# **Lenguajes de programación**

# **Introducción**

En los años 30, se construyeron las primeras máquinas capaces de realizar ciertas operaciones científicas. Aunque se dice que el ENIAC, 1946, se puede considerar el primer computador realmente de propósito general, pero con programación cableando directamente sus circuitos.

Las primeras máquinas de computación tenían programas fijos. Actualmente algunos equipos muy simples siguen utilizando este diseño. Por ejemplo, una calculadora. Cambiar el programa de una máquina de programa fijo requiere rediseñar la máquina. El rediseñar en caso de ser posible, era un proceso complicado ya que había que cambiar tanto el diseño lógico como el físico resultando en mucho tiempo perdido para preparar un nuevo programa.

En esta época, Von Neumann propuso su [arquitectura](https://es.wikipedia.org/wiki/Arquitectura_de_Von_Neumann), donde el programa se almacena en una memoria principal antes de ejecutarse por la (CPU), evitando así la necesidad de cablear todos los componentes para cada nuevo problema.

Un lenguaje de programación es una notación especial para comunicarse con el ordenador: un conjunto de símbolos con una sintaxis y una semántica que permite definir instrucciones interpretables por una CPU.

Para la implementación de los programas, los lenguajes de programación nos proporcionan:

* Estructuras de datos para el almacenamiento de la información a procesar.
* Instrucciones y operadores para el procesado de dicha información.

## **Clasificación histórica y basada en el nivel de abstracción**

Se agrupan ambas clasificaciones ya que la segunda es concurrente a las primeras etapas de la primera. El nivel de abstracción es la proximidad al lenguaje natural (con el que nos comunicamos los humanos).

### **Bajo nivel**

Los lenguajes con poca o ninguna abstracción con respecto al conjunto de instrucciones de la CPU.

#### **Primera generación: lenguaje máquina**

Se escribía directamente en binario al ser la transcripción lógica de las señales eléctricas. Cada procesador, dispone de un conjunto de instrucciones de las operaciones que entiende (ej.: sumar, guardar en memoria). Los programas escritos en código máquina solo funcionan en ordenadores que comparten el mismo conjunto de instrucciones.

#### **Segunda generación: lenguaje ensamblador**

Establece una serie de reglas mnemotécnicas asociadas a las instrucciones soportadas por la máquina (ADD, SUB, LOAD, STORE, etc.) facilitando la lectura y escritura de programas. La traducción a lenguaje máquina es directa y sencilla. Actualmente se utilizan para programar *drivers* para dispositivos o determinadas partes de los sistemas operativos. Además, permite comentarios y las instrucciones añaden etiquetas para establecer puntos de acceso por control de flujo condicional o repetitivo.

### **Tercera generación: lenguajes de alto nivel**

En los años 50 se empezó a desarrolar lenguajes capaces de expresar las acciones del programador de una manera sencilla. Entonces apareció FORTRAN el considero el primer lenguaje de alto nivel el cual permitía escribir fórmulas matemáticas de manera traducible para un ordenador.

Los lenguajes de alto nivel permiten hacer programas más portables, ya que, en ensamblador, cada programa tiene que rehacerse para el conjunto de instrucciones específico de cada procesador. Además, para programar en ensamblador es necesario un profundo conocimiento de la máquina en la que se va a ejecutar.

Los primeros son:

* Compilados e imperativos: Fortran, Algol y Cobol.
* Lisp (primer lenguaje de alto nivel interpretado y primero funcional).

En esta generación incluiríamos lenguajes de uso común hasta la actualidad como C, C++, Java o Go. Estos lenguajes se denominan de alto nivel por estar bastante alejados del lenguaje máquina y más cerca del lenguaje natural. Pero aun así dentro del alto nivel hay subniveles dependiendo de su abstracción haciendo que por ejemplo se considere un bajo nivel comparado con C# o JAVA.

### **Cuarta generación**

Se consideran lenguajes de cuarta generación a los lenguajes diseñados para solucionar problemas muy concretos. Por ejemplo, SQL que está dirigida a bases de datos.

### **Quinta generación**

Se llamaron lenguajes de quinta generación a algunos utilizados en inteligencia artificial. Se trata de lenguajes que permiten especificar restricciones que se le indican al sistema y resuelve un determinado problema sujeto a estas restricciones (Constraint programming). Algunos ejemplos serían Prolog o Mercury.

## **Clasificación por ámbito**

Un lenguaje específico de dominio o DSL es un lenguaje diseñado específicamente para facilitar y realizar un tipo de tareas. Lo contrario es un lenguaje de propósito general o GPL. Algunos ejemplos de DSL son:

* SPSS o R para análisis de datos
* SQL para consultas de bases de datos relacionales

Un DSL es un lenguaje de programación con un nivel superior de abstracción optimizado para una clase específica de problemas. Es por ello normalmente menos complejo en su uso que un lenguaje de propósito general. En muchos casos, los DSL están destinados no para ser usados por expertos en software, sino por no programadores que son versados en el dominio de aplicación del DSL.

# **Paradigmas de programación**

## **Características**

### **Abstracción**

Cuando hablamos de lenguajes de alto o bajo nivel, estamos hablando de hasta que punto el lenguaje utiliza datos o estructuras cuya correspondencia con en binario (lenguaje máquina) no es directa.

#### **Abstracción de datos**

En bajo nivel podemos trabajar, por ejemplo, con número enteros (basta una conversión de decimal a binario como BCD para almacenar físicamente en una memoria un número) o con caracteres (basta usar una codificación como ASCII o Unicode). Estas conversiones son directas (cada número o caracter se corresponde con una codificación binaria).

#### **Abstracción de control**

Resume las propiedades de la transferencia de control, esto es de la modificación de la estrategia de ejecución de un programa en una situación determinada (p. ej., los bucles, las sentencias condicionales o las llamadas a subprogramas.

### **Modularidad**

La modularidad consiste en la separación de las funcionalidades de un programa en módulos independientes e intercambiables, de modo que cada módulo contiene todo lo necesario para realizar una determinada tarea.

Un módulo esta definido por su interfaz, esto es: los elementos que recibe y los que devuelve.

La modularidad permite en primera instancia reutilizar código: una serie de sentencias que se repiten en varias partes de un código variando solo los datos sobre los que actuan pueden definirse como una subrutina (un procedimiento o función) que tenga esos datos como interfaz (entradas y salidas).

De este modo, se escriben menos líneas de código en total, se mejora la legibilidad del código y con ello su escalabilidad, mantenimiento; y se reduce el tiempo de desarrollo.

### **Ocultación de información**

Es un principio e diseño que protege datos del programa de ser modificados por bloques de código que no deban acceder a ellos.

### **Separation of Concerns**

Principio de diseño que enfatiza que cada subrutina o bloque de código debe encargarse por completo de una determinada funcionalidad o aspecto y solo de esa.

## **Paradigmas más relevantes**

Un paradigma de programación es es un conjunto de conceptos y técnicas para realizar cómputos y la manera en que se deben estructurar y organizar las tareas que debe llevar a cabo un programa. Ninguno de ellos tiene una definición unánime, precisa o estandarizada; y la mayor parte se superponen en distinto grado en cada lenguaje, dando lugar al concepto de lenguajes multiparadigma.

## **Programación imperativa**

Inicialmente los lenguajes de programación se basaron en el modelo de computación de Von Neumann, que propuso que el programa fuese almacenado en la máquina antes de ejecutarse. De ese modo, el programa es una serie de bytes en memoria que contienen instrucciones que serán ejecutadas secuencialmente. De este modo, los programas se fundamentaban en:

* La ejecución secuencial de instrucciones
* El uso de variables para la representación de las posiciones de memorias
* El uso de la asignación para cambiar el valor de las variables.

La programación imperativa se centra en los cambios de estado del programa (instrucciones que modifican variables a lo largo de su ejecución).

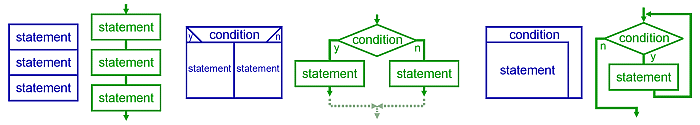
El concepto de "instrucción" se asocia con el lenguaje máquina (o ensamblador), mientras que en lenguajes de alto nivel imperativos, se hablará normalmente de "sentencias (statements)". Una sentencia se compila o interpreta y equivale a varias instrucciones a nivel máquina. En la práctica suelen utilizarse muchas veces indistintamente ambos términos. Java utiliza el término sentencia.

### **Programación estructurada**

Paradigma que busca mejorar la legibilidad (y con ello el tiempo de desarrollo) mediante la utilización exclusiva de

* Secuencia: ejecución de sentencias secuencialmente (incluyendo llamadas a subprogramas)
* Control de flujo (if/else)
* Iteración: bucles (while, for)

Se fundamenta en el [teorema de la programación estructurada (1966)](https://es.wikipedia.org/wiki/Teorema_del_programa_estructurado) y el artículo ["Go To Statement considered Harmful" (1968)](https://es.wikipedia.org/wiki/GOTO#Controversia_sobre_el_uso_de_GOTO).



### **Programación procedimental / Programación modular**

La programación procedimental (a menudo usado como sinónimo de imperativa) enfatiza el uso de llamadas a procedimientos (subrutinas) y es la primera forma de [modularidad](https://github.com/avidaldo/ed22/blob/master/paradigmas-programacion.md#modularidad). Los procedimientos son una serie de sentencias agrupadas en un bloque que es el ámbito (scope) en el que una variable es visible. Hoy en día los procedimientos en general han evolucionado a funciones (o métodos en OOP).

### **Programación orientada a objetos (OOP)**

Se trata de un paradigma basado en el concepto de "objectos" que pueden contener datos (campos (fields)) y código (métodos) y que interactúan entre ellos. Normalmente los métodos están encapsulados en el objeto junto con sus campos y son los que permiten acceder y modificar sus campos.

El primer lenguaje orientado a objetos es Simula (1967) y Smalltalk (1972) el que más desarrolla la teoría de la OOP.

La mayor parte de los lenguajes más populares (C++, Java, C#, Python, JavaScript, Ruby, Perl, R, PHP, Visual Basic.NET...) son multiparadigma pero implementan OOP (a menudo como parte central).

Un objeto se puede describir como una colección de posiciones de memoria junto con todas las operaciones que pueden cambiar los valores de dichas posiciones. Un ejemplo muy básico de objeto es una variable con operaciones de asignación de valor y de recogida de su valor.

#### **Programación basada en clases**

La programación basada en clases es el modelo más popular y desarrollado de OOP. Los objetos son siempre instancias de clases; la clase es el tipo del objeto, de forma análoga a como una variable tiene un determinado tipo. La clase es una definición, un plano, que define la estructura y comportamiento del objeto. Cada clase representa un tipo con un estado (definido por unos campos) y operaciones (métodos) que pueden invocar desde los objetos (instancias) de la clase.

La Programación basada en prototipos es un tipo de OOP en el cual la reutilización de comportamiento (herencia) se desarrolla mediante la reutilización de objetos que sirven como prototipos.

El uso de herencia entre clases (unas clases pueden heredar campos y métodos de otras) permite extender clases, creando jerarquías de clases, y polimorfismo, que permite manipular un conjunto de objetos de distinto tipo como algo uniforme sujeto a ciertas restricciones.

Destacan como lenguajes basados en clases C++ o especialmente Java.

#### **Programación basada en prototipos**

La Programación basada en prototipos es un tipo de OOP en el cual la reutilización de comportamiento (herencia) se desarrolla mediante la reutilización de objetos que sirven como prototipos.

En lugar de definir clases como tipo e instanciar estas clases en objetos, directamente se trabaja con objetos que pueden ser clonados y extendidos. Casi todos los lenguajes basados en prototipos son interpretados y tienen tipado dinámico.

El primer lenguaje orientado a prototipos fue Self a mediados de los 80. El más destacado es JavaScript (aunque también permite definir clases).

## **Programación declarativa**

En los paradigmas imperativos, el énfasis se pone en el "cómo" (las estructuras de control). Los programas consisten en una serie de ordenes, instrucciones o sentencias. Por contraposición, en los declarativos, el objetivo es describir el problema de forma detallada, declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones. La solución es obtenida mediante mecanismos internos de control (inferencia), sin especificar exactamente cómo encontrarla (tan solo se le indica al sistema qué es lo que se desea obtener o qué es lo que se está buscando).

Entre los elementos que forman parte de este tipo de lenguajes, contaremos con herramientas para describir hechos y reglas, con las que podemos construir una base de conocimientos. Además contaremos con un sistema de inferencia que permite extraer conocimiento a partir de los elementos de representación de un problema.

Como no se describen los pasos a realizar, la programación declarativa permite delegar una mayor optimización al proceso de traducción a código máquina; el traductor podrá elegir el mejor algoritmo y/o paralelizar tareas. Un buen ejemplo sería el planificador de consultas (Query planner) de SQL.

Los lenguajes de propósito específico (DSLs) como SQL o lenguajes de marcas como XML o HTML se podrían considerar declarativos.

### **Programación funcional**

Surge paralelamente a la imperativa, con el desarrollo de LISP en 1958, primer lenguaje funcional de la historia, enfocado al área de la inteligencia artificial. Los lenguajes funcionales están basado en el concepto matemático de función. En los lenguajes funcionales, los programas son expresiones que pueden ser definidas como funciones y pueden recibir parámetros. La evaluación de un programa consiste entonces en evaluar dichas expresiones en base a unos parámetros de entrada. Esto es similar a cómo se comporta una hoja de cálculo. Además de LISP y sus dialectos más modernos como Scheme o Clojure, otros lenguajes destacados que implementan este paradigma serían Haskell, Miranda o CAML.

Durante la última década, algunos elementos de programación funcional han sido incorporados a lenguajes imperativos, como es el caso de las funciones Lambda incorporadas a Java en su versión 8.

En la programación funcional, las funciones son tratadas como entidades de primer nivel (first-class citizen), igual que cualquier objeto en OOP. Estos significa que las funciones pueden ser pasadas como argumentos a otras funciones, devueltas como valores o asignadas a variables.

En la programación funcional, las funciones no deben modificar estados (variables o objetos). Las modificaciones en estado son lo que se llama efectos secundarios (side effects).

### **Programación lógica**

Es un paradigma que aplica el conocimiento proveniente del campo de la lógica matemática al desarrollo de programas. Este paradigma ha sido utilizado sobre todo en el campo de la inteligencia artificial tradicional. El programador define una serie de reglas y hechos, y posteriormente se le pueden plantear al sistema una serie de consultas que resolverá en base a las reglas y hechos proporcionados. En este paradigma destaca el lenguaje Prolog.

## **Programación concurrente**

Es una forma de diseño de programas en la cual estos se construyen como colecciones de procesos o hilos (threads) que se ejecutan concurrentemente y que interactúan entre ellos. La programación concurrente surge de forma natural en determinadas aplicaciones como aquellas que tienen interfaz de usuario (UI). En aplicaciones no concurrentes, cuando la aplicación está realizando una tarea pesada no es posible interactuar con su interfaz de usuario dado que la aplicación está ocupada haciendo otras cosas. Por eso las aplicaciones con interfaz de usuario suelen ser concurrentes: al menos hay un hilo encargado de recoger la interacción del usuario con la misma, mientras que puede haber otros hilos que estén realizando otras acciones dentro del mismo programa. Muchos lenguajes incluyen características de programación concurrente. Así, Java es un lenguaje concurrente orientado a objetos y Haskell un lenguaje funcional que también soporta concurrencia.

## **Programación dirigida por eventos (Event-driven)**

La programación dirigida por eventos es un paradigma de programación en el que el flujo de ejecución de los programas va determinado por los sucesos que ocurran en el sistema, definidos habitualmente por acciones del usuario. Un ejemplo sería los eventos que desencadenan los usuarios mediante la interacción con los componentes de las librerías gráficas Java FX y Java Swing.

## **Programación reactiva**

Tras la publicación del Reactive Manifesto en 2014, se ha ido volviendo muy popular la idea de la programación reactiva. Se trata de una programación orientada a los datos (cambios en datos desencadenan la ejecución de subrutinas) a diferencia de la programación basada en eventos, que es orientada a procesos. Su idea fundamental es que los eventos son datos y los datos son eventos.

# **Traductores o procesadores de lenguajes**

Al aparecer los lenguajes de alto nivel se hace necesario el uso de traductores que los conviertan a un lenguaje entendible por la máquina.

Un traductor es un *software* que convierte un código fuente escrito en alto nivel a otro en lenguaje objeto, por ejemplo, máquina o ensamblador, pero también otro lenguaje de alto nivel.

Ensamblador es un traductor ya que convierte código ensamblador en código maquina.

## **Compiladores**

Tambien aparece el concepto de compilador, es el programa que convierte el código fuente en código máquina ejecutable. El código en alto nivel debe ser compilado con el compilador adecuado para convertirlo al conjunto de instrucciones de la CPU y Sistema operativo correspondiente.

Normalmente un programa se compone de varios ficheros los cuales hay que diferenciar:

* Código objeto: el de cada fichero compilado.
* Código ejecutable: el resultante de compilar y enlazar los ficheros objeto para que el código pueda llamar a subrutinas que están en otros ficheros. Esta tarea la realiza el enlazador o linker.

Además, el compilador realiza tareas de optimización para que el código ejecutable sea más rápido.

Se habla de cross-compilar cuando se produce código fuente para un sistema distinto de aquel en el que se ejecuta el compilador.

### **Fases de compilación**

El proceso de compilación se compone de las siguientes fases:

1. Análisis o frontend
   1. Análisis léxico: separación en palabras clave (class, public, int...)
   2. Análisis sintáctico: comprueba que el orden en que se combinan esas palabras se adecúen a las reglas del lenguaje de programación.
   3. Análisis semántico: interpreta el significado de operadores, operandos, variables y demás símbolos para generar el código intermedio.
2. Síntesis o backend
   1. Generación del código máquina
   2. Optimización
   3. Enlazador y generación de código objeto.

## **Intérpretes**

Son programas que analizan y ejecutan una a una las instrucciones que se encuentran en un código fuente. Por tanto coexisten en la memoria con el programa fuente, de modo que el proceso de traducción se realiza en tiempo de ejecución (*runtime*).

Un compilador en general es más difícil de programar que un interprete, de modo que los programas interpretados son en general más portables pero presentan un peor rendimiento por tener que realizar el proceso de interpretación en tiempo de ejecución.

Los lenguajes interpretados facilitan el desarrollo y depuración ya que el tiempo total de traducir el código y ejecutarlo es menor que compilando, pero si lo importante es el tiempo final de ejecución, un programa compilado es, en general, más rápido que uno interpretado.

Los lenguajes interpretados reducen la carga para el programador y aumentan la que recae en el interprete. En general los lenguajes interpretados sirven para trabajar con mayor nivel de abstracción y suelen tener tipado dinámico.

Algunos ejemplos serían Bash, muchos LISP o las primeras versiones de JavaScript.

En lenguajes interpretados no existe código objeto, ni binario. Solo código fuente.

Hoy en día, existen pocos interpretes puros, ya que se utilizan estrategias mixtas para llegar a las ventajas de ambos sistemas.

### **Lenguajes de scripting**

A menudo se utiliza como sinónimo de interpretado, pero "script" tiene la connotación de ser programas cortos de propósito específico (DSL) que permiten manipular, personalizar y automatizar funcionalidades de un sistema ya existente.

Por ejemplo, con Bash se puede hacer un script que cambie el nombre de todos los ficheros en un directorio siguiendo un patrón.

Otro ejemplo: JavaScript en sus inicios era un lenguaje que permitía hacer pequeñas tareas dentro de una página web (aunque hoy en día es un lenguaje con muchos más usos).

Python, por su parte, es un lenguaje muy utilizado en el mundo de Machine Learning para ejecutar series de tareas, a menudo haciendo uso de librerías precompiladas en C.

## **Transpiladores**

Los transpiladores son traductores que convierten un lenguaje de alto nivel a otro lenguaje también de alto nivel. Por ejemplo, Kotlin permite ser compilado a la [JVM](https://github.com/avidaldo/ed22/blob/master/traductores.md#java-virtual-machine-jvm) (por ejemplo para aplicaciones Android) pero también ser transpilado a JavaScript para realizar páginas web.

## **Lenguajes de bytecode**

En algunos casos, las dos operaciones recién citadas se han separado por completo, de modo que se tiene un compilador que traduce el código a un código intermedio, comúnmente denominado bytecode, y un intérprete de bytecode (en general llamado máquina virtual, dado que es análogo a un ordenador).

Al igual que los lenguajes interpretados, presentan la ventaja de la portabilidad: el mismo código binario puede ser ejecutado en diferentes plataformas y arquitecturas. Sin embargo, como el bytecode es en general menos abstracto, más compacto y más orientado a la máquina que un programa pensado para su modificación por humanos, su rendimiento suele ser mejor que el de los lenguajes interpretados.

### **Java Virtual Machine (JVM)**

En este caso, la *Java Virtual Machine* (JVM) es la encargada de traducir bytecode Java, generado por el compilador de Java a partir de un código fuente, a instrucciones en código máquina para la arquitectura hardware sobre la que esté corriendo la MV. En este caso, la JVM se ejecuta como un proceso dentro de un SO. Su objetivo es proporcionar un entorno de ejecución independiente de la plataforma y del sistema operativo, que oculte los detalles de la arquitectura y permita que un programa se ejecute siempre de la misma forma sobre cualquier arquitectura y SO.

*Write Once, Run Everywhere* (escribe una vez, ejecuta en cualquier lugar) fue un eslogan creado para ejemplificar los beneficios de Java como lenguaje multiplataforma (cross-platform).

## **Compilación Just-In-Time (JIT)**

La compilación dinámica o JIT es una técnica para mejorar el rendimiento de programas interpretados. Es una estrategia intermedia donde el código se compila durante la ejecución del programa en lugar de antes de ella. Esto permite optimizar el programa más durante su ejecución en lugar de simplemente hacer una interpretación directa.

Por contraposición con el concepto de JIT, la compilación tradicional a código máquina es llamada ahead-of-time compilation (AOT). Simplificando, la compilación JIT busca combinar la velocidad de ejecución del código compilado con la flexibilidad de la interpretación.

Hoy en día, la JVM utiliza un compilador JIT para compilar el bytecode en código máquina nativo antes de su ejecución. Del mismo modo, destaca el motor V8 de Google que incorpora un compilador JIT para JavaScript.

# **Java**

Java es un lenguaje de programación y a su vez un ecosistema de frameworks y software desarrollado por James Gosling en los años 90. Principalmente está orientada a objetos, siendo estrictamente basado en clases. Se basa en C++, pero eliminando la herencia múltiple para utilizar las jerarquías de clases para ordenar librerías de un modo jerárquico.

Java es un lenguaje que se compila a **bytecode**, un código intermedio que se interpreta en la JVM para garantizar una gran portabilidad. El mismo bytecode puede ser ejecutado en distintas máquinas sobre su JVM correspondiente.

Los ficheros de código fuente de Java están organizados en clases que se almacenan en ficheros .java. Estos deben ser compilados para crear los ficheros objeto de la JVM. Cuando se compila un fichero fuente .java se generan uno o varios archivos .class, uno por cada clase que tenga el fichero. Estos ficheros contienen el bytecode.

Las principales partes a diferenciar de Java son:

1.Java Virtual Machine (JVM): La máquina virtual java ejecuta el bytecode en un entorno virtual independiente, para no tener dependencia del sistema en el que se ejecuta el programa. Utiliza compilación Just-In-Time para interpretar el bytecode.

2.Java Runtime Environment (JRE):

Para poder ejecutar aplicaciones Java, necesitamos tener instalado en nuestro ordenador el entorno de ejecución. Además de la JVM, incluye las librerías básicas de Java.

3.Development Kit (JDK):

El Kit de desarrollo incluye, además del JRE, las librerías necesarias para desarrollar nuevas aplicaciones Java. Para ejecutar un programa Java solo necesitamos el JRE, mientras que para desarrollar nuevas aplicaciones es necesario un entorno de desarrollo, denominado JDK, que además del JRE (mínimo imprescindible) incluye, entre otros, un compilador para Java (**javac**).