sale]: ");
        scanf("%d", & nroAsteriscos);

        if (nroAsteriscos != 0) {

            printf("AQUI ESTAN: ");
            for (int i = 0; i < nroAsteriscos; i++) {
                printf("*");
            }
            printf("\n");

        } else {

            salir = true;
        }

    } while (!salir);

}
```

La salida de este código será:

```
¿Cuál es su nombre? Jose Luis
Bievenido al 'asterisquero', Jose Luis
¿Cuántos asteriscos quieres [Cero sale]: 5
AQUI ESTAN: *****

¿Cuántos asteriscos quieres [Cero sale]: 0
```

Vemos que el simple hecho que le código este indentado nos facilita la lectura, también vemos que no existen sentencias goto, todo está estructurado con sentencias if (condicionales), y de interacción (while) y en definitiva es mucho más sencillo de entender, que el código en BASIC.

Existen algunos lenguajes que si bien son estructurados (como C), todavía tiene la sentencia goto entre sus sentencias posible (aunque de ninguna forma se recomienda), otros, los más modernos (como java), generalmente no lo tienen o tienen su uso restringido.

Existe un debate, sobre si las sentencias, break (para interrumpir una interacción), continue (para ir a la siguiente interacción), o más de un return por función son recomendables, porque son sentencias "goto" enmascaradas. La respuesta es **"si", si deben usarse,** son sentencias que nos llevan a un punto muy concreto del código y estas asociadas a las mismas sentencias de control, si no las usuramos deberías crear sentencias de control a bases de if, que complicarían nuestro código y su legibilidad. Recordemos que el objeto de la programación estructura no es eliminar los saltos de línea como tal, sino facilitar la creación de un código de calidad que sea mantenible y entendible.

## 9.2 Programación Modular

La programación modular divide un programa grande y complejo, en "programas", más pequeños y sencillos, de maneras que son más sencillos de resolver. A cada uno de estos subprogramas los llamamos módulos.

Cada modulo se encarga de solucionar un problema, de esta manera son más sencillos de crear y de mantener, el problema final resultara de la combinación de dichos módulos.

Cada modulo tiene sus propias estructuras de datos (que idóneamente) estas aislado del resto de los módulos, de forma que cada modulo, expone solo lo necesario para poder comunicarse con los demás. Cuanto menos expongamos, menos acoplamiento generamos y más sencillo es mantener el código.

Al facilitar que todo lo que necesita un modulo para funcionar este junto, y que además este aislado de lo que no necesita, aumentamos la cohesión, esto es garantizar la proximidad en código de los elementos que consumimos (funciones, estructuras de datos) en mismo lugar y garantizar que se usan entre sí, nuevamente hacemos un código más sencillo y mas mantenible.

Pascal es uno de los lenguajes que implementa la programación modular (y posiblemente unos de los más estructurados), fuente de inspiración para multitud de lenguajes modernos, veamos el siguiente ejemplo en pascal:

El ejemplo solicita dos puntos (con coordinadas X e Y) y calcula la distancia entre los dos. Vamos a crear un modulo (llamado unidad en pascal), con todo lo que requerimos para trabajar con puntos.

```
unit Puntos;

interface

type

//Representan un punto
TPunto = record
```

```
     x:integer;
     y:integer;
end;

//Obtiene un punto desde el teclado
function obtenerPunto(): TPunto;

//Cacula la distancia entre dos puntos
function calcularDistancia(punto1: TPunto; punto2:TPunto): real;

implementation
Uses Math;

function obtenerPunto(): TPunto;
var
  punto:TPunto;
begin
   write('Introcuzca la X del punto: ');
   readln(punto.x);
   write('Introcuzca la Y del punto: ');
   readln(punto.y);

   obtenerPunto:= punto;
end;

function calcularDistancia(punto1: TPunto; punto2:TPunto): real;
begin
   calcularDistancia:=  sqrt( power(punto2.x-punto1.x,2) +
power(punto2.y-punto1.y,2))
end;

end.
```

El código principal usara el modulo anterior, para solicitar los datos y mostrar los resultados:

```
program Ejemplo;
uses
    Puntos, SysUtils;

var
   punto1 :TPunto;
   punto2 :TPunto;
   distancia: real;


begin

  writeln('Introduzca el punto uno: ');
```

```
  punto1:= obtenerPunto();
  writeln();

  writeln('Introduzca el punto dos: ');
  punto2:= obtenerPunto();
  writeln();

  distancia:= calcularDistancia(punto1, punto2);

  writeln();
  write('La distancia entre el punto uno y el punto dos es: ');
  writeln(Format('%.2f', [distancia]));
  writeln();
  writeln('Pulse enter para salir...');
  readln();

end.
```

La salida será

```
Introduzca el punto uno:
Introduzca la X del punto: 10
Introduzca la Y del punto: 10

Introduzca el punto dos:
Introduzca la X del punto: 5
Introduzca la Y del punto: 5


La distancia entre el punto uno y el punto dos es: 7.07

Pulse enter para salir...
```

# Capítulo X. **Paradigma orientado a objetos**

En este paradigma orientado a objetos ya nos acercamos más a como organizamos el pensamiento las personas. Nosotros para comprender el mundo real no nos fijamos muchos en los detalles, si no que abstraemos y clasificamos lo que nos rodea para poder comprenderlo, a cada "ser" le atribuimos propiedades y en base a que características de dichas propiedades les conceden, sabemos cómo interactuar con ellos.

Un ejemplo de esto son, los coches, poca gente sabe (en profundidad) cómo funciona el motor de un coche, o su sistema eléctrico, pero mucha gente sabe como operar un coche, incluso no tendría problemas para cambiar de modelo de coche y poder conducirlo inmediatamente, a pesar de que existe centenares de modelos de coches; automáticos, estándares, y de características muy variadas (incluso sin consideramos detalles estéticos como el color). El hecho de que una vez que hemos aprendiendo a conducir un coche, podamos sin mucho problemas, conducir varios modelos de coche, no implica que sepamos cómo funciona un coche realmente, si no que sabemos que es lo que estos pueden hacer, y en base a esto, podemos utilizarlo.

La POO, consiste en clasificar nuestros problemas, en "entidades", que tienen características (llamadas atributos), y comportamientos conocidos, con los que podemos interactuar, a la vez que tienen características y comportamientos ocultos que no son necesarios conocer para poder interactuar con los anteriores.

El cómo se relacionen a través de comportamientos y características todas estas entidades es lo que conocemos como programación orientada a objetos.

Las entidades con sus características definidas y establecidas se llaman objetos, y los objetos de un mismo tipo son clasificadas en clases, por ejemplo si mi coche es un Nissan Versa Blanco del 2017, ese sería considerado el objeto, el modelo Nissan Versa seria la clase, con las características de color y año.

## 10.1 Elementos que posee una clase

- **Atributos:** Representa el estado de la clase (una vez convertida en objeto), es lo que le da "personalidad" a la clase (en el ejemplo del coche seria el año o el color). Para acceder a los atributos, según el lenguaje, pueden usarse métodos o propiedades.
- **Mensajes:** Las objetos pueden definir mensajes, que son instrucciones para realizar acciones, esto está representado por métodos (funciones y procedimiento), que pueden recibir cero o más parámetros.
    - **Relaciones:** las clases tienen relaciones de diversos tipos con otras clases:
    - **Asociación:** Dos clases (o más) están asociadas de alguna forma, puede ser que una clase sea un conjunto de otra clase (como una lista), o que sus atributos estén compuestos de otras clases.
    - **Dependencia:** Una clase usa a la otra.
    - Generalización/Especialización: Aquí entran todos los mecanismos de herencia

## 10.2 Características de la programación orientada a objetos

- **Encapsulamiento/Ocultación:** La programación modular, ya tenía lago de encapsulamiento y ocultación.

**El encapsulamiento** consístete en juntar en un mismo lugar todo lo que define a un objeto, esto es los datos (atributos), y la funcionalidad (métodos), de forma indivisible (uno de los grandes errores de la programación es considerar que los datos, deben estar aislados del negocio, esto es incorrecto, **en la POO, los datos y el negocio, son una entidad unificada, un objeto, no tiene sentido, si no tiene datos y si no tiene negocio, todo esto hace referencia a la cohesión de una clase.**

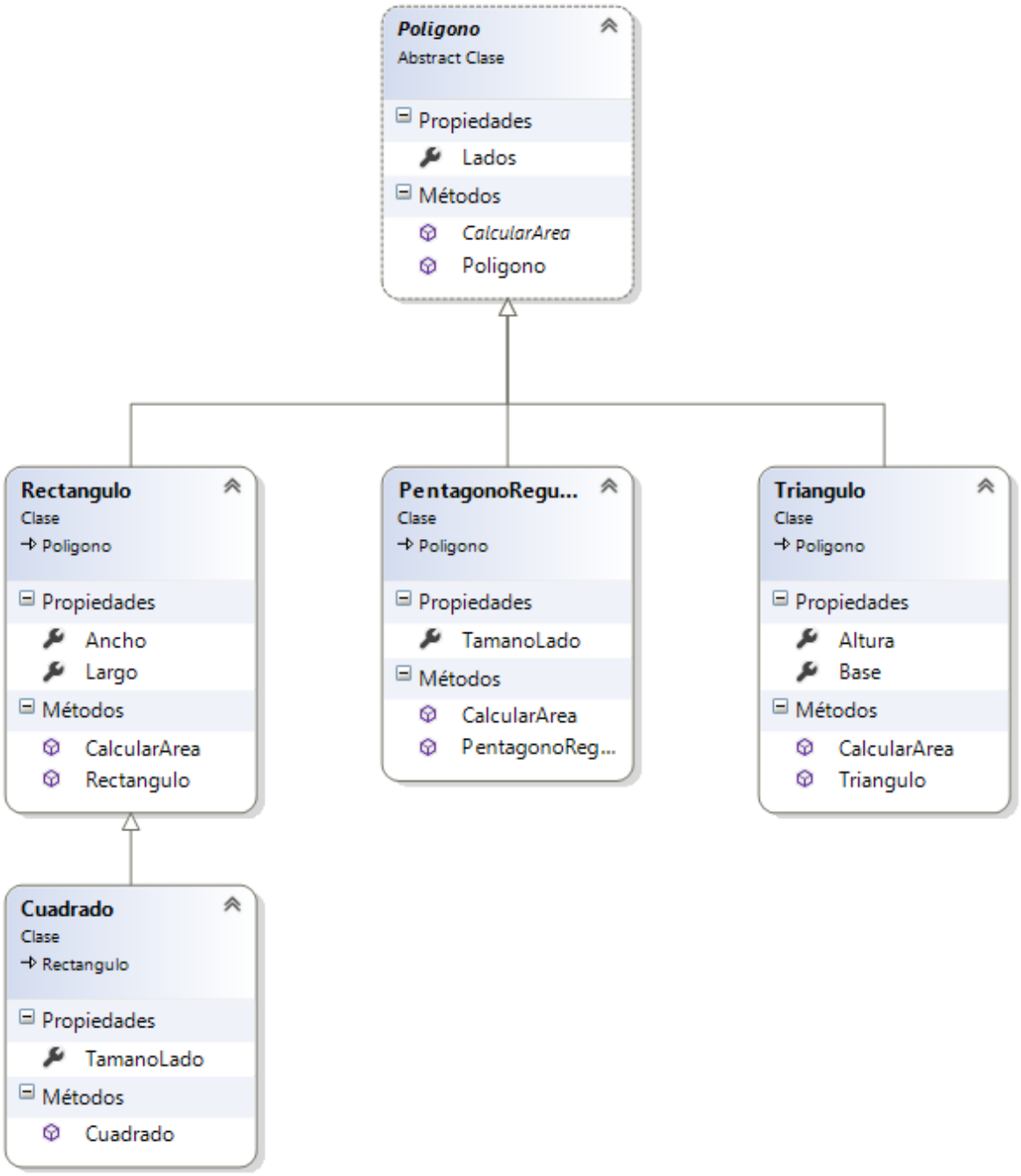
**El ocultamiento** a su vez, significa que la clase debe mostrar lo menos posible sobre sí misma. Su funcionamiento y estructuras internas, no le interesa al resto de las clases, cuando menos expongamos más mantenible va a ser la clase, esto refuerza el bajo acoplamiento.

- **Herencia:** es una de las principales relaciones entre clases, permite extender una clase creando otra que "herede" todas sus características y comportamientos, pero permitiéndola agregar nueva funcionalidad. La clase de la cual hereda se llama "clase" madre (o base) y la otra clase "hija". En el caso del coche, la clase "coche", seria la clase madre, una clase con ciertas características comunes a todos los coches, pero la marca y el modelo seria la clase hija, "Nissan Versa". Es posible que luego se hagan más coches que hereden de la marca, como particularidades o agregados por ejemplo podemos tener un "Nissa Versa Sense" que herede del anterior. El coche como tal, el que conducimos, es el objeto de clase, la "instancia" en sí.

- **Polimorfismo:** El polimorfismo en la programación siempre hace referencia a que "algo" pueda tener distintas "caras", según el contexto y la necesidad. En el POO, significa que se puedan usar las clases madre, sin preocuparnos la instancia de la clase hija que realmente es, por ejemplo podemos tener una función que envía "coches" al extranjero, sin preocuparnos si es un Nissan, un volvo, o un Renault, porque es irrelevante, lo que importa es que son coches (Aquí lo relevante serían sus dimensiones)

## 10.3 Ejemplo en C# de programación orientada a objetos.

Vamos a hacer un ejemplo con una colección de clases basadas en polígonos (cuadrados, rectángulos, etc.).Veamos el diagrama**:**




- **Polígono** es una clase abstracta que representa una figura geométrica, que tiene lados, y como también un área, el cómo se calcula esta área depende del tipo de figura en sí,

vemos que esta es una clase "incompleta", sabe que es lo que tiene que hacer (Calcular un área), pero no sabe cómo hacerlo, de allí lo abstracto de la clase.

```
internal abstract class Poligono
{
    /// <summary>
    /// Numero de lados del poligono.
    /// </summary>
    public int Lados
    {
        get;
        private set;
    }

    /// <summary>
    /// Constructor del poligono.
    /// </summary>
    /// <param name="lados">Numero de lados</param>
    public Poligono(int lados)
    {
        this.Lados = lados;
    }

    /// <summary>
    /// Calcula la area, es una funcion abstracta, segun la figura la
area se calcula de forma diferente.
    /// </summary>
    /// <returns>Area.</returns>
    public abstract double CalcularArea();
}
```

- **Clase Rectángulo:** Un rectángulo es propiamente un polígono, que ya tiene entidad, ya es algo concreto, la clase como tal tiene ancho y largo, y una forma específica de calcular su área.

```
/// <summary>
/// Representa un rectangulo, es un poligono que tiene ancho y largo.
/// </summary>
internal class Rectangulo : Poligono
{
    /// <summary>
    /// Ancho.
    /// </summary>
    public int Ancho
    {
        get;
        private set;
```

```
    }

    /// <summary>
    /// Largo.
    /// </summary>
    public int Largo
    {
        get;
        private set;
    }

    /// <summary>
    /// Constructor del rectangulo.
    /// </summary>
    /// <param name="ancho">Ancho.</param>
    /// <param name="largo">Largo.</param>
    public Rectangulo(int ancho, int largo) : base(4)
    {
        this.Ancho = ancho;
        this.Largo = largo;
    }

    /// <summary>
    /// Calcula el area del rectangulo.
    /// </summary>
    /// <returns>Area</returns>
    public override double CalcularArea()
    {
        return Ancho * Largo;
    }
}
```

- **Clase cuadrado:** El cuadrado es a su vez un tipo de rectángulo en el que los lados miden lo mismo, aquí vemos una relación de herencia en la que un cuadrado es una particularización de un rectángulo y a su vez un rectángulo es un tipo de polígono.

```
/// <summary>
/// Tamaño del lado (todos los lados son iguales).
/// </summary>
public int TamanoLado
{
    get
    {
        return Largo;
    }
}
```

```
/// <summary>
/// Constructor de cuadrado.
/// </summary>
/// <param name="tamanoLado">tamaño de los lados.</param>
public Cuadrado(int tamanoLado) : base(tamanoLado, tamanoLado)
{
}
```

Veamos el siguiente código donde usamos todos los elementos en conjunto. En el siguiente método se suman el área de varios polígonos y se devuelve como resultado, notase que puede ser un polígono de cualquier forma, a la función eso le da igual, solo llama a la función de CalcularArea que es diferente en cada tipo de polígono (eso se llama polimorfismo, a pesar de que nuestra función trabaja con la clase base, la abstracta, se ejecuta cada uno de los métodos de CalcularArea de cada polígono).

```
private static double SumarAreas(params Poligono[] poligonos)
{
    double result = 0;
    foreach (Poligono poligono in poligonos)
    {
        //Ejecuto por cada poligo su función calcular area, que es
diferente segun el caso.
        result += poligono.CalcularArea();
    }
    return result;
}
```

Veamos cómo se consume toda esta funcionalidad

```
//Creo los poligonos, es de notar que cada poligono es de un tipo
diferente.

Poligono rectangulo = new Rectangulo(2, 5);
Poligono cuadrado = new Cuadrado(8);
Poligono triangulo = new Triangulo(10, 12);
Poligono pentagono = new PentagonoRegular(2);

// Muestro el area de cada poligono

Console.WriteLine("El area de rectangulo es: " +
rectangulo.CalcularArea().ToString("0.##"));
Console.WriteLine("El area del cuadrado es: " +
cuadrado.CalcularArea().ToString("0.##"));
```

```
Console.WriteLine("El area del triangulo es: " +
triangulo.CalcularArea().ToString("0.##"));
Console.WriteLine("El area del pentagono es: " +
pentagono.CalcularArea().ToString("0.##"));

// Muestro el area tota,
Console.WriteLine("El area total es: " + SumarAreas(rectangulo,
cuadrado, triangulo, pentagono).ToString("0.##"));
Console.ReadLine();
```

Como vemos se crean los polígonos, indicando el tipo, pero asociándolo a una variable de tipo "polígono", usando características polimórficas, vamos ejecutando cada uno de los métodos CalularArea, hasta que llamamos a SumarAreas, donde le pasamos el listado de figuras geométricas sin ningún orden en particular, puesto que esta función solo recibe polígonos.

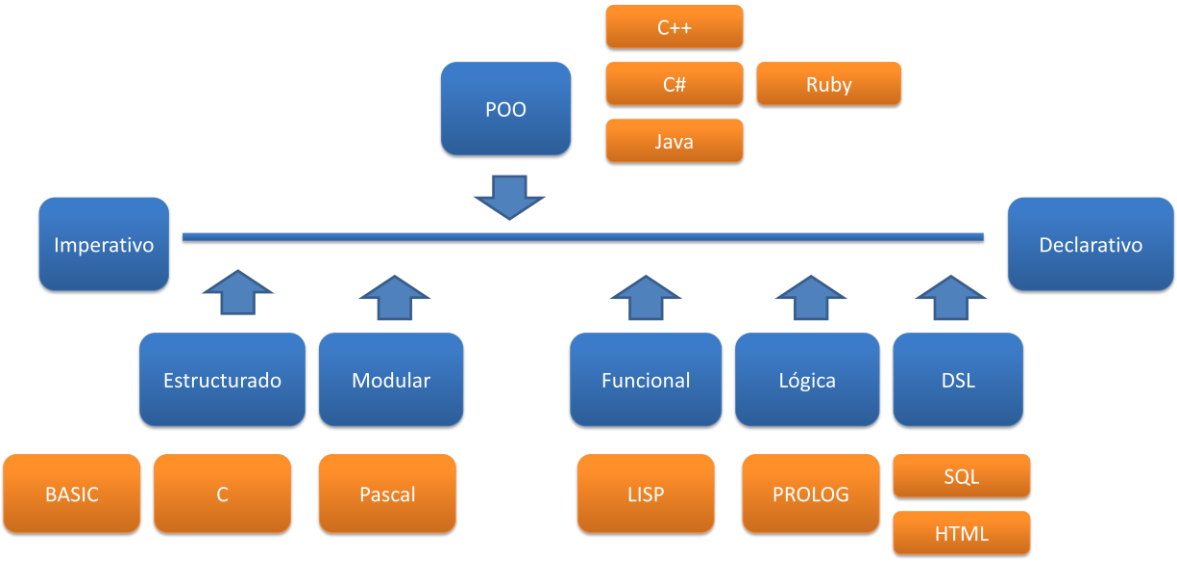
## 10.4 ¿Qué fue antes la clase o el objeto?

Una broma que encontré en internet, emulando a la famosa frase "que fue antes el huevo o la a pesar gallina", es "¿Qué fue antes la clase o el objeto?". A pesar que es un meme, se me hizo practico contestar la pregunta

Lo primero fue el objeto, esto es porque había datos y comportamiento antes de los que los clasificáramos, esto es además, el proceso de análisis normal, primero identificamos los elementos que queremos programar en un ejemplo práctico y después lo abstraemos para generalizarlo. Entonces podemos decir que la clase es realmente, una abstracción del objeto.

# Capítulo XI. **Programación declarativa**

Si pusiéramos ordenados en una línea los lenguajes de programación, estarían en un punto los más cercanos a cómo funciona una computadora, que nos hacen estar al pendiente de temas tal como la memoria, el rendimiento, o los periféricos, y en el otro punto de la línea, tendremos lenguajes llamados declarativos. En estos lenguajes se especifica que es lo que deseamos obtener y no como obtenerlo.




No hay que confundir la programación declarativa con usar el lenguaje natural para resolver un problema, si no con dejar de pensar en cómo lo resolvería la computadora y comenzar a pensar en cómo lo resolveríamos nosotros (como personas). Algunos tipos de lenguajes declarativos se parecen al lenguaje natural (en la forma de expresar necesidades), pero la mayoría se parece a declaraciones matemáticas (en las que también pensamos y estructuramos nuestras ideas)

## 11.1 Programación funcional

En la programación funcional, toda resolución de una necesidad es el resultado de una función (que devuelve un valor), es decir todo se tiene que resolver mediante el uso de funciones, (y no se secuencias de operaciones, como en la programación declarativa).

Aquí vemos el primer enfoque matemático, asociando el resultado de una operación al resultado de una función.

No existen bucles como tal, sino que todo tipo de iteración, debe ser el resultado de una recursividad. Tampoco existen variables, como tal.

Por otro lado una función, siempre devuelve el mismo resultado si se le proporciona los mismos parámetros de entrada, a diferencia de por ejemplo, la programación orientada a objetos, en la que el resultado de una función depende del estado del objeto (de sus atributos).

El principal representante de la programación funcional es LISP, y aquí hay una curiosidad, por el orden que hemos seguido, pareciera que los lenguajes declarativos están al final de la historia de los lenguajes de programación. Pues bien LISP es anterior a todos los lenguajes de programación de alto nivel, solo fortran es más viejo que él. LISP fue creado en el año 58.

¿Por qué el pronto surgimiento, de LISP?, posiblemente por facilidad de expresar pensamientos matemáticos, y la naturaleza científica/matemática que tenían de origen los primeros científicos de la computación.

Entonces, ¿Por qué se impusieron, los otros tipos de lenguajes, como los imperativos?, posiblemente por la dificultad de implementar un compilador LISP para diferentes plataformas y arquitecturas, y la dificultad de crear sistemas de aplicación (como por ejemplo las usadas en un banco o para gestionar información). Es decir en ese momento, y cuando se vio un tema comercial en las computadoras, parecía más sencillo la creación de lenguajes imperativos, que de lenguajes de corte más científico. Aquí vemos como la ingeniera (resolución practica de problemas), se impuso a la ciencia (teoría de cómo se deben resolver problemas).

Vamos a ver un ejemplo de cómo resolver un problema de forma imperativa y de forma funcional.

En este caso resolveremos un factorial. El factorial de un entero es la multiplicación de todos los números que son memores que él, por ejemplo el factorial de cinco, se calcularía así:

```
5! = 1 x 2 x 3 x 4 x 5= 120
```

Si quisiéramos resolver este problema en C, seria de la siguiente forma:

```c
#include <stdio.h>

int main ()
{
    int n=5, fact=1;

    for (int c = n; c >= 1; c--) {

        fact = fact * c;
    }

    printf ("Factorial= %d\n", fact);

    return 0;

}
```

Si quisiéramos darle otro enfoque pudiéramos, establecer las siguientes reglas.

- El factorial de uno es uno (el de cero también): **factorial(1)=1**
- El factorial de cualquier otro número (positivo y entero) es igual a el mismo número, multiplicado por el factorial del numero que lo precede: **factorial(N)=N*factorial(N-1)**

Para resolver el problema del factorial en LISP, procedemos de la siguiente forma:

```
; Declaro una funcion para calcular el factorial
(defun factorial (n)
  (if (= n 0)
      1 ;Si n es cero el resultado es 1
      (* n (factorial (- n 1))) ) ) ; En cualquier otro lugar el
resultado es N * factorial del numero anterior


(print (factorial 5))
```

En LISP todo es una función, por ejemplo si quiero realizar una resta, la función es el signo menos y los parámetros el numero original y el numero que quiero restar, por ejemplo para restar uno a n (n-1), tengo que ponerlo de la siguiente forma (- n 1).

Nótese que se expresa la necesidad, de una forma más clara y directa que en un lenguaje imperativo, estamos en sí, expresando que queremos y no como obtenerlo.

Evidentemente, es posible solucionar el tema usando recursión de una forma muy parecida en C, pero originalmente la recursividad era mal vista en los primeros lenguajes debido a que agregaban un costo extra a la ejecución de los programas, tanto es así que lenguajes imperativos como fortran no la implementaban de origen, obligando a buscar una solución interactiva a todo problema. Hoy en día, la recursividad da claridad a los problemas y hace sistemas fácilmente comprensibles, debe haber una justificación muy buena, para no buscar una solución recursiva a un problema que lo permita.

## 11.2 Programación lógica

La programación lógica es otra forma de resolver problemas expresando lo que queremos y no como resolverlo. En si se expresan una serie de condiciones que debe cumplir la solución, y el lenguaje (motor) encuentra una solución al problema.


El representante clásico sobre la programación lógica es PROLOG, creado en los años 70, veamos algunos ejemplos en PROLOG.

Como comentábamos matemáticamente hablando, el factorial se define de esta forma

- factorial(1)=1
- factorial(N)=N*Factorial(N-1)

El código el PROLOG seria:

```
% La sintaxis es factorial(N, F) -> Factorial de N es F (el resultado se guarda en F)
factorial(0, 1). %El Factorial de 0 es 1, esto es un HECHO
factorial(1, 1). %El Factorial de 1 es 1, esto es un HECHO
factorial(N, F) :-  %El factorial de F de N
       N>0,                 %N tiene que ser mayor que 0, esto es una REGLA
       N1 is N - 1,  %N1 es el entero anterior a N
       factorial(N1, F1), %F1 es Factorial de N1
       F is N * F1. %F es el numero N * el factorial anterior.
```

Una vez definido nuestra reglas (y hechos), podemos preguntarle a PROLOG, varias cosas, por ejemplo

- **¿Cuál es el Factorial de uno?**

```
? - factorial(1,F)
```

La respuesta es **`F = 1`**

- **¿Cuál es el Factorial de cinco?**

```
? - factorial(5,F)
```

La respuesta es **`F = 120`**

Ahora viene lo interesante. Puedo preguntarle si un número es factorial de otro, y me debe indicar si es cierto o falso, por ejemplo

- **¿Es 120 factorial de 5?**

```
? - factorial(5,120)
```

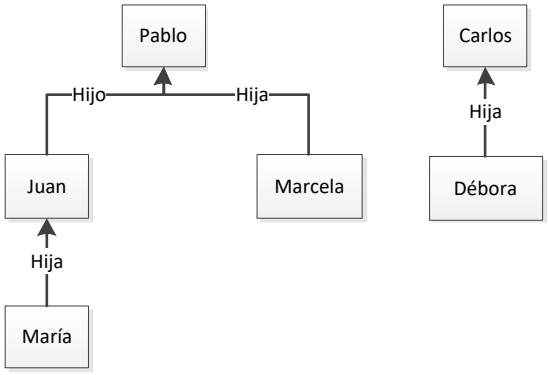
La respuesta es **`true`** (verdadero)

- **¿Es 121 factorial de 5?**

```
? - factorial(5,121)
```

La respuesta es **`false`** (falso)

Veamos otro ejemplo algo más elaborador, supongamos que tenemos este árbol familiar.




**Podemos generar en PROLOG los siguientes hechos:**

```
%%
%% declaraciones
%%
padrede('Juan', 'María'). % Juan es padre de María
padrede('Pablo', 'Juan'). % Pablo es padre de Juan
padrede('Pablo', 'Marcela'). % Pablo es padre de Marcela
padrede('Carlos', 'Débora'). % Carlos es padre de Débora
```

Y las siguientes reglas:

- **% A es hijo de B si B es padre de A**

```
hijode(A,B) :-        padrede(B,A).
```

- **% A es abuelo de B si A es padre de C y C es padre B**

```
abuelode(A,B) :-    padrede(A,C), padrede(C,B).
```

- **% A y B son hermanos si el padre de A es también el padre de B y si A y B no son lo mismo**

```
hermanode(A,B) :-    padrede(C,A), padrede(C,B), A \== B.
```

- **% A y B son familiares si A es padre de B o A es hijo de B o A es hermano de B**

```
familiarde(A,B) :-   padrede(A,B).
familiarde(A,B) :-   hijode(A,B).
familiarde(A,B) :-   hermanode(A,B).
familiarde(A,B) :-   abuelode(A,B).
```

Podemos realizar las siguientes preguntas al motor:

- **¿Juan es hermano de Marcela?**

```
?- hermanode('Juan', 'Marcela').
```

La respuesta es `true` (verdadero).

- **¿Carlos es hermano de Juan?**

```
?- hermanode('Carlos', 'Juan').
```

La respuesta es `false` (falso).

- **% ¿Pablo es abuelo de María?**

```
?- abuelode('Pablo', 'María').
```

La respuesta es `true` (verdadero).

- **¿María es abuela de Pablo?**

```
?- abuelode('María', 'Pablo').
```

La respuesta es `true` (verdadero).

- **¿Pablo y Carlos son familiares?**

```
?- familiarde('Pablo', 'Carlos').
```

La respuesta es `false` (falso).

- **¿Pablo y Maria son familiares?**

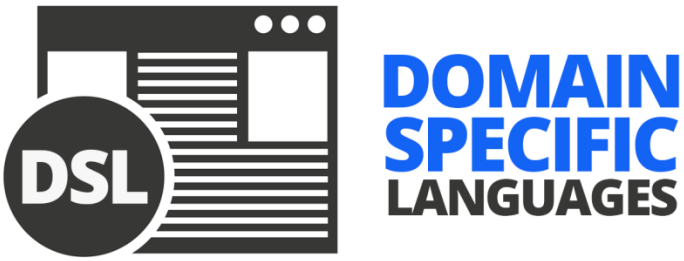
```
?- familiarde('Pablo', 'María').
```

La respuesta es `false` (falso).

Estos dos lenguajes, LISP y PROLOG, han sido influencia en muchos lenguajes de programación. Sus ideas y enfoque se han aplicado a lenguajes imperativos y orientados a objetos, debido a lo sencillo y potente que son para expresar ideas. En breve veremos algunos ejemplos de su influencia en el desarrollo de los lenguajes de programación.

## 11.3 Lenguajes específicos del dominio

Los lenguajes específicos del domino (Domain Specific Languages, DSL) son aquellos que están diseñados para resolver un problema en concreto. Lo contrario a los DSL, son los lenguajes de propósito general (es decir aquellos que no tiene un propósito en particular). Generalmente los DSL suelen ser lenguajes declarativos, debido que al enfocarse en la resolución de una necesidad en concreto intentan eliminar cualquier tipo de conocimiento sobre la arquitectura de la maquina y son más cercano el lenguaje propio de la necesidad (del dominio de la aplicación).




Algunos ejemplos de lenguaje declarativos

### Structure Query Lenguaje (SQL)

SQL es uno de los lenguajes más importantes a nivel empresarial, y es un gran ejemplo de programación declarativa y de DSL.


Es un DSL por que tiene un objetivo en concreto, que es realizar consultas sobre bases de datos, no sirve para ningún otro tipo de tarea.

¿Por qué es declarativo?, es declarativo por que especificamos (quizás en este lenguaje de forma más clara que en otros), lo que deseamos obtener, y no tenemos ningún conocimiento acerca de cómo se va a obtener.

Profundicemos más en lo anterior. SQL es estándar, que no sabe desde que plataforma se está consumiendo (puede ser Java, C++, C#, o cualquier lenguaje que se pueda conectar a una base de datos), y no sabe a qué base de datos está realizándose la consulta (puede ser MySQL, SqlServer, Oracle…), lo único que contempla el lenguaje es la posibilidad de indicar que queremos obtener, el cómo hacerlo será trabajo del motor de la base de datos.

Veamos un ejemplo en la base de datos Northwind (base de datos ejemplo de Microsoft, que se simula los datos de una empresa):

```
-- Ventas por empleado
SELECT Employees.EmployeeID
    ,Employees.FirstName
    ,Employees.LastName
    ,count(Orders.OrderId) NumeroVentas
FROM Employees
INNER JOIN Orders ON Employees.EmployeeID = Orders.EmployeeID
GROUP BY Employees.EmployeeID
    ,Employees.FirstName
    ,Employees.LastName

--Obtener los clientes que no han realizado una compra desde 1997
SELECT CustomerID
    ,CompanyName
FROM Customers
WHERE NOT EXISTS (
        SELECT *
        FROM orders
        WHERE Customers.CustomerID = ORders.CustomerID
            AND OrderDate > '1997-12-31'
        )

--Obtener los jefes intermedios, estos son jefes que tiene personal a su
cargo, en el ultimo nivel jeraquico
-- (la gente que depende de el, no tiene a su vez a gente que dependa de
ellos)
SELECT *
FROM Employees JefesIntermedios
WHERE EXISTS (
        SELECT *
        FROM Employees Dependientes
        WHERE Dependientes.EmployeeID = JefesIntermedios.ReportsTo
```

```
            AND NOT EXISTS (
                SELECT *
                FROM Employees DependientesUltimoNivel
                WHERE DependientesUltimoNivel.EmployeeID =
Dependientes.ReportsTo
                )
        )
```

En ningún caso sabemos cómo se almacena este información, ni como la va a recuperar la base de datos, solo indicamos que es lo que queremos conseguir.

## Expresiones Regulares

Las expresiones regulares son un DSL, creado para analizar cadenas de textos, si bien, en un principio parece un lenguaje complicado, una vez que se domina es casi imposible prescindir de ellos. Es un lenguaje funcionar, por que expresamos lo que queremos obtener de las cadenas de texto, y no la forma de obtenerlo.


Vamos a ver un ejemplo de expresiones regulares.

En México, cada contribuyente tiene un identificador asignado conocido como RFC (Registro federal de contribuyentes). A través de este número cada trabajador paga sus impuestos, un ejemplo, de RC, es el siguiente:

```
BETC 70 12 01 U76
```

Los campos de los componen un RFC, son los siguientes:

- Cuatro letras extraídas de los nombres y apellidos del contribuyente: BETC
- Año de nacimiento del contribuyente: 70
- Mes de nacimiento del contribuyente 12
- Día de nacimiento de contribuyente: 01
- Tres dígitos verificadores, compuestos de letras y números.

Una expresión que validara que un RFC fuera correcto (en estructura), sería el siguiente:

```
^([A-ZÑ&]{4})([0-9]{2})(0[1-9]|1[0-2])(0[1-9]|1[0-9]|2[0-9]|3[0-1])([A-Z\d]{3})$
```

- **/^** Debe ser el comienzo de la palabra
- **([A-ZÑ&]{4})** Debe contener cuatro caracteres, de la A a la Z, o Ñ o &
- **([0-9]{2})** dos numero del cero al nueve (año)
- **(0[1-9] | 1[0-2])** a un cero seguido de un numero, o un uno seguido de un cero, uno o dos (mes)
- **(0[1-9]|1[0-9]|2[0-9]|3[0-1])**, combinaciones posibles para la fecha del mes,
- **([A-Z\d]{3})** cualquier combinación de números y letras que midan tres
- **$** fin de palabra

## HyperText Markup Language (HTML)


HTML5, junto a CSS3 y JavaScript, son parte de muchas de las aplicaciones que se usan en la actualidad. Podemos tener del lado del servidor multitud de lenguajes y herramientas, pero del lado del cliente solo tendremos esta tres trabajando en conjunto.

Es común encontrar bromas acerca del uso de HTML5, pero lo cierto es que es un DSL declarativo, fundamental hoy en un día.

Es un DSL, porque tiene una misión específica, que es estructurar el contenido de un sitio web (el CSS3, le dará apariencia, y el JavaScript funcionalidad).

Es declarativo, porque no indica en ningún lado, ni forma, ni el cómo se convertirá el contenido en los gráficos y texto que vera el usuario. Es el trabajo de cada navegador el renderizar el HTML, para convertirlo en algo visual para el usuario.

Un ejemplo pudiera ser el siguiente:

Nota: el ejemplo fue extraído de https://codepen.io/Dentz/pen/wMYRXY

```
<DOCTYPE! html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Basic html layout example</title>
</head>
<body class="hello">
  <header>
    <h1>< header ></h1>
  </header>
  <nav>
    < nav >
    <ul>
      <li></li>
```

```
      <li></li>
      <li></li>
    </ul>
  </nav>
  <section>
    < section >
    <header>< header ></header>
    <article>< article ></article>
    <footer>< footer ></footer>
  </section>
  <aside>
    < aside >
  </aside>
  <footer>
    < footer >
  </footer>
</body>
</html>
```

Si mostramos el documento HTML anterior, veremos algo no muy espectacular como

< header >

< nav >

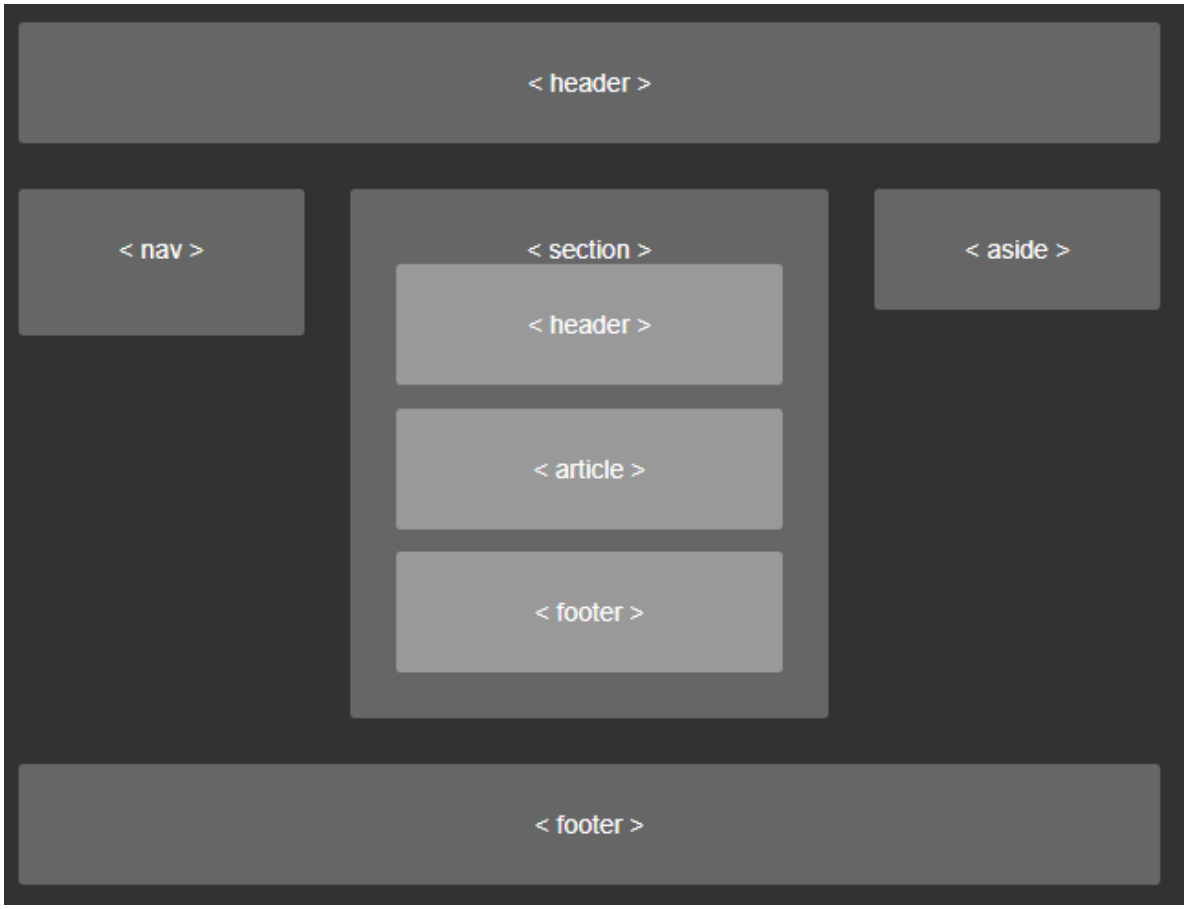
- 
- 
- 

< section >
< header >
< article >
< footer >
< aside >
< footer >

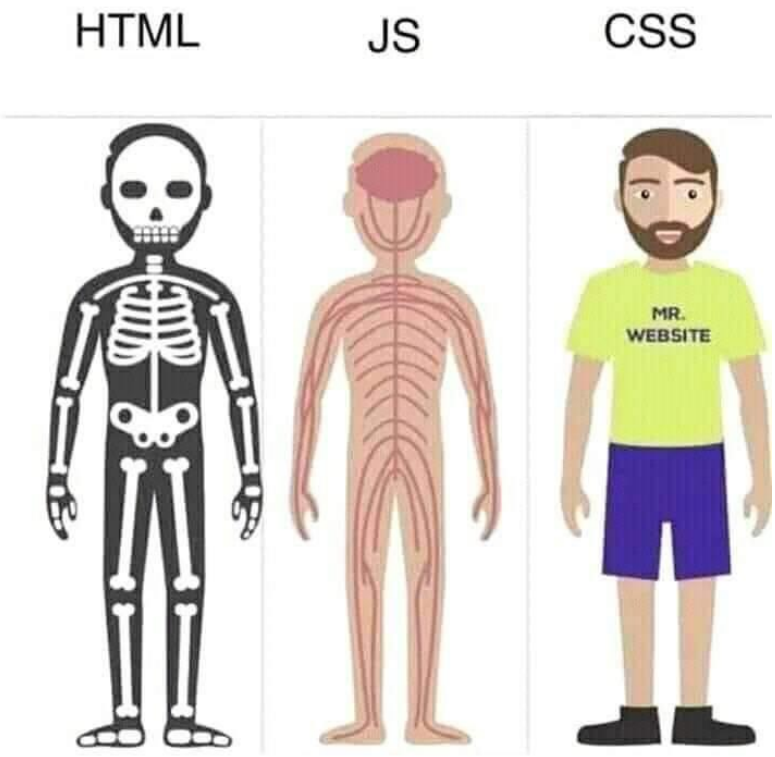
Pero en el HTML ya hemos definido los elementos estructurales de nuestra página, como la cabecera, un partes de secciones, barras de navegaciones, etc., Todos estos

componentes pueden ser además analizados fácilmente por un buscador y indexador (puesto, que tienen una identidad fácilmente compresible.

Con un poco de CSS, nuestra página puede verse así:




Una de las cosas que me gustan de HTML, CSS y JS, es como ha sabido evolucionar para tener una identidad claramente diferenciada. HTML se encarga de la estructura, CSS de la apariencia, y JS de la funcionalidad, todo está perfectamente separado, y todo tiene un lugar, con lo que se cumple la separación de responsabilidades (la 'S' de SOLID). Esto es particularmente hermoso, porque HTML nació como un lenguaje que tenía todo incrustado en su interior, apariencia, contenido y estructura, y que era realmente difícil de mantener, ahora es un modelo de como se debieran organizar las cosas, para que sean escalables.

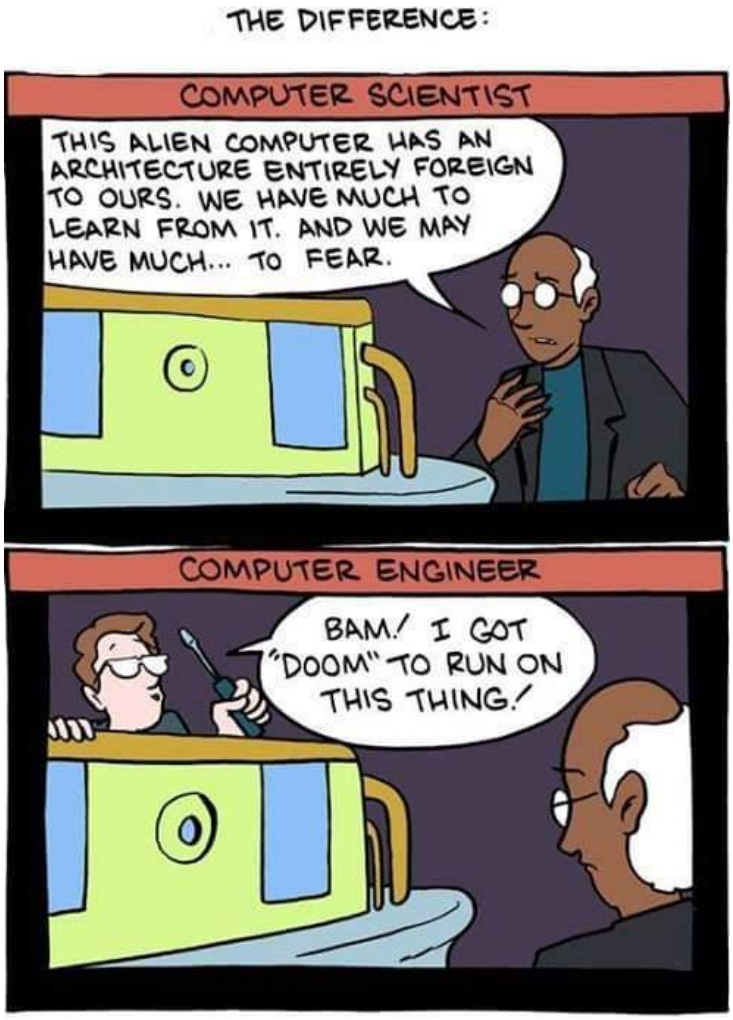

HTML
JS
CSS
MR.
WEBSITE

# Capítulo XII. Programación declarativa en lenguajes empresariales

Aunque casi todos los lenguajes empresariales se presentaban como lenguajes multiparadigma, lo cierto es que casi siempre eran lenguajes que admitían programación imperativa (con alguna de sus evoluciones) junto a programación orientada a objetos, sin acercase a algún otro tipo de paradigma. Esto cambio en los últimos años donde se incorpora, en los lenguajes de programación funciones y sintaxis declarativas que facilitan el desarrollo de software.

La programación declarativa se basa en crear código que expresa las necesidades que queremos resolver y cómo resolverlas (mas información en nuestra entrada anterior), a diferencia de la programación imperativa que se basa en seguir diversos pasos en los que indicamos como resolver un problema. Es por lo tanto mucho mas practico (y sencillo) un

código declarativo (en que queda claro el problema), que uno imperativo para el tenemos que hacer un análisis más profundo para comprenderlo.




Como curiosidad, la programación declarativa es anterior a la programación imperativa, entonces ¿Por qué se impuso la programación imperativa, sobre la declarativa entonces? Fueron motivos comerciales más que nada, la programación declarativa tenía un deje mas científico que practico (de origen), y la programación imperativa se ajustaba mas a la comprensión del cómo funcionaba una maquina (considerando sus capacidades de procesamiento y su velocidad). Parecía más sencillo de comprender programas que indicaban

una instrucción tras otra. A esto se debe añadir lo difícil de construir compiladores, ya que no se poseían las herramientas (ni las metodologías de análisis), actuales. En un mercado emergente como la computación, que se comenzaba a incorporar a las empresas, perecía más razonable diseñar lenguajes imperativos que sirvieran para vender computadoras, que lenguajes declarativos.


Un ejemplo de lo anterior es COBOL, que tiene un origen muy peculiar. Es uno de los lenguajes más antiguos existentes (del 1959). En su momento empresas como IBM estaban viendo la viabilidad de vender comercialmente y de forma masiva sus computadoras a negocios que necesitaban grandes herramientas de procesamiento de información, tal como bancos. El problema es que esas maquinas, eran monstruos enormes y terriblemente caras, y los directores de los bancos que autorizaban dichas compras no se sentían felices pagando dichas millonadas, para algo que difícilmente entendía (y para colmo tampoco entendían a los

primeros informativos y matemáticos que si las podían manejar). La solución fue COBOL, un lenguaje que "parecía", que era como escribir inglés, y con el que directores bancarios creían que podían estar más cerca de cómo funcionaban dichas maquinas, sintiéndose mas cómodos al desembolsar lo que constaba una computadora de la época (evidentemente todo era un efecto placebo, independiente de la potencia y capacidades de COBOL, solo querían saber que pudieran comprenderlo, aunque evidentemente no lo hacían). Esto es como un ejemplo de cómo un lenguaje imperativo ayudo a la venta comercial de computadoras.

Participants in COBOL's 25th Anniversary Celebration at The Computer Museum on May 16, 1985, surround the COBOL Tombstone, a gift in 1960 from Howard Bromberg (far right) to the COBOL Committee."

En lo personal creo que todavía en la actualidad es inviable crear un sistema empresarial usando completamente un paradigma declarativo, pero realmente algunas partes es mejor crearlas de forma declarativa por su sencillez y claridad, es aquí donde se pueden mezclar la programación imperativa, la programación orientada a objetos, y la programación declarativa, para usar lo mejor de cada uno, según la necesidad.

A continuación algunos ejemplos de programación declarativa en lenguajes con enfoque empresarial.

## 12.1 Ruby

Ruby es posiblemente unos de mis lenguajes preferidos, tuvo un auge muy importante a principios de siglo, aunque en esta década (a partir de 2010) comenzó a decaer, frente a su principal competidor Python.

Ruby es un lenguaje multiparadigma, con una inspiración declarativa muy fuerte, como vemos en el siguiente comentario de su creador Yukihiro "Matz" Matsumoto:

*A menudo la gente, especialmente los ingenieros en computación, se centran en las máquinas. Ellos piensan, "Haciendo esto, la máquina funcionará más rápido. Haciendo esto, la máquina funcionará de manera más eficiente. Haciendo esto..." Están centrados en las máquinas, pero en realidad necesitamos centrarnos en las personas, en cómo hacen programas o cómo manejan las aplicaciones en los ordenadores. Nosotros somos los jefes. Ellos son los esclavos.*

### Características declarativas de Ruby

- **Facilidad para expresar nuestras necesidades.**

Ruby tiene muchas facilidades para expresar lo que necesitamos de una forma clara y con pocas líneas de código:

- **Declaración de variables**

En Ruby siempre es necesario asignar un valor a una variable, al momento de declararla. El tipo no se indica explícitamente, si no que se toma del valor de la variable.

```
mensaje = 'Hola Mundo!!!'
```

Para declara un array simplemente se lo asignamos a la variable.

```
mensajes = ['Hola Juan', 'Hola Maria', 'Hola Pedro']

puts mensajes.inspect
```

Para declarar un Dictionary (tambien llamdo Hash o Map, solo declaramos los valores y las llaves:

```
# Hash vacio

hash = {}

# Hash con valores
hash =  {
            saludo_juan: 'Hola juan',
            saludo_maria: 'Hola Maria',
            saludo_pedro: 'Hola pedro',
        }

puts hash.inspect
#  Salida:  {:saludo_juan=>"Hola  juan",  :saludo_maria=>"Hola  Maria",
:saludo_pedro=>"Hola pedro"}
```

**Salida:** `["Hola Juan", "Hola Maria", "Hola Pedro"]`

- **Agregar un elemento a un array**

```
mensajes << 'Hola Jose'

puts mensajes.inspect
```

**Salida:** `["Hola Juan", "Hola Maria", "Hola Pedro", "Hola Jose"]`

- **Agrego un elemento en una posición aleatoria**

```
mensajes[6] = 'Hola Elias'

puts mensajes.inspect
```

**Salida:** `["Hola Juan", "Hola Maria", "Hola Pedro", "Hola Jose", nil, nil, "Hola Elias"]`

- **Agrego un elemento al final**

```
mensajes << 'Hola Gustavo'

puts mensajes.inspect
```

**Salida:** `["Hola Juan", "Hola Maria", "Hola Pedro", "Hola Jose", nil, nil, "Hola Elias", "Hola Gustavo"]`

- **Agrego un elemento al principio**

```
mensajes.unshift('Hola Roberto')

puts mensajes.inspect
```

**Salida:** `["Hola Roberto", "Hola Juan", "Hola Maria", "Hola Pedro", "Hola Jose", nil, nil, "Hola Elias", "Hola Gustavo"]`

- **Buscar elementos**

Buscar un elemento, aquellos mensajes que acaben con la letra o, aquí se usan expresiones lamba, las veremos después

```
encontrados=mensajes.select {|e| e=~/o$/}

puts encontrados.inspect
```

**Salida:** `["Hola Roberto", "Hola Pedro", "Hola Gustavo"]`

- **Eliminar elementos de un array (por ejemplo los nulos)**

```
mensajes.delete_if {|e| !e }

puts mensajes.inspect
```

**Salida:** `["Hola Roberto", "Hola Juan", "Hola Maria", "Hola Pedro", "Hola Jose", "Hola Elias", "Hola Gustavo"]`

- **Trabajar con un array como una cola (último en entrar último en salir)**

```
mensajes << 'Hola Edgar'
mensajes << 'Hola Luz'

puts mensajes.inspect
```

**Salida:** `["Hola Roberto", "Hola Juan", "Hola Maria", "Hola Pedro", "Hola Jose", "Hola Elias", "Hola Gustavo", "Hola Edgar", "Hola Luz"]`

```
mensajes.shift()

puts mensajes.inspect
```

**Salida:** `["Hola Juan", "Hola Maria", "Hola Pedro", "Hola Jose", "Hola Elias", "Hola Gustavo", "Hola Edgar", "Hola Luz"]`

- **Trabajar con un array como una pila (ultimo en entrar, primero en salir)**

```
mensajes.push('Hola Ruben')

puts mensajes.inspect
```

**Salida:** `["Hola Juan", "Hola Maria", "Hola Pedro", "Hola Jose", "Hola Elias", "Hola Gustavo", "Hola Edgar", "Hola Luz", "Hola Ruben"]`

```
mensajes.pop()

puts mensajes.inspect
```

**Salida:** `["Hola Juan", "Hola Maria", "Hola Pedro", "Hola Jose", "Hola Elias", "Hola Gustavo", "Hola Edgar", "Hola Luz"]`

Como vemos todo el tratamiento de los arrays es bastante declarativo, parecido a PROLOG y LISP, en ningún caso debemos preocuparnos por cómo se almacenan los elementos del array, ni en tareas tales como reservar memoria o liberarla. Tampoco debemos

conocer estructuras específicas para cada tarea propias de la programación orientada a objetos como Listas, Pilas, o Colas.

- **Bloques que contiene código**

Ruby da una mezcla muy buena entre programación orientada a objetos y programación declarativa. Básicamente casi todo está orientado a objetos, tanto es así que no existen estructuras como los bucles for para recorrer colecciones, existe un método each (de la colección) y recibe como parámetro una función que será ejecutada por cada elemento de la colección.

La función que ejecutar el método recibe el nombre de bloque y puede ser especificada de una forma declarativa entre llaves o entre las clausulas do/end.

- **Bloque especificado por llaves**

```
#Imprimir el cuadro de cada numero

[1,2,3].each { |x| puts x*2 }
# Salida:
# 2
# 4
# 6
```

- **Bloque do/end**

```
# Otra forma semejante a la anterior

[1,2,3].each do |x|
  puts x*2
end
# Salida:
# 2
# 4
# 6
```

- **Variable que almacena un bloque**

Como muestra de lo anterior es posible asociar un bloque a una variable y pasársela a un método each, para que se ejecute por cada uno de los elementos de la colección.

```
# Crear variables que almacenan un proceso

p = Proc.new { |x| puts x*2 }
[1,2,3].each(&p)

# Salida:
# 2
# 4
# 6
```

Nótese que la variable solo almacena el código que se va a ejecutar pero que no se ejecuta realmente hasta que dicha variable se pasa como parámetro al each. También es de apreciar que todos los métodos son equivalentes.

- **Comprobar si existe un valor en un array (mediante bloques).**

```
# ******************************************************
# Comprobar si un saludo elemento existe en un array
# ******************************************************

mensajes = ["Hola Juan", "Hola Maria", "Hola Pedro", "Hola Jose", "Hola
Elias", "Hola Gustavo", "Hola Edgar"]

puts( mensajes.any? {|e| e=~ /Maria/} )
# Salida: true
```

- **Eliminar elementos de un array (mediante bloques).**

```
# ******************************************************
# Eliminar elementos de un array (elimino a Maria y a Jose)
# ******************************************************

mensajes.delete_if {|e| e=~ /Maria|Jose/ }

puts mensajes.inspect
# Salida: ["Hola Juan", "Hola Pedro", "Hola Elias", "Hola Gustavo",
"Hola Edgar"]
```

- **Cambiar un array a masculas (mediante bloques).**

```
# ************************************************
# Cambiar elementos dentro de un array (Cambio el nombre a mayusculas)
# ************************************************
mensajes.map! { |e| e.upcase }

puts mensajes.inspect
#  Salida: ["HOLA JUAN", "HOLA PEDRO", "HOLA ELIAS", "HOLA GUSTAVO",
"HOLA EDGAR"]
```

## 12.2 C#

C# es un lenguaje que en los últimos tiempos ha tenido una evolución muy rápida, nació con un paradigma orientado a objetos, muy parecido a Java, y ha sabido ganarse su identidad, agregando multitud de funcionalidades declarativas que lo hacen un lenguaje claro y sencillo.

### Características declarativas de C#

- **Facilidad para declarar variables**

En momento de la creación de variables se ven ciertos aspectos declarativos, como por ejemplo:

- **Declaración de arrays**

```
// Puede probar este codigo en https://repl.it/languages/csharp

using System;

public static class Program
{
    private static void Imprimir(this string[] arrString)
    {
        Console.WriteLine($"[a \"{String.Join("\", \"", arrString)}\"]
");
    }

    public static void Main()
    {
        string[] mensajes = new string[] { "Hola Juan", "Hola Maria",
"Hola Pedro", "Hola Jose", "Hola Elias", "Hola Gustavo", "Hola Edgar" };

        mensajes.Imprimir();
        Console.ReadLine();
    }
}
```

Salida: `[ "Hola Juan", "Hola Maria", "Hola Pedro", "Hola Jose", "Hola Elias", "Hola Gustavo", "Hola Edgar"]`

- **Declaración de diccionarios**

```
// Puede probar este codigo en https://repl.it/languages/csharp

using System;
using System.Collections.Generic;
using System.Linq;

public static class Program
{
    private static void Imprimir(this Dictionary<string, string> dicionario)
    {
        var lines = dicionario.Select(e => $"\t{e.Key}: {e.Value}");

        Console.WriteLine($"[{Environment.NewLine}{String.Join(Environment.NewLine,
lines)}{Environment.NewLine}]");
    }
```

```
    public static void Main()
    {
        Dictionary<string, string> mensajes = new Dictionary<string, string>
        {
            ["saludo_juan"] = "Hola juan",
            ["saludo_maria"] = "Hola Maria",
            ["saludo_pedro"] = "Hola pedro",
        };

        //Salida:
        /*
         * [
         *   saludo_juan: Hola juan
         *   saludo_maria: Hola Maria
         *   saludo_pedro: Hola pedro
         * ]
         */

        mensajes.Imprimir();
        Console.ReadLine();
    }
}
```

- **Declaración mediante el uso de var**

Existe una clausula especial para declarar variables sin necesidad de especificar el tipo:

```
using System;

public static class Program
{
    public static void Main()
    {
        var mensaje = "Hola mundo";
        var mensajes = new string[]{"Hola Juan", "Hola Maria", "Hola
Pedro", "Hola Jose", "Hola Elias", "Hola Gustavo", "Hola Edgar"};
        var mensajesUnidos = String.Join("[\", \"", mensajes + "]");
        Console.ReadLine();
    }
}
```

Con esto se infiere el tipo (en tiempo de compilación) de la variable, en este caso son String y String[].

En lo personal no me gusta un uso excesivo de var, ya que no queda claro el tipo de la variable(al momento de leer el código), sobre todo si es el resultado deuda función. Pero esto cambia cuando estamos funciones Linq, donde sin duda, es mucho más sencillo usar la cláusula var, que especificar el tipo.

Hasta ahora hemos visto algunos tipos de declaración, que nos ayudan a especificar la creación de objetos de forma más declarativo que imperativa, aunque es, desde luego, mucho menos declarativo que Ruby.

### *Expresiones Lamba*

Clásicamente, en la programación imperativa, se tienen variables que apuntan a funciones (en lugar de a datos), de forma que se pueden almacenar la función que se va a llamar en una variable, y para realizar la llamada posteriormente, por ejemplo en C, seria de la siguiente forma:

```
#include <stdio.h>

int sumar_por_2(int numero){
  //Suma dos al número especificado
  return numero + 2;
}

int multiplar_por_2(int numero){
  //Multiplica por dos el número especificado
  return numero *  2;
}


int main(void) {

  int valor=5;

  //Puntero a funcion que devuelve recibe un int y devuelve un int
  int (*funcion) (int);
```

```
  //La función apunta a sumar 2
  funcion = &sumar_por_2;

  printf("Valor función: %d\n",funcion(valor));

  //La función apunta a sumar 2
  funcion = &multiplar_por_2;

  printf("Valor función: %d\n",funcion(valor));

  // Salida:
  /*
  * Valor función: 7
  * Valor función: 10
  */

  return 0;
}
```

De origen Java, elimino esta funcionalidad por lo que no era posible tener variables que apuntaran a código, si no que era necesario tener un objeto, que contuviera el código que queríamos ejecutar, a continuación un ejemplo de swing, donde se ve la implementación mediante Listeners (se muestra un mensaje por pantalla, al pulsar un botón).

Nótese que para el botón "uno" se usa programación orientada a objetos para indicar que debe hacerse al pulsar dicho botón, creando una clase anónima Listener. Se incluye un botón "dos" que hace lo mismo mediante programación declarativa y expresiones lamba, es, como se ve, mucho menos código y mas compresible,

```
package com.desdelashorasextras;

import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Codigo11 {
    JFrame frame;
    JButton boton1;
    JButton boton2;
```

```
    Codigo11() {
        frame = new JFrame();
        boton1 = new JButton("Pulsa el boton Uno !!");
        boton2 = new JButton("Pulsa el boton Dos!!!");

        frame.setSize(550, 500);
        frame.setLayout(null);
        frame.add(boton1);
        frame.add(boton2);

        boton1.setBounds(10, 100, 200, 60);
        //Configuro un evento de forma tradicional para cuando el
usuario haga clic,
        //se ve que necesito crear una clase anonima que capture el
evento
        boton1.addActionListener( ( new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(frame, "Gracias desde el
UNO!!");
            }
        }));

        boton2.setBounds(250, 100, 200, 60);
        //Misma situacion que la anterior pero usando programación
declarativa
        //Disponible en las versiones modernas de Java
        boton2.addActionListener(e ->
JOptionPane.showMessageDialog(frame, "Gracias desde el UNO!!"));


        frame.setVisible(true);
    }

    public static void main(String[] args) {
        new Codigo11();
    }
}
```

Java estaba en lo correcto al usar un paradigma orientado a objetos puro, donde incluso las llamadas a código mediante variables tenían que pasar por un Listener o alguna estructura semejante, pero eso hacia el código demasiado largo, enredoso, y difícil de comprender. En versiones posteriores Java introdujo las expresiones lambda y comenzó a añadir funciones declarativas y de otra índole multiparadigma.

C# (oficialmente inspirado en C++, pero en muchos aspectos claramente basada en Java) introdujo desde su primera versión la posibilidad de que las variables apuntaran a direcciones de código (a funciones directamente) de una forma bastante imperativa al principio (mediante delegados y delegados anónimos), hasta convertirse en algo mucho más declarativo (mediante expresiones lambda)

Veamos directamente el uso de expresiones lambda de forma declarativa

- **Calculo del cuadro de un array de enteros**

```
int[] numeros = new int[] { 1, 2, 3 };

// Imprimir el cuadro de cada numero
var cuadrados = numeros.Select(x => x * x);

//Salida: ["1", "4", "9"]

cuadrados.Imprimir();
Console.ReadLine();
```

El método Select, selecciona los elementos de un array, aplicándoles una posible transformación, que viene especificado mediante una expresión lamba. En este caso por cada x recibida (que es cada uno de los elementos del array), lo multiplica por sí mismo.

**x=> x * x** , en esta expresión x es algo así como el parámetro de la expresión lamba y x * x el resultado.

Otro detalle interesante, es el método Select, no pertenece al tipo de datos `int[]` sino que es un método extensor. Los métodos extensores son métodos que se definen fuera de la definición de la clase, y amplían las capacidades de esta sin modificarla, a veces incluso son aplicables a interfaces, por ejemplo, este es el método extensor que nos permite imprimir el resultado de los cuadrados.

```
private static void Imprimir(this IEnumerable<int> arraInt)
{
   Console.WriteLine($"[\"{String.Join("\",      \"",      arraInt.Select(x      =>
x.ToString()))}\"]");
}
```

El código anterior significa que para todas las clases que implemente IEnumerable <int> (Arrays, Listas, colecciones y demás), le agrega un método para imprimir su contenido, de forma que podemos llamarlo así:

```
cuadrados.Imprimir();
```

En el caso del Select y la expresión lambda asociada (x=> x * x) vemos como se combinan de forma muy simple y funcional la programación orientada a objetos y la declarativa, haciendo que el código resultante, es sencillo y fácil de entender.

- **Consultas a colecciones de forma declarativas**

Es posible aplicar consultas a colecciones de objetos por ejemplo podemos tener el objeto Persona, y podemos realizar transformaciones y selecciones de forma muy simple y clara.

```
 // Creo una colección de personas.
Persona[] personas = new Persona[]{
    new Persona { Nombre="Maria", Apellido="Jimenez"},
    new Persona { Nombre="Juan", Apellido="Perez"},
    new Persona { Nombre="Ana", Apellido="Moreno"},
    new Persona { Nombre="Ruben", Apellido="Gomez"},
    new Persona { Nombre="Pedro", Apellido="Sanchez"},
    new Persona { Nombre="Roberto", Apellido="Hernandez"}
};

// ******************************
// Comprobar si un saludo elemento existe en un array
// ******************************
```

```
Console.WriteLine($"{personas.Any(x => x.Nombre == "Maria")}");

// Salida: true

// ******************************
// Selecciono elementos no sean Maria o Juan
// ******************************

var sinMariaJose = personas
        .Where(x => x.Nombre != "Maria" && x.Nombre != "Jose")
        .Select(x => x);

sinMariaJose.Imprimir();

//Salida [ "Juan Perez", "Ana Moreno", "Ruben Gomez", "Pedro Sanchez", "Roberto
Hernandez"]

// ************************************************
// Devolver un array de elementos a mayusculas
// ************************************************

var mayusculas = personas
    .Select(x =>
        new Persona
        {
            Nombre = x.Nombre.ToUpper(),
            Apellido = x.Apellido.ToUpper(),
        }
    );

mayusculas.Imprimir();

// Salida: ["HOLA JUAN", "HOLA PEDRO", "HOLA ELIAS", "HOLA GUSTAVO", "HOLA EDGAR"]

//************************************************
// Buscar un elemento que acaben en o
// Busco aquellos mensajes que acaben con la letra o, aquí se usan expresiones
lamba, las veremos después
//************************************************

var elementosAcabanO = personas
    .Where(x => Regex.IsMatch(x.Nombre, @"o$"))
    .Select(x => x);

elementosAcabanO.Imprimir();

//Salida: [ "Pedro Sanchez", "Roberto Hernandez"]
```

C# tiene una sintaxis alternativa para las consultas parecida a SQL (aunque en lo personal no es mi forma favorita, simplifica la lectura):

```
// ******************************
// Selecciono elementos no sean Maria o Juan
// ******************************

var sinMariaJoseAlternativo = from x in personas
                              where x.Nombre != "Maria" && x.Nombre != "Jose"
                              select x;

sinMariaJoseAlternativo.Imprimir();
```

- **Consultas a colecciones de forma paralela**

Generalmente la programación paralela es complicada, hay muchos puntos a tener en cuenta, como sincronizaciones, esperas, candados, pero mediante C# y de forma declarativa, podemos recorrer un array de forma paralela, simplemente exponiéndolo con la opción "AsParallel()".

```
// ******************************
// Selecciono elementos no sean Maria o Juan
// ******************************

var sinMariaJose = personas.AsParallel()
        .Where(x => x.Nombre != "Maria" && x.Nombre != "Jose")
        .Select(x => x);

sinMariaJose.Imprimir();
```

**Salida:** `[ "Juan Perez", "Ana Moreno", "Ruben Gomez", "Pedro Sanchez", "Roberto Hernandez"]`

Simplemente poniendo AsParallel() a continuación del array se recorre de forma paralela, optimizando el proceso entre los varios núcleos de una CPU.

- Las **expresiones** lambda pueden ser datos y código

Como vemos las expresiones lambda son ideales para expresar lo que queremos conseguir, en lugar de cómo obtenerlo.

Lambda nos permite unir la programación en C# y las consultas en SQL, mediante alguno ORM, por ejemplo Entity Framework.

Por ejemplo teniendo la base de datos Northwind, pudiéramos hacer una consulta a la tabla employe de la siguiente forma:

```
using (var context = new NorthwindEntities())
{
    var consulta = from e in context.Employees
                   where e.FirstName.Contains("a")
                   select  (e.TitleOfCourtesy  +  "  "  +  e.FirstName  +  "  "  +
e.LastName);

    var resultado = consulta.ToArray();

    //SQL GENERADO
    // SELECT CASE
    //        WHEN ([Extent1].[TitleOfCourtesy] IS NULL)
    //                THEN N''
    //        ELSE [Extent1].[TitleOfCourtesy]
    //        END + N' ' + [Extent1].[FirstName] + N' ' + [Extent1].[LastName] AS
[C1]
    // FROM [dbo].[Employees] AS [Extent1]
    // WHERE [Extent1].[FirstName] LIKE N'%a%'

    var                                     sql                                     =
((System.Data.Entity.Core.Objects.ObjectQuery)consulta).ToTraceString();

    Console.WriteLine(String.Join(Environment.NewLine, resultado));

}
```

**Salida:**

```
Ms.Nancy Davolio
Dr.Andrew Fuller
Ms.Janet Leverling
Mrs.Margaret Peacock
Mr.Michael Suyama
Ms.Laura Callahan
```

```
Ms.Anne Dodsworth
```

El código anterior realiza las siguientes acciones:

1. Se conecta  la base de datos Northwind
2. Ejecuta un código de consulta SQL sobre la tabla empleados
3. Recupera el nombre completo de todos los empleados
4. Cierra la conexión.

Todo esto sin realizar los pesados pasos para conectarse a una base de datos, y ejecutar código SQL, todo se especifica de forma declarativa.

La parte declarativa puede expresarse de dos formas, y las dos son exactamente equivalentes.

- **Primera forma:**

```
var consulta = from e in context.Employees
               where e.FirstName.Contains("a")
               select (e.TitleOfCourtesy + " " + e.FirstName + " " + e.LastName);
```

- **Segunda forma:**

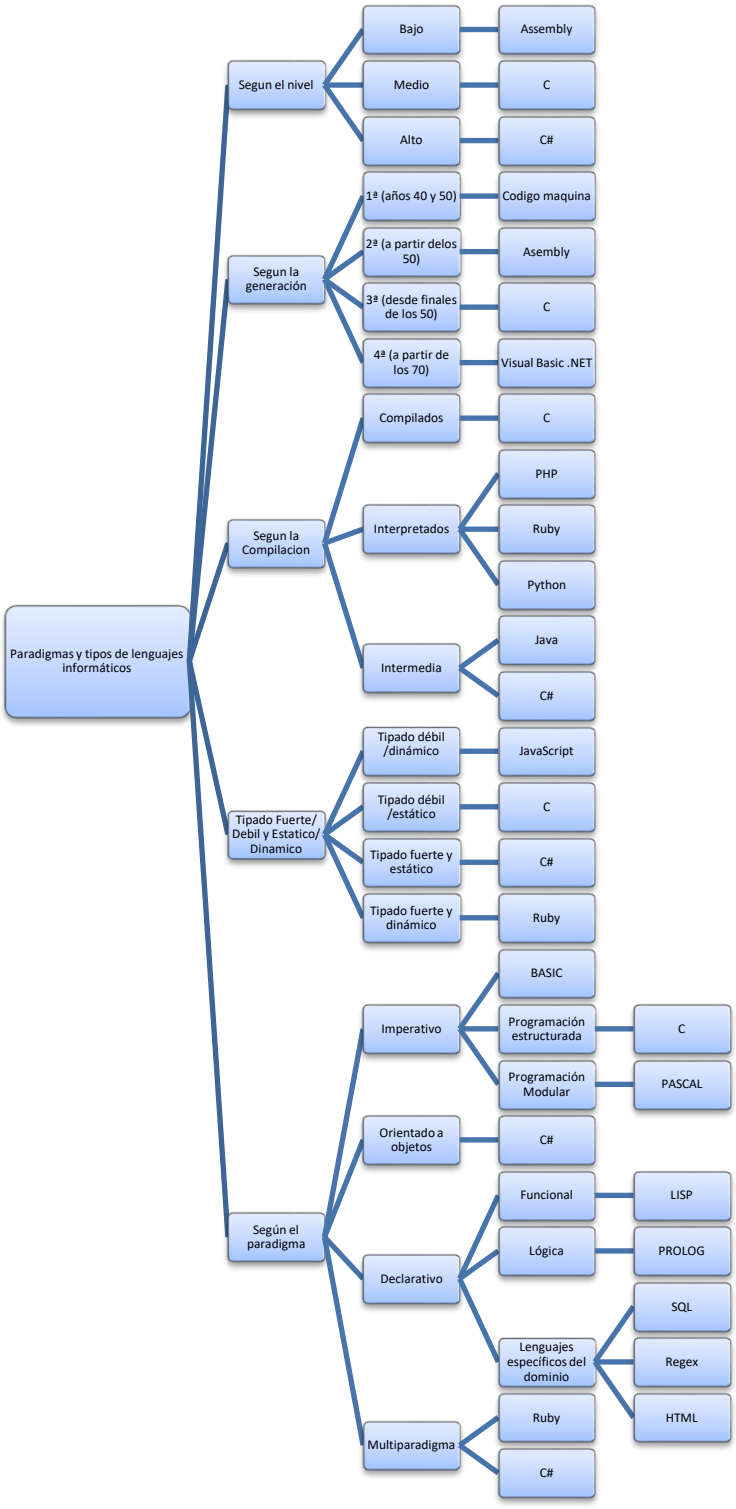
```
var consulta = context.Employees
    .Where(e => e.FirstName.Contains("a"))
    .Select(e => e.TitleOfCourtesy + " " + e.FirstName + " " + e.LastName);
```

Ahora bien, no es muy inteligente, traer una tabla completa en memoria (de la maquina cliente), y realizar un buscara allá, pero eso es lo que pensaríamos al ver la sentencia tal (sobre todo en la segunda forma, pero realmente hacen lo mismo). El caso es que la

expresión lambda en este escenario no hace la función de ser código, si no datos, que serán convertidos en una sentencia SQL, de forma completa al momento de ejecutarse, es decir mediante código podemos expresar una consulta SQL de forma íntegra y declarativa.

# ANEXO I: MAPA

Se muestra el siguiente mapa conceptual que puede facilitar la comprensión de este documento, se ilustran todas las clasificaciones de lenguajes tratadas y se muestran ejemplos de lenguajes para cada una. La selección de lenguajes de ejemplo solo corresponde al gusto personal del autor.


Paradigmas y tipos de lenguajes informáticos
Segun el nivel
Bajo
Assembly
Medio
C
Alto
C#
Segun la generación
1ª (años 40 y 50)
Codigo maquina
2ª (a partir delos 50)
Asembly
3ª (desde finales de los 50)
C
4ª (a partir de los 70)
Visual Basic .NET
Segun la Compilacion
Compilados
C
Interpretados
PHP
Ruby
Python
Intermedia
Java
C#
Tipado Fuerte/ Debil y Estatico/ Dinamico
Tipado débil /dinámico
JavaScript
Tipado débil /estático
C
Tipado fuerte y estático
C#
Tipado fuerte y dinámico
Ruby
Según el paradigma
Imperativo
BASIC
Programación estructurada
C
Programación Modular
PASCAL
Orientado a objetos
C#
Declarativo
Funcional
LISP
Lógica
PROLOG
Lenguajes específicos del dominio
SQL
Regex
HTML
Multiparadigma
Ruby
C#

# ANEXO I: CODIGO FUENTE DE EJEMPLO

**Código fuente de los ejemplos (y del documento)**

https://github.com/jbautistamartin/ParadigmasTiposLenguajes

- **Northwind.sql**
  - ParadigmasTiposLenguajes/**Base de Datos**/

En la carpeta **Base de Datos** encontrara el script para crear la base de datos Northwind, es una base de datos de Microsoft que se usa en algunos ejemplos de este documento, puede instalar una versión de SQL Server Express e implementarla allí

A continuación las carpetas donde se encuentra el códigos fuente ilustrativo de los capítulos:

- **Capítulo VII. Combinaciones de lenguajes estáticos/dinámicos y débiles/fuertes**
  - ParadigmasTiposLenguajes/Fuentes/**Tipado**/
- **Capítulo IX. Paradigma Imperativo**
  - ParadigmasTiposLenguajes/Fuentes/**Imperativo**/
- **Capítulo X. Paradigma orientado a objetos 48**
  - ParadigmasTiposLenguajes/Fuentes/POO
- **Capítulo XI. Programación declarativa**
  - ParadigmasTiposLenguajes/Fuentes/**Declarativo**/
- **Capítulo XII. Programación declarativa en lenguajes empresariales 73**
  - ParadigmasTiposLenguajes/Fuentes/**Multiparadigma**/

# ANEXO II: ENLACES ONLINE DE ESTE DOCUMENTO

**Documento en PDF**

PENDIENTE

**Enlaces en el Blog**

- **Enlace principal**

PENDIENTE

- **Paradigmas y tipos de lenguajes informáticos (1 de 3)**

https://desdelashorasextras.blogspot.com/2019/03/paradigmas-y-tipos-de-lenguajes.html

- **Paradigmas y tipos de lenguajes informáticos (2 de 3)**

https://desdelashorasextras.blogspot.com/2019/05/paradigmas-y-tipos-de-lenguajes.html

- **Paradigmas y tipos de lenguajes informáticos (3 de 3). Programación Imperativa**

https://desdelashorasextras.blogspot.com/2019/06/paradigmas-y-tipos-de-lenguajes.html

- **Paradigmas y tipos de lenguajes informáticos (3 de 3). Orientación a Objetos**

https://desdelashorasextras.blogspot.com/2019/10/paradigmas-y-tipos-de-lenguajes_30.html

- **Paradigmas y tipos de lenguajes informáticos (3 de 3). Programación declarativa**

https://desdelashorasextras.blogspot.com/2019/10/paradigmas-y-tipos-de-lenguajes_80.html

- **Paradigmas y tipos de lenguajes informáticos (3 de 3). Programación declarativa en lenguajes empresariales**

https://desdelashorasextras.blogspot.com/2019/10/paradigmas-y-tipos-de-lenguajes_0.html

**Codigo fuente de los ejemplos (y del documento)**

https://github.com/jbautistamartin/ParadigmasTiposLenguajes