Tema 1. Lenguajes y técnicas de programación. Introducción a Java

Contenido

Objetivos 2
1. Pasos para la resolución de un problema 2
1.1. Fase de Resolución 3
1.1.1. Análisis del problema 4
1.1.2. Diseño del Algoritmo 4
1.1.3. Verificación "manual" del algoritmo. 6
1.2. Fase de Implementación 7
1.2.1. Codificación 7
1.2.2. Traducción a código máquina 9
1.2.3. Depuración 11
1.2.4. Pruebas 11
1.3. Fase de Explotación y Mantenimiento. 12
2. Técnicas de resolución de problemas 12
2.1. Realiza preguntas 13
2.2. Buscar cosas familiares 13
2.3. Resolución por analogía 13
2.4. Análisis medios-fines 14
2.5. Divide y vencerás 14
2.6. Construcción por bloques 14
2.7. Bloqueo mental: el miedo a empezar. 14
3. Paradigmas de programación 15
3.1. Programación imperativa 15
3.2. Programación declarativa 15
3.3. Programación funcional 15
3.4. Programación lógica 15
3.5. Programación concurrente 16
3.6. Programación orientada a objetos 16
4. Técnicas de programación: Programación estructurada, modular y orientada a objetos. 16
4.1. Programación estructurada 16
4.2. Programación modular 17
4.3. Programación orientada a objeto. 17

Objetivos

- Entender qué es un programa de ordenador
- Comprender qué es un algoritmo
- Aprender qué es un lenguaje de alto nivel
- Entender los procesos de compilación, ejecución e interpretación
- Enumerar algunas técnicas de resolución de problemas
- Conocer algunos paradigmas de programación
- Diferenciar programación estructurada de modular y orientada a objetos
- Comprender el concepto de objeto en el contexto de la resolución de problemas

¿Qué es un ordenador?

ordenador, ra Del lat. *ordinātor, -ōris.* 1. adj. Que ordena. U. t. c. s. 2. m. Jefe de una ordenación de pagos u oficina de cuenta y razón. 3. m. Esp. computadora electrónica.	computador, ra 1. adj. Que computa (‖ calcula). U. t. c. s. 2. m. calculadora (‖ aparato para cálculos matemáticos). 3. m. computadora electrónica. 4. f. calculadora (‖ aparato para cálculos matemáticos). 5. f. computadora electrónica.

computadora electrónica

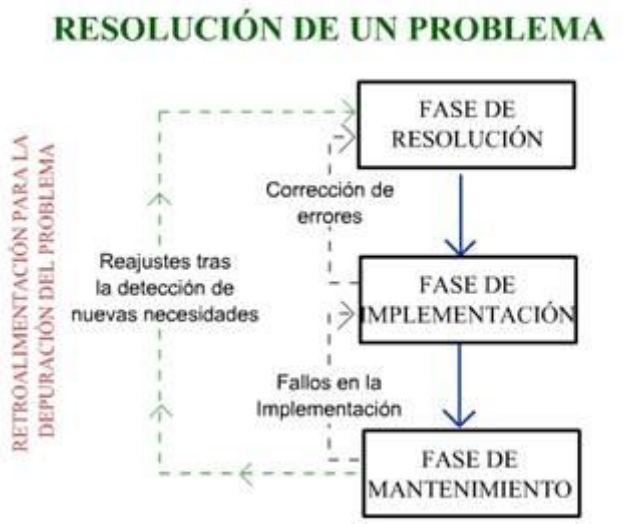
1. f. Máquina electrónica que, mediante determinados programas, permite almacenar y tratar información, y resolver problemas de diversa índole.

Breve definición para algo que, en sólo unas décadas, ha cambiado por completo la forma de vida de las sociedades industrializadas. Los ordenadores afectan a todas las áreas de nuestra vida; de hecho, sería más fácil decir en qué aspectos de nuestra vida no los utilizamos.

1. Pasos para la resolución de un problema

Un ordenador no es inteligente, no puede analizar el problema para llegar a una solución. Debe ser un ser humano, el programador, quien analice el problema, desarrolle los objetos e instrucciones para resolverlo y a continuación, hacer que el ordenador lleve a cabo dichas instrucciones. El motivo por el que mucha gente aprende lenguajes de programación es para poder usar los ordenadores como una herramienta para resolver los problemas.

La solución de un problema comienza con la **definición** del mismo y termina con la verificación de que la solución encontrada es válida para todas las situaciones posibles. La **fase de resolución** abarca hasta ese punto.



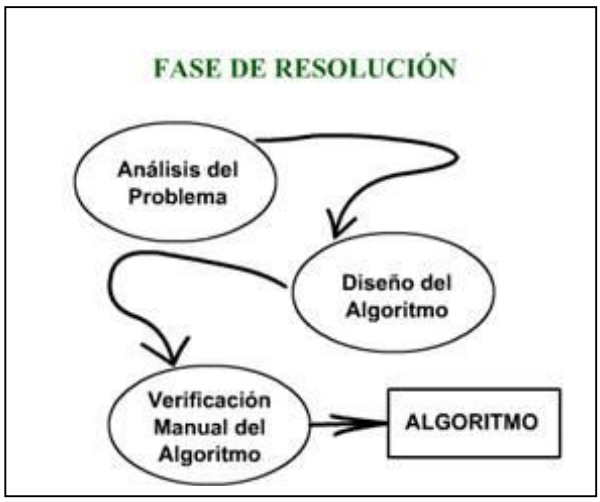
Pero en el caso de una solución informática, hay que ir un poco más lejos, ya que deberemos:

- "traducir" esa solución para que sea comprensible y ejecutable por un ordenador,
- corregir los errores que contenga y
- probar que su funcionamiento es el correcto.

Hasta aquí abarca la ***fase de implementación***.

Pero incluso una vez que nuestra aplicación (o programa de ordenador) está terminada hay que procurar mantenerla actualizada, haciéndole ajustes, mejoras, adaptaciones, etc. siempre que sean necesarias. Esa es la fase de mantenimiento.

1.1. Fase de Resolución



En esta fase **tratamos de acercarnos al problema**, **definirlo correctamente, e idear una forma de resolverlo**. También deberemos verificar que la solución ideada es válida para todos los casos posibles. Todo ello con independencia de que esa solución se vaya a llevar a cabo manualmente, con papel y lápiz, o con un ordenador. Intentamos describir esa solución de forma genérica, sin que todavía entremos en detalles asociados a un lenguaje de programación concreto o a las características de un ordenador concreto. **El resultado de esta fase será un algoritmo, ya verificado**, expresado mediante alguna herramienta descriptiva. Los pasos que incluye esta fase (y que se muestran en la imagen) se detallan en los apartados siguientes.

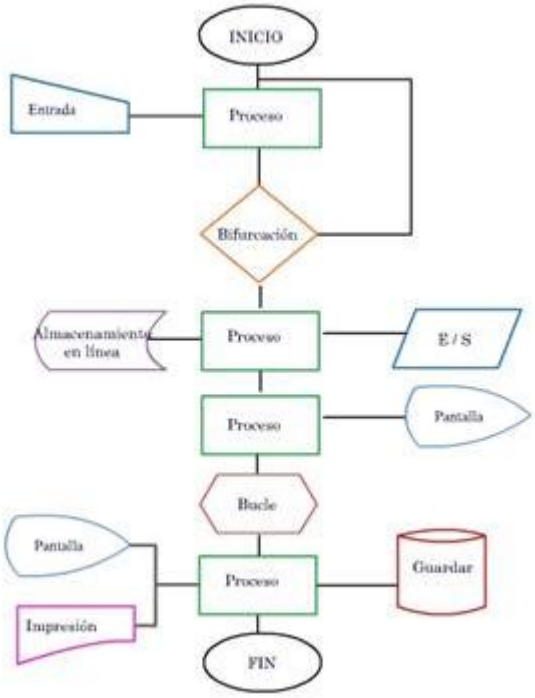
1.1.1. Análisis del problema

El análisis es la primera aproximación al problema. Consiste en **estudiar el problema**, asegurándonos de que lo entendemos bien, de que hemos tenido en cuenta todos los casos y situaciones posibles, etc. Una vez comprendido el problema, será necesario dar una definición lo más exacta posible del mismo, identificando qué tipo de **información** se debe producir y qué **datos** o elementos dados en el problema pueden usarse para obtener la solución.

Como resultado de un buen análisis del problema, obtendremos un **conjunto de especificaciones de datos de entrada, transformaciones y datos de salida**, que nos definen los **requisitos que nuestra solución** del problema debe cumplir.

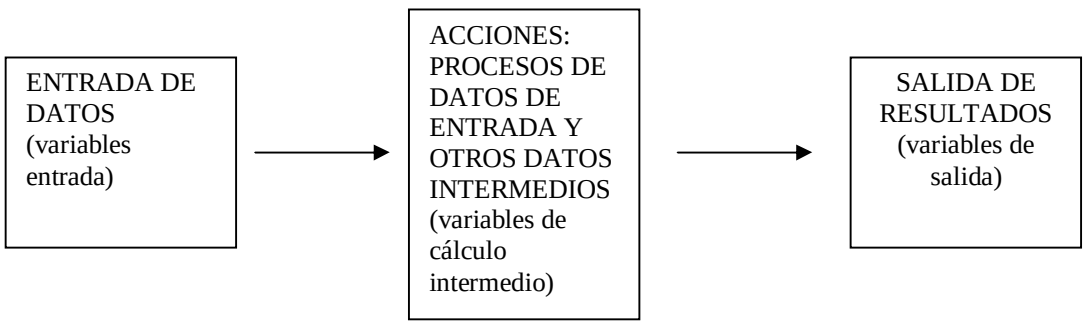
1.1.2. Diseño del Algoritmo.

En este paso se trata de idear y representar de forma más o menos normalizada, **la solución del problema**. Para ello se usan métodos descriptivos o herramientas estándar, que puedan ser entendidos por cualquier **programador**, y que no admitan ambigüedades ni distintas interpretaciones, pero sin entrar en detalles de sintaxis del **lenguaje de programación**.



Si vamos al diccionario un **algoritmo** se define como: " un conjunto finito y ordenado de operaciones que permiten la resolución de un problema o tarea".

La estructura general de un algoritmo es la siguiente:



En el esquema anterior la parte más importante es la central, es decir, cómo procesar y manipular los datos combinándolos correctamente para solucionar el problema deseado.

Antes de entrar en aprender a describir algoritmos vamos a ver cómo, en actividades de la vida diaria, nuestra cabeza está en muchas situaciones planteando algoritmos (programas) para resolver situacion cotidianas.

Un ejemplo muy claro que podemos poner sería el de cruzar una calle. En este caso el dato de entrada es el color del semáforo y la salida es una accion, la de "cruzar la calle".

```
Algoritmo cruzar
Repetir
     Observar color semaforo
Hasta que color del semaforo sea verde
Cruzar la calle
Fin algoritmo cruzar
```

Fijarse que las características de finito y ordenado son importantes porque obviamente, si cruzáramos antes de mirar, los resultados podrían ser catastróficos. Y él termino finito está implícito en la **repetición** de "observar color semáforo", puesto que si está bien reglado habrá un momento en el que cambiara de color y la repetición finalizará pasando a la acción siguiente

Aunque el ejemplo visto es muy sencillo por lo maquinal e inmediato que se desarrolla en nuestra cabeza, la experiencia nos indicará que diseñar un algoritmo a veces no es tan sencillo. Es más, obtendremos incluso dos ó más soluciones para resolver un mismo problema y la pregunta que surge entonces es: ¿qué algoritmo es mejor? o, ¿qué criterio aplicamos para elegir entre varios? Pues bien, la respuesta se ajusta a los siguientes criterios:

- Entre dos algoritmos que resuelven un problema nos quedamos con el que más rápido se ejecuta en la máquina, es decir, una vez traducido a un cierto lenguaje probaríamos el tiempo de cada uno y el que menor tiempo de máquina ocupe será el elegido. Esto de denomina EFICIENCIA: decir que un algoritmo es más eficiente que otro significa que se ejecuta con mayor rapidez.
- Ahora bien, si dos algoritmos (programas) son muy similares en su ejecución en el tiempo: ¿con cual nos quedamos?, pues el siguiente criterio seria el TAMAÑO, es decir el que menor número de líneas posea, lo cual lo hará más LEGIBLE y COMPRENSIBLE, siendo por tanto más fácil de mantener si en el futuro hay que modificarlo. Aquí juega un papel importante la DOCUMENTACION del programa que estemos escribiendo, es decir, es obligación del programador indicar con documentación interna (comentarios) ó externa en un archivo de texto, ó físicamente en papel, los requisitos e ideas con los que hemos diseñado el

programa para su comprensión por nosotros o por otros programadores que tuvieran que mantenerlo en el futuro

- Por último si hubiera dos algoritmos que fueran igualmente eficientes y de tamaño similar, nos quedaríamos con el que menos RECURSOS INTERNOS consumiese: menos memoria RAM, menos espacio en disco (datos en archivos) y menos cantidad de variables y estructuras de datos para desarrollarse.

La idea es describir la solución al problema (algoritmo) con la claridad suficiente como para que trasladar la solución a cualquier lenguaje de programación concreto sea inmediata.

En esta fase, frecuentemente nos encontraremos con la necesidad de resolver un problema complejo, en cuyo caso lo más adecuado es usar la técnica de "**divide y vencerás**", que consiste en **dividir el problema en subproblemas más sencillos, que pueden ser a su vez subdivididos hasta conseguir problemas de fácil solución.**

Esta técnica se conoce también como ***diseño descendente***, o ***diseño por refinamiento paso a paso o sucesivo***.

Se llama diseño descendente porque partimos de lo más complejo hasta llegar a lo más simple, descendiendo en nivel de dificultad en cada paso o refinamiento.

1.1.3. Verificación "manual" del algoritmo.

Una vez que hemos descrito el algoritmo que da solución a nuestro problema, mediante alguna herramienta descriptiva (o lenguaje algorítmico) es necesario asegurarse de que efectivamente realiza todas las tareas para las que se ha diseñado de forma correcta, produciendo el resultado esperado. Para ello será necesario "ejecutar" manualmente el algoritmo, **probándolo para un conjunto de datos que incluyan todos los casos posibles,** incluidos los más extremos y los menos frecuentes, abarcando todo el rango de valores de entrada posibles. Deberemos ir anotando en una hoja los valores intermedios de todos los **datos** que se vayan calculando, y los resultados finalmente obtenidos.

Es muy posible que durante la verificación del algoritmo nos demos cuenta de que algunos fallos se deben a un mal diseño del algoritmo, o incluso a un análisis equivocado. Esto nos puede llevar a **volver a cualquiera de estas fases anteriores, bien añadiendo o modificando especificaciones, o bien proponiendo cambios en la manera de resolver el problema**. Después, sea cual sea el caso, deberemos volver a verificar el algoritmo, como es lógico, hasta que supere todas las pruebas realizadas.

Al proceso de ir pasando por la cada instrucción del algoritmo siguiendo los cambios que se producen en las variables y actuando según esos valores se denomina **traza**.

1.2. Fase de Implementación

A esta fase llegamos con una solución a nuestro problema, que ya sabemos que es correcta. Y además la tenemos descrita de una forma clara, sin ambigüedades, por lo que si queremos resolver el problema con un ordenador bastará con **pasar (traducir o codificar) del lenguaje algorítmico a un lenguaje de programación concreto**, como por ejemplo Java, o C, o Delphi, o Visual Basic, etc.

En esta fase de implementación, además deberemos corregir los errores del programa escrito en el lenguaje de programación elegido hasta que pueda ejecutarse

Y una vez ejecutado, deberemos hacer pruebas, al igual que con el algoritmo, para verificar que todo funciona como se esperaba.

Los pasos a seguir en la fase de implementación son los que se describen en los apartados siguientes.

1.2.1. Codificación

Este paso consiste en pasar de la descripción del algoritmo a un lenguaje de programación concreto, tal como Java. Una vez que hayamos elegido el lenguaje de programación concreto, sólo tendremos que ir siguiendo el algoritmo, y **escribiendo con la sintaxis de ese lenguaje las sentencias o instrucciones que hacen lo que se indica en el algoritmo**.

Por ejemplo, si el algoritmo indica:

```
nombre ← "Juan"
Si (nombre = "Pepe")
   Escribir ( "El nombre del empleado es ", nombre)
Caso Contrario
   Escribir ("No lo conozco, pero se llama ", nombre)
Fin-Si
```

En Java deberemos escribir algo como lo que sigue:

```
String nombre = "Juan" ;
if (nombre.equals("Pepe")){
 System.out.println("El nombre del empleado es "+ nombre) ;
}else {
 System.out.println("No lo conozco, pero se llama " + nombre) ;
}
```

Aún sin conocer todavía la sintaxis de Java ni la forma de describir el algoritmo en lenguaje algorítmico, vemos que existe bastante similitud entre la descripción más o menos formal, pero cercana a nuestra forma de hablar del algoritmo y el código escrito en lenguaje Java. La diferencia es que en el segundo tenemos que tener en cuenta una serie de detalles de **sintaxis del lenguaje**, tales como que las sentencias terminan con punto y coma, o las llaves para delimitar bloques de sentencias, o las palabras concretas que usamos para comprobar si se cumple o no una condición, o la forma de asignar un valor a una variable... En Java la sintaxis es esa y hay que escribirlo así, porque si no el ordenador no lo entendería.

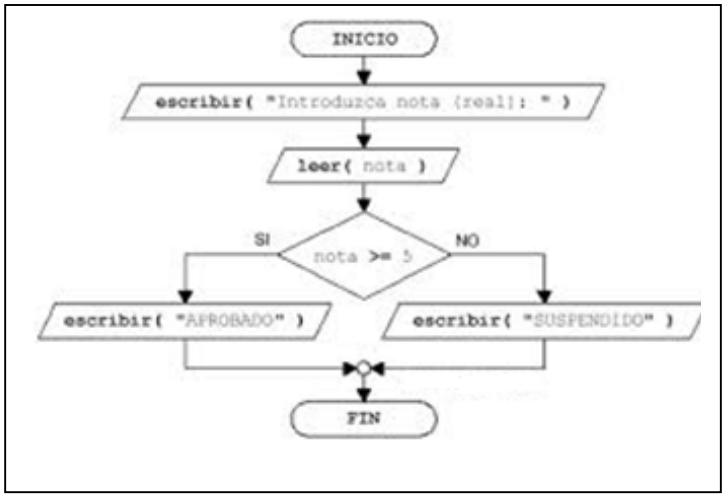
Por el contrario, en el algoritmo la sintaxis no es nada rígida. Podíamos haberlo expresado de otra forma, ya que el único requisito a cumplir es que quede claro para cualquier persona lo que hay que hacer en cada paso y el orden en el que se deben dar, sin ambigüedades.

Dicho de otra forma, **el lenguaje algorítmico es independiente del lenguaje de programación**, y por tanto no tiene por qué atenerse a una sintaxis determinada, por lo que cada persona puede expresarlo de forma distinta, siempre y cuando cumpla con las **dos reglas**:

- Sea fácil de comprender por cualquier persona.
- No sea ambiguo.

A continuación mostramos dos formas de representar el algoritmo para mostrar, dependiendo de la nota que se lea, si corresponde a un aprobado o a un suspenso. Estas dos representaciones del algoritmo nos pueden llevar al mismo código en el lenguaje de programación elegido. Por tanto la codificación, por ejemplo en Java, va a consistir en la traducción de cada uno de los pasos de una de estas representaciones de la solución, a instrucciones del lenguaje, obteniendo como resultado un programa que posteriormente se escribirá en el ordenador para su compilación y ejecución.

```
Inicio
  Visualizar (“Introduzca nota (real): “)
  Leer (nota)
  Si (nota >= 5)
    Visualizar (“APROBADO”)
 sino
    Visualizar (“SUSPENDIDO”)
  finsi
Fin
```



1.2.2. Traducción a código máquina

Los lenguajes de programación son herramientas de software que se han construido para hacer posible el desarrollo de algoritmos teóricos en un ordenador. Un algoritmo no es más que un conjunto ordenado y finito de operaciones que permiten la resolución de un problema dado.

Lenguajes de bajo nivel

Los ordenadores funcionan con corriente, y eso es lo único que pueden entender. Concretamente, funcionan con sólo dos valores de corriente distintos y bien diferenciados (Por Ejemplo: En una línea de un circuito, en un momento dado hay una tensión de +5V o una tensión de -5V, y ningún otro valor es posible). **A esos dos valores distintos los identificamos como 0 y 1**, y decimos que los ordenadores son binarios, o que trabajan en binario. Pues bien, cualquier instrucción que queramos que ejecute el ordenador, deberemos expresársela en forma de ceros y unos, que es lo único que entiende. Ese lenguaje es lo que llamamos **lenguaje máquina**, y decimos que es un **lenguaje de bajo nivel**, ya que está alejado de nuestra forma de pensar y expresarnos.

En los albores de la informática la manera de introducirle un programa a un ordenador era básicamente mediante **lenguaje de máquina**. Eso hacía muy pesado y engorroso la construcción de programas y además era muy específico para cada máquina. Por el contrario era muy rápido en ejecutarse ya que no había que hacer traducción intermedia.

Más adelante se inventaron los lenguajes **ensambladores** que permitían introducir instrucciones mediante palabras, generalmente en inglés y estructuras de instrucciones como repeticiones y bifurcaciones. Este lenguaje era más sencillo para el programador pero requería de traducción del texto introducido a lenguaje máquina.

¿Quién realiza esa traducción de lenguaje de alto nivel a código máquina? La respuesta es que **la realizan automáticamente unos programas llamados Traductores**. Existen dos tipos de traductores, que son los **compiladores** y los **intérpretes**.

Un **intérprete** es un programa traductor al que se le pasa como dato de entrada el fichero de texto con el programa escrito en lenguaje de alto nivel (**código fuente**), leyendo una a una las instrucciones del programa, la analiza y si contiene errores detiene el proceso y lo notifica para editar esa instrucción en el código fuente. Si no hay errores, **la traduce a código máquina en memoria y la ejecuta**, sin guardar la traducción en ningún sitio. Por tanto, si quiero volver a ejecutar de nuevo el programa, deberé volver a traducir una a una cada línea antes de ejecutarla, por lo que la ejecución será más lenta. Por el contrario, el tiempo invertido en la depuración suele ser menor, ya que los mensajes de error suelen ser más precisos y estar más localizados.

Los compiladores traducen todo el programa escrito en un determinado lenguaje de alto nivel, obteniéndose como resultado un fichero con el programa traducido a código máquina (**código objeto**) que es ejecutable sin tener que volver a traducir. Además de traducir, los modernos compiladores suelen darnos un informe de errores, las líneas donde se han producido y sus causas. Los entornos IDE actuales ayudan mucho en esta tarea.

El lenguaje de máquina se denomina de bajo nivel por estar más cerca de la maquina que del hombre. Los lenguajes de programación que emplean frases y sentencias con palabras (por supuesto en inglés) se denominan de alto nivel por estar más cerca del hombre que de la máquina. Ello redunda, como veremos a continuación, en una mayor portabilidad.

Java utiliza un método intermedio para obtener aún más portabilidad. El código fuente de Java se traduce a un código máquina estandarizado denominado **Bytecode**. No existen máquinas que utilicen *Bytecode* como lenguaje nativo. En su lugar, en cada máquina en la que se ejecute, debe existir otro programa denominado **Máquina Virtual de Java (JVM)**, que sirve como intérprete de *Bytecode*. La JVM puede utilizar diferentes estrategias de traducción, aunque la más habitual es el uso de los compiladores denominados **Just In Time (JIT)**, que traducen el código conforme lo ejecutan pero guardan un buffer para acelerar la traducción en algunos casos.

Lenguajes de alto nivel

Esta gama de lenguajes se crearon con tres objetivos principales:

- Lograr la independencia del lenguaje con el ordenador ó sistema operativo, es decir, dar PORTABILIDAD a los programas.
- Hacer más sencillo el desarrollo de programas, aproximando la sintaxis de las instrucciones al lenguaje humano más que al lenguaje de la maquina (binario).
- Proporcionar librerías de rutinas y utilidades, para poder ser reutilizadas con frecuencia y no tener que volver a escribirlas.

Entre estos lenguajes podemos citar los siguientes:

FORTRAN

Lenguaje de ámbito científico-técnico creado en 1955 por IBM, posee gran cantidad de rutinas matemáticas, derivación, integración numérica sistemas de ecuaciones etc. etc. Su nombre deriva de la frase Formula Translator.

COBOL

Su nombre procede de las iniciales de Common Business Oriented Language, es decir leguaje común orientado a los negocios, diseñado para aplicaciones de gestión y sobre todo para manejar información almacenada en archivos en disco, su fecha de creación fue 1960.

BASIC

Diseñado en 1965 por un grupo de profesores americano del Darmouth College, para la enseñanza de la programación, es un lenguaje de propósito general que se popularizó mucho al ser incluido como software en los ordenadores personales a partir de los años 80.

PASCAL

Desarrollada por el matemático suizo Niklaus Wirth, es un lenguaje creado en los años 70 y muy difundido durante años para la enseñanza de la programación en Universidades, es un lenguaje estructurado y de propósito general, es decir no está orientado a ningún tipo de aplicaciones concretas.

C

Desarrollado por Denis Ritchie en 1970, en los laboratorios Bell en Estados Unidos, en 1978 este autor escribió el libro "The C language", que fue un estándar de facto para los lenguajes C y en 1989 el comité ANSI estandarizó el C marcando los requisitos que todo compilador de C debía cumplir, para ser denominado así. Es un lenguaje de propósito general, con multitud de rutinas ya prediseñadas y potente manejo de memoria dinámica, los que le convierten en una ágil herramienta para construir programas estructurados compactos y eficientes. A partir de éste Bjarne Strousup creó C++, que es un lenguaje orientado objetos.

JAVA

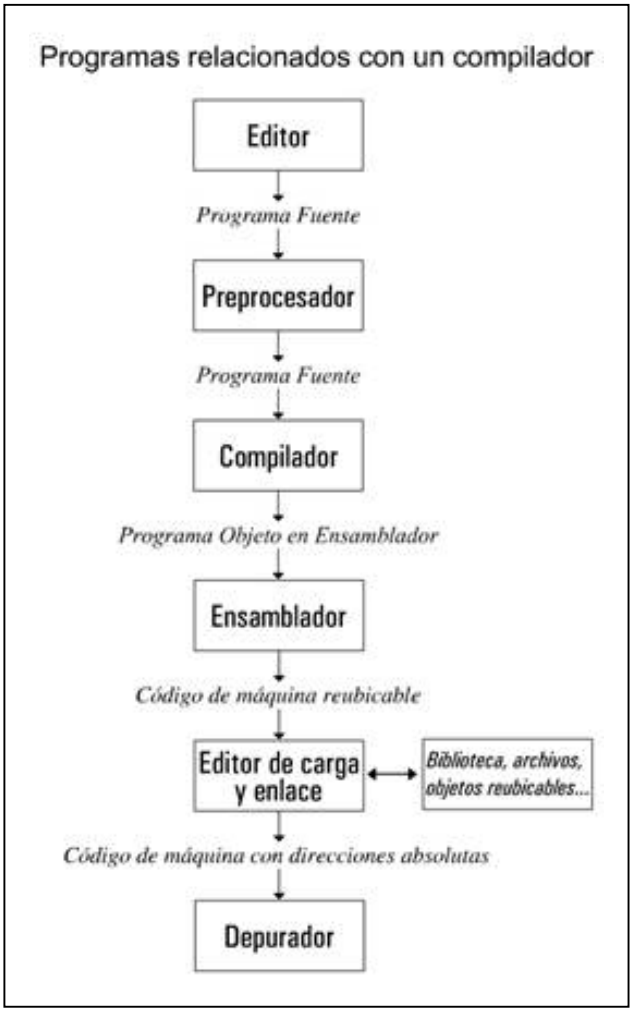
Creado por la empresa Sun Microsystems a partir de 1991, orientado totalmente a objetos y precompilado es decir independiente de la plataforma, es uno de los lenguajes más populares en la actualidad.

1.2.3. Depuración

El proceso de depuración incluye la realización de un ciclo de tareas, hasta que el resultado es correcto. **Ése ciclo consiste fundamentalmente en:**

1. **compilar el programa**
2. **analizar el listado de errores**
3. **editar y corregir el texto del programa**
4. **volver a compilar el programa.**

Para entender ese proceso hay que detenerse un momento a explicar el significado de la palabra **compilar.** En la imagen se puede ver el camino que sigue un programa durante la compilación.



1.2.4. Pruebas

Una vez finalizada la depuración, dispondremos de la posibilidad de ejecutar el programa. Ahora bien, es posible que aunque el programa funcione no haga exactamente lo que se necesita, o que funcione mal para algún dato de entrada, etc.

Será por ello necesario probar el funcionamiento del programa con un conjunto de **datos** lo suficientemente significativos, incluyendo valores extremos o raros.

También debemos asegurarnos de haber probado toda la casuística prevista en el programa, de forma que no queden bloques de código sin someter a su oportuno test de prueba. **Cualquier fallo detectado nos puede llevar a replantear cualquiera de las etapas anteriores**, incluyendo el análisis o el diseño del algoritmo, o la codificación. No obstante, lo más frecuente es que se trate de un error en la etapa inmediata anterior.

1.3. Fase de Explotación y Mantenimiento.

Una vez que la aplicación ya ha sido probada con éxito es el momento de instalarla para ponerla en marcha. Esta es la **fase de explotación** de la aplicación, y suele ser la más larga en el tiempo, y la que mayor coste supone a lo largo de la vida útil de la aplicación. A lo largo de la misma, será necesario ir haciendo algunas **tareas de mantenimiento de la aplicación**, para corregir errores, introducir mejoras, adaptaciones, etc.

En las grandes empresas de desarrollo del software, suele utilizarse un equipo de personas especializado en esta fase que además recoge todos los problemas encontrados en un documento de evaluación de la aplicación. Una vez concluida esta fase comprobando que la aplicación hace lo que debe sin errores y sin incoherencias es cuando se da por finalizado el servicio.

2. Técnicas de resolución de problemas

Como hemos comentado en el punto 1.1.2, solucionamos problemas todos los días, a menudo, sin ser conscientes del proceso que seguimos para hacerlo. En un entorno de aprendizaje, por lo general, nos dan la mayor parte de la información que necesitamos:

- un claro planteamiento del problema,
- la información necesaria,
- y la salida requerida.

En la vida real, por supuesto, el proceso no es siempre tan simple: en general, tenemos que definir el problema nosotros mismos y después decidir con que información tenemos que trabajar y cuáles deben ser los resultados.

Después de comprender y analizar un problema, hay que llegar a una solución potencial, un algoritmo. Anteriormente, hemos definido un algoritmo como un conjunto ordenado y finito de operaciones que permiten la resolución de un problema dado, esto es, el procedimiento paso a paso para resolver el problema en una cantidad finita de tiempo usando una cantidad finita de datos.

Aunque trabajamos con algoritmos todo el tiempo, la mayor parte de nuestra experiencia con ellos se presenta en el contexto en el que los utilizamos: seguir una receta, jugar a un juego, etc. En la fase de resolución de problemas, sin embargo, diseñamos algoritmos realmente. Esto significa que debemos ser conscientes de las estrategias que utilizamos para resolver problemas de manera que podamos aplicarlos a la programación eficazmente.

2.1. Realiza preguntas

Si alguien nos propone oralmente alguna tarea, lo primero que hacemos es realizar preguntas -¿Cómo?, ¿Dónde?, ¿Cuándo?, ¿Por qué?, etc.- hasta comprender exactamente qué hacer. Si ésta está escrita, haremos anotaciones al margen, subrayaremos o indicaremos de alguna manera aquellas partes del texto que no tengamos claras. De ese modo, las cuestiones que se presenten pueden ser discutidas posteriormente.

En el contexto de la programación, algunas de las preguntas más comunes que podemos hacernos son:

- Con qué objetos tengo que trabajar
- Qué tareas realizan
- Qué información se introduce
- Cómo sé que ha introducido toda la información
- Cómo debe ser la salida
- Qué errores se pueden producir

2.2. Buscar cosas familiares

No debemos reinventar la rueda: si ya existe una solución, usémosla. Las personas reconocemos fácilmente situaciones similares. No necesitamos aprender cómo comprar el pan, después cómo comprar la leche o los huevos; Sabemos que “ir a la tienda” es siempre lo mismo y sólo cambia qué compramos cada vez. Por ello, si hemos solucionado el mismo problema o uno similar con anterioridad, simplemente, reutilicemos dicha solución.

En programación, muchos problemas se producen una y otra vez en diferentes formas. Un buen programador enseguida reconoce algo que ya resuelto con anterioridad y aplica la solución. Un ejemplo muy sencillo es el de encontrar las temperaturas máxima y mínima de una jornada o las notas más alta y más baja de un examen; consiste en obtener los valores mayor y menor en un conjunto de números.

2.3. Resolución por analogía

A menudo, un problema puede recordar a otro que se ha visto antes. Es posible encontrar la solución del problema más fácilmente si se recuerda como se resolvió el anterior. En otras palabras, se puede trazar una analogía entre los dos problemas a medida que se resuelve el nuevo, descubriendo las cosas que son diferentes y tratar estos detalles de menor importancia uno a uno.

La analogía es en realidad una aplicación más amplia de la estrategia de buscar cosas que son familiares. Cuando tratas de encontrar un algoritmo para resolver un problema, no te limites a buscar “soluciones informáticas”, trata de obtener una visión más amplia del problema. No te preocupes si tu analogía no coincide perfectamente: la única razón para utilizar una analogía es que proporciona algo para empezar. Los mejores programadores son personas que tienen una amplia experiencia en la resolución de todo tipo de problemas.

2.4. Análisis medios-fines

Por lo general, conocemos el estado inicial y el estado final; el problema consiste en definir un conjunto de instrucciones que te conduce de un estado a otro. Supongamos que quieres ir de Zaragoza a Huelva. Sabes el estado inicial (Zaragoza) y el estado final (Huelva): el problema es cómo llegar de un lugar a otro.

En este ejemplo, tienes un montón de opciones. Puedes coger un avión, caminar, hacer dedo, ir en bici... El método que elijas dependerá de tus circunstancias. Supongamos que quieres ir en tren pero que no existe un tren directo de Zaragoza a Huelva.

Una vez que hayas identificado los objetos esenciales y sus características (viaje por tren, con transbordo), hay que determinar los detalles: establecer objetivos intermedios que son más fáciles de cumplir que el objetivo general. Suponemos que sí existen trenes de Zaragoza a Madrid, Madrid a Sevilla y Sevilla a Huelva, que podemos utilizar.

La estrategia global del análisis de medios-fines consiste en definir los extremos y luego de analizar los medios para alcanzarlos. Este proceso se traduce fácilmente a un programa de ordenador. Esto es, comienzas escribiendo lo que es la entrada y cuál debe ser la salida. Después, tienes en cuenta los objetos disponibles y las acciones que pueden realizar y eliges una secuencia de acciones que puedan transformar la entrada en los resultados deseados.

2.5. Divide y vencerás

Como ya hemos comentado con anterioridad, a menudo separamos un problema de grandes dimensiones en unidades más pequeñas que son más fáciles de gestionar. Limpiar toda la casa puede parecer abrumador; sin embargo, la limpieza de cada habitación por separado parece algo más llevadero. El mismo principio se aplica a la programación: romper un gran problema en trozos más pequeños que podemos resolver de forma individual.

2.6. Construcción por bloques

Otra manera de atacar a un gran problema es ver si existen soluciones para problemas más pequeños. Resulta posible combinar estas soluciones para resolver la mayor parte de los grandes problemas. Esto es sólo una combinación de las técnicas buscar cosas familiares y de divide y vencerás. Observas el problema y buscas como dividirlo en problemas más pequeños para los cuales ya existe solución. La solución al problema consiste en unir las soluciones ya existentes, como cuando construyes una pared utilizando ladrillos y mortero.

2.7. Bloqueo mental: el miedo a empezar.

Aunque nos resulta familiar la expresión del “miedo al folio en blanco” que normalmente asociamos a los escritores, también los programadores se encuentran con esa dificultad la primera vez que afrontan un nuevo problema.

En estos casos conviene siempre escribir en un papel, con tus propias palabras, cuál es el problema para de esa manera entenderlo. Una vez que has descrito el problema puedes fijar tu atención en cada una de sus partes en lugar de tratar de resolverlo todo de vez. Este proceso proporciona una visión global, lo que facilita encontrar partes que resultan familiares o que son análogas a otros problemas que ya has resuelto. Además, resaltan aspectos que no están claros y para los que se necesita más información.

La mayoría de los bloqueos mentales son debidos a que no hay una comprensión real del problema. Describir el mismo con tus propias palabras ayuda a reconocer los estados inicial y final, así como las operaciones que hay que realizar, sus subpartes, comprendiendo cuáles son los requisitos para solucionarlo.

3. Paradigmas de programación

Un paradigma representa un enfoque particular o una filosofía para diseñar soluciones a los problemas planteados. La diferencia entre los mismos la encontramos en los conceptos y la forma en que cada uno de ellos abstrae los elementos involucrados en el problema y la forma de resolverlo.

3.1. Programación imperativa

Describe el programa en función de una serie de estados y las operaciones que modifican dicho estado. Los programas imperativos son un conjunto de instrucciones que le indican al ordenador cómo hacer una tarea.

Los primeros lenguajes imperativos son los lenguajes máquina. Desde la perspectiva de las máquinas, el programa, contenido en la memoria, es una serie de instrucciones que modifica áreas de la memoria.

3.2. Programación declarativa

Describe el programa en función de lo que se quiere obtener. Las instrucciones solo establecen el objetivo a alcanzar no cómo se ha de conseguir. Es decir se define qué queremos obtener no cómo lo obtendremos.

3.3. Programación funcional

Se basa en el uso de funciones matemáticas, de manera que no sea necesario describir el proceso de cómputo y evitar el concepto de estado, propio de la programación imperativa.

Entre sus características destaca la no existencia de variables y construcciones estructuradas, aunque algunos lenguajes híbridos admiten el uso de estos conceptos. Se han usado ampliamente en el ambiente académico pero no tanto en el comercial.

3.4. Programación lógica

Se basa en el concepto de predicado o de relación entre elementos, utilizando la teoría lógica de primer orden. En general, toma en consideración una serie de hechos

de manera que, cuando formula una hipótesis se apoya en dichos hechos. Si todos los hechos en los que se apoya la hipótesis son ciertos, entonces también la hipótesis es cierta; si alguno de los hechos no es cierto, no se puede inferir que la hipótesis sea cierta, aunque eso no implica que la hipótesis sea falsa.

Se utiliza ampliamente en aplicaciones de inteligencia artificial como los sistemas expertos, la demostración automática o el reconocimiento del lenguaje.

3.5. Programación concurrente

Consiste en la simultaneidad en la ejecución de dos o más tareas interrelacionadas. Por ello, se centra en el estudio de de las interacciones entre las diferentes tareas, de la comunicación entre las mismas y el acceso coordinado a los recursos que comparten.

Normalmente, los lenguajes que siguen este paradigma también están basados en otro, como la programación imperativa o la programación orientada a objetos.

3.6. Programación orientada a objetos

Los elementos fundamentales de construcción de programas son objetos, abstracciones de algún ente del mundo real con sus propias características y comportamiento. De este modo, los objetos –cada uno con su propia funcionalidad- manipulan los datos de entrada para obtener los datos de salida deseados.

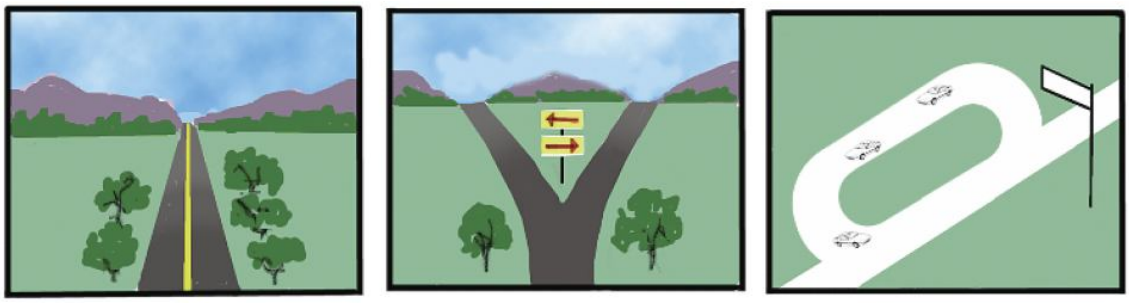
Aunque surge a finales de los 60, es a mediados de los 80 cuando se populariza hasta ser el paradigma dominante a partir de los 90 consolidado, en gran parte, por el auge de las interfaces gráficas de usuario. Tanto es así que la mayoría de lenguajes clásicos han incorporado las características de la POO o bien han surgido versiones del lenguaje que las incorporan.

4. Técnicas de programación: Programación estructurada, modular y orientada a objetos.

4.1. Programación estructurada

Esta técnica de desarrollo de programas se encamina a conseguir programas estructurados. Los programas estructurados responden a una técnica trabajada por E. Dijkstra profesor de computación en una Universidad de Holanda , que demostró que los programas estructurados son más fáciles de comprender , elaborar y sobre todo de mantener, los programas estructurados responden al Teorema Fundamental de la estructura enunciado el 1966 que dice:

"Todo programa propio puede ser escrito usando solamente las estructuras secuencial (flujo de ejecución de arriba hacia abajo), selectiva (toma de decisión) y repetitiva (bucles)".



Un programa se define como **propio** si:

- **Posee un solo punto de entrada y de salida**
- **Todas las sentencias son *alcanzable*s, es decir, no contiene *código muerto*, que no se ejecutará nunca**
- **Todos los posibles caminos llevan desde el punto de entrada al de salida.**
- **Todas las instrucciones son ejecutables y no existen bucles infinitos**

Indicaremos aquí que el lenguaje Java es un lenguaje que permite elaborar programas estructurados y con manejo de objetos con facilidad y rapidez.

C++ también, aunque es algo más complicado porque usa punteros.

4.2. Programación modular

Esta técnica combinada con la estructurada hace que la productividad de la programación de los proyectos a realizar sea mayor pues permite trabajo en paralelo, por varias personas de un equipo. Se basa en descomponer una tarea en tareas más sencillas y luego armonizarlas todas en un programa principal. Cada una de esas tareas será:

- sencilla.
- programa estructurado.
- ejecutado desde un programa denominado principal.

Adelantando algo del tema siguiente diremos que esto se consigue mediante la construcción de **funciones en C++ ó métodos en Java**, y el paso de parámetros.

4.3. Programación orientada a objeto.

La programación orientada a objetos se basa en un modelo de lo que vemos en la vida real. El mundo se ve desde el punto de vista de los objetos que hay en él. Los objetos tienen características **(propiedades)** y con ellos pueden realizarse acciones **(métodos)**. En esta filosofía se busca encontrar los objetos, en vez de las funciones (que es lo que se hace en programación estructurada).

Todo programa en Java se construye a base de objetos. Pero en realidad, el programador de Java no define objetos, sino clases. Y en concreto, clases de objetos.

Pero, ¿qué es una clase? Una clase es una plantilla de un conjunto de objetos que tienen características similares.

La clase es la definición de un tipo de objeto. Al definir una clase lo que se hace es indicar cómo funciona un determinado tipo de objetos. Luego, a partir de la clase, podremos crear objetos de esa clase.

Por ejemplo, si quisiéramos crear el juego del parchís en Java, una clase sería la casilla, otra las fichas, otra el dado, etc., etc. En el caso de la casilla, se definiría la clase para indicar su funcionamiento y sus propiedades, y luego se crearía tantos objetos casilla como casillas tenga el juego.

Lo mismo ocurriría con las fichas, la clase ficha definiría las propiedades de la ficha (color y posición por ejemplo) y su funcionamiento mediante sus métodos (por ejemplo un método sería mover, otro llegar a la meta, etc., etc.,), luego se crearían tantos objetos ficha, como fichas tenga el juego.