

# MÓDULO PROFESIONAL ENTORNOS DE DESARROLLO

UD 1 – Elementos de desarrollo de  
software

# CONTENIDO

- Conceptos básicos
  - Programa
  - Lenguaje de programación
  - Algoritmo
- Algoritmos
  - Diagramas de flujo
  - Pseudocódigo
- Clasificación de los lenguajes de programación
- Código fuente, código objeto y código ejecutable
- Traductores: compiladores e intérpretes
- Máquinas virtuales
- Características de los lenguajes de programación más difundidos
- Desarrollo de software
  - Ciclo de vida
  - Metodología
  - Roles que interactúan en el desarrollo de software

# PROGRAMA

Es un conjunto ordenado de instrucciones que se dan a la computadora indicándole las operaciones o tareas que se desea realice.

(Introducción a la Informática, Alberto Prieto, McGrawHill)

# PROGRAMADOR

La tarea de un **programador informático** es escoger qué órdenes constituirán un programa de ordenador, en qué orden se deben llevar a cabo y sobre qué datos hay que aplicarlas para que el programa lleve a cabo la tarea que debe resolver.

## **Ejecutar un programa**

Por "ejecutar un programa" se entiende hacer que el ordenador siga todas sus órdenes, desde la primera hasta la última.

# SOFTWARE

Es imposible operar el mundo moderno sin **software**.

Como disciplina, la **ingeniería el software** ha progresado mucho en un corto periodo de tiempo.

**¿Pero, el software qué comprende?**

# SOFTWARE



- **Programas:** Instrucciones que cuando se ejecutan proporcionan las características, función y desempeño buscados
- **Datos:** este componente incluye los datos necesarios para manejar y probar los programas y las estructuras requeridas para mantener y manipular estos datos.
- **Documentos:** este componente describe la operación y uso del programa.

# LENGUAJE DE PROGRAMACIÓN

Para especificar las órdenes que debe seguir un ordenador, es decir construir un programa, lo que se usa es un **lenguaje de programación**.

# LENGUAJE DE PROGRAMACIÓN

Se trata de un lenguaje **artificial** diseñado expresamente para crear **algoritmos** que puedan ser llevados a cabo por el ordenador

## **artificial**

Por *artificial* entendemos lo que no ha evolucionado a partir del uso entre humanos, sino que ha sido creado expresamente, en este caso para ser usado con los ordenadores.



# PROGRAMA

Las siguientes son un conjunto de instrucciones de un programa escrito en el **lenguaje de programación Java**. Este conjunto de instrucciones recibe el nombre de **código fuente**.



```
/*  
    El programa Hola Mundo muestra un saludo en la pantalla  
*/  
public class HolaMundo {  
    public static void main(String[] args){  
        //Muestra "Hola Mundo!"  
        System.out.println("Hola Mundo!");  
    }  
}
```

# PROGRAMA

Normalmente, el **conjunto de instrucciones** de un programa se almacena dentro de un **conjunto de archivos**.

Estos archivos los edita el programador (vosotros) para crear o modificar el programa. Para los programas más sencillos basta con un único archivo, pero para los más complejos puede ser necesario más de uno.

# ALGORITMO

Los **algoritmos** son secuencias precisas de instrucciones para resolver un problema.

No debe confundirse **algoritmo** con **programa**. Ya que un programa es la codificación de un algoritmo en un lenguaje de programación.

Un algoritmo puede implementarse en distintos lenguajes de programación

# ALGORITMO

Veamos lo que sería un algoritmo para freír un huevo

- 1 *Poner aceite en la sartén*
- 2 *Colocar la sartén en el fuego*
- 3 *Romper el huevo haciendo caer el contenido en la sartén*
- 4 *Tirar las cáscaras a la basura*
- 5 *Poner sal en la yema*
- 6 *Si el huevo está sólido ir a 7, si no esperar*
- 7 *Servir el huevo, fregar la sartén*
- 8 *Fin*

# ALGORITMO

Las características fundamentales que debe cumplir todo algoritmo son:

- **Un algoritmo debe ser preciso** e indicar el orden de realización de cada paso.
- **Un algoritmo debe estar bien definido.** Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- **Un algoritmo debe ser finito.** Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

# ALGORITMO

En un algoritmo se plasman las tres partes fundamentales de una solución informática: **entrada, proceso y salida**.

Un algoritmo describe una transformación de los datos de entrada para obtener los datos de salida a través de un procesamiento de la información.

Por ejemplo, en el cálculo de la edad de una persona, conociendo su año de nacimiento, la definición del algoritmo, quedaría de la siguiente manera:

- Entrada: información del año de nacimiento y del año actual.
- Proceso: realizar la diferencia del año actual menos el año de nacimiento.
- Salida: visualización del resultado generado. Es decir, el resultado es la edad.

# ALGORITMO

Los algoritmos pueden ser representados simbólicamente por **diagramas de flujo** o en **pseudocódigo (falso lenguaje)**, dónde la secuencia de instrucciones se representa por medio de frases o proposiciones

1. Inicio
2. Leer A, B
3.  $S = A + B$
4. Escribir S
5. Fin

Pseudocódigo Algoritmo para determinar la suma de dos números cualesquiera.

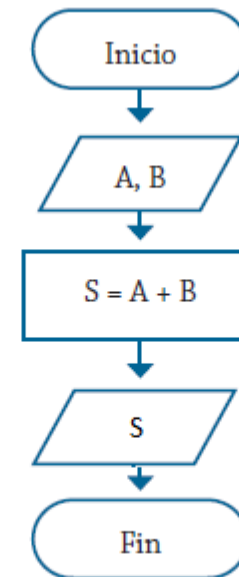


Diagrama de flujo Algoritmo para determinar la suma de dos números.

# ALGORITMOS. DIAGRAMA DE FLUJO

Un **diagrama de flujo** es una representación gráfica de un algoritmo, el cual muestra gráficamente los pasos o procesos a seguir para alcanzar la solución de un problema.


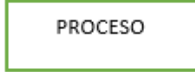




Los pasos son representados por varios tipos de bloques y el flujo de ejecución es indicado por flechas que conectan los bloques.



# ALGORITMOS. DIAGRAMA DE FLUJO

Los símbolos utilizados han sido normalizados por las organizaciones **ANSI** (American National Standard Institute) y por **ISO** (International Standard Organization).

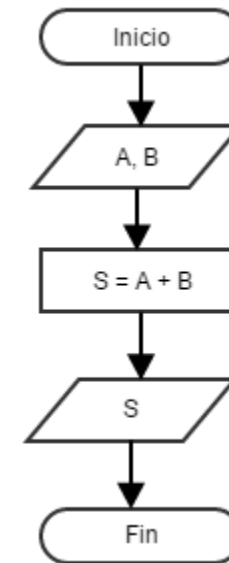
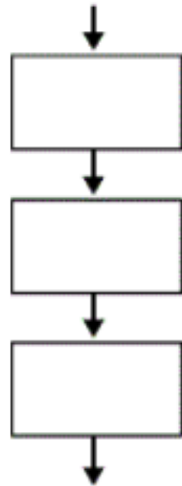
Algunos de los más utilizados son:

	Terminal (representa el comienzo, "inicio", y el final, "fin" de un programa. Puede representar también una parada o interrupción programada que sea necesario realizar en un programa
	Proceso (cualquier tipo de operación que pueda originar cambio de valor, formato o posición de la información almacenada en memoria, operaciones aritméticas, de transferencia, etc.).
	Entrada/Salida (cualquier tipo de introducción de datos en la memoria desde los periféricos, "entrada", o registro de la información procesada en un periférico, "salida").
	Decisión (indica operaciones lógicas o de comparación entre datos y en función del resultado de la misma determina cuál de los distintos caminos alternativos del programa se debe seguir
	Indicador de dirección o línea de flujo (indica el sentido de ejecución de las operaciones).
	Conector, sirve para enlazar dos partes cualesquiera de un diagrama a través de un conector en la terminación del primero y otro al comienzo del segundo. Se refiere a la conexión de partes del diagrama en la misma página

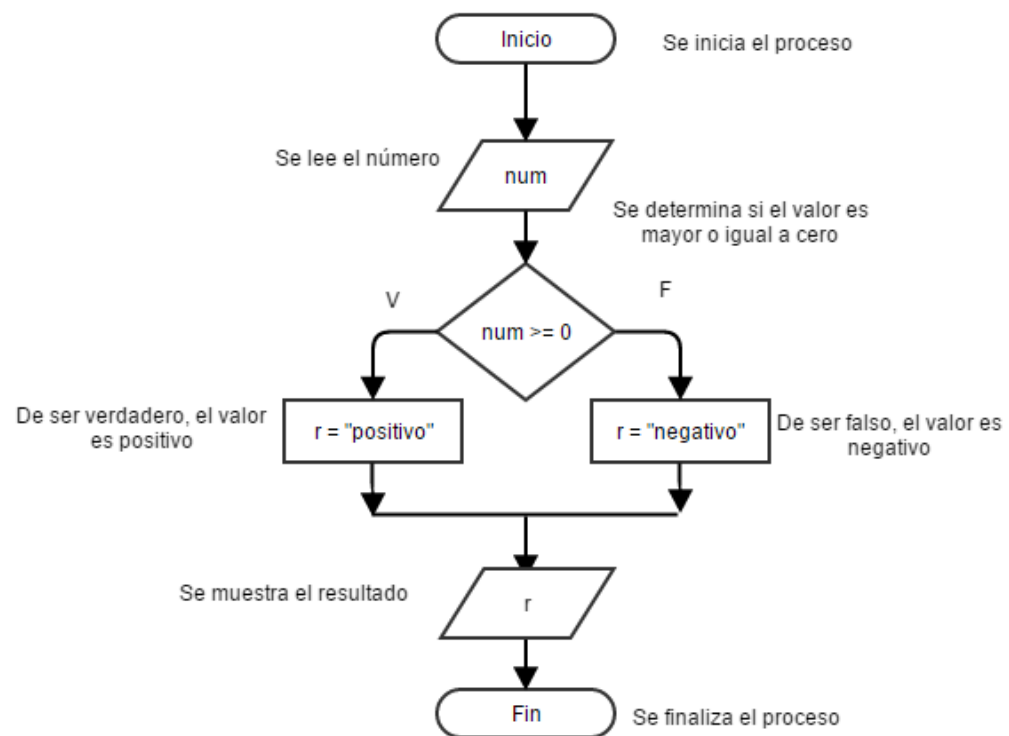
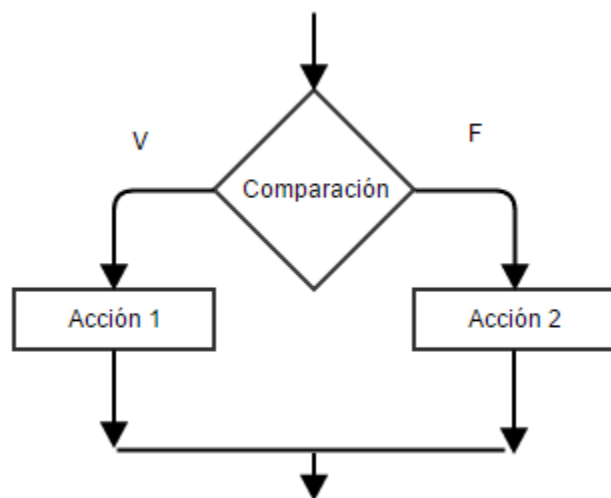
## DIAGRAMA DE FLUJO: ESTRUCTURAS SECUENCIALES

Algoritmo para obtener la suma de dos números cualesquiera.

**Secuencia**

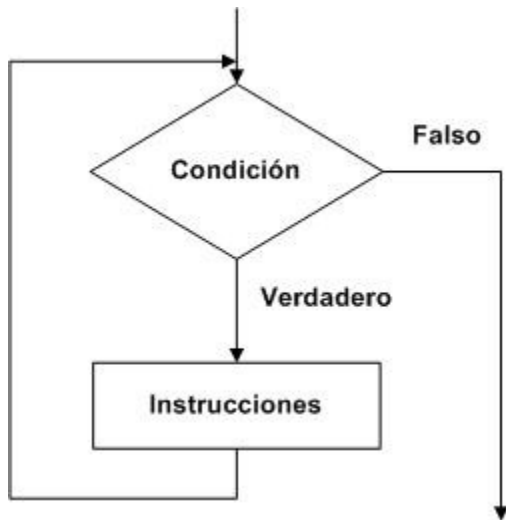


## DIAGRAMA DE FLUJO: ESTRUCTURAS SELECTIVAS

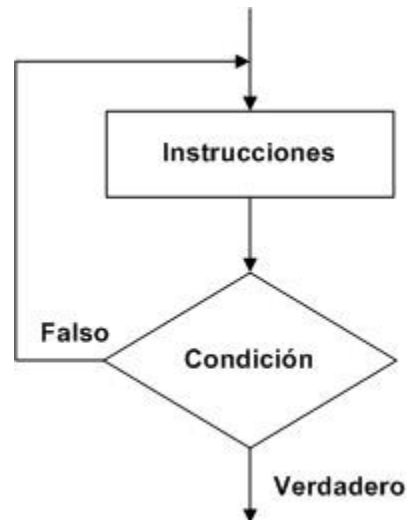


Algoritmo para determinar si un número es positivo o negativo.

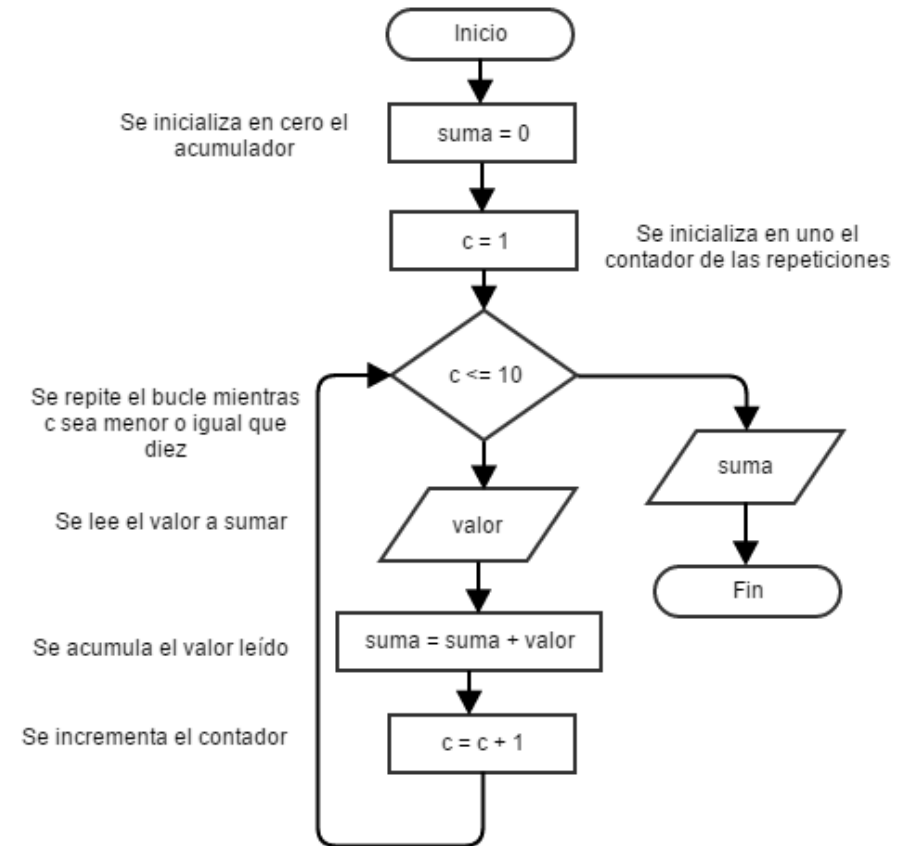
## DIAGRAMA DE FLUJO: ESTRUCTURAS REPETITIVAS



**Mientras:** repite mientras la condición se cumpla



**Hasta:** repite hasta que la condición se cumpla



Algoritmo para obtener la suma de diez cantidades

# DIAGRAMA DE FLUJO

Su correcta construcción es sumamente importante porque a partir del mismo se escribe un programa en algún lenguaje de programación.

Si el diagrama de flujo está completo y correcto el paso del mismo a un lenguaje de programación es relativamente simple y directo

# DIAGRAMAS DE FLUJO

**Construcción:** para la construcción de un diagrama de flujo se tienen que seguir los tres pasos siguientes:

1. **Análisis:** identificar datos de entrada y salida, grupo de pasos, puntos de toma de decisión, etc. y ordenarlos de manera cronológica.
2. **Construcción:** organizarlo gráficamente. Normalmente la organización es
  - De arriba hacia abajo
  - De izquierda a derecha
3. **Prueba:** se basa en hacer pruebas para ver si funciona.



# ALGORITMOS. PSEUDOCÓDIGO

Los programas deben ser escritos en un lenguaje que pueda entender el ordenador, pero no olvidemos que nuestra forma normal de expresar algo es en lenguaje natural. De la aproximación entre ambos surge una herramienta para la descripción de algoritmos: el **pseudocódigo**.

Es por tanto un **lenguaje algorítmico que permite representar las construcciones básicas de los lenguajes de programación, pero a su vez, manteniéndose próximo al lenguaje natural.**

# ALGORITMOS. PSEUDOCÓDIGO

## **pseudocódigo**

El pseudocódigo es un lenguaje informal de alto nivel que usa las convenciones y la estructura de un lenguaje de programación, pero que está orientado a ser entendido por los humanos.



# ALGORITMOS. PSEUDOCÓDIGO

La ventaja del pseudocódigo es que en su uso, en la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje específico.

Además es fácil de aprender y utilizar, es independiente del lenguaje de programación que se vaya a utilizar. Facilita el paso del programa al lenguaje de programación.

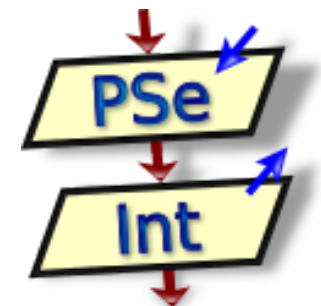
# ALGORITMOS. PSEUDOCÓDIGO

No existe una sintaxis estándar para el pseudocódigo, se utiliza una mezcla de lenguaje natural (utilizando como base la lengua nativa del programador) y una serie de símbolos, términos y otras características propias de los lenguajes de programación de alto nivel.

La escritura de pseudocódigo exige normalmente la indentación (sangría en el margen izquierdo) de diferentes líneas.

Una herramienta libre, gratuita y multiplataforma usada ampliamente en entornos educativos para iniciarse en la programación es PSeInt.

Actividad voluntaria: Escribe con PSeInt el pseudocódigo de los ejercicios de la diapositiva 25. Esta aplicación nos permite además generar el diagrama de flujo automáticamente a partir del código.



# LENGUAJES DE PROGRAMACIÓN

Igual como hay muchas lenguas diferentes, también hay muchos **lenguajes de programación**, cada uno con sus características propias, que los hacen más o menos indicados para resolver un tipo de tareas u otras.

Todos, sin embargo, tienen una **sintaxis** muy definida, a seguir para que la computadora interprete correctamente cada orden que se le da.

```
std::cout << "Hola, mundo!" << std::endl;  
<?php echo 'Hola, mundo!'; ?>  
trace("Hola, mundo!");  
Hola, mundo!  
printf("Hola, mundo!\n");  
System.out.println("Hola, mundo!");  
document.writeln("Hola, mundo!");
```

# LENGUAJES DE PROGRAMACIÓN

Es exactamente lo mismo que ocurre con las lenguas del mundo: para expresar los mismos conceptos, el español y el inglés usan palabras y normas de construcción gramatical totalmente diferentes entre sí.



**¡Hola Mundo!**



# CLASIFICACIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

Los lenguajes de programación se pueden **clasificar** según varios criterios. Nosotros tomaremos con más relevancia los siguientes:

- Según el **nivel de abstracción**.
- Según el **paradigma de programación**.
- Según la **manera de ejecutarse**.

# CLASIFICACIÓN. NIVEL DE ABSTRACCIÓN

Según el nivel de abstracción, es decir, según el grado de cercanía a la máquina:

- Lenguajes **de bajo nivel.**
- Lenguajes **de nivel intermedio.**
- Lenguajes **de alto nivel.**

# CLASIFICACIÓN. NIVEL DE ABSTRACCIÓN

**Lenguajes de bajo nivel o lenguajes máquina:** El lenguaje máquina es el único lenguaje de programación que entiende directamente la máquina, es su "lenguaje natural". En él solamente se pueden utilizar dos símbolos: el cero (0) y el uno (1). Por ello, al lenguaje máquina también se le denomina lenguaje binario. La computadora solo puede trabajar con bits, sin embargo, para el programador no resulta fácil escribir instrucciones tales como:

```
10100010  
11110011  
00100010  
00010010
```

Además, las instrucciones en lenguaje máquina dependen del hardware, en concreto del conjunto de instrucciones usado por el procesador, por lo tanto las instrucciones diferirán de un equipo a otro. A favor tiene que es un código muy eficiente pero en contra que es muy difícil de escribir.

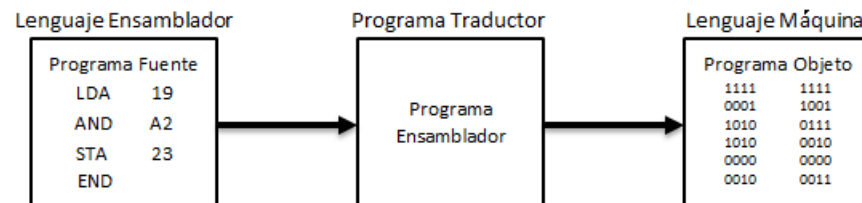
Por esta razón, se inventaron lenguajes de programación más fáciles de aprender y de utilizar.

# CLASIFICACIÓN. NIVEL DE ABSTRACCIÓN

Así, aparecieron los **lenguajes de nivel intermedio**, dentro de los cuales encontramos los llamados **lenguajes ensambladores**, los cuales permiten al programador escribir las instrucciones de un programa usando abreviaturas del inglés, también llamadas palabras nemotécnicas, tales como: **ADD, DIV, SUB**, etc, en vez de utilizar ceros y unos. Por ejemplo, la siguiente instrucción:

**ADD a, b, c**

- Un programa escrito en lenguaje ensamblador no puede ser ejecutado directamente por el ordenador sino que requiere una fase de traducción al lenguaje máquina. Este “traductor” recibe el nombre de “**ensamblador**”. Los lenguajes ensambladores también son dependientes del procesador. Código muy eficiente y más fácil de escribir en comparación con el lenguaje máquina.





# CLASIFICACIÓN. NIVEL DE ABSTRACCIÓN

**Lenguajes de alto nivel:** dada la complejidad de desarrollar estos programas se desarrollaron lenguajes de alto nivel independientes del procesador, diseñados para que las personas escriban y entiendan los programas de un modo más fácil que los lenguajes máquina y ensambladores, y por otro lado traducibles a código máquina.

Un lenguaje de alto nivel permite al programador escribir las instrucciones de un programa utilizando palabras o expresiones sintácticas muy similares a su lenguaje natural (en este caso al inglés). Por ejemplo, en Java se pueden usar palabras reservadas tales como: **if, else, for, while**, etc. para construir con ellas instrucciones como:

```
if (iNumero1>iNumero2)
    System.out.println(iNumero1 + " es mayor que " + iNumero2);
else
    System.out.println(iNumero2 + " es mayor que " + iNumero1);
```

Que traducido al castellano viene a ser: si iNumero1 es mayor que iNumero2 se escribe por pantalla un mensaje para el usuario.

## CLASIFICACIÓN. NIVEL DE ABSTRACCIÓN

Otra característica importante de los lenguajes de alto nivel es que, para la mayoría de las instrucciones de estos lenguajes, se necesitarían varias instrucciones en un lenguaje ensamblador para indicar lo mismo.

De igual forma que, la mayoría de las instrucciones de un lenguaje ensamblador, también agrupa a varias instrucciones de un lenguaje máquina.

Por otra parte, un programa escrito en un lenguaje de alto nivel tampoco se libra del inconveniente que tiene el hecho de no ser comprensible para la computadora y, por tanto, para traducir las instrucciones de un programa escrito en un lenguaje de alto nivel a instrucciones de un lenguaje máquina, también hay que utilizar otro traductor que en este caso recibe el nombre de "**compilador**".

# CLASIFICACIÓN. VENTAJAS E INCONVENIENTES

## **Lenguajes de alto nivel:**

### Ventajas:

- Son lenguajes más cercanos al lenguaje natural.
- El código es más sencillo y comprensible.
- Son lenguajes independientes del hardware además de portables, es decir, se pueden ejecutar en distintos equipos.

### Inconvenientes:

- Son lenguajes más lentos que los de bajo/intermedio nivel ya que existen más capas intermedias entre el programador y la máquina.

## **Lenguajes de bajo/intermedio nivel:**

### Ventajas:

- Su principal ventaja es que son lenguajes mucho más eficientes ya que están más próximos al lenguaje que la máquina entiende.

### Inconvenientes:

- Son mucho más difíciles de comprender para un humano.
- Son dependientes del hardware, es decir, un programa desarrollado para una determinada máquina no funcionará en otra máquina distinta.

# CLASIFICACIÓN. PARADIGMA DE PROGRAMACIÓN

Según el paradigma de programación:

- Lenguajes de programación **estructurados**.
- Lenguajes de programación **orientados a objetos**.

# PARADIGMAS DE PROGRAMACIÓN

Pero antes que nada...

¿Qué entiendes por **paradigma**?

# PARADIGMAS DE PROGRAMACIÓN

Un **paradigma de programación** es un enfoque particular para la construcción de software, un estilo de programación que facilita la tarea de programación.

Algunos lenguajes soportan varios paradigmas a la vez, y otros sólo uno. Se puede decir que históricamente **han ido apareciendo para facilitar la tarea de programar según el tipo de problema a abordar**.

Entonces, esta **clasificación** de los lenguajes de programación es atendiendo al paradigma de programación.

Existe **una gran cantidad de paradigmas de programación**, pero nosotros nos centraremos únicamente en los paradigmas más usados de los lenguajes de alto nivel, que son el paradigma de la programación estructurada y el paradigma de la programación orientada a objetos.

# PARADIGMAS DE PROGRAMACIÓN

Los **lenguajes de programación estructurados** se caracterizan por el empleo exclusivo de tres estructuras para la creación de cualquier programa: la estructura secuencial, la condicional y la repetitiva.

En esta metodología, la atención **se centra en los procesos o tareas que deben realizarse para resolver un determinado problema** y estos procesos o tareas se distribuyen en módulos que los ejecutan.

Estos módulos realizan determinadas tareas, para lo que requiere utilizar determinados datos.

Los módulos pueden ser de dos tipos: **procedimientos** o **funciones**. La diferencia entre estos es que las funciones devuelven algún dato como resultado de su ejecución, mientras que los procedimientos no lo hacen.

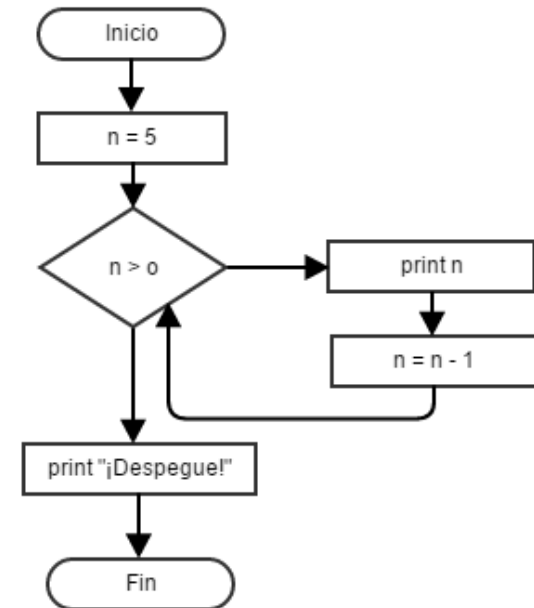
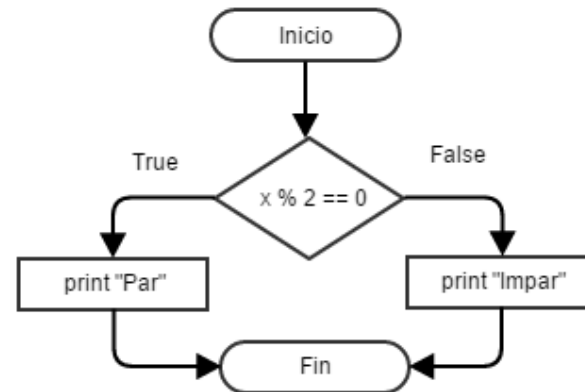
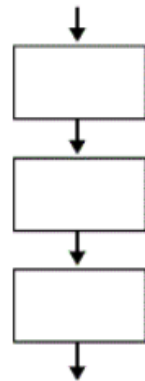
Ejemplos de lenguajes de programación estructurados: C, COBOL, Pascal...

## EJEMPLO PROGRAMACIÓN ESTRUCTURADA

La idea es que cualquier programa estructurado, por complejo y grande que sea, puede ser representado mediante tres tipos de estructuras:

- Secuencial.
- Condicional.
- Repetitiva.

**Secuencia**





## EJEMPLO PROGRAMACIÓN ESTRUCTURADA

Ejemplo:

```
const float SOU_BASE = 1.000;

struct Administrativo
{
    string nombre;
    string DNI;
    float Salario;
}

struct Profesor
{
    string nombre;
    string DNI;
    int numHores;
    float salario;
}

void AssignarSalariAdministratiu (Administrativo administratiu1)
{
    administratiu1. salario = SOU_BASE * 10;
}

void AssignarSalariProfessor (Profesor profesor1)
{
    profesor1. salario = SOU_BASE + (numHores * 12);
}
```

# PARADIGMAS DE PROGRAMACIÓN

Los **lenguajes de programación orientados a objetos**, que surgieron más tarde, fueron un gran cambio de visión (paradigma) en relación con el desarrollo de software. **Según este paradigma una aplicación informática consta de un conjunto de objetos que interactúan entre sí a través de mensajes (llamadas a métodos).**

Un **objeto** es una representación directa de algo del mundo real, como un libro, una persona, un pedido, un empleado... En nuestro código, un objeto será una instancia de una clase, la cual está formada por una serie de datos y un conjunto de métodos que la clase puede realizar.

Por ejemplo, supongamos que queremos modelar una aplicación para una pequeña tienda de productos informáticos. Podríamos crear distintos objetos que representaran a los actores/objetos que intervienen como los “clientes”, “proveedores”, “productos”, “almacén”, etc.

La construcción de programas con este paradigma está basado por tanto en una abstracción del mundo real.

Algunos ejemplos de lenguajes orientados a objetos son: Python, Java, C# y C++.

## EJEMPLO PROGRAMACIÓN ORIENTADA A OBJETOS

Ejemplo:

```
class Trabajador {
    private:
        string nombre;
        string DNI;
    protected:
        static const float SOU_BASE = 1.000;
    public:
        string GetNom () {return this.nom;}
        void SetNom (string n) {this.nom = n;}
        string GetDNI () {return this.DNI;}
        void SetDNI (string dni) {this.DNI = dni;}
        virtual float salario () = 0;
}

class Administrativo: public Trabajador {
    public:
        float Salario () {return SOU_BASE * 10;}
}

class Profesor: public Trabajador {
    private:
        int numHores;
    public:
        float Salario () {return SOU_BASE + (numHores * 15);}
}
```

# CÓDIGO FUENTE, CÓDIGO OBJETO, CÓDIGO EJECUTABLE.

Las siguientes son un conjunto de instrucciones de un programa escrito en el **lenguaje de programación Java**. Este conjunto de instrucciones recibe el nombre de **código fuente**.



```
/*  
    El programa Hola Mundo muestra un saludo en la pantalla  
*/  
public class HolaMundo {  
    public static void main(String[] args){  
        //Muestra "Hola Mundo!"  
        System.out.println("Hola Mundo!");  
    }  
}
```

El **código fuente** está escrito en un lenguaje de programación determinado elegido por el programador, como pueden ser: C, C++, C#, Java, Python o PHP.

# CÓDIGO FUENTE, CÓDIGO OBJETO, CÓDIGO EJECUTABLE.

Se considera **código fuente** aquel que el programador escribe directamente, por lo que en casi todos los casos, consiste en instrucciones escritas siguiendo las normas de un lenguaje de programación de alto nivel, que es el tipo de lenguaje más próximo a nuestro lenguaje natural.

Sin embargo, como el ordenador no comprende este lenguaje, debe ser transformado a **código objeto**.

## CÓDIGO FUENTE, CÓDIGO OBJETO, CÓDIGO EJECUTABLE.

Un **traductor** es un programa que recibe como entrada un texto escrito en un lenguaje de programación concreto y produce, como salida, el lenguaje máquina equivalente. El programa inicial se denomina **código fuente** y el programa obtenido, **código objeto**.

El **código objeto** en ocasiones no es directamente ejecutable por lo que de nuevo tiene que ser “convertido” en **código ejecutable** para ser entendido por la máquina:



## TRADUCTORES: COMPILADORES E INTÉRPRETES

Dependiendo de como se lleve a cabo este proceso de traducción o de transformación del código fuente en código objeto, hay dos tipos de traductores: los **compiladores** y los **intérpretes**.

### Compiladores

Los compiladores son traductores que **en un único proceso** analizan todo el código fuente y generan el código objeto correspondiente almacenando el resultado.

Este almacenado se realizará si el código fuente no tiene errores, en el caso de que no sea así, se mostrarán los mensajes de error que hubiera.

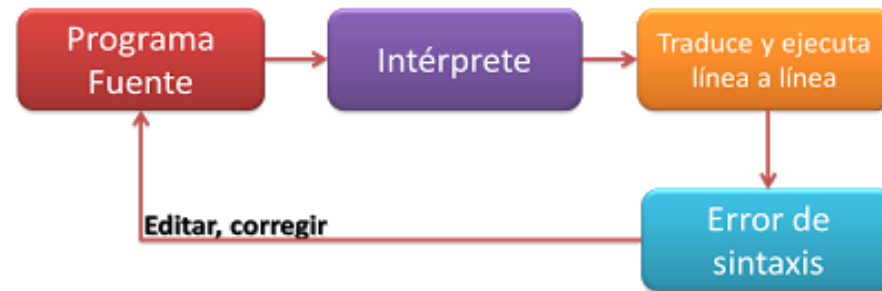
Dependiendo del compilador, el archivo con **código objeto generado puede ser directamente ejecutable o necesitar otros pasos previos a la ejecución**, tales como ensamblado, enlazado y carga.



# TRADUCTORES: COMPILADORES E INTÉRPRETES

## Interpretes

Los intérpretes son traductores que hacen en forma simultánea el proceso de traducción y el de ejecución. Su forma de trabajo es ir analizando línea a línea el programa fuente, generando el código máquina correspondiente, y ejecutándolo. A continuación, repiten este proceso con el siguiente bloque, así hasta que acaba el programa.



A diferencia de los compiladores, los intérpretes no crean un archivo ejecutable que se almacena en memoria secundaria para su posterior uso.



# VENTAJAS E INCONVENIENTES

- **Compiladores**

**Ventajas:**

- La ejecución del código una vez que ha sido compilado es mucho más rápida ya que el resultado se almacena en un fichero ejecutable.

- **Inconvenientes:**

- La compilación en sí misma necesita más tiempo ya que se procesa todo el código en un único bloque.
- Cualquier modificación del código (resolución de errores, desarrollo del software, etc.) requiere volver a compilar todo el código.
- El fichero ejecutable creado solo funcionará en la plataforma donde ha sido creado.

- **Intérpretes**

**Ventajas:**

- Permiten una fácil depuración y una mayor interactividad con el código en tiempo de desarrollo al ser el proceso de traducción más sencillo por ir de línea en línea.
- Multiplataforma. El intérprete suele estar en varios sistemas operativos, así que no tienes que adaptar tu código a una plataforma en concreto. Esto también favorece la portabilidad. El mismo programa puede llevarse a diferentes plataformas.
- No necesita almacenar en un sistema de almacenamiento el resultado en un fichero ejecutable (aunque sí en la memoria principal).

- **Inconvenientes:**

- La principal desventaja es que la velocidad de ejecución del programa es más lenta ya que para cada línea de código es necesario realizar la traducción.

# PROCESO DE OBTENCIÓN DEL CÓDIGO EJECUTABLE A PARTIR DEL CÓDIGO FUENTE

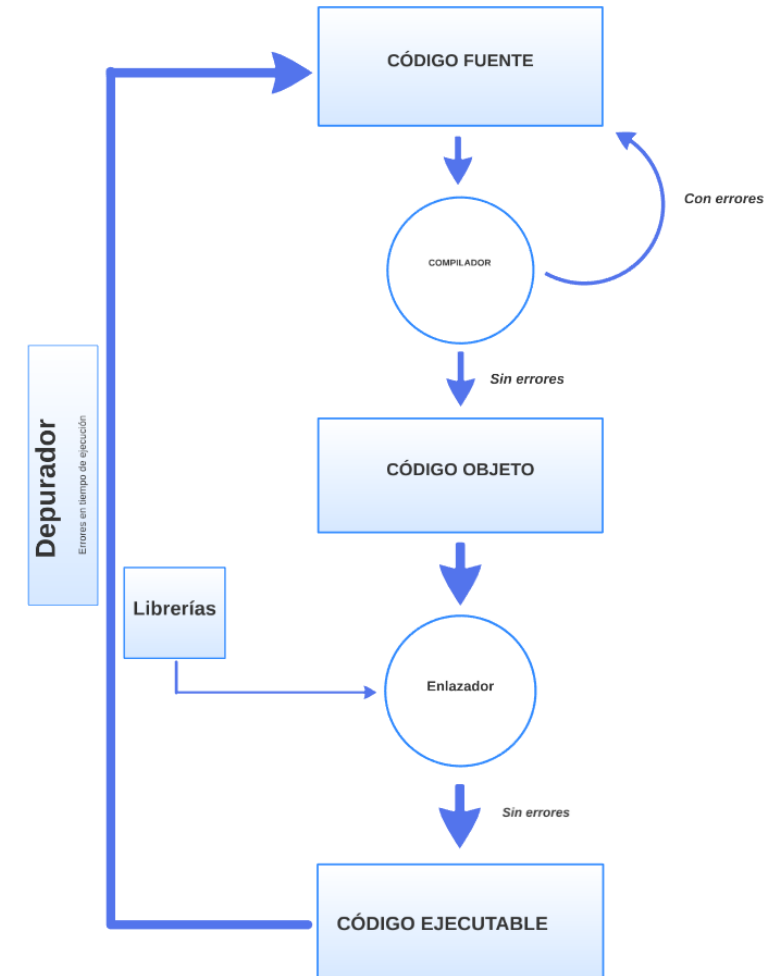
## Compilación y enlace

Tal y como se ha comentado anteriormente, el código objeto en ocasiones no es directamente ejecutable por lo que de nuevo tiene que ser “convertido” en código ejecutable para ser entendido por la máquina.

También hay que tener en cuenta que **una aplicación informática suele estar compuesta por un conjunto de programas o subprogramas.**

Por tanto, **el código objeto de todos ellos deberá ser enlazado (unido)** para obtener el deseado programa ejecutable.

Para ello, se utiliza un programa llamado **enlazador**, el cual generará y guardará, en disco, un archivo ejecutable. En Windows, dicho archivo tendrá extensión (.exe), abreviatura de *executable*.



# PROCESO DE OBTENCIÓN DEL CÓDIGO EJECUTABLE A PARTIR DEL CÓDIGO FUENTE

## Fases:

- **Análisis lexicográfico:** se leen de manera secuencial todos los caracteres de nuestro código fuente, buscando palabras reservadas, operadores, caracteres de puntuación, identificadores y organizándolos todos en cadenas de caracteres denominados **lexemas**.
- **Análisis sintáctico-semántico:** agrupa los lexemas en frases gramaticales. Con el resultado del análisis sintáctico se revisa la coherencia de las frases gramaticales, si su “significado” es correcto, si los tipos de dato son correctos, si los arrays tiene el tamaño y tipo adecuados, y así con todas las reglas gramaticales de nuestro lenguaje.

# PROCESO DE OBTENCIÓN DEL CÓDIGO EJECUTABLE A PARTIR DEL CÓDIGO FUENTE

- **Generación de código intermedio:** una vez finalizado el análisis se genera una representación intermedia con el objetivo de facilitar la traducción a código objeto.
- **Optimización de código:** se revisa el código generado en el paso anterior optimizándolo para que el código resultante sea más fácil y rápido de interpretar por la máquina.
- **Generación de código:** se genera el código objeto de nuestro programa en un código de lenguaje máquina.
- **Enlazador de librerías:** se enlaza nuestro código objeto con las librerías necesarias, produciendo en último termino el código final o código ejecutable.

# PROCESO DE OBTENCIÓN DEL CÓDIGO EJECUTABLE A PARTIR DEL CÓDIGO FUENTE

En el mercado existen aplicaciones informáticas, llamadas **Entornos Integrados de Desarrollo, en inglés Integrated Development Environment (IDE)**, que incluyen a todos los programas necesarios para realizar todas las fases de puesta a punto de un programa.

Además, un IDE suele proporcionar otras herramientas software muy útiles para los programadores, tales como el **depurador de código** con el fin de ayudar y facilitar el trabajo al programador.

# PROCESO DE OBTENCIÓN DEL CÓDIGO EJECUTABLE A PARTIR DEL CÓDIGO FUENTE

Un **depurador de código** permite al programador ejecutar un programa paso a paso, es decir, instrucción a instrucción, parando la ejecución en cada una de ellas, y visualizando en pantalla qué está pasando en la memoria del ordenador en cada momento, esto es, qué valores están tomando las variables del programa.

Cuando decidimos parar la ejecución de un programa en una línea de código concreta, este punto de ruptura se denomina **breakpoint**.

De esta forma, el programador puede comprobar si el hilo de ejecución del programa es el deseado. De no ser así, esto puede ser debido a múltiples causas que tendremos que analizar.

# PROCESO DE OBTENCIÓN DEL CÓDIGO EJECUTABLE A PARTIR DEL CÓDIGO FUENTE

Ejemplo de un **breakpoint** en la línea 13 del siguiente código usando el IDE Netbeans:



```
6 public class ej08 {
7
8     public static void main(String[] args) {
9         int num1, num2, i = 2, posible_divisor, numMenor ;
10
11         Scanner leer = new Scanner(System.in);
12
13         System.out.println("Introduce un número: ");
14         num1 = leer.nextInt();
15
16         System.out.println("Introduce otro número: ");
17         num2 = leer.nextInt();
18
19         if (num1 < num2)
20             numMenor = num1;
21         else
22             numMenor = num2;
23
24         for (posible_divisor = 2; posible_divisor <= numMenor/2; posible_divisor++){
25             if ((num1 % posible_divisor == 0) && (num2 % posible_divisor == 0)){
26                 while ((posible_divisor % i != 0) && (i <= posible_divisor/2))
27                     i++;
28                 if ((i > posible_divisor/2))
29                     System.out.println("El número primo: " + posible_divisor + " divide a: " + num1 + " y a su vez a: " + num2);
30                 i = 2;
31             }
32         }
33     }
34 }
```

# PROCESO DE OBTENCIÓN DEL CÓDIGO EJECUTABLE A PARTIR DEL CÓDIGO FUENTE

Lógicamente, lo primero que hay que hacer siempre cuando estamos desarrollando un programa es comprobar si hemos escrito la sintaxis del lenguaje correctamente.

En estos casos, corregir el error solamente afectará al código fuente del programa.

Sin embargo, si la sintaxis del algoritmo es correcta, entonces el problema estará, precisamente, en el **diseño** de dicho algoritmo, el cual habrá que revisar, modificar y volver a testear ayudándonos del **depurador de código**.



## CLASIFICACIÓN. MANERA DE EJECUTARSE

Una vez entendido los conceptos de código fuente, código objeto y código ejecutable, ahora sí, ya podemos ver la tercera forma de clasificación de los lenguajes de programación, que es según la manera de ejecutarse:

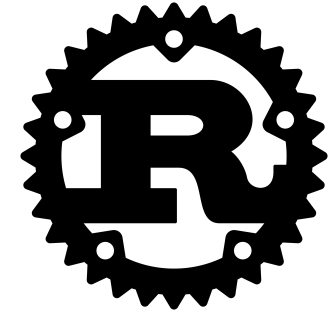
- Lenguajes **compilados**.
- Lenguajes **interpretados**.
- Lenguajes **mixtos**.

# LENGUAJES COMPILADOS

En los lenguajes compilados, una vez escrito el **código fuente** de un programa, este se traduce en su totalidad **en un único proceso**, por medio de un **compilador**, a un archivo ejecutable para una determinada plataforma.

Ejemplos:

- C / C ++ / C#
- Rust
- Go



**Go**

# LENGUAJES INTERPRETADOS

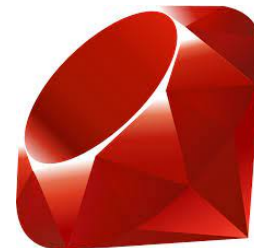
Por el contrario, un lenguaje de programación **interpretado** era aquel en el que las instrucciones se traducen o interpretan **una a una**.

El código es traducido por el **interprete** a un lenguaje entendible para la máquina paso a paso, instrucción por instrucción. El proceso se repite cada vez que se ejecuta el código en cuestión.

No se genera ningún código objeto ni ningún archivo ejecutable. Se trabaja directamente con el archivo de código fuente.

Ejemplos:

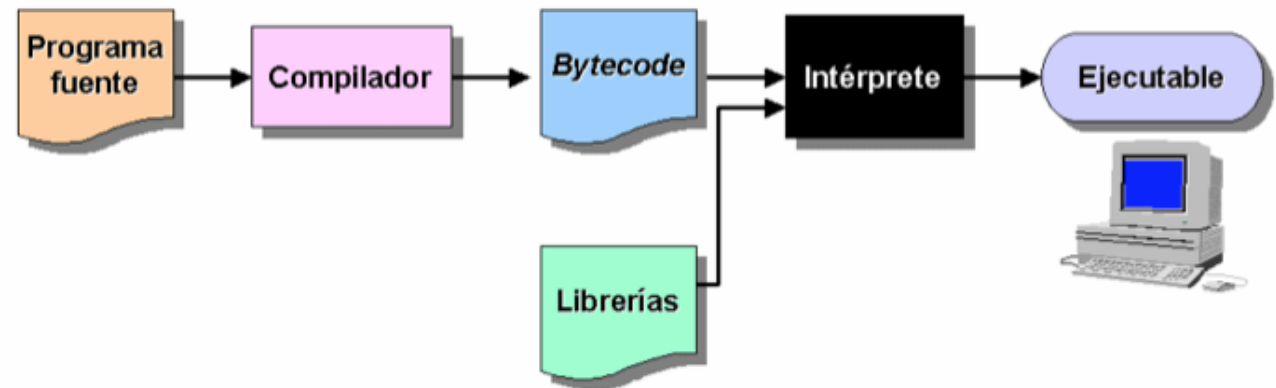
- Python
- PHP
- Ruby
- JavaScript



# APROXIMACIÓN MIXTA

Algunos lenguajes tienen una forma de ejecución **mixta**.

En este caso el código fuente es en primer lugar **compilado**, produciendo una representación intermedia del programa o **bytecode**. Este fichero, que es completamente independiente de la máquina en la que se ejecuta, es posteriormente ejecutado por un **intérprete** que genera el código específico en cada procesador.



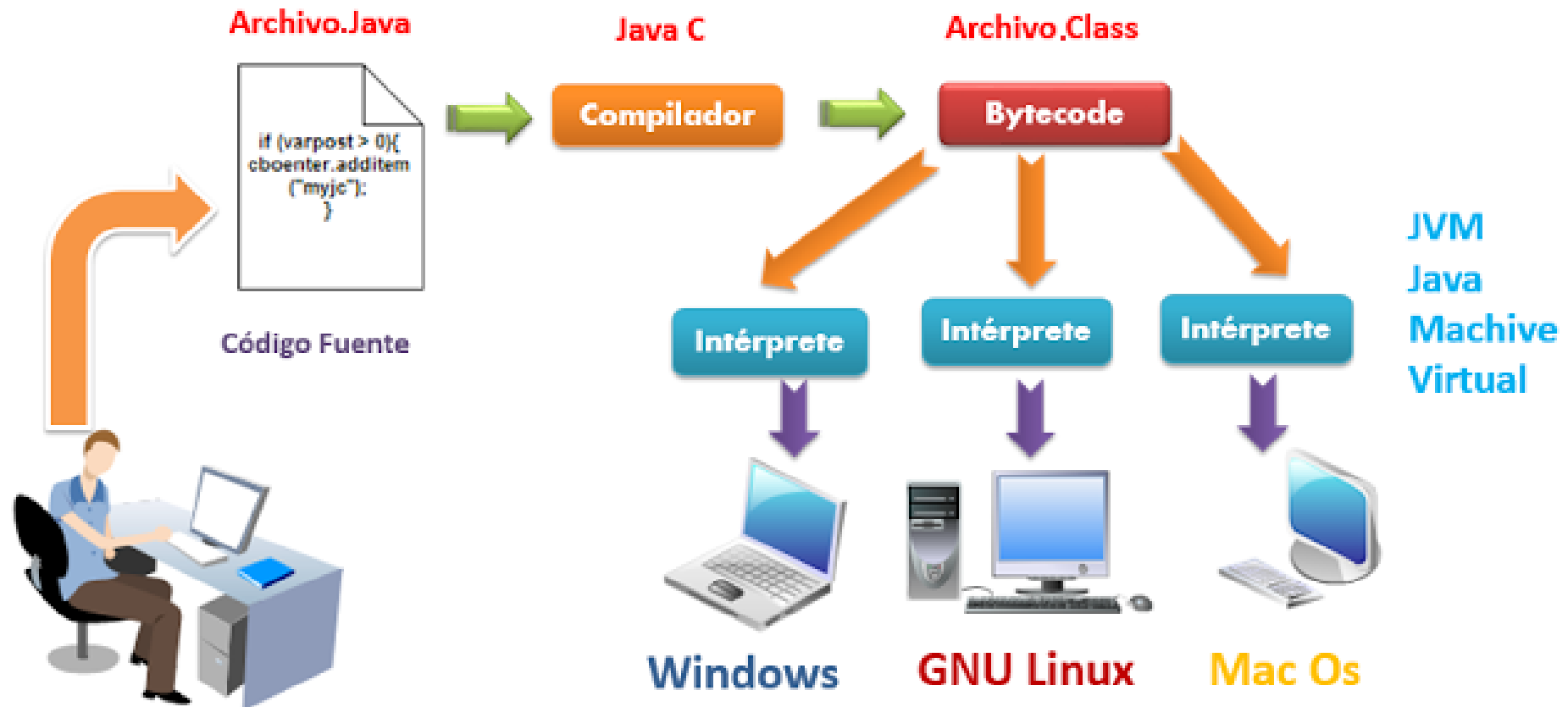
Ejemplos:

- Java
- Kotlin



# MÁQUINAS VIRTUALES

Tanto en el caso de **Java** como de **Kotlin**, el intérprete es la propia Máquina Virtual de Java o JVM



# LA MÁQUINA VIRTUAL JAVA

La **Máquina Virtual Java (JVM)** es el entorno en el que se ejecutan los programas Java.

Es un programa nativo, es decir, ejecutable en una plataforma **específica**, que es capaz de interpretar y ejecutar instrucciones expresadas en un código de bytes (el bytecode de Java), que es generado por el compilador del lenguaje Java.

# LA MÁQUINA VIRTUAL JAVA

La gran ventaja de la JVM es que posibilita la portabilidad de la aplicación a diferentes plataformas y, así, un programa Java escrito en un sistema operativo Windows se puede ejecutar en otros sistemas operativos (Linux, Solaris y Apple OS X) con el único requerimiento de disponer de la JVM para el sistema correspondiente.

# LA MÁQUINA VIRTUAL JAVA

Las máquinas virtuales se pueden clasificar en dos tipos:

1. **Máquinas virtuales de proceso.**
2. **Máquinas virtuales de sistema.**



# MÁQUINAS VIRTUALES

**1. Máquinas virtuales de proceso:** Estas máquinas ejecutan un proceso concreto dentro de un sistema operativo. Este tipo de máquinas se inician cuando se lanza el proceso que se desea ejecutar y se detienen cuando este termina. El objetivo de estas máquinas es permitir que un programa se ejecute de igual forma en cualquier plataforma, proporcionando un entorno de ejecución independiente del hardware y del sistema operativo.

El ejemplo más popular de máquina virtual de proceso es el que acabamos de ver de la máquina virtual de Java. Su ventaja es que dota de portabilidad al lenguaje, de manera que un programa compilado en Java, se puede ejecutar en cualquier plataforma. Esto se debe a que el programa escrito en Java realmente no es ejecutado por el procesador del ordenador, sino por la JVM.

# MÁQUINAS VIRTUALES

**2. Máquinas virtuales de sistema:** Son aplicaciones que emulan a un ordenador por completo, de forma que se puede instalar en su interior otro sistema operativo con su propio disco duro, memoria, tarjeta gráfica, etc. Se puede trabajar en la máquina virtual de igual forma que si se tratase de una máquina real. De esta forma se puede disponer de manera sencilla de varios sistemas operativos en el mismo ordenador.

Ejemplos de aplicaciones de este tipo son VirtualBox y Vmware Workstation.

# CARACTERÍSTICAS DE LOS LENGUAJES DE PROGRAMACIÓN MÁS DIFUNDIDOS

Existen muchos lenguajes de programación diferentes (más de 700), cada uno de estos lenguajes tiene una serie de particularidades que lo hacen diferente del resto.

Los lenguajes de programación más difundidos son aquellos que más se utilizan en cada uno de los diferentes ámbitos de la informática.

En el **ámbito educativo**, por ejemplo, se considera un lenguaje de programación muy difundido aquel que se utiliza en muchas universidades o centros educativos para la docencia de la iniciación a la programación. Por ejemplo, **Python**.

# DESARROLLO DE SOFTWARE:

## CICLO DE VIDA

Toda **aplicación informática**, ya sea utilizada en un ordenador de sobremesa, un portátil, un teléfono móvil, una tableta, etc, ha seguido una serie de **fases para su desarrollo**.

Estas fases irán desde **la concepción de la idea** hasta su puesta exitosa **en funcionamiento y su mantenimiento**. Este conjunto de fases en el desarrollo de software se corresponde con el concepto de **ciclo de vida** del software, que consta de las siguientes etapas:

- Análisis
- Diseño
- Programación/Codificación
- Pruebas
- Explotación/Implantación
- Mantenimiento

# DESARROLLO DE SOFTWARE:

## FASES

**Análisis ¿qué debe hacer el sistema?** En esta fase se definen los requisitos de software que hay que desarrollar. Se trata de una fase fundamental para obtener un software que responda a las necesidades reales de los usuarios y que sea usable.

Dentro de esta fase se realizan reuniones con el cliente, a partir de las cuales surge una memoria llamada **documento de especificación de requisitos**, que contiene la especificación completa de lo que debe hacer el sistema sin entrar en detalles técnicos.

Es importante señalar que esta fase es la base del ciclo de vida, y el resultado se utilizará en las siguientes fases, por lo que una vez finalizado el documento de especificación de requisitos no se deberían poder añadir nuevas características al Software más adelante.

Otra característica muy importante de esta fase de análisis es que es siempre independiente de la tecnología a usar.

# DESARROLLO DE SOFTWARE:

## FASES

**Diseño ¿cómo se va a hacer?** Durante esta fase se traducen los requisitos de la fase de análisis en componentes de software (diagramas lógicos de base de datos, diagramas UML de clases, interfaz de usuario, etc) que definirán la arquitectura general del software.

Durante esta fase se toman decisiones concretas sobre tecnología, que se aplicarán a nuestro análisis de la fase anterior. Por ejemplo, en esta fase, una de las decisiones que habrá que tomar es decidir qué sistema gestor de bases de datos (SGBD) se va a emplear para gestionar la información usada por la aplicación: Oracle, MySQL, etc.

# DESARROLLO DE SOFTWARE:

## FASES

**Programación/Codificación.** Durante esta fase se traduce el análisis y diseño en una forma legible por la máquina, es decir, se implementa todo el código fuente del programa usando lenguajes de programación.

La programación se hará siempre de la forma más modular posible para facilitar el posterior mantenimiento o manipulación del código por parte de otros programadores.

# DESARROLLO DE SOFTWARE:

## FASES

**Pruebas.** Durante esta fase se comprueba que todo funciona correctamente. Se buscan sistemáticamente y se corrigen todos los errores antes de entregar el Software al usuario final.

Por lo general, las pruebas las realiza personal diferente al que realizó la programación. En proyectos de gran tamaño existen equipos de personas especializadas solo en el testing de aplicaciones.



# DESARROLLO DE SOFTWARE:

## FASES

**Pruebas.** Esta fase a su vez se divide en diferentes etapas, las principales son:

- Pruebas unitarias: Se comprueba de manera aislada, el correcto funcionamiento de cada línea de código.
- Pruebas de integración: Se comprueba que los anteriores elementos unitarios también funcionan bien juntos.
- Pruebas de sistema: Son pruebas en donde se reproduce el sistema del propio cliente para probar el programa completo en su conjunto y con otros sistemas con los que se relaciona.
- Pruebas de aceptación o validación: Es la última etapa dentro de las pruebas. Aquí lo habitual es probar el Software dentro de las oficinas del cliente y con sus equipos. En esta fase, es el propio cliente el que termina dando la validación final o no a nuestro Software.

# DESARROLLO DE SOFTWARE:

## FASES

**Explotación/Implantación** En esta fase, se lleva a cabo la instalación y puesta en marcha del software en el entorno de trabajo del cliente, donde los usuarios finales lo utilizarán.

Cabe destacar que en caso de que nuestro software sea una versión sustitutiva de un software anterior es recomendable valorar la convivencia de ambas aplicaciones durante un proceso de adaptación.

# DESARROLLO DE SOFTWARE:

## FASES

**Mantenimiento.** El software sufrirá cambios después de que se entregue al cliente por diversos motivos.

Son muy escasas las ocasiones en las que un software no vaya a necesitar de un mantenimiento continuado. En esta fase de desarrollo de software **se arreglan los fallos o errores o se agregan nuevas funcionalidades** que suceden cuando el programa ha sido implementado.

Llegados a este punto, es importante también distinguir la diferencia entre un producto y un servicio. El producto suele ser una cosa "tangible", que primero se diseña, se replica ininidad de veces, y finalmente el cliente paga por recibir una copia del mismo.

# DESARROLLO DE SOFTWARE:

## FASES

**Mantenimiento.** Mientras que un servicio busca una necesidad insatisfecha que tengan los clientes, y el cliente paga por tener esta necesidad atendida.

Volviendo a nuestro caso del desarrollo de Software, el producto sería el Software que hemos ido elaborando, y que en esta fase de mantenimiento, ya es algo "tangible" que ha sido entregado a nuestro cliente como un paquete completo y funcional. A partir de ese momento, tenemos la oportunidad de ofrecer también el servicio de mantenimiento, cuyas condiciones, duración, precio, etc tendrán que ser pactadas con nuestro cliente.

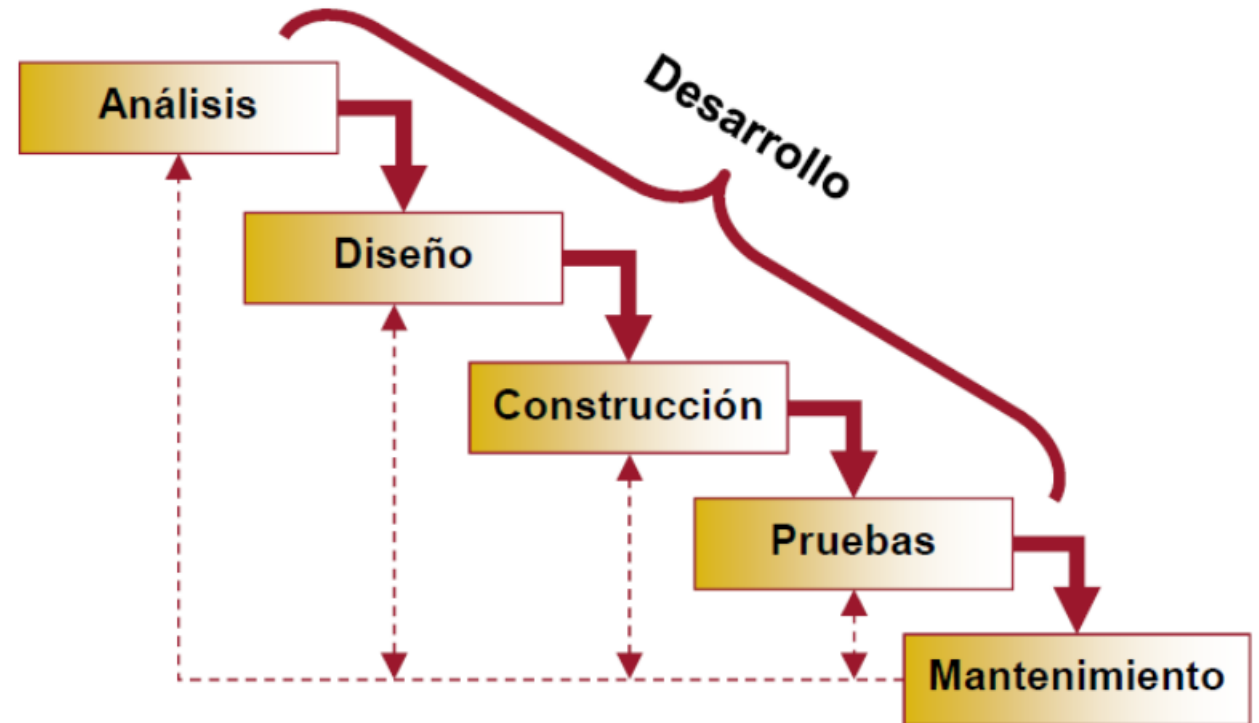
# DESARROLLO DE SOFTWARE: CICLO DE VIDA

Algunos modelos de ciclo de vida

El modelo de ciclo de vida clásico  
o en cascada

La principal característica es que hasta que no se ha finalizado una etapa, no se pasa a la siguiente.

Si bien, en el caso de que se detecte algún error en una etapa anterior, se permite volver hacia atrás.



# DESARROLLO DE SOFTWARE:

## CICLO DE VIDA

Algunos modelos de ciclo de vida

El modelo de ciclo de vida evolutivo

Este modelo sigue un recorrido en espiral donde continuamente se va pasando por todas las etapas. Este modelo sigue por tanto una filosofía completamente distinta al anterior modelo, donde hasta que una etapa no se completaba, no se pasaba a la siguiente.

La ventaja de este modelo es que al cliente se le van haciendo diversos entregables durante el desarrollo del software.



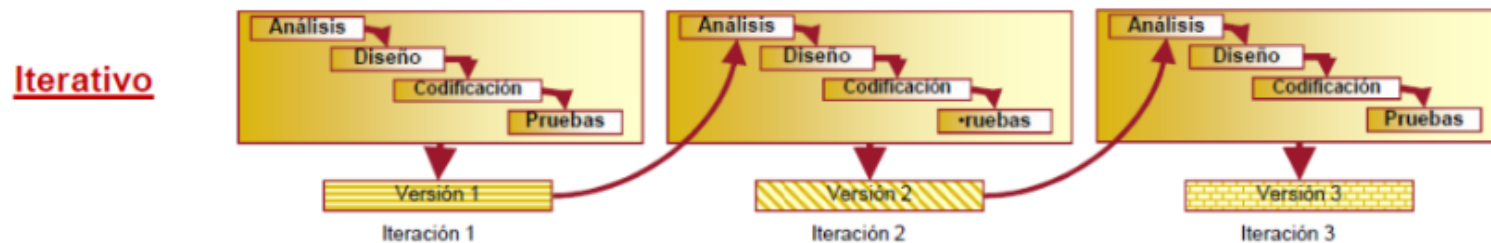
# DESARROLLO DE SOFTWARE:

## CICLO DE VIDA

### El modelo de ciclo de vida evolutivo

El modelo evolutivo a su vez puede ser de dos tipos:

**1º Iterativo:** Se completan todas las fases del ciclo de vida clásico (excepto el mantenimiento) y se entrega una primera versión completa del Software. A continuación se vuelve a repetir el mismo proceso pasando por las etapas de análisis, diseño, codificación y pruebas y se entrega la segunda versión completa. Este proceso se repetirá las veces que sean necesarias hasta terminar el producto.

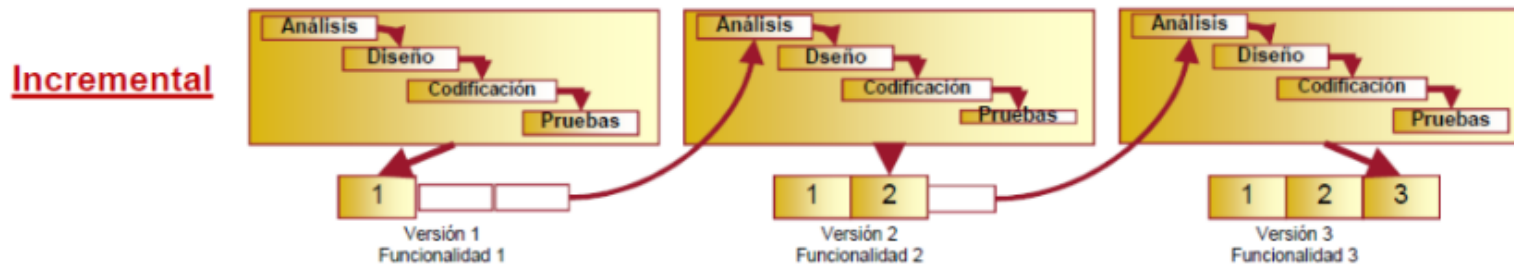


# DESARROLLO DE SOFTWARE:

## CICLO DE VIDA

### El modelo de ciclo de vida evolutivo

**2º Incremental:** En este caso, a diferencia del modelo anterior, no se entrega al cliente en cada iteración una versión completa, sino que se entrega una determinada **funcionalidad**. Supongamos que estamos haciendo el Software de una agencia de viajes, la primera funcionalidad a entregar podría ser el módulo de perfil del usuario con sus datos: nombre, apellidos, foto, etc. Esta funcionalidad se entrega completa, pero el resto de requisitos de la aplicación aún no se han empezado. La siguiente funcionalidad a entregar podría ser el listado de todos los viajes disponibles de la agencia, y la última funcionalidad sería el módulo de añadir un viaje al carrito de la compra y pagarlo. Con este modelo incremental, en cada una de estas entregas al cliente hemos pasado por las etapas de análisis, diseño, codificación y pruebas, y hasta que no hemos entregado todas las funcionalidades, el cliente no tendrá el producto completamente terminado.

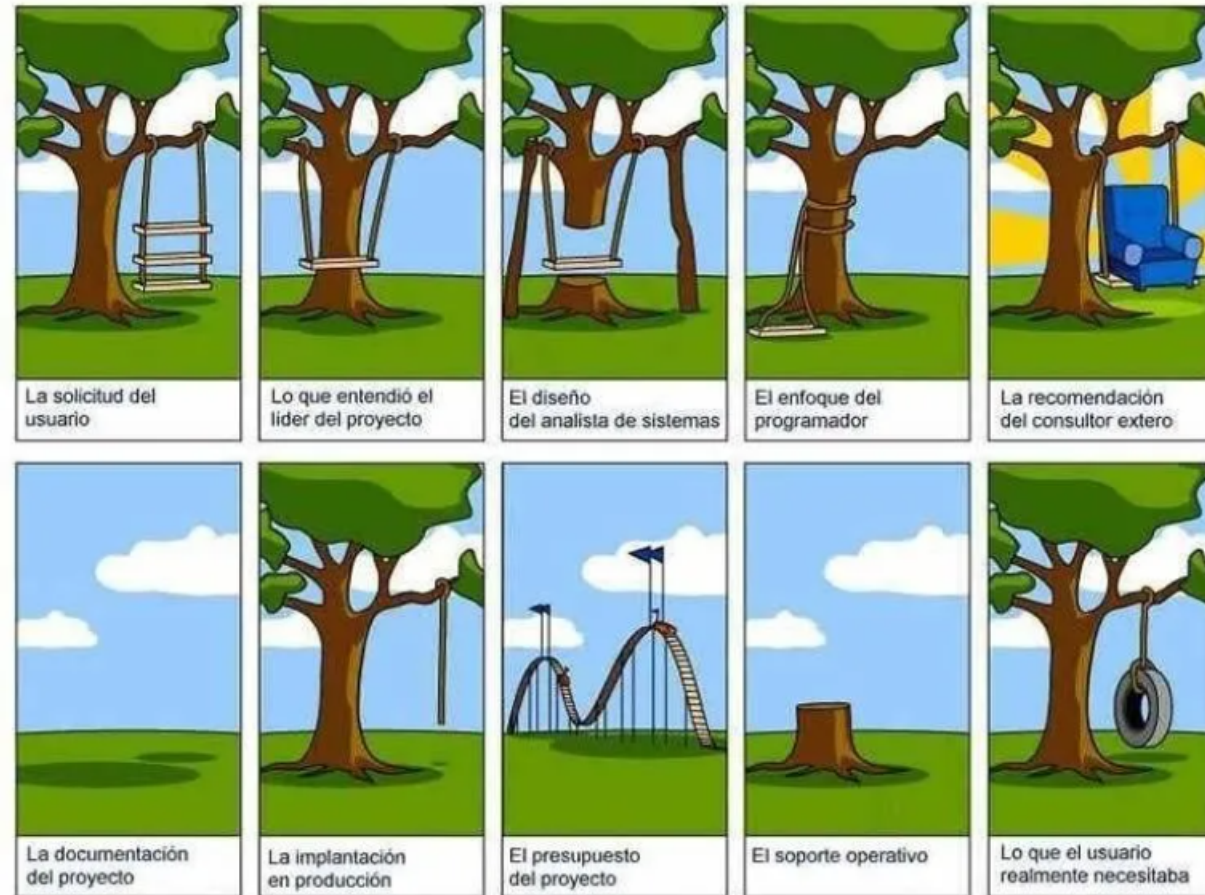




# DESARROLLO DE SOFTWARE:

## CICLO DE VIDA

A pesar de todo lo visto anteriormente, el siguiente cómic nos indica como funcionan realmente los proyectos de desarrollo de Software:



# DESARROLLO DE SOFTWARE: METODOLOGÍA

Un objetivo de décadas en el desarrollo de software ha sido encontrar **procesos y metodologías** que sean sistemáticas, predecibles y repetibles, a fin de mejorar la productividad en el desarrollo y la calidad del software.

Una **metodología** es un conjunto integrado de técnicas y métodos que permite abordar de forma homogénea y abierta cada una de las actividades del ciclo de vida de un proyecto de desarrollo de software.

Existen numerosas propuestas metodológicas adaptándose unas mejor que otras a los distintos paradigmas. Estas propuestas han demostrado ser efectivas durante muchos años y necesarias en un gran número de proyectos, pero también han presentado problemas en muchos otros.

# DESARROLLO DE SOFTWARE: METODOLOGÍA

A partir del 2001, surge un punto de inflexión, con el surgimiento de las llamadas metodologías ágiles para el desarrollo de software.

Las metodologías ágiles cuales dan mayor valor a la colaboración con el cliente y al modelo de ciclo de vida evolutivo de tipo incremental para el desarrollo de software con iteraciones muy cortas.

Este enfoque está mostrando su efectividad en proyectos pequeños/medianos con requisitos muy cambiantes y cuando se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad.

# DESARROLLO DE SOFTWARE: METODOLOGÍA SCRUM

- Metodologías Ágiles: **SCRUM**
- Scrum es un modelo de desarrollo ágil que tiene como objetivo la entrega de valor (productos) al cliente en cortos periodos de tiempo y se basa en tres pilares: transparencia, inspección y adaptación.
- - Transparencia: Todas las personas involucradas en el proyecto conocen en todo momento qué ocurre y cómo, lo que permite que haya una visión global del proyecto.
- - Inspección: Todos los miembros del equipo inspeccionan de manera autoorganizada el progreso del proyecto para detectar posibles problemas.
- - Adaptación: Cuando es necesario realizar algún cambio en el proyecto, el equipo se adapta para conseguir el objetivo.
- La metodología Scrum se compone de los siguientes eventos: [Sprint Planning](#), [Sprint](#), [Daily Scrum](#), [Sprint Review](#) y [Sprint Retrospective](#) que detallaremos a continuación.

# DESARROLLO DE SOFTWARE: METODOLOGÍA

## SCRUM

- En la metodología SCRUM el tiempo dedicado al desarrollo de un proyecto, se divide en una serie de sprints. Un **sprint** abarca el periodo de tiempo que se emplea para realizar el trabajo necesario para entregar valor al cliente, siendo la duración máxima de un sprint de un mes, aunque es aconsejable que sea inferior (unas dos semanas). La duración se determina en función del nivel de comunicación que el cliente desea tener con el equipo de desarrollo.
- Antes de iniciarse un sprint, tiene lugar una reunión llamada **Sprint Planning**, en la que todo el equipo establece qué tareas se van a realizar en el sprint, es decir, se deciden los elementos del product backlog\* que se van a llevar a cabo (sprint backlog). Una vez que se cierra esta reunión, el equipo puede olvidarse durante ese sprint del resto de requisitos del product backlog.
- El equipo de trabajo organiza reuniones diarias o **Daily Scrum** dentro de cada sprint con una duración máxima de 15 minutos. El objetivo es que cada miembro del equipo conozca que están haciendo los demás. En las reuniones se responde a las siguientes preguntas: ¿qué hicimos ayer? ¿qué vamos a hacer hoy? Y ¿tenemos algún problema que solucionar? En cualquier caso, en un daily Scrum nunca se tratará en detalle como resolver un problema, ya que la reunión debe ser breve.

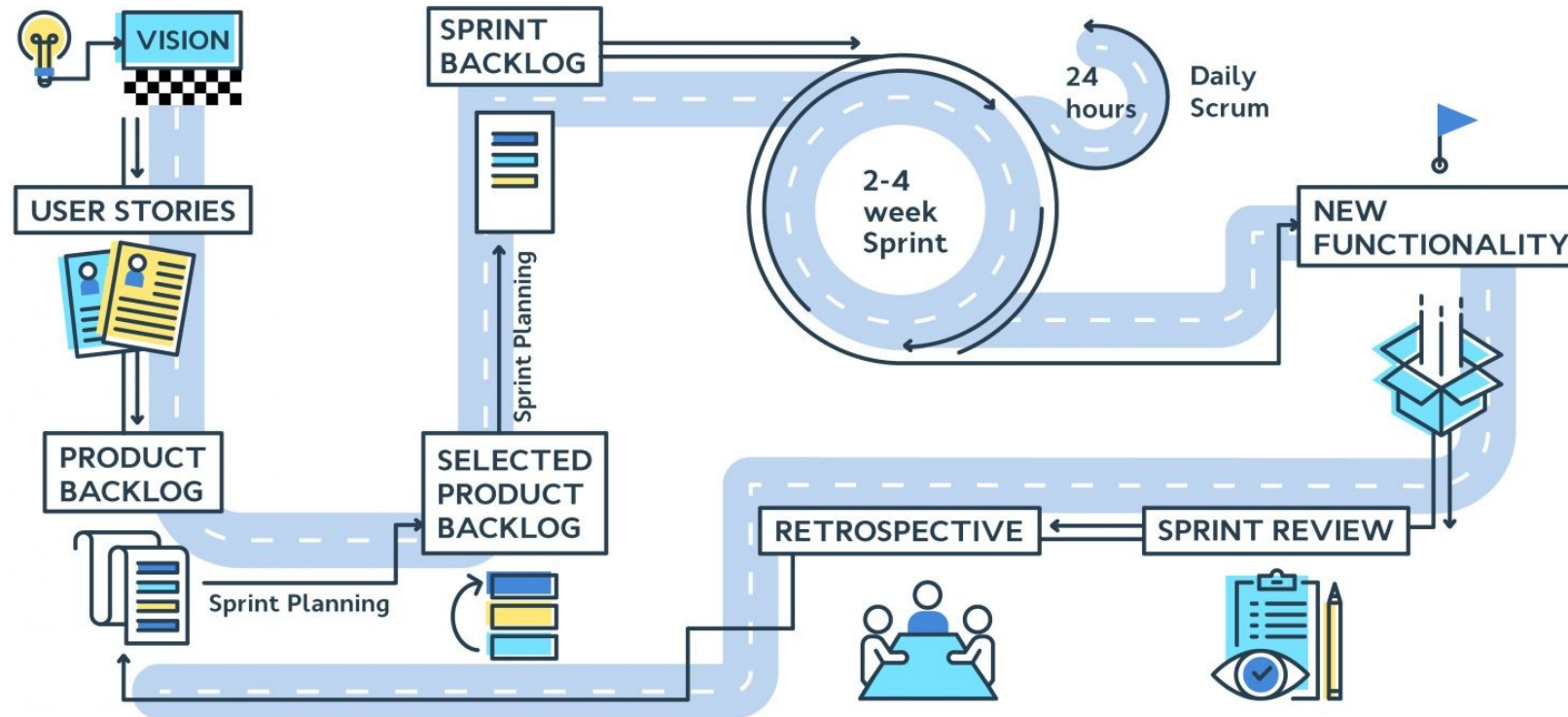
\* Product Backlog: Es la lista ordenada de funcionalidades de la que se compone el Software, priorizadas de mayor a menor importancia. El Product Owner es la persona responsable de mantener esta lista.

# DESARROLLO DE SOFTWARE: METODOLOGÍA SCRUM

- Una vez concluido el sprint, se entrega y se presenta la funcionalidad desarrollada al cliente en un nuevo evento ([Sprint Review](#)). El equipo scrum al completo muestra el funcionamiento del producto al cliente, quien tendrá que validarlo. Además, durante esta reunión, el cliente proporciona retroalimentación útil para futuras tareas. Se analiza la situación y el product backlog es revisado.
- Tras el sprint review, se efectúa una retrospectiva del sprint ([Sprint Retrospective](#)), en la que se hace una evaluación del trabajo realizado durante el sprint, proponiendo mejoras para el siguiente sprint. Finalizada esta reunión, se comienza con un nuevo sprint seleccionando nuevas tareas del product backlog.

# DESARROLLO DE SOFTWARE: METODOLOGÍA SCRUM

## SCRUM PROCESS



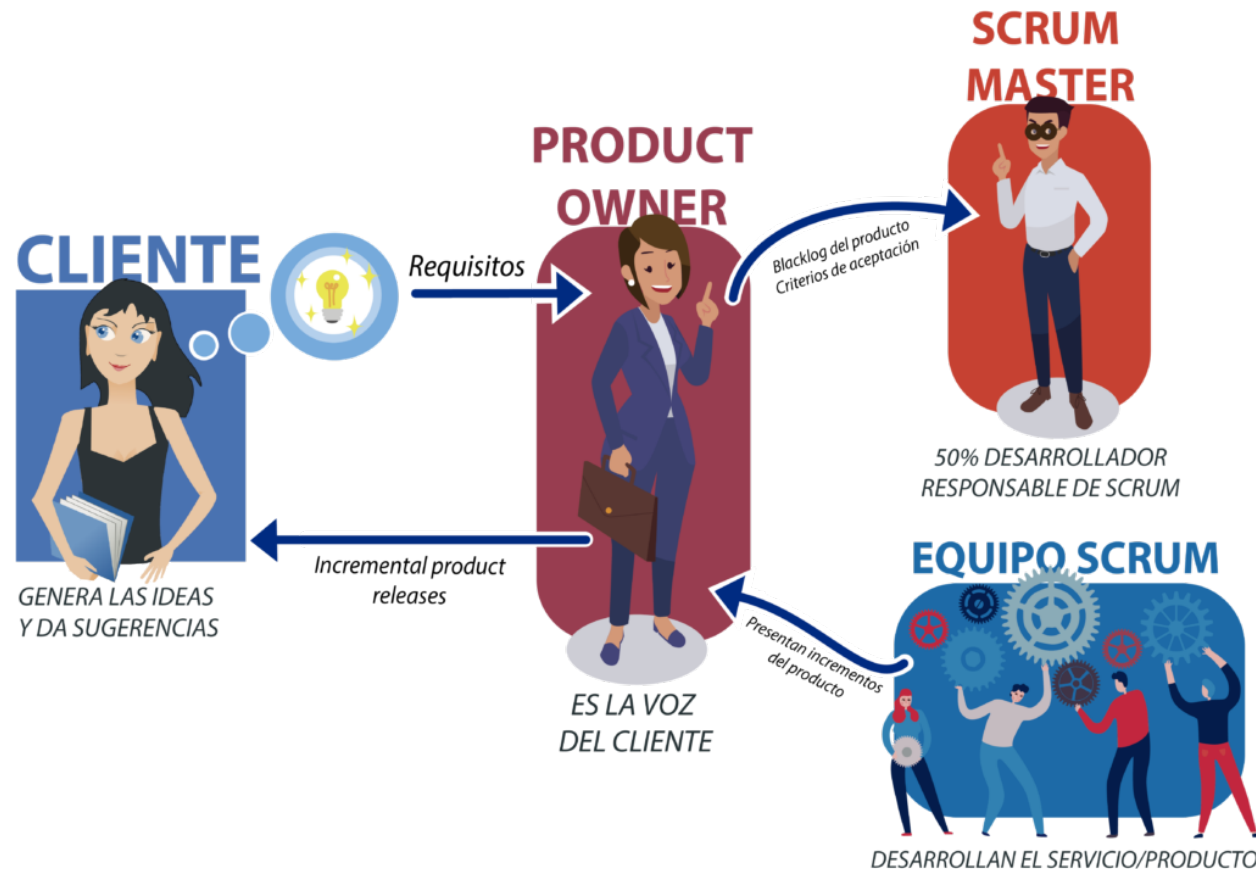
# DESARROLLO DE SOFTWARE: METODOLOGÍA SCRUM

- **Roles Scrum**
- **Product Owner:** (propietario de la visión del producto): Persona individual que se dedica a recoger los requisitos del cliente y lo pasa al equipo de desarrollo. Se encarga también de mantener priorizado el Product Backlog.
- **Scrum Master:** Al igual que el PO, se trata de una única persona. El Scrum master suele ser un miembro del equipo de desarrollo. Su tarea es velar por que el método de scrum se cumpla correctamente: que se ejecuten todos y cada unos de los eventos de scrum. Es una persona que facilita a los demás, ayudando continuamente. Debe eliminar impedimentos. El scrum master es el que hace de conexión con el exterior, debe de ayudar también al PO a que priorice las tareas del Product Backlog.
- **Scrum Development Team:** Se trata del equipo que lleva a cabo las tareas de desarrollo del servicio/producto.



# DESARROLLO DE SOFTWARE: METODOLOGÍA SCRUM

- Roles Scrum



# DESARROLLO DE SOFTWARE: ROLES

A lo largo del proceso de desarrollo de software debemos realizar, como ya hemos visto anteriormente diferentes y diversas tareas. Es por ello que el personal que interviene en el proceso es diverso. Los roles no son necesariamente rígidos y es habitual que participen en varias fases del proceso de desarrollo:

- **Jefe de proyecto:** Dirige todo el proyecto de desarrollo de software. Se encarga de entregar un producto de **calidad**, manteniendo el coste dentro de las limitaciones del **presupuesto** del cliente y entregar el proyecto a **tiempo**.
- **Analista funcional:** Participa en la fase de análisis. Se encarga del análisis de requisitos del software a construir.
- **Diseñador de software:** Participa de la fase de diseño. Diseña la solución a desarrollar a partir de los requisitos.

# DESARROLLO DE SOFTWARE: ROLES

- **Arquitecto:** Investiga y conoce frameworks y tecnologías, revisando que todo el proceso se lleva a cabo de la mejor forma y con los recursos más apropiados.
- **Analista programador:** Comúnmente llamado desarrollador, domina una visión más amplia de la programación por lo que también participa en tareas de análisis y organización del trabajo de los programadores.
- **Programador:** Participa únicamente en la escritura del código fuente del Software.