CFGS DAM

Primer Curso

PROGRAMACIÓN

TEMA 1:

Introducción a la Programación

Objetivos

- Conocer las bases y jerga del mundo de la programación.

Contenidos

1. INTRODUCCIÓN

1.1. Conceptos básicos y definiciones

Para entender el concepto de programación y todo lo relativo a ella necesitamos una base conceptual y terminológica que nos permita entender los muchos conceptos que veremos a lo largo del curso.

Por ello empezaremos por comentar una serie de definiciones indispensables en este mundillo:

Tecnología:

La tecnología es un concepto amplio que abarca un conjunto de **técnicas, conocimientos y procesos, que sirven para el diseño y construcción de entes, cosas u objetos para satisfacer necesidades humanas.**

Proviene del griego tekne (técnica, oficio) y logos (ciencia, conocimiento).

En el mundo empresarial, la tecnología es empleada para eliminar tareas repetitivas, lo cual redunda en una mayor rapidez/facilidad para desempeñar la labor diaria.
Si se consigue mayor eficiencia en el trabajo, estamos consiguiendo mayor productividad para la empresa, y por tanto, y hablando siempre en términos generales, mayores beneficios para la empresa.

Informática:

Ciencia que estudia el tratamiento automático de la información en dispositivos electrónicos y sistemas informáticos.

Proviene del francés informatique y fue acuñado por el ingeniero Philippe Dreyfus en 1962. Formó una conjunción entre las palabras "information" y "automatique".

Trata de **automatizar** todas las **tareas repetitivas** que tengan que ver con la **información**.

Datos/información:

Son aquellos elementos considerados como unidades de tratamiento dentro de un sistema de proceso de datos.

Pueden ser básicamente de 2 tipos: de Entrada (aquellos pendientes de procesar/elaborar) **y de Salida** (aquellos obtenidos una vez elaborados los datos iniciales).

Al conjunto de los datos se les denomina *información*, y ésta es tan importante que se dice que vivimos en una **sociedad de la información**.

Nota:

> Una sociedad de la información es una sociedad en la que la creación, distribución y manipulación de la información forman parte importante de las actividades culturales y económicas.
> La sociedad de la información es vista como la sucesora de la sociedad industrial.
> Tanta información ha podido comprimirse en soportes de almacenamiento (como los HD) o enviarse a través de señales, gracias a que han sido traducidos a **formatos digitales**.

Programación:

La programación es ***el instrumento que permite la ejecución de las tareas automatizadas de un sistema informático.***

Trata de imitar el pensamiento humano, el cual ejecuta unas u otras acciones en base a un razonamiento lógico a partir de unas premisas (conjunto de datos o información).

Las *herramientas* que utilizaremos para programar son los lenguajes de programación, a través de las cuales ***codificaremos*** los programas escritos en un lenguaje cercano a nosotros (como Java o Python) a un lenguaje cercano a la máquina (como los ejecutables).

Lenguaje de programación:

Conjunto de notaciones, símbolos y reglas sintácticas para posibilitar la escritura de un *algoritmo*, que posteriormente será interpretado por el hardware del ordenador.

Algoritmo:

Procedimiento paso a paso para resolver un problema en una cantidad finita de pasos.
Ej.: receta de cocina que resuelve el prob de hacer una comida.

Instrucción:

Regla, norma o paso dado para la realización de una tarea.
Es decir, **a cada paso del algoritmo se le llama orden o instrucción.**

Normalmente, si hablamos en un entorno no informático se utiliza la expresión “orden”, haciéndolo con el término instrucción en caso contrario.

Así para que algo o alguien realice el trabajo que nosotros queremos que haga, debemos darle un conj ordenado y organizado de instrucciones.

Programa:

Vamos a ver 3 definiciones (=3 Puntos de vista) para comprender mejor el concepto:

- Conjunto ordenado de instrucciones, perfectamente legibles por el ordenador, que permiten realizar un trabajo o resolver un problema.
- Descripción/implementación de un algoritmo en un lenguaje inteligible por la máquina.
- Conjunto de órdenes, diseñadas y creadas a través del razonamiento lógico, que respetan la sintaxis de un determinado lenguaje de programación. Este conj de órds se transmiten al ord para la realización y ejecución de tareas concretas.

Características:

- **Finito**: Todos los progs deben acabar alguna vez.
- **Preciso**: para q un programa escrito con un lenguaje sea preciso, hay q detallar el orden.
  Todos los lengs imponen al programador q siga un orden. (al = q las frases en español/ing/francés...)
- **Correcto**: Si le introducimos los mismos datos de entrada, el programa debe obtener siempre el mismo resultado.

Es decir:

> Un programa debe ser finito, es decir, tiene que tener un inicio y un fin, tiene que estar bien confeccionado para que, al introducir un dato, salga una solución y si se volviese a introducir el mismo dato, saliese de nuevo la misma solución.

Aplicación informática:

Es la unión o conjunto de uno ó más programas relacionados entre sí, **junto con la documentación** generada durante el proceso de desarrollo de dicha aplicación.
También reciben el nombre de paquete informático.

Metodología de programación:

Se entiende como metodología de la programación al *conjunto de normas, métodos y anotaciones que nos indican la forma de programar*, esto es, el conjunto de conceptos relativos a la forma de razonar y estructurar un problema para su resolución.
Cada lenguaje de programación sigue una metodología distinta.

Entorno de desarrollo:

Conjunto de herramientas utilizadas para elaborar y ejecutar un programa.

Se utiliza mucho la expresión ***IDE*** (Integrated Development Envelopment) o entorno integrado de desarrollo.

1.2. Fases de elaboración de un programa. Ubicación de la fase de codificación dentro del ciclo de vida clásico.

La realización de trabajos mediante computadora, como cualquier otra actividad (ingeniería, arquitectura, etc.) requiere un método que explique de un modo ordenado y secuencial hasta los últimos detalles a realizar por la máquina.

Al igual que para construir una casa, la empresa constructora no comienza por el tejado, sino que se encarga a un arquitecto el diseño de unos planos, al jefe de proyectos un calendario de actividades, etc., una aplicación informática, sobre todo si tiene cierta complejidad, no debe comenzar nunca por la codificación del programa, sino que exige una serie de fases previas destinadas a conocer todos los aspectos del problema planteado y estudiar las posibles soluciones.

Cualquiera que tenga unos conocimientos básicos de programación puede escribir un programa. Pero eso no implica que esté bien hecho.

Para entenderlo mejor, pensemos en que un arquitecto fuera directamente a hacer la obra sin hacer previamente los cálculos de estructuras, inspección del terreno,... si te haces un castillo de arena en la playa evidentemente no vas a hacer cálculos para hacerlo; te pones y lo haces; pero si se trata de un producto complejo debes planificar y detallar muchos aspectos antes de acometer su construcción.

La principal razón para que las personas aprendan lenguajes y técnicas de programación es utilizar la computadora como una herramienta para resolver problemas. Debido a ello, no hay más remedio que conocer las distintas etapas por las que pasa una aplicación desde que se concibe hasta que cae en desuso.

La resolución de un problema exige, al menos, los siguientes pasos, denominados **ciclo de desarrollo** de una aplicación:

1. **Definición del problema→** enunciado q os proporcionará el profesor, y que, en la vida real, proporciona el cliente tras una entrevista.
2. **Análisis del problema→** entender el enunciado y realizar organigrama para tener una visión general de la posible solución.
3. **Diseño del algoritmo→** realizar un ordinograma / pseudocódigo q detalle la solución
4. **Transformación del algoritmo en un programa→** codificar el paso anterior en un lenguaje de programación (C por ejemplo), de forma q se pueda traducir a código máquina.
5. **Ejecución y validación(pruebas) del programa.**

El desarrollo de una aplicación se basa en un concepto llamado ***ciclo de vida*****,** que es el **proceso desde que nace ésta hasta muere (=deja de utilizarse).**
Está compuesto de:

- **El ciclo de desarrollo más**
- **El proceso de explotación y mantenimiento.**

Ciclo de Vida clásico

En su aspecto más clásico el ciclo de vida se divide en **7 etapas o fases que hay que seguir secuencialmente y de forma ordenada** cuando se desea desarrollar un determinado producto de software:

1. Análisis del sistema (estudio y planificación)
2. Análisis de los requisitos del sw (análisis)
3. Diseño del sistema.
4. **Programación o codificación del sistema.**
5. Pruebas del sistema.
6. Explotación del sistema.
7. Mantenimiento del sistema.

En la programación es muy importante seguir un estilo, ***una metodología***, y no porque el código fuente "quede más bonito", sino porque se gana en legibilidad y eficiencia.

Hay que entender y asumir la diferencia entre alguien que consigue que sus programas funcionen y alguien que no sólo elabora sus programas de forma coherente, sino **que consigue mejorar la ejecución de dichos programas**, ya sea en velocidad o en el consumo de recursos, tanto en el presente como en el futuro.

> Por ejemplo, si documentas y estructuras bien un programa, éste podrá ser manipulado y mejorado por otros programadores; en caso contrario es muy probable que muera.

Por ello, nosotros, a la hora de resolver un problema o ej (podríamos ir directamente a la fase 4 y resolverlo como hacen muchos malos programadores), *vamos resolverlo* (incluso si se trata de un problema muy sencillo) de la forma en la que se deberían hacer todos (o, casi todos) los programas:

1. Haremos un **estudio (análisis) del problema a resolver**, proponiendo un algoritmo implementado a base de dibujos y razonamientos lógicos.
   Aquí generalmente haremos un organigrama.

2. A partir de ese análisis **construiremos un ordinograma y/o pseudocódigo** que lo resuelva.

   *Nota: Normalmente después de esta fase de diseño se entrega el **cuaderno de carga** (con toda la doc. de las fases anteriores) al programador/conj de programadores.*

3. Finalmente realizaremos la **fase de codificación**, en la que realizaremos las pruebas oportunas así como la **documentación** del programa.

   Nota1: Realmente la fase de documentación abarca todas las etapas, pero nosotros sólo la haremos en esta para aplicaciones sencillas.

   Nota2: Además procuraremos que nuestros programas tiren siempre antes de memoria que de disco, el cual usaremos sólo para lo imprescindible.

2. LENGUAJES DE PROGRAMACIÓN.

Como ya intuimos al llegar a estas líneas, y como corroboraremos a lo largo de todo este curso, el objetivo de la asignatura es **realizar programas** que resuelvan algún problema o que mejoren algún automatismo **utilizando un lenguaje programación**.

Si estuviéramos **en el mundo actual** y nos tuviéramos que comunicar con alguien que acabamos de conocer de otro país, **¿Qué idioma usarías?**

> Es evidente, que s**i los dos hablaseis un idioma común sería sencillo**, pero esa posibilidad, y más en países como el nuestro (y como Francia, USA, china, etc.) donde todavía gran parte de la población solo conoce bien un idioma.

Si no se habla un idioma común se necesita algún tipo de traductor que conozca ambos idiomas y que se encargue de traducir entre ellos, para hacer así efectiva la comunicación.

Si esto lo extrapolamos al mundo de la informática, veremos que el ordenador habla un lenguaje único: el ***lenguaje máquina***; y si queremos decirle a la máquina que construya algo, debemos darle órdenes en ese lenguaje, o no nos entenderá para poderlas llevar a cabo (al estilo de un obrero al recibir órdenes de su capataz en una obra).

Pero, ¿hablas el lenguaje máquina?
¿Eres capaz de escribir tus programas en un lenguaje complejo que además varía según la arquitectura Hardware en la que te mueves?

Si no lo haces –lo normal- entonces deberás recurrir a un **traductor**; un traductor entre el lenguaje máquina y algún lenguaje de alto nivel de entre los disponibles actualmente en el mercado.

En el mundo real todos conocemos un idioma "de nacimiento", ya que nuestros padres y nuestro entorno nos enseñan el suyo para poder convivir y relacionarnos en él.

Nosotros debemos hacer lo mismo, y en este curso haremos lo que en nuestro primero años de vida: aprender un lenguaje que conozca algún traductor para que nos comuniquemos con el ordenador.

Pero ahora la pregunta es: ¿**Qué lenguaje elegir**? ¿Por cuál me decanto para construir la aplicación si hay casi tantos como en el mundo hablado?

Evidentemente mi consejo es que aprendas cuantos más mejor, y a ser posible de **distinta raíz o paradigma**.

¿Y porque? ¿No hay un "inglés" en la informática? ¿Qué ventajas tengo al estudiar una variedad de lenguajes diferentes que es poco probable que uno llegue a utilizar?
Pues son:

- Hacer posible una mejor elección del lenguaje de programación en relación al proyecto que vaya a desarrollar.
- Mejorar la habilidad para desarrollar algoritmos eficaces.
- Acrecentar el propio vocabulario con construcciones útiles sobre programación.
- Facilitar el aprendizaje de un nuevo lenguaje.

Nosotros durante este curso elegiremos 1: **Java**

Los motivos los abordaremos con detalle a lo largo del módulo, pero básicamente son su paradigma, que es ampliamente empleado en diferentes ámbitos o sectores, y por ser la base de otros lenguajes de programación.

2.1 Lenguajes de alto y bajo nivel

Cuando escribimos un programa lo que hacemos es crear un fichero de texto con numerosos símbolos y caracteres, que unidos y siguiendo una sintaxis concreta forman lo que se denomina programa. (= q cuando vemos un libro/artículo en lenguaje natural -español por ejemplo-).

Dichos símbolos y caracteres son traducidos a un conjunto de señales eléctricas representadas en código binario (0 Y 1). El motivo de la conversión es que el micro (= cerebro del ordenador) sólo entiende un lenguaje, q es el código binario ó código máquina.

Los lenguajes de programación se pueden clasificar en 2 grandes grupos:

2.1.1. Lenguajes de bajo nivel

Son aquellos q x sus características se encuentran más próximos a la arquitectura de la máquina q al lenguaje natural.

Son totalmente dependientes de la estructura de la maq.
(Es decir, si cambio el HW de mi ordenador, mi programa probablemente deje de funcionar).

Se Diseñaron para aprovechar al máximo las características de la máquina.

En este grupo se encuentran el lenguaje máquina y el lenguaje ensamblador:

Lenguaje Máquina

Una computadora sólo puede entender el lenguaje máquina. El lenguaje de máquina ordena a la computadora (a través del micro) realizar sus operaciones fundamentales (leer de memoria, acceder a un dispositivo de E/S, hacer una operación aritmética,...).

En los inicios de la informática los programadores escribían en un archivo una "interminable" ristra de 0's y 1's que luego pasaban a ejecutar inmediatamente. Esto lo podían hacer porque conocían perfectamente la máquina a nivel de HW.

Ventajas:

- Usar este lenguaje directamente reportaba la ventaja de que se ejecutaba rapidísimamente (dentro de las posibilidades de la época, claro) y además
- era código 100% optimizado para la máquina en q se ejecutara el código.

Desventajas:

- Es difícil de usar/manejar para lar persona porque trabajar con números no es muy cómodo; además estos números están en formato binario.
- Los archivos "fuente" son enormes, por lo que encontrar fallos o simplemente modificarlos (aunque fuesen por la misma persona) es una odisea.
- No es nada portable. Cada procesador entiende su propio y particular lenguaje máquina q no podrá ser entendido x cualquier otro.
- En realidad cada micro tiene un conjunto de microinstrucciones escritas en SU lenguaje maq, q no tienen por qué coincidir con las escritas para otro micro.

Lenguaje ensamblador

Para facilitar y agilizar su labor a los programadores, se buscaron nuevos lenguajes. El lenguaje ensamblador surge como sustituto del lenguaje máquina y está basado en pequeñas abreviaturas de letras y números procedentes del inglés (llamadas mnemotécnicos).

Surgieron porque se observaba q muchas secuencias de ceros y unos se repetían continuamente al realizar ciertas operaciones. Entonces se decidió agruparlas (empaquetarlas) y darle un nombre al paquete→el mnemotécnico.

Con la aparición de este lenguaje, se tuvieron q crear los programas traductores para convertir los programas escritos en lenguaje ensamblador a lenguaje máquina. Estos lenguajes aun requerían muchas instrucciones para realizar operaciones simples.

Ventajas:

- Códigos rápidos, eficientes y optimizados para una arquitectura determinada.
- Los códigos maquina generados son más cortos, eficientes, se ejecuta más rápido y ocupan menos memoria q si se hubieran desarrollado en un lenguaje de alto nivel.

Nota:

Hoy en día no hay más q ver la falta de optimización en favor de la abstracción al triunfar los lenguajes y entornos de cada vez más alto nivel.
Esto donde mejor se ve es en herramientas generadoras de código tales como dreamweaver y frontPage, q generan gran cantidad de código genérico válido "para todo" pero muy poco optimizado.

Desventajas:

- Dependen de la arquitectura, por lo que no son portables a otros sis.
- Son difíciles de escribir, probar y de mantener.
- Son difíciles de aprender.

Ejemplos de código escrito con el lenguaje ensamblador:

ejemplo ensamblador: Hola Mundo!

```
DOSSEG
.model small
.stack 100h
.data
msgHello DB "Hola mundo!",13,10,"$"
.code
  mov ax,@data              ; Strichpunkt leitet Kommentar ein!
  mov ds,ax                 ; Datensegment initialisiert

  mov dx,offset msgHello    ;
  mov ah,9                  ; Ausgabe eines String
  int 21h                   ;

  mov ax,4C00h              ; Programm beenden
  int 21h                   ;
END
```

```
; Primer programa de prueba. Inicializa el Puerto B y pone todos los
; pines del puerto en un estado lógico "1"
; Fecha: 17.01.07   Autor: Jorge A. Bojórquez   micropic.wordpress.com

        list        p=16f628a       ; Declaración del procesador
        include     p16f628a.inc    ;
        __config    0x3F38          ; Declaración de la configuración

                                    ; Inicio del programa
        org     0x00                ; Vector de Inicio
        goto    Inicio              ; Ir a la etiqueta 'Inicio'

Inicio  bsf     STATUS,RP0  ; Seleccionar el banco de memoria 1
        clrf    PORTB       ; Configurar puerto B como salida
        bcf     STATUS,RP0  ; Seleccionar el banco de memoria 0

        movlw   0xFF        ; Cargar al acumulador W el valor 0xFF
        movwf   PORTB       ; Pone todos los pines del Puerto B en "1"

Ciclo   goto    Ciclo

        end
```

2.1.2. Lenguajes de alto nivel

Son aquellos q x sus características se encuentran más próximos al lenguaje natural q al lenguaje máquina, haciéndonos más sencilla su codificación.

Se encuentran por tanto más próximos al programador (usuario) q a la máquina, estando dirigidos a solucionar problemas mediante el manejo y tratamiento de algoritmos + EDD's (Estructuras de Datos).

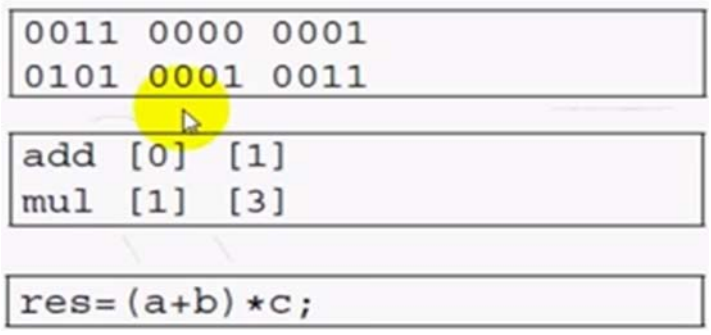
Además, un mismo programa en alto nivel puede compilarse para diferentes arquitecturas, por lo que son "independientes" de la arquitectura del ordenador, lo q implica que puedan ser usados en distintos micros y SSOOs (plataformas), siempre y cuando el compilador contemple al micro/SSOO para el q se creará el código máquina.

Nota: Evidentemente me refiero a los archivos fuente como archivos independientes de la arquitectura, y no a los archivos exe generados por el compilador, pues estos si q son dependientes de un determinado micro y SSOO.

Características:

- Permiten **abstraer** al programador del funcionamiento interno de la máquina.
- 1 instrucción de alto nivel equivale a varias(1..N) en ensamblador.
- Los hay de propósito general (=se pueden aplicar a cualquier tipo de aplicación) y de propósito específico (como COBOL para negocios o FORTRAN para entornos científicos).
- Necesitan de un traductor q entienda tanto el cod fuente como las características de la máquina.
- Usan tipos de datos.

Al final, estos lenguajes son los que se usan para la mayoría de desarrollos, ya que, a pesar de que son menos eficientes y aprovechan peor los recursos, son más sencillos de usar y modificar.
Miremos esta imagen para comprenderlo:



En ella vemos cómo hacemos una operación matemática en lenguaje máquina, ensamblador y C.
¿Cuál te resulta más familiar?

NOTA: También podéis oír hablar en estos términos, de los *lenguajes de medio nivel*, dentro de los que se encuadra C.

Estos lenguajes se denominan así porque tienen características de ambos tipos; así C puede acceder a los registros del sistema, trabajar con direcciones de memoria, etc... (características todas ellas de bajo nivel), y a su vez puede (características de alto nivel)

2.2. El Proceso de Traducción

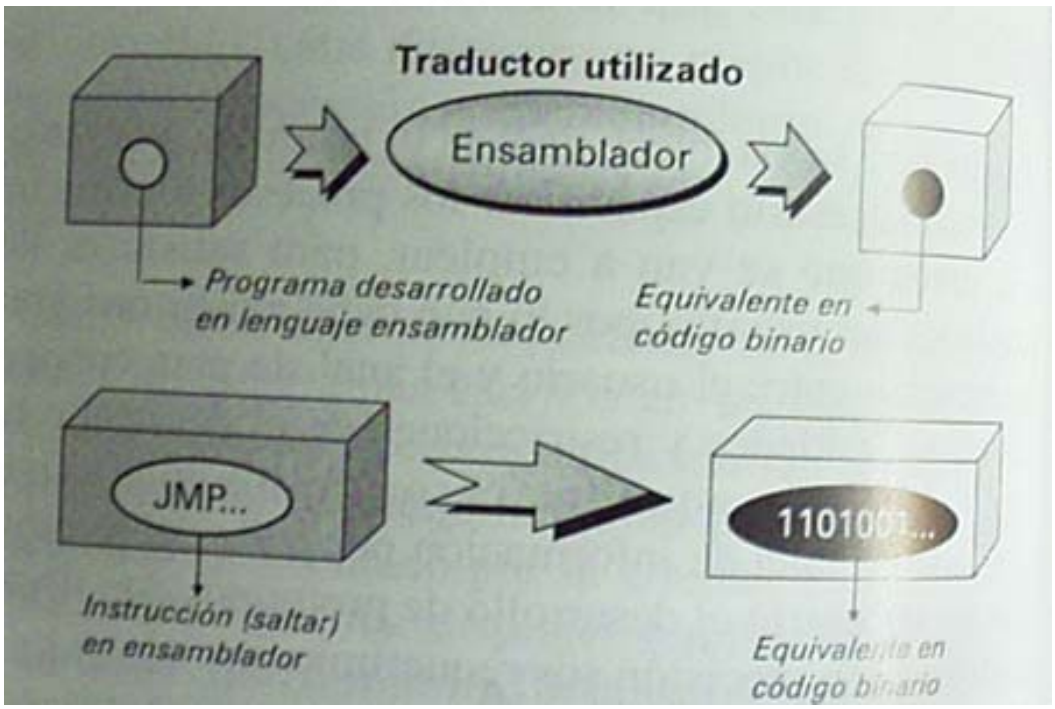
Cuando programamos en un lenguaje distinto al lenguaje máquina, los programas diseñados (escritos en código fuente y denominados "programas fuente") deben ser traducidos a código binario, para q así las instrucciones en ellos especificadas puedan ser entendidas y ejecutadas x la CPU.

El sistema o programa de SW encargado de realizar este proceso de traducción entre un lenguaje y otro recibe el nombre de Traductor del lenguaje.

Los traductores pueden ser de tres tipos: ensambladores, compiladores e intérpretes.

2.2.1. Ensambladores

Son los encargados de traducir los progs escritos en ensamblador a cod máq.



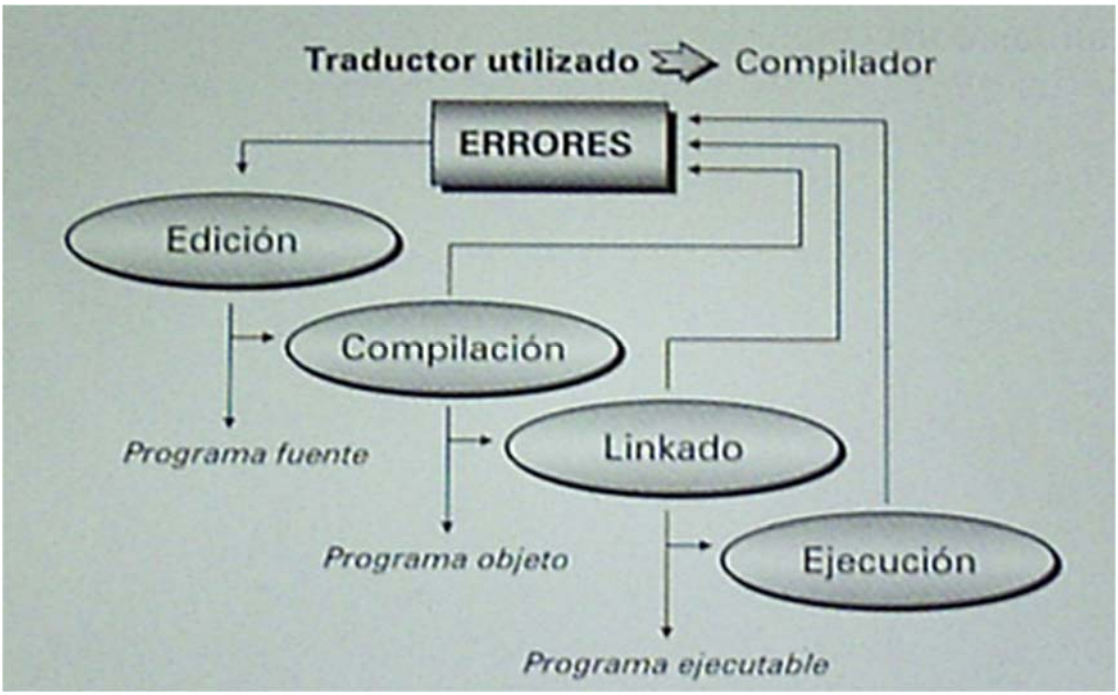
2.2.2 Compiladores

Un compilador es un programa que lee el código escrito en un lenguaje (lenguaje origen), y lo traduce o convierte en un programa equivalente escrito en otro lenguaje (lenguaje objetivo).

Como una parte fundamental de este proceso de traducción, el compilador le hace notar al usuario la presencia de errores en el código fuente del programa. (Ver figura).

EJ: C++ es un lenguaje que utiliza un compilador y su trabajo es el de traducir el código fuente escrito en C++ a código escrito en lenguaje máquina.

Etapas del proceso de compilación



1) Edición

Esta fase consiste en la escritura del programa empleando algún lenguaje de programa y un editor (que puede proporcionar el SOP o un entorno integrado del lenguaje q suele incluir editor, compilador y otras utilidades) en un soporte de AMT permanente (Disco normaly).

El resultado de esta operación se conoce como "programa fuente".

2) Compilación
En esta fase se traduce el cod fuente a su equivalente en cód máq, obteniendose, en caso de no producirse ningún error, el denominado "programa objeto".
En caso de errores→ el compilador los muestra usando los msjs correspondientes, q nos ayudarán a corregir el cod fuente y proceder de nuevo a su compilación.

El compilador de C++ al realizar su tarea realiza una comprobación de errores en el programa, revisa que todo este en orden (por ejemplo variables y funciones bien definidas), revisa todo lo referente a cuestiones sintácticas (por ej q no haya ninguna palabra mal escrita), etc...

NOTA: esta fuera del alcance del compilador que por ejemplo el algoritmo utilizado en el problema funcione bien.

3) Linkado
Esta fase también recibe el nombre de montaje o enlazado.

Consiste en unir/enlazar el objeto con determinadas rutinas internas del lenguaje y, si el método de programa es modular, se enlazan los distintos módulos xa obtener así el programa ejecutable.

En nuestro código fuente solemos tener incluidas funciones como printf(), fgets(), fopen(), ...
¿Dónde está el código de todas estas funciones?

Pues ya están compilados en forma de DLL's (Windows) o en forma de objetos (.obj en Unix) en las librerías que podamos encontrar en directorios como /lib /usr/lib o /usr/local/lib (Unix).

Lo que tenemos que hacer es añadir dinámicamente o estáticamente estos binarios a nuestro código ya compilado.

Hay dos tipos principales de linkado:

- **Linkado Estático**: Los binarios de las librerías se añaden a nuestros binarios compilados generando un "gran" archivo ejecutable.
- **Linkado Dinámico**: Esto no añade los binarios de las librerías a nuestro binario, si no que hará que nuestro programa, cargue en memoria la librería en el momento que la necesite. Esto permite que nuestro programa no engorde demasiado.

4) Ejecución
Esta fase, aunq no forma parte del proceso de compilación propiamente dicho es interesante conocerla en este punto porque también produce errores, q habrá q depurar a traves de un nuevo proceso de compilación.

Consiste en llevar el programa a mem xa su ejecución, cosa q hace el SOP cdo el usu invoca la exe del programa.
Inicialmente se debe comprobar el buen funcionamiento del programa mediante el uso de unos juegos de prueba.

2.2.3. Intérpretes

Los intérpretes en lugar de producir un Lenguaje objetivo, como en los compiladores, lo que hacen es realizar la operación de traducción y ejecución "simultánea". Un intérprete lee el código como esta escrito y luego lo convierte en acciones, es decir, lo ejecuta en ese instante.

Existen lenguajes que utilizan un intérprete, como por ejemplo VBasic, y su intérprete traduce en el instante mismo de lectura, el código leído en lenguaje máquina para que pueda ser ejecutado.

La siguiente figura muestra el funcionamiento de un intérprete:

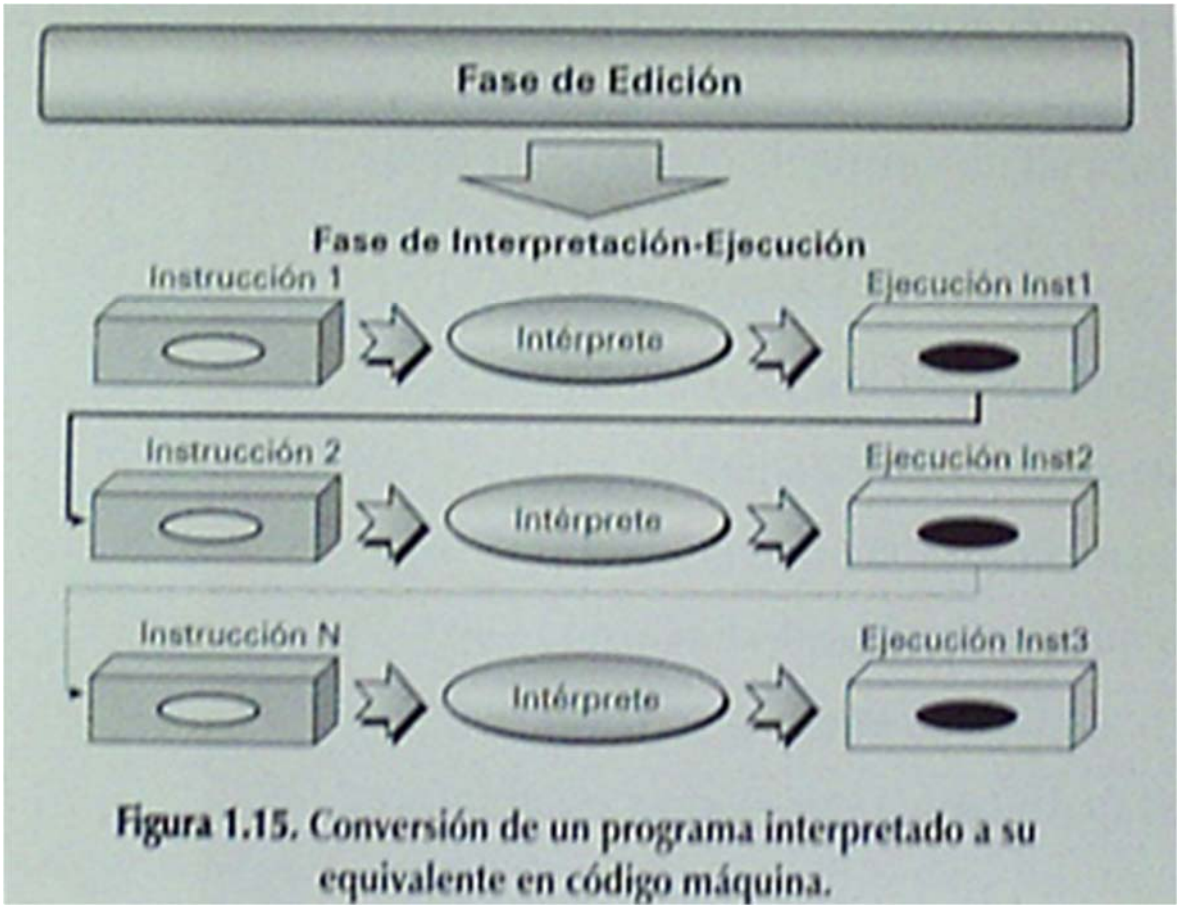


Figura 1.15. Conversión de un programa interpretado a su equivalente en código máquina.

Diferencia entre Compilador e Intérprete

Los compiladores difieren de los intérpretes en varios aspectos:
Un programa que ha sido compilado puede correr por si solo, pues en el proceso de compilación se lo transformo en otro lenguaje (lenguaje máquina).

Un intérprete traduce el programa cuando lo lee, convirtiendo el código del programa directamente en acciones.

La ventaja del intérprete es que dado cualquier programa se puede interpretarlo en cualquier plataforma (sistema operativo), en cambio el archivo generado por el compilador solo funciona en la plataforma en donde se lo ha creado.

Pero por otro lado un archivo compilado puede ser distribuido fácilmente conociendo la plataforma, mientras que un archivo interpretado no funciona si no se tiene el intérprete.

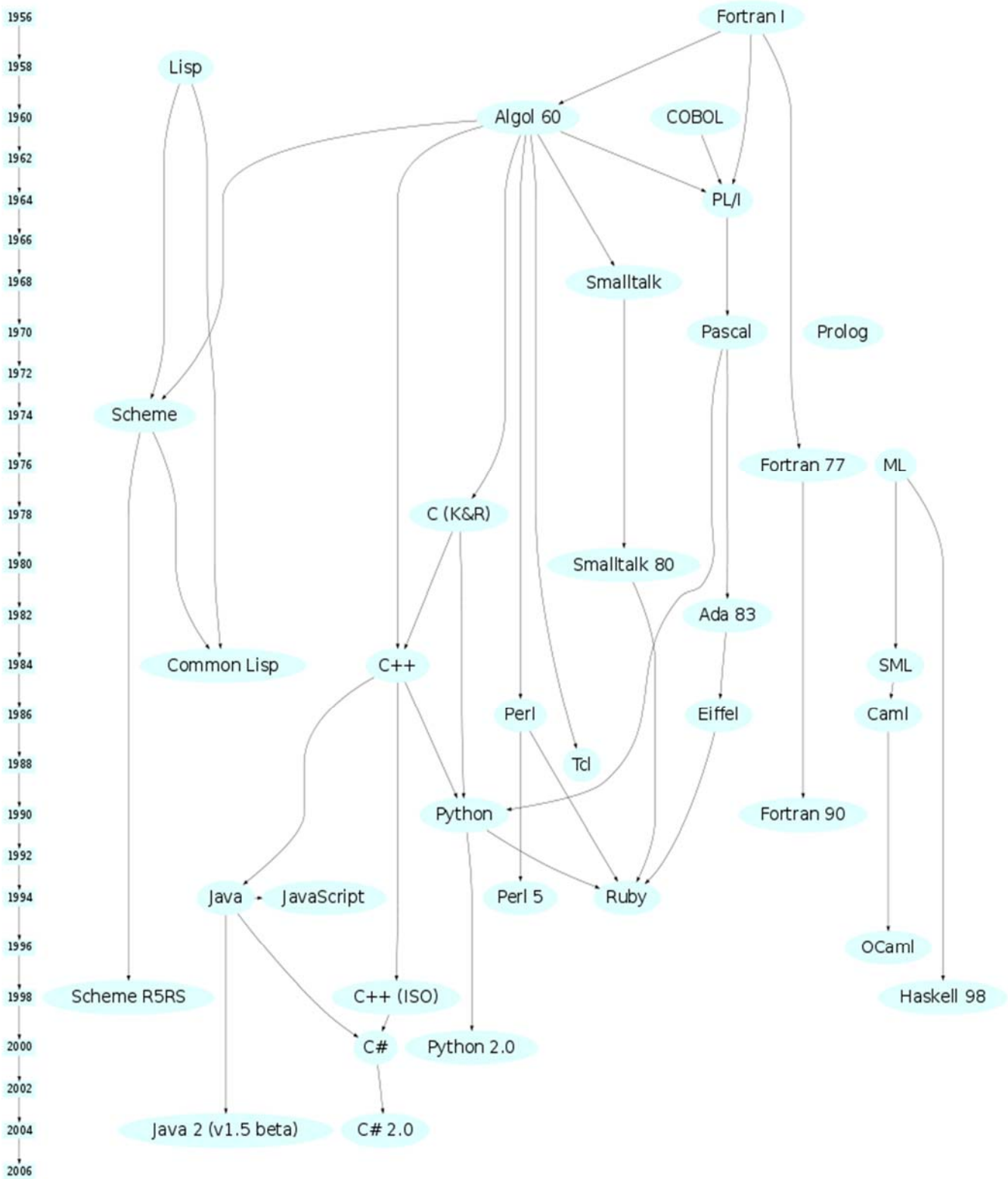
Hablando de la velocidad de ejecución un archivo compilado es de 10 a 20 veces más rápido que un archivo interpretado.

2.3. Los lenguajes más importantes

Dado que existen muchos tipos de lenguajes, vamos a conocer, al menos a modo orientativo y de características, los lenguajes más conocidos a lo largo de la historia.

Evolución

Empecemos por conocer de un vistazo la evolución de los lenguajes más importantes a lo largo de la historia:



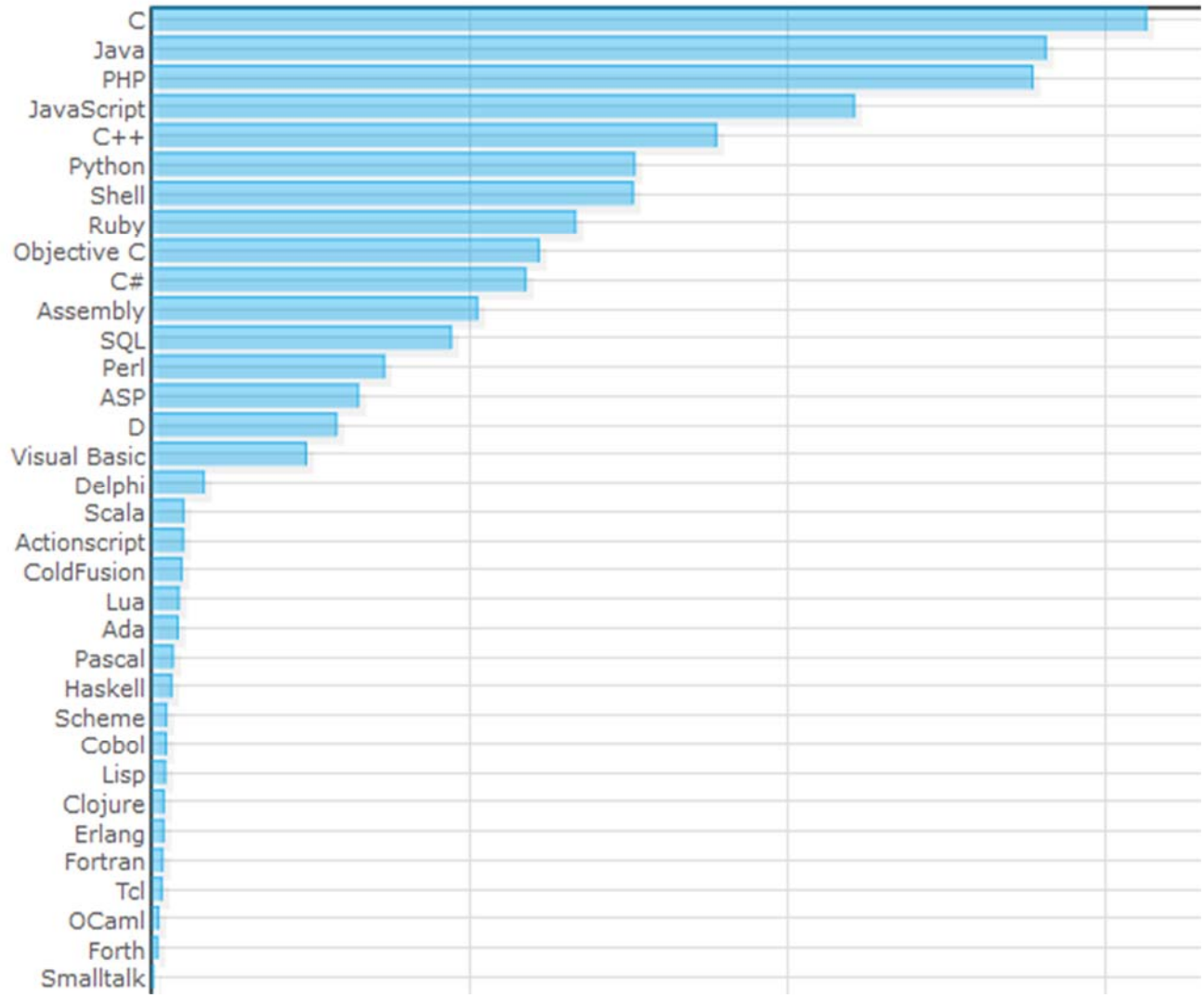
Una lista parcial de algunos de los lenguajes de programación más importantes, junto con su año de creación:

1957 FORTRAN	1962 SIMULA	*1975 Pascal*	1983 Ada	1990 Haskell	*2000 C#*
1958 ALGOL	1964 BASIC	1975 Scheme	*1986 C++*	1991 Python	2003 Scala
1960 Lisp	1964 PL/I	1975 Modula	1986 Eiffel	1993 Ruby	2003 Groovy
1960 COBOL	1970 Prolog	1983 Smalltalk-80	1987 Perl	*1995 Java*	2009 Go
1962 APL	*1972 C*	1983 Objective-C	1988 Tcl/Tk	1995 PHP	

Rankings

Veamos ahora algunos estudios por diferentes entidades.

El más conocido /reputado puede ser el ínidce TIOBE: https://www.tiobe.com/tiobe-index/ donde vemos que Java y C son los más empleados hoy en día (2019) seguidos de Python, que ganando en auge, como se ve respecto a la imagen de 2015:



Tras este ya hay muchos otros:

- El de StackOverflow: https://insights.stackoverflow.com/survey/2019#technology-_-most-loved-dreaded-and-wanted-languages

- El de Spectrum: http://spectrum.ieee.org/computing/software/top-10-programming-languages)):

- El de GitHub: https://madnight.github.io/githut/#/pull_requests/2019/1

Cada uno con diferentes resultados e ¿intereses?

Tipos de Lenguajes

Podemos agrupar los lenguajes en grupos, en relación a su función principal o fin para el que principalmente fueron construidos.

Así tenemos:

- **Lenguajes científicos** (Ejemplo: FORTRAN).
- **Lenguajes para negocios** (Ejemplo: COBOL).
- **Lenguajes para Inteligencia Artificial** (Ejemplo: LISP).
- **Lenguajes para sistemas** (Ejemplo: C).
- *Lenguajes de propósito general* (Ejemplo: Java)

Características principales de los lenguajes más importantes

Veamos las características de **los más relevantes**.

BASIC

BASIC (Beginner's All Purpose Symbolic Instruction Code) fue desarrollado en 1964 por John Kemeny y Thomas Kurtz. Fue desarrollado para enseñar a los estudiantes de Dartmouth College a cómo usar computadoras y escribir programas. BASIC se dice ser un lenguaje muy simple de aprender y uno fácil de traducir, además de ser capaz de hacer casi todas las tareas de una computadora desde inventario hasta cálculos matemáticos.

BASIC incluyó componentes de FORTRAN y ALGOL. El primer BASIC tenía 14 comandos, entre ellos LET, PRINT, GOTO, IF THEN. BASIC era un programa compilado, queriendo decir que las instrucciones era traducidas a lenguaje o código de máquina antes de ser ejecutadas.

Para el 1970, existían más de 20 versiones diferentes de BASIC corriendo en computadoras "mainframe" y micro-computadoras.

Diez años más tarde, una versión de BASIC fue escrita por Bill Gates y Pauel Allen e incluida en la "Altair", (la primera computadora personal) y luego de tres décadas, BASIC se convirtió el lenguaje programación de computadoras más popular. En adición a las versiónes de BASIC de Microsoft y Eubanks, se escribieron muchas más, incluyendo RadioShack Level 1 BASIC, Apple Ineteger BASIC, RMBASIC, Better BASIC, QuickBASIC, y Professional BASIC.

Para mediados de los 1980's, varias cien versiónes de BASIC estaban siendo utilizadas.

En 1998, 14 años luego de comenzar esta tarea, el "American National Standards Institute" (ANSI), sometió el Standard BASIC.

Hoy día, las versiones de BASIC estan cayendo a un lado, algunas han sido remplazadas con versiones que se unieron a Standard Basic, y nuevas versiones estan influenc iadas por programación orientada a objetos (OOP). OOP crea módulos de data y procedimientos de manipulación. En 1989, en un artículo de la revista BYTE, Bill Gates habla de una nueva versión visual de BASIC, que sería una mezcla de código, objetos "programador-written", y objetos visualmente específicados.

Tres años más tarde, Microsot lanza al mercado Visual BASIC, una versión diseñada específicamente para desarrollar y realzar aplicaciones para Microsoft Windows.

C/C++
C es un lenguaje de programación creado en 1972 por Dennis M. Ritchie en los Laboratorios Bell como evolución del anterior lenguaje B, a su vez basado en BCPL.

Tras la aparición de B y BCPL, dos de los primeros lenguajes creados en los Laboratorios Bell, Ken Thompson, con B y a partir de las especificaciones BCPL, creó las primeras versiones del sistema operativo UNIX, durante el año 1970, en una computadora DEC PDP-7.

En 1972, Dennis Ritchie, estaba finalizando su proyecto "El lenguaje C", una evolución del B, lo usó para crear una nueva versión, mucho más operativa de UNIX.
Es decir en 1972 se tomó la decisión de escribir nuevamente UNIX, pero esta vez en el lenguaje de programación C. Este cambio significaba que UNIX podría ser fácilmente modificado para funcionar en otras computadoras (de esta manera, se volvía portable) y así otras variaciones podían ser desarrolladas por otros programadores.
AT&T puso a UNIX a disposición de universidades y compañías, también al gobierno de los Estados Unidos, a través de licencias. Una de estas licencias fue otorgada al Departamento de Computación de la Universidad de California, con sede en Berkeley. En 1975 esta institución desarrolló y publicó su propio sucedáneo de UNIX, conocida como Berkeley Software Distribution (BSD), que se convirtió en una fuerte competencia para la familia UNIX de AT&T.

Al contrario que sus antecesores, C era un lenguaje con tipos, es decir, que cada elemento de información ocupaba un 'palabra' en memoria y la tarea de tratar cada elemento de datos como número entero, real, o arrays, no recaía en el programador.

Actualmente la mayoría de los sistemas operativos se codifican en C o sucedáneos.

A finales de los años 70 y principios de los 80 C alcanzó gran auge entre los programadores; esto supuso un inconveniente: al ser tan universal, habían muchas variantes, además bastante incompatibles, creando serios problemas para los desarrolladores de software, que necesitaban escribir código para diferentes plataformas, lo que eliminaba las características de portabilidad y de compatibilidad.

Por tanto se pedía “a gritos” que se creara una versión universal de C. Esta tarea fué llevada a cabo por el organismo estadounidense ANSI, quien en 1989 aprobó el estándar “ANSI C”. Este organismo, que cooperó con su homónimo europeo ISO, publicó el documento en 1990 y es posible pedirle una copia de ese manuscrito.

Ahora, volvamos al principio de los ochenta. Bjarne Stroustrup, diseñó una extensión del lenguaje C, llamándolo C con Clases. El término clase provenía de Simula 67 (lenguaje de programación), y servía para entender mejor el comportamiento del mundo real y llevarlo a los códigos de programación, ocultando los detalles de su implementación.

En 1984, C con Clases fue rediseñado en un compilador al que se denominó C++, tal y como lo relata la nota Data Abstraction in C, en el Technical Journal de AT&T Bell Laboratories. (vol. 63, núm 8, Octubre 1984).

En 1985 estuvo disponible la primera versión del lenguaje C ++ y se realizó el libro de Bjarne Struostrup: The C ++ Programming Languaje, publicado por Addison-Wesley en 1986.

El nombre de C ++ se debió a que era una variante del C original. En el lenguaje C, el operador ++ significa incrementar la variable; por tanto se eligió el nombre de C ++ ya que éste agregaba al C original el término de Programación Orientada a Objetos (POO), basadas en Simula 67.

Al ser C ++ una variación de C, los programas codificados en C pueden correr tranquilamente en C ++.

En 1990, el lenguaje ha sido descrito por Stroustrup y Ellis en el Annotated C ++ Reference Manual editado por Addison -Wesley, existiendo una versión en español del mismo, con el título de C ++. Manual de Referencia con anotaciones publicado por Addison-Wesley/Días de Santos en 1994.

COBOL

El deseo de desarrollar un lenguaje de programación que fuera aceptado por cualquier marca de computadora, reunió en Estados Unidos, en Mayo de 1959, una comisión (denominada CODASYL: Conference On Data Systems Languages) integrada por fabricantes de computadoras, empresas privadas y representantes del Gobierno. Dando lugar a la creación del lenguaje COBOL, llamándose esta primera versión COBOL-60, por ser éste el año que vio la luz.

Así, el lenguaje COBOL (acrónimo de COmmon Business-Oriented Language, Lenguaje Común Orientado a Negocios) fue creado en el año 1959 con el objetivo de crear un lenguaje de programación universal que pudiera ser usado en cualquier ordenador, ya que en los años 1960 existían numerosos modelos de ordenadores incompatibles entre sí, y que estuviera orientado principalmente a los negocios, es decir, a la llamada informática de gestión.

COBOL, estaba en constante evolución gracias a las sugerencias de los usuarios y expertos dando lugar a las revisiones de 1.961, 1.963 y 1.965. La primera versión standard nació en 1968, siendo revisada en 1.974, llamadas COBOL ANSI o COBOL-74, muy extendidas todavía. En la actualidad es COBOL-85 la última versión revisada del lenguaje COBOL, estando pendiente la de 1.997.

¿Porque se hablaba de fabricantes de computadoras y no de S.O., como en la actualidad? Si que es significativo, pero por aquellos años no existían Sistemas Operativos abiertos, sino que cada fabricante tenía su propio S.O. y por lo tanto cada Cobol debería valer para cada computadora. Ciertamente no había mucha diferencia entre ellos.

Cobol es un lenguaje compilado, es decir, existe el código fuente escrito con cualquier editor de textos y el código objeto (compilado) dispuesto para su ejecución con su correspondiente runtime. Cuando se ve un programa escrito en Cobol saltan a la vista varios aspectos:

Existen unos márgenes establecidos que facilitan su comprensión.

Está estructurado en varias partes, cada una de ella con un objetivo dentro del programa.

Nos recuerda mucho al lenguaje inglés, puesto que su gramática y su vocabulario están tomados de dicho idioma. En contraste con otros lenguajes de programación, COBOL no se concibió para cálculos complejos matemáticos o científicos, de hecho solo dispone de comandos para realizar los cálculos mas elementales, suma, resta, multiplicación y división, sino que su empleo es apropiado para el proceso de datos en aplicaciones comerciales, utilización de grandes cantidades de datos y obtención de resultados ya sea por pantalla o impresos.

Con Cobol se pretendía un lenguaje universal, sin embargo, los numerosos fabricantes existentes en la actualidad han ido incorporando retoques y mejoras, aunque las diferencias esenciales entre ellos es mínima.

Con la llegada del Sistema Operativo Windows, son muchos los que intentan proveer al Cobol de esa interface gráfica, Objective Cobol, Visual Object Cobol de Microfocus, Fujitsu PowerCobol, Acucobol-GT, Vangui y Cobol-WOW de Liant (RM), etc. que están consiguiendo que éste lenguaje siga estando presente en moda visual de ofrecer los programas. Sin embargo, son muchas las empresas que siguen dependiendo del Cobol-85 tradicional para sus proyectos debido principalmente a la estructura de su sistema informático.

FORTRAN

Fortran que originalmente significa Sistema de Traducción de Fórmulas Matemáticas pero se ha abreviado a la FORmula TRANslation, es el más viejo de los establecidos lenguajes de "alto-nivel", fue diseñado por un grupo en IBM durante los años 50 (1950).

El idioma se hizo tan popular en los 60s fue cuando otros vendedores empezaron a producir sus propias versiones y esto llevó a una divergencia creciente de dialectos (a través de 1963 había 40 recopiladores diferentes).

Fue reconocido que tal divergencia no estaba en los intereses de los usuarios de la computadora o los vendedores de la computadora y para que FORTRAN 66 se volviera el primer idioma en ser regularizado oficialmente en 1972 La publicación de la norma significó que ese Fortran se llevó a cabo más ampliamente que cualquier otro idioma.

A mediados de los años setenta se proporcionaron virtualmente cada computadora, mini o mainframe, con un sistema FORTRAN 66 normal. Era por consiguiente posible escribir programas en Fortran en cualquier sistema y estar bastante seguro que éstos pudieran moverse para trabajar en cualquier otro sistema bastante fácil. Esto, y el hecho que pudieran procesarse programas de Fortran muy eficazmente.

La definición normal de Fortran se puso al día en 1970 y una nueva norma, ANSI X3.9-1978, fueron publicadas por el Instituto de las Normas Nacional americana. Esta norma era seguida (en 1980) adoptado por la Organización de Normas Internacionales (ISO) como una Norma Internacional (ES 1539: 1980). El idioma es normalmente conocido como FORTRAN 77 (desde que el proyecto final realmente se completó en 1977) y es ahora la versión del idioma en su uso extendido.

Muchos rasgos deseables no estaban disponibles, por ejemplo, en FORTRAN 77 es muy difícil de representar datos estructura sucintamente y la falta de cualquier medios del almacenamiento dinámico que todas las series deben tener un tamaño fijo que no puede excederse; estaba claro de una fase muy temprana, más moderno, el idioma necesitó ser desarrollado.

El trabajo empezó en los 80s en un idioma conocido como "Fortran 8x". El trabajo tomó 12 años en parte debido al deseo de guardar FORTRAN 77 un subconjunto estricto y también para asegurar esa eficacia, pero el idioma no se compuso. Idiomas como Pascal, ADA y Algol son muy fáciles de usar pero no pueden igualar la eficacia de Fortran.

Fortran 90 es un desarrollo mayor del idioma pero no obstante incluye todos los de FORTRAN 77 como un subconjunto estricto y para que cualquier FORTRAN conformando normalmente como el programa del 77 continuará siendo un programa valido en Fortran 90. Muchos hombre han puesto toda su vida en escribir estos programas que, después de tantos años de uso, es muy fiable.

Además de las viejas estructuras de FORTRAN 77, Fortran 90 permite expresar los programas de maneras que se satisfacen más a un ambiente de la informática moderna y han quedado obsoletos muchos de los mecanismos que eran apropiados en FORTRAN 77.

En Fortran 90 algunos rasgos de FORTRAN 77 han sido reemplazados por rasgos mejores, más seguros y más eficaces, muchos de éstos fueron quitados de la siguiente revisión interina del idioma Fortran 95.

Como la norma de Fortran 90 es muy grande y compleja hay (inevitablemente) un número pequeño de ambigüedades y conflictos, las áreas grises.

Las tales anomalías a menudo sólo vienen a observarse cuando se desarrollan compiladores. En los últimos años el idioma basado en Fortran 90 conocido como High Performance Fortran (HPF) se ha desarrollado. Este idioma contiene todo de Fortran 90 y también incluye otras extensiones que son muy deseables. Fortran 95 incluirá muchos de los nuevos rasgos de HPF.

JAVA

En 1991 un grupo de ingenieros de Sun Microsystems liderados por Patrick Naughton y James Gosling comienza el desarrollo de un lenguaje destinado a generar programas independientes de la plataforma en la que se ejecutan. Su objetivo inicial nada tiene que ver con lo que hoy en día es Java, sus creadores buscaban un lenguaje para programar los controladores utilizados en la electrónica de consumo.

Existen infinidad de tipos de CPU distintas, y generar código para cada una de ellas requiere un compilador especial y el desarrollo de compiladores sabemos que es caro.

Después de dieciocho meses de desarrollo aparece la primera versión de un lenguaje llamado OAK que más tarde cambiaría de nombre para convertirse en Java. La versión

de 1992 está ampliada, cambiada y madurada, y **a principios de 1996 sale a la luz la primera versión de Java**.

Los inicios son difíciles, no se encuentran los apoyos necesarios en Sun y el primer producto que sale del proyecto, un mando a distancia muy poderoso y avanzado, no encuentra comprador. Pero el rumbo de Java cambiaría debido a una tecnología completamente ajena a los controladores de electrodomésticos: Internet.

Mientras Java se estaba desarrollando, el mundo de las comunicaciones crecía a una velocidad de vértigo, Internet y principalmente el mundo World Wide Web dejaban los laboratorios de las universidades y llegaban a todos los rincones del planeta. Se iniciaba una nueva era y Java tuvo la suerte de estar allí y aprovechar la oportunidad.

En 1993, con el fenómeno Internet en marcha, los desarrolladores de Java dan un giro en su desarrollo al darse cuenta de que el problema de la portabilidad de código de los controladores es el mismo que se produce en Internet, una red heterogénea y que crece sin parar, y dirigen sus esfuerzos hacia allí.

En 1995 se libera una versión de HotJava, un navegador escrito totalmente en Java y es en ese mismo año cuando se produce el anuncio por parte de Netscape de que su navegador sería compatible con Java. Desde ahí otras grandes empresas se unen y Java se expande rápidamente. No obstante, las primeras versiones de Java fueron incompletas, lentas y con errores. Han tenido que pasar varios años de desarrollo y trabajo para que Java sea un lenguaje perfectamente asentado y lleno de posibilidades.

Actualmente es ampliamente utilizado en entornos tanto relacionados con Internet como completamente ajenos a la Red. El mundo Java está en constante desarrollo, las nuevas tecnologías surgen y se desarrollan a gran velocidad haciendo de Java un lenguaje cada día mejor y que cubre prácticamente todas las áreas de la computación y comunicaciones, desde teléfonos móviles hasta servidores de aplicaciones.

PASCAL

La historia de Pascal estaría incompleta sin trazar primero la historia del ALGOL, del cual Pascal es una evolución. ALGOL comenzó en 1.958, cuando un comité de representantes del GAMM (una organización europea de científicos en informática) y ACM (su contrapartida en USA) se reunieron en Zurcí y produjo un informe preliminar sobre un "International Algebraic Language", o IAL.

Este lenguaje, conocido más tarde como ALGOL 58, atrajo mucho interés y fue implementado sobre varias computadoras. Los representantes europeos y estadounidenses se reunieron de nuevo en París en 1.960 para considerar una versión completamente nueva de este lenguaje, conocida como ALGOL 60.

Durante este período, ALGOL fue extremadamente popular entre los científicos informáticos, y su definición rigurosa marcó nuevos estándares para el diseño e implementación de lenguajes. ALGOL se convirtió en un lenguaje universal para la definición de algoritmos publicados en revistas.

Con el paso del tiempo, fueron apareciendo nuevas versiones revisadas de ALGOL 60, como ALGOL W (desarrollado por Niklaus Wirth) o ALGOL 68, que fue intencionadamente un lenguaje de propósito general con aplicaciones en un amplio rango de interés, aunque rápidamente se reconoció como un lenguaje demasiado ambicioso para ser práctico. De esta manera, Wirth diseñó un sucesor más reducido del ALGOL 60 y lo llamó PASCAL.

El primer compilador de PASCAL se implementó en 1.970 y una versión revisada fue definida e implementada en 1.973. PASCAL fue claramente diseñado para servir como un lenguaje para enseñar diseño de algoritmos y metodología de programación.

Como el ALGOL, PASCAL ha jugado un papel único como el principal lenguaje usado para publicar algoritmos en las revistas y libros. A pesar de sus fuertes mejoras sobre ALGOL, -especialmente en el área de entrada-salida, archivos, registros, gestión dinámica de memoria y estructuras de control- PASCAL también fue cuestionado por sus deficiencias, y por ello se propusieron sucesores importantes como algunos de los que describimos a continuación.

DELPHI
En el año 1995 se crea el nuevo sucesor de Pascal, al que se llamó Delphi, heredero directo de Object Pascal, y que fue además la primera herramienta con un entorno de desarrollo visual construida por Borland, una compañía puntera durante muchos años en el desarrollo de compiladores para diferentes lenguajes de programación.

Está caracterizado por ser un lenguaje orientado a eventos, es decir, que la ejecución del programa no es secuencial, sino que depende de los eventos que suceden durante la ejecución de la aplicación.

Delphi es una **herramienta de Desarrollo Rápido de Aplicaciones (RAD)**.

Los componentes que incorpora facilitan el acceso a bases de datos, comunicación a través de Internet, calidad en impresiones, desarrollo de aplicaciones multimedia, enlaces DDE, componentes OLE y VBX, etc. Borland ha introducido al mercado varias versiones de Delphi, aportando mejoras notables, entre las que cabe destacar el CodeInsight, un asistente que muestra automáticamente las listas de parámetros de procedimientos, métodos y eventos. En el año 2001 Borland lanzo al mercado la versión de Delphi 6.0 que funciona bajo Windows y es compatible con todas las versiones anteriores.

Junto con esta versión se introdujo en el mercado la primera versión Kylix, una versión de Delphi que funciona bajo Linux.
La última versión disponible en el mercado es Delphi 7.0. Entre las nuevas características se incluye un nuevo compilador que permite construir aplicaciones basadas en la plataforma .NET

VISUAL BASIC

La programación visual evoluciona al surgir la pregunta de porqué se persiste en comunicarse con las computadoras usando lenguajes de programación textual, si podríamos ser más productivos y el campo de las computadoras estaría mas accesible a una gran cantidad de personas si simplemente dibujáramos las imágenes que nos vienen a la mente cuando consideramos soluciones a algún problema en particular.

La mayoría de las personas piensan con dibujos, los lenguajes de programación han sido probados de ser difíciles de aprender efectivamente para muchas personas y algunas aplicaciones como visualización científica y creación simulada han sido buenos para métodos de desarrollo visual. Estas preguntas marcaron la motivación para el estudio de lenguajes de programación visual (VPLs).

El campo de la programación visual ha crecido de una unión en trabajos de gráficas de computadoras, lenguajes de programación, e interación de computadora-humano ("human-computer interaction").

Visual BASIC es un producto hecho por Microsoft que permite al una persona crear fácilmente porgramas para Windows muy poderosos y completos, como por ejemplo programas para bases de datos, controles "Active X", y programas cliente/servidor para redes.

Microsoft, compañía fundada por Bill Gates, lanza su primera versión de Visual BASIC 1.0 en 1991 (llamada en codigo, "Thunder"). Fue la primera herramienta visual de Microsoft, y estaba supuesta a competir con C, C++, Pasacal y cualquier otro lenguaje de programación bien conocido.

No fue exitosa, hasta no salir la versión de Visual BASIC 2.0 en 1993 cuando la gente realmente comenzó a descubrir el poder de este lenguaje, y cuando Microsoft lanza la versión VB 3.0 se convirtió en el lenguaje de programación de mayor crecimiento en el mercado.

Inmediatamente después de esto, Microsoft recibe pedidos de cientos de vendedores de progamas independientes (ISVs) para licenciarlo para uso de sus propias aplicaciones.

En 1997, sale al mercado la versión 5.0 de Visual Basic para Aplicaciones.

Ya para abril de 1999, más de 80 aplicaciones "host" de Visual Basic estaban disponibles para que los desarrolladores las adquirieran, permitien do así la creación e integración de soluciones personalizadas con una variedad de componentes horizontales y verticales.

"Microsoft Visual Basic for Applications" (VBA) es una tecnología de desarrollo muy poderosa para personalizar aplicaciones enlatadas e integrarlas a sistemas y data existente.

PYTHON
Python fue creado a finales de los ochenta por Guido van Rossum en el Centro para las Matemáticas y la Informática (CWI, Centrum Wiskunde & Informatica), en los Países Bajos.
El nombre del lenguaje proviene de la afición de su creador por los humoristas británicos Monty Python.

Características principales:

- Es un lenguaje de programación de propósito general, orientado a objetos, que también puede utilizarse para el desarrollo web.
- Es multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional.
- Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.
- Es administrado por la Python Software Foundation y posee una licencia de código abierto.

RUBY
Ruby es un lenguaje de programación orientado a objetos creado por el programador japonés Yukihiro "Matz" Matsumoto, quien lo publicó en 1995.

Combina una sintaxis inspirada en Python y Perl con características de programación orientada a objetos similares a Smalltalk.

Ruby es un lenguaje de programación interpretado en una sola pasada y su implementación oficial es distribuida bajo una licencia de software libre.

Su creador lo diseño conlos siguientes propósitos:

- Ruby está diseñado para la productividad y la diversión del desarrollador, siguiendo los principios de una buena interfaz de usuario.
- Ruby es un "lenguaje de guiones (scripts) para una programación orientada a objetos rápida y sencilla".

Pero, ¿qué significa esto?

Rápido y sencillo:
- Son innecesarias las declaraciones de variables
- Las variables no tienen tipo
- La sintaxis es simple y consistente
- La gestión de la memoria es automática

Programación orientada a objetos:
- Todo es un objeto
- Clases, herencia, métodos,...
- Métodos singleton→ Un método que pertenece sólo a un objeto se conoce como método singleton.
- Mixins por módulos→Módulos para simular la herencia múltiple.
- Iteradores y cierres

Otras características son:
- Enteros de precisión múltiple
- Modelo de procesamiento de excepciones
- Carga dinámica
- Hilos

Su alcance parece ilimitado y hoy se encuentra presente en aplicaciones que van desde el desarrollo web hasta la simulación de ambientes complejos.

Por último destacar que asociado a Ruby está *Rails*.
Rails es un framework para Ruby de la misma manera que Simphony, Cakephp o Yii lo son lo es para Php, es decir, una utilidad diseñada para facilitar el desarrollo de proyectos en Ruby (especialmente back-end).

OBJECTIVE-C
Objective-C es un lenguaje de programación orientado a objetos creado como un superconjunto de C para que implementase un modelo de objetos parecido al de Smalltalk.

En 1992 fue liberado bajo licencia GPL para el compilador GCC.
Actualmente se usa como lenguaje principal de programación en Mac OS X e iOS.

C#, Visual Basic.Net, J#.Net, Visual C++.Net
Visual Basic .NET, Visual C++ .NET, C# .NET y J# .NET utilizan el mismo entorno de desarrollo integrado (IDE), que les permite compartir herramientas y facilita la creación de soluciones en varios lenguajes.
Este entorno es el llamado Visual Studio.Net.

Asimismo, dichos lenguajes aprovechan las funciones de **.NET Framework** (o plataforma .NET), que ofrece acceso a tecnologías clave para simplificar el desarrollo de aplicaciones Web ASP y servicios Web XML.

.NET es una plataforma sencilla y potente para distribuir el software en forma de servicios que puedan ser suministrados remotamente y que puedan comunicarse y combinarse unos con otros de manera totalmente independiente de la plataforma, lenguaje de programación y modelo de componentes con los que hayan sido desarrollados.

ASP, PHP, JSP
Son los principales lenguajes para desarrollar aplicaciones web.
Se interpretan en el servidor web (o back-end) y producen un resultado (o vista) que puede ser entendido por un navegador o cliente web (parte front-end).

Están basados en Visual Basic, c y java respectivamente.

Ejemplos

Aquí os dejo unos enlaces donde podéis ver diferentes "HolaMundo" en diferentes lenguajes:

- http://rosemary.umw.edu/~finlayson/class/spring14/cpsc401/notes/02-evolution.html
- https://www.youtube.com/watch?v=h6YY46Wjlmk
- https://www.youtube.com/watch?v=94RborlSzsI

Y videos interesantes:

- https://www.youtube.com/watch?v=8lp20JFiB4s
- https://www.youtube.com/watch?v=XnjcmH-lx_o
- https://www.youtube.com/watch?v=QXysWeHNRpU
- https://www.youtube.com/watch?v=Min6zYrtxWc

2.4. Atributos de un buen lenguaje

Con tanto lenguaje, ¿cuál elijo para implementar mi proyecto?

La respuesta es compleja, ya que siempre dependerá del tipo de proyecto, pero voy a intentar darte unas pautas que te ayuden a la hora de elegir un lenguaje (además de la experiencia claro):

- **Legibilidad (Claridad, sencillez y unidad)**: La sintaxis del lenguaje afecta la facilidad con la que un programa se puede escribir, por a prueba, y más tarde entender y modificar.

- **Ortogonalidad**: Capacidad para combinar varias características de un lenguaje en todas las combinaciones posibles, de manera que todas ellas tengan significado.

- **Potencia/velocidad/rendimiento/Fiabilidad**: La velocidad del código generado también es un factor muy a tener en cuenta, así como el rendimiento y fiabilidad del lenguaje en entornos "hostiles".

- **Apoyo para la abstracción**: Una parte importante de la tarea del programador es proyectar las abstracciones adecuadas para la solución del problema y luego implementar esas abstracciones empleando las capacidades más primitivas que provee el lenguaje de programación mismo.

- **Facilidad para verificar (probar) programas**: La sencillez de la estructura semántica y sintáctica ayuda a simplificar la verificación de programas.

- **Entorno de programación**: Facilita el trabajo con un lenguaje técnicamente débil en comparación con un lenguaje más fuerte con poco apoyo externo.

- **Portabilidad de programas**

- **Costo de uso:**
  - Costo de ejecución del programa.
  - Costo de traducción de programas.
  - Costo de creación, prueba y uso de programas.
  - Costo de mantenimiento de los programas: costo total del ciclo de vida.

3. PARADIGMAS DE PROGRAMACIÓN

Comúnmente, las personas empiezan a **aprender a programar escribiendo** programas pequeños y sencillos consistentes en **un solo programa principal**.

> Aquí "programa principal" se refiere a una secuencia de comandos o instrucciones que modifican datos que son a su vez globales para todo el programa.

Estas técnicas de programación ofrecen **tremendas desventajas una vez que el programa se hace suficientemente grande**.

Por ejemplo:

- Si **la misma secuencia de instrucciones se necesita en diferentes situaciones** dentro del programa, la secuencia **debe ser repetida**. Esto ha conducido a la idea de extraer estas secuencias, darles un nombre y ofrecer una técnica para llamarlas y regresar desde estos procedimientos.

- Si no se sigue una estructura, **seguir y modificar el programa** por una persona ajena al desarrollo (si el programa es muy grande incluso x la misma), **se convierte en una "misión imposible".**

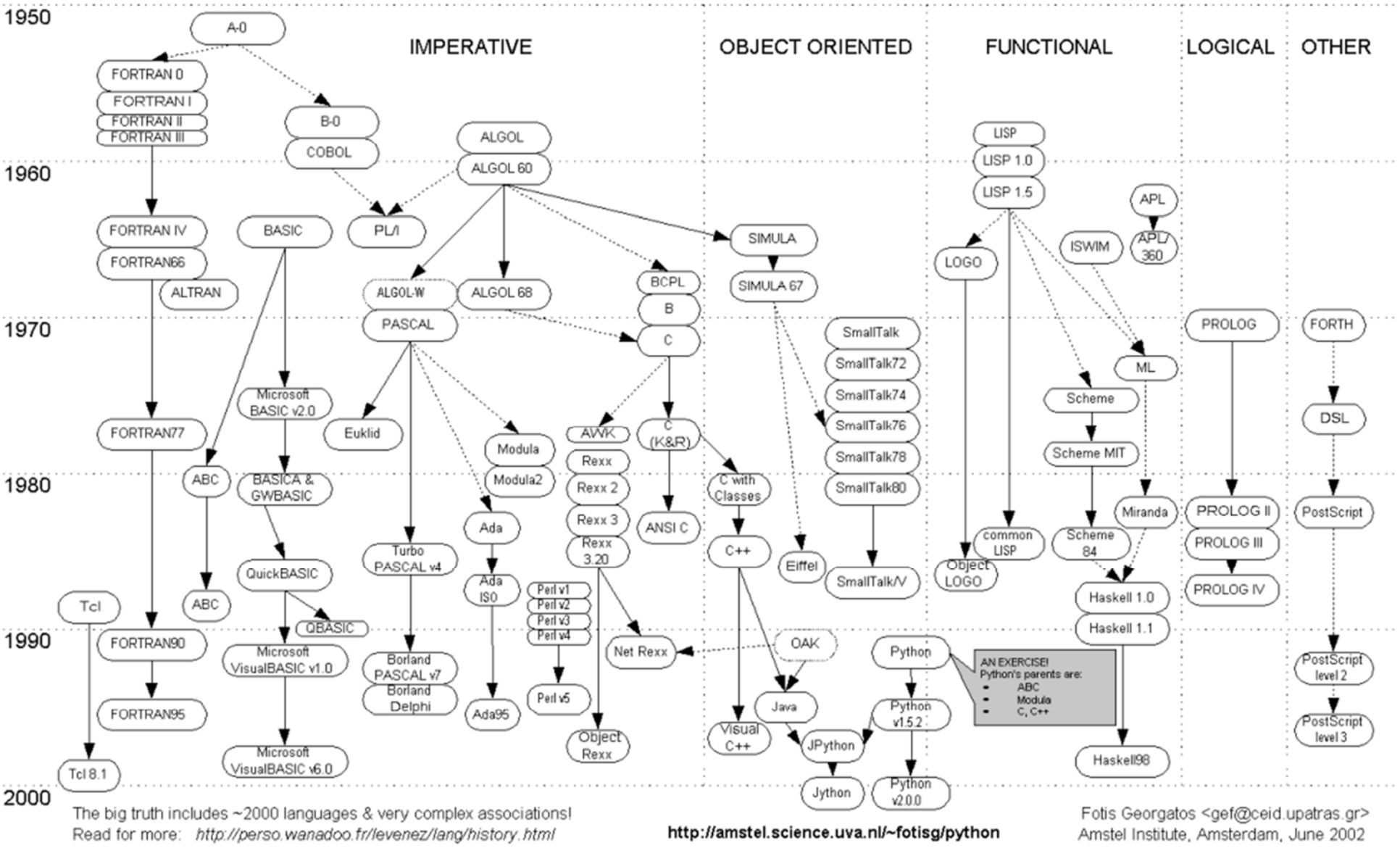
Para resolver estos y otros muchos problemas se "crean" diferentes **paradigmas,** formas o modelos de abordar la solución.

El término "paradigma", etimológicamente significa "modelo", "ejemplo" o "patrón". Sobre este punto de partida, podemos hablar de un paradigma como un conjunto de creencias, prácticas y conocimientos que guían el desarrollo de una disciplina durante un período de tiempo.

Así, los paradigmas de programación establecen que existen diferentes formas y estilos de programar que representan un enfoque particular o filosofía para la construcción del software.

Actualmente existen una gran variedad de paradigmas de programación, pero una clasificación general podría ser la siguiente:

La siguiente imagen nos muestra los principales lenguajes ordenados en base a su paradigma principal:


1950
1960
1970
1980
1990
2000
IMPERATIVE
OBJECT ORIENTED
FUNCTIONAL
LOGICAL
OTHER
A-0
FORTRAN 0
FORTRAN I
FORTRAN II
FORTRAN III
B-0
COBOL
ALGOL
ALGOL 60
FORTRAN IV
FORTRAN66
ALTRAN
BASIC
PL/I
SIMULA
SIMULA 67
LISP
LISP 1.0
LISP 1.5
APL
APL/360
ISWIM
LOGO
ALGOL-W
ALGOL 68
PASCAL
BCPL
B
C
SmallTalk
SmallTalk72
SmallTalk74
SmallTalk76
SmallTalk78
SmallTalk80
SmallTalk/V
PROLOG
FORTH
ML
Scheme
Scheme MIT
DSL
FORTRAN77
Microsoft BASIC v2.0
Euklid
Modula
Modula2
AWK
Rexx
Rexx 2
Rexx 3
Rexx 3.20
C (K&R)
ANSI C
C with Classes
C++
Eiffel
Ada
Ada IS0
Ada95
Perl v1
Perl v2
Perl v3
Perl v4
Perl v5
Net Rexx
Object Rexx
OAK
Java
Visual C++
Python
Python v1.5.2
Python v2.0.0
JPython
Jython
AN EXERCISE!
Python's parents are:
ABC
Modula
C, C++
ABC
BASICA & GWBASIC
QuickBASIC
QBASIC
Microsoft VisualBASIC v1.0
Microsoft VisualBASIC v6.0
Turbo PASCAL v4
Borland PASCAL v7
Borland Delphi
Tcl
Tcl 8.1
FORTRAN90
FORTRAN95
common LISP
Object LOGO
Scheme 84
Miranda
Haskell 1.0
Haskell 1.1
Haskell98
PROLOG II
PROLOG III
PROLOG IV
PostScript
PostScript level 2
PostScript level 3
The big truth includes ~2000 languages & very complex associations!
Read for more: http://perso.wanadoo.fr/levenez/lang/history.html
http://amstel.science.uva.nl/~fotisg/python
Fotis Georgatos <gef@ceid.upatras.gr>
Amstel Institute, Amsterdam, June 2002

El **Paradigma por procedimientos o paradigma imperativo** es el más común y más conocido.
Los programas imperativos son un **conjunto de instrucciones** que le indican a la computadora cómo realizar una tarea. Está representado por C, Basic, Fortran, etc

La mayoría de los actuales lenguajes de programación usan técnicas de programación modular y programación estructurada (ambas imperativas), basándose en el concepto de descomponer el problema en trozos o partes (llamados módulos), y en programar cada módulo mediante métodos estructurados.

La **Programación Declarativa** es un paradigma de programación que está basado en el desarrollo de programas especificando o "declarando" un conjunto de **condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución**.

La solución es obtenida mediante mecanismos internos de control, sin especificar exactamente cómo encontrarla (tan sólo se le indica a la computadora qué es lo que se desea obtener o qué es lo que se está buscando).

Es decir, **describe que** se debe calcular, **sin explicitar el cómo**.
Además:

- No existe un orden de evaluación prefijado.
- Las variables son nombres asociados a definiciones, y una vez instanciadas son inmutables.

Por ejemplo, supongamos que un lenguaje declarativo dice que las funciones F(x) y G(y) son tales que su composición es conmutativa, es decir, que tanto da hacer F(G(z)) que hacer G(F(z)).

En ese caso, tenemos una información que nos sirve para cualquier función y dará igual quién y cómo la programen, nosotros podremos jugar con esa información.

¿Para qué sirve saber eso?, supón que nosotros tenemos que hacer un programa que compute F(G(z)), pero resulta que computar G conlleva un esfuerzo muy importante (ej. 30 segundos); por otro lado, computar F es muy rápido y eficiente y, además, F nos indica en el 90% de los casos, que el cómputo final no fallará y en el 10% que fallará.
Como sabemos que el proceso es conmutativo, podemos computar primero F y sólo en el 10% de los casos computaremos G, ¡con un ahorro en tiempo del 90%!.

Pero lo más importante, es que esto sirve para cualquier par de funciones que cumplan la conmutatividad y por tanto nuestro compilador/intérprete/máquina podrá realizar esta importantísima optimización en cualquier programa.

Diferencia entre imperativo y declarativo

En la programación imperativa se describe paso a paso un conjunto de instrucciones que deben ejecutarse para variar el estado del programa y hallar la solución, es decir, un algoritmo en el que se describen los pasos necesarios para solucionar el problema.

En la programación declarativa las sentencias que se utilizan lo que hacen es describir el problema que se quiere solucionar, pero no las instrucciones necesarias para solucionarlo. Esto último se realizará mediante mecanismos internos de inferencia de información a partir de la descripción realizada.

3.1. Programación Estructurada (PE)

La programación estructurada permite la escritura de programas fáciles de leer y modificar. La PE se refiere a un conjunto de técnicas que han ido evolucionando. Estas técnicas aumentan considerablemente la productividad del programa reduciendo el tiempo requerido para escribir, verificar, depurar y mantener los programas.

La programación estructurada utiliza un número limitado de estructuras de control que minimizan la complejidad de los problemas y que reducen los errores.

Esta incorpora: diseño descendente, recursos abstractos y estructuras básicas.

Diseño descendente.

Consiste en efectuar una relación entre las sucesivas etapas de estructuración. Es decir, se descompone el problema en etapas o estructuras jerárquicas, de modo que se puede considerar cada estructura desde dos puntos de vista: ¿que hace? y ¿cómo lo hace?

Recursos abstractos.

Consiste en descomponer una determinada acción compleja en términos de un número de acciones más simples capaz de resolverla.

Estructuras básicas.

Se ha demostrado que un programa propio puede ser escrito utilizando solamente tres tipos de estructuras de control: —secuenciales, selectivas y repetitivas.

- **Estructura secuencial**. Es aquella en que una acción sigue a otra en secuencia. Las tareas se suceden de tal modo que la salida de una es la entrada de la siguiente.
- **Estructura selectiva.** Se utilizan para tomar decisiones lógicas. En éstas se evalúa una condición y en función del resultado de la misma se realiza una opción u otra. Las condiciones se especifican usando expresiones lógicas. En pseudocódigo estas palabras son if, then, else.
  Las estructuras selectivas pueden ser: – simples, dobles o múltiples.
- **Estructuras repetitivas.** Las estructuras que repiten una secuencia de instrucciones un número determinado de veces se llaman bucles, e iteración al hecho de repetir la ejecución de una secuencia de acciones.

Finalmente comentaremos que **un programa es correcto desde el punto de vista estructurado si:**

- Posee un sólo punto de entrada y uno de salida,
- Existen de 1 a N caminos desde el inicio hasta el fin que se pueden seguir y que pasan por todas las partes del programa y,
- Todas las instrucciones son ejecutables sin que hayan bucles infinitos.

Ejs de lenguajes de programación estructurados=Pascal, C, C++, Ada, Modula-2,...
Ejs de lenguajes de programación no estructurados=Cobol, Fortran, (No tienen estructuras de datos), Basic.

Identación.
El uso de la identación no es obligatorio pero casi;,es importante debido a que, cuando se es consistente en su utilización, facilita la lectura del programa al mostrar en una forma gráfica las relaciones existentes entre las distintas instrucciones.

Ventajas de la programación estructurada.

- **Los programas son más fáciles de entender.** Un programa estructurado puede ser leído en secuencia, de arriba hacia abajo, sin necesidad de estar saltando de un sitio a otro en la lógica, lo cual es típico de otros estilos de programación. La estructura del programa es más clara puesto que las instrucciones están mas ligadas o relacionadas entre sí, por lo que es más fácil comprender lo que hace cada función.

- **Reducción del esfuerzo en las pruebas.** El programa se puede tener listo para producción normal en un tiempo menor del tradicional; por otro lado, el seguimiento de las fallas (debugging) se facilita debido a la lógica más visible, de tal forma que los errores se pueden detectar y corregir más fácilmente.

- Reducción de los costos de mantenimiento.

- Programas más sencillos y más rápidos.

- Aumento de la productividad del programador.

- Se facilita la utilización de otras técnicas para el mejoramiento de la productividad en programación. Los programas quedan mejor documentados internamente.

3.2 Programación Modular

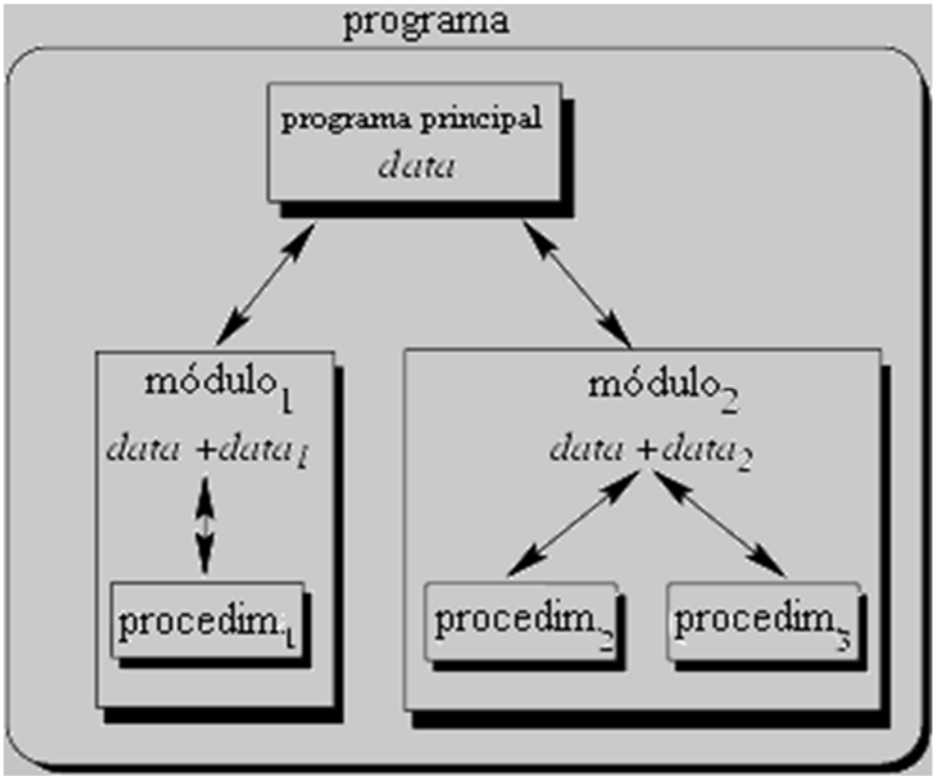
En la programación modular, las secuencias de código con una funcionalidad común son agrupados en módulos separados. Un programa, por consiguiente, ya no consiste solamente de una sección. Ahora está dividido en varias secciones más pequeñas que interactúan a través de llamadas a procedimientos y que integran el programa en su totalidad.

Es uno de los métodos de diseño más flexibles y potentes para mejorar la productividad en la programación. Se fundamenta en una serie de descomposiciones sucesivas del problema inicial, y está inspirado en la técnica "Divide y Vencerás" que usaba el conquistador Alejandro Magno para derrotar a sus enemigos. Su utilización tiene muchos beneficios, entre los que se encuentran la facilidad en la escritura, lectura y comprensión de los programas y el permitir ahorrar espacio que de otro modo estaría ocupado por código duplicado.

En esta técnica se combinan una serie de instrucciones repetibles en un solo lugar, llamado función o procedimiento.
Una llamada de procedimiento se utiliza para invocar al procedimiento. Después de que la secuencia es procesada, el flujo de control regresa al punto de llamada para proseguir con la ejecución de la instrucción ubicada inmediatamente después de la que hizo la llamada.

El código completo del programa opera directamente sobre datos globales y datos locales.
Al introducir parámetros así como procedimientos de procedimientos (subprocedimientos) los programas ahora pueden ser escritos en forma más estructurada y libres de errores.

Por ejemplo, si un procedimiento ya es correcto, cada vez que es usado produce resultados correctos. Por consecuencia, en caso de errores, se puede reducir la búsqueda a aquellos lugares que todavía no han sido revisados.



De este modo, un programa puede ser visto como una secuencia de llamadas a procedimientos.

El programa principal es responsable de pasar los datos a los procedimientos, invocándolos a modo de director de orquesta; a continuación los datos son procesados por los procedimientos y, una vez que terminan, los datos resultantes son presentados.

Así, el flujo de datos puede ser ilustrado como una gráfica jerárquica en forma de árbol.

Con esta metodología el programa se divide en partes independientes, cada una de las cuales ejecuta una única actividad o tarea y se codifican independientemente de los demás.
Cada una de ellas se analiza, codifica y ponen a punto por separado.

Siguiendo un método ascendente o descendente de desarrollo se llegará a la descomposición final del problema en módulos en forma jerárquica.

Si la tarea asignada a cada procedimiento es demasiado compleja, éste deberá descomponerse en otros procedimientos más pequeños. Este proceso de subdivisión sucesiva continúa hasta que cada procedimiento tenga solamente una tarea específica que realizar. Esta tarea puede ser entrada, salida, procesamiento de datos, control de otros procedimientos o una combinación de estos.

Dado que los procedimientos son independientes, diferentes programadores pueden trabajar simultáneamente en diferentes partes del mismo programa. Esto reducirá el tiempo de diseño del algoritmo y posterior codificación. Además un procedimiento se puede modificar radicalmente sin afectar a los demás, incluso sin alterar su función principal.

Cada módulo puede contener sus propios datos. Esto permite que cada módulo maneje un estado interno que es modificado por las llamadas a procedimientos de ese módulo. Sin embargo, solamente hay un estado por módulo y cada módulo existe cuando más una vez en todo el programa.

3.3 Programación Orientada a Objetos

El concepto de programación orientada a objetos no es nuevo, ya que lenguajes clásicos como SmallTalk se basan en ella. De hecho su origen data de los años 60, concretamente con el lenguaje Simula 67. Sin embargo no se populariza hasta mediado de los 90.

Es un técnica de programación cuya principal ventaja es que aumenta la velocidad de desarrollo de los programas gracias a q se pueden incorporar/volver a utilizar "objetos"(de mi cosecha o de la de otros) que tienen comportamientos, características, y relaciones asociadas con el programa.

La orientación a objetos, respecto a la programación tradicional, promete mejoras de amplio alcance en la forma de diseño, desarrollo y mantenimiento del software ofreciendo una solución a largo plazo a los **problemas y preocupaciones** que han existido desde el comienzo en el desarrollo de software: la **falta de portabilidad del código y reusabilidad**, código que es difícil de modificar, ciclos de desarrollo largos y técnicas de codificación no intuitivas.

Un lenguaje orientado a objetos ataca estos problemas.

La OOP se basa en la idea natural de la existencia de un mundo lleno de objetos y que la resolución del problema se realiza en términos de objetos, un lenguaje se dice que está basado en objetos si soporta objetos como una característica fundamental del mismo.

El elemento fundamental de la OOP es, como su nombre lo indica, **el objeto**.
Podemos definir un objeto como un conjunto complejo de datos y programas que poseen estructura y forman parte de una organización.

Un objeto no es un dato simple, sino que contiene en su interior cierto número de componentes bien estructurados.

Estos elementos internos se ven o no según quiera su programador=**ocultación/visibilidad**.

El objeto reúne/recoge sus atributos, funciones y estados en una unidad auto contenida=**encapsulación.**

Otras características importantes son el polimorfismo y la herencia, que veremos con detalle en el capítulo de la 3ª evaluación dedicado x completo a este tema.

Los objetos del programa interactúan mandando mensajes unos a otros.

La POO resuelve problemas importantes de la programación modular.

Entre ellos tenemos los siguientes:

- Ahora los datos no deben crearse y destruirse explícitamente, sino de manera automatica.
- Los datos y las operaciones no están desacoplados, conduciendo a una estructura centrada en los datos en lugar de centrada en los procedimientos (o los algoritmos) .
- Mayor protección y seguridad de los datos y operaciones de los objetos, tanto de accesos externos como internos

3.4. Otros paradigmas

Programación concurrente

La programación concurrente se usa cuando tenemos o queremos realizar varias tareas a la vez.
Sirve para resolver un tipo de problemas determinados, pero no se debe aplicar a cualquier problema.

Del mismo modo que cuando desarrollamos una aplicación tradicional no buscamos el lugar adecuado para colocar un bucle o una sentencia condicional, no tenemos q pensar dónde podemos aplicar el uso de varios hilos concurrentes, más bien lo contrario, este hecho debe surgir como una necesidad.

De hecho, en contra de lo q cree la gente, el rendimiento de una aplicación es inversamente proporcional al nº de hilos en ejecución concurrentes q utilizo (siempre pensando en programa para un solo micro).
Esto es debido al cambio de contexto.

Ej. gráfico→Si tienes q hacer 3 agujeros en la pared, como tardas menos, haciendo uno a uno o haciendo un poquitín de uno y cambiando continuamente de agujero??? Evidentemente, cuando más te pases taladrando y menos cambiando de agujero más rápido acabarás la tarea.

En el mundo de los SOP's se usa mucho para controlar los accesos de usuario y programas a un recurso de forma simultánea.

En el mundo de la programación también se usa para aplicaciones concretas: accesos de L/**E** a la BDD, aplicaciones servidor q deben atender "simultáneamente" a multitud de clientes (ej Chats), aplicaciones q deben tener una IFZ o un PS q debe estar siempre ejecutándose (pase lo q pase), etc...

Tanto *Pascal-FC* (ojo Pascal NO admite concurrencia) como Java o C# admiten programa concurrente.

C too la admite pero sólo se suele usar en para gestionar probs de concurrencia en Unix

Programación lógica

La programación lógica se usa principalmente para inteligencia artificial y para pequeños programas infantiles, con gran aplicación probable en el futuro, ya q se están intentando crear bases de conocimiento para permitir q un sabio esté en todos sitios y nos ceda su sabiduría para siempre.

La programación lógica es un paradigma de los lenguajes de programación en el cual los programas se consideran como una serie de aserciones lógicas. De esta forma, el conocimiento se representa mediante reglas, tratándose de sistemas "declarativos".

Una representación declarativa es aquella en la que el conocimiento está especificado, pero en la que la manera en que dicho conocimiento debe ser usado no viene dado.

Este tipo de lenguajes se basan en el cálculo de predicados, la cual es una teoría matemática que permite entre otras cosas, lograr que un ordenador basándose en un conjunto de hechos y de reglas lógicas, pueda derivar en soluciones inteligentes.

El más popular de los sistemas de programación lógica es el *PROLOG*.

Programación funcional

La programación funcional es un paradigma de programación declarativa basado en la utilización de funciones que no maneja datos mutables o de estado.

Enfatiza la aplicación de funciones, en contraste con el estilo de programación imperativa, que enfatiza los cambios de estado. La programación funcional tiene sus raíces en el cálculo lambda, un sistema formal desarrollado en los años 1930 para investigar la definición de función, la aplicación de las funciones y la recursión.

Inicialmente los lenguajes de programación funcionales, se utilizaron en el ambiente académico y no tanto en el desarrollo de software comercial.

Sin embargo, posteriormente, lenguajes de programación importantes tales como Scheme, Rusto Haskell, han sido utilizados en aplicaciones comerciales e industriales por muchas organizaciones.

La programación funcional también es utilizada en la industria a través de lenguajes de dominio específico como

- R (estadística),
- Mathematica (matemáticas simbólicas),
- J y K (análisis financiero),
- F# en Microsoft.NET y XSLT (XML),
- Etc.

Java en su versión 8, está incorporando la programación funcional.

4. CALIDAD DE LOS PROGRAMAS. DOCUMENTACIÓN.

4.1. Programando con calidad

La documentación de programas que constituyen una aplicación es sumamente importante tanto en el desarrollo de la aplicación como en el futuro mantenimiento de la misma. Asimismo, una correcta documentación permitirá la reutilización de parte de los programas en otras aplicaciones, si éstas se desarrollan con un diseño modular.

La documentación de los programas debe comenzar con la primera fase del ciclo de vida de un proyecto informático y no esperar a tener finalizada la construcción de todos los programas que lo componen.
Con el objeto de pasar de una fase a otra de forma clara y sencilla, la elaboración de dicha documentación deberá ser especialmente cuidada.

La documentación que finalmente se obtiene deberá coincidir con la versión final de los programas que componen el proyecto, que en definitiva será la que se instalará y probará en su entorno habitual. Esto exigirá un perfecto control de la configuración de todas las versiones que se vayan realizando.

Una vez concluido el proyecto los documentos que se deben obtener son una guía técnica, una guía de uso y una guía de instalación.

4.2. Documentación interna y externa.

La documentación que se incluye en las aplicaciones se puede dividir en dos categorías claramente diferenciadas:

- **Documentación interna**
  Es aquella q se incrusta dentro del código de cualquier elemento q forme parte de la aplicación, como por ejemplo la q se incluye dentro de un archivo que contiene el código fuente de un programa en forma de comentarios.

- **Documentación externa**
  Es aquella q no se incrusta dentro del código, es decir, cualquier forma escrita ajena al conjunto de programas q conformen la aplicación. Ej: manual del usuario.