Descargar PDF

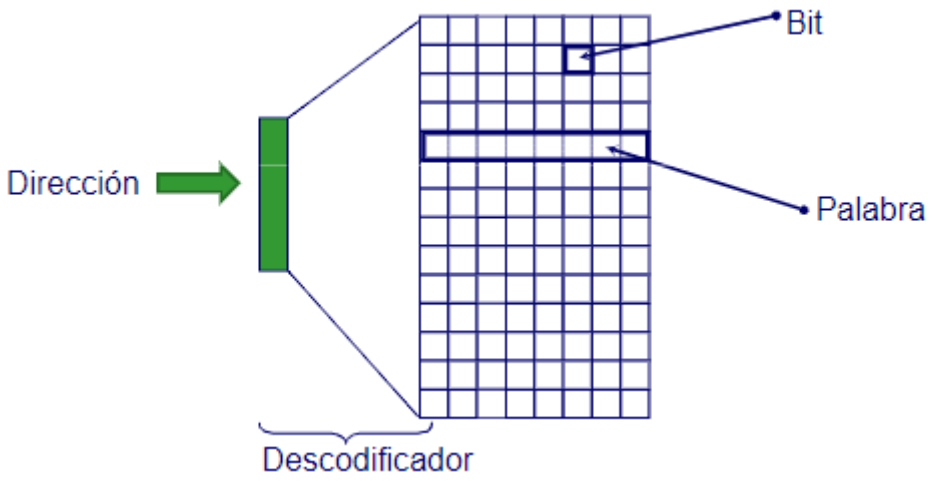
La informática es la ciencia que estudia todo lo regerente al procesamiento de la información de manera automática. La programación es una de las disciplinas más importantes de la informática, ya que es la que permite a los seres humanos comunicarse con las máquinas.

Un programa es una secuencia de instrucciones que le indican a una máquina qué hacer. Estas instrucciones se escriben en un lenguaje de programación, que es un lenguaje formal que se utiliza para expresar algoritmos.

## Arquitectura básica de un computador

Un computador es una máquina que está compuesta por varios elementos que trabajan juntos para procesar la información. El elemento más importante de un ordenador junto a su procesador es la memoria principal.

La memoria principal es el lugar donde se almacenan los datos y las instrucciones que se están utilizando en un momento dado. Se producen operaciones básicas de lectura y escritura.




La memoria de un ordenador tiene una organización matricial, en la que cada celda tiene una dirección única. La memoria se divide en dos tipos:

- Memoria RAM (Random Access Memory): es una memoria volátil, lo que significa que los datos se pierden cuando se apaga el computador.
- Memoria ROM (Read Only Memory): es una memoria no volátil, lo que significa que los datos se mantienen aunque se apague el computador.

El direccionamiento de la memoria se realiza mediante un sistema de coordenadas, en el que cada celda tiene una dirección única. El tamaño de la memoria es el número de celdas que tiene:

```
Capacidad = (Número de palabras * Tamaño de la palabra) - 1
Número de palabras = 2^16 = 65536
Tamaño de la palabra = 8 bits = 1 byte
Capacidad = (65536 * 1) - 1 = 65535
```

Las unidades de almacenamiento de la memoria son los bits y los bytes. Un bit es la unidad mínima de información, que puede tener dos valores: 0 o 1. Un byte es un conjunto de 8 bits, que puede representar 256 valores diferentes.

```
Número de valores = 2^8 = 256
Octeto o byte = 8 bits
Kilobyte (KB) = 2^10 bytes = 1024 bytes
Megabyte (MB) = 2^20 bytes = 1048576 bytes o 1024 KB
Gigabyte (GB) = 2^30 bytes = 1073741824 bytes o 1024 MB
```

## Introducción a los sistemas operativos

Un sistema operativo es un conjunto de programas que se encargan de gestionar los recursos de un computador. Presenta una interfaz entre el usuario y el hardware, y se encarga de gestionar los recursos de la máquina. Los sistemas operativos se dividen en dos tipos:

- Sistemas operativos de usuario: son los que se instalan en los computadores personales y permiten a los usuarios interactuar con la máquina.
- Sistemas operativos de tiempo real: son los que se utilizan en los sistemas embebidos y en los sistemas de control.

Un programa debe estar en memoria para ser ejecurado. El sistema operativo se encarga de cargar el programa en memoria, asignarle un espacio de memoria y ejecutarlo. Pueden existir varios programas en memoria al mismo tiempo, para esto el sistema operativo se encarga de gestionar la memoria, asignando y liberando espacio de memoria.


## Introducción a los lenguajes de programación

Se utilizan lenguajes de programación para escribir programas que se ejecutarán en un computador. Los lenguajes de programación se dividen en dos tipos según su cercanía al lenguaje máquina:

- Lenguajes de bajo nivel: son lenguajes que están muy cerca del lenguaje máquina, por lo que son difíciles de entender y de escribir. Ejemplos: ensamblador.

- Lenguajes de alto nivel: son lenguajes que están más alejados del lenguaje máquina, por lo que son más fáciles de entender y de escribir. Ejemplos: C, C++, Java, Python.

Los lenguajes de programación se dividen en dos tipos según su forma de ejecución:

- Lenguajes compilados: son lenguajes que se traducen a lenguaje máquina antes de ser ejecutados. Ejemplos: C, C++.
- Lenguajes interpretados: son lenguajes que se traducen a lenguaje máquina mientras se ejecutan. Ejemplos: Python, Ruby.

Los lenguajes de bajo nivel son más rápidos que los lenguajes de alto nivel, pero son más difíciles de entender y de escribir. Son dependientes de la máquina, por lo que un programa escrito en un lenguaje de bajo nivel no se puede ejecutar en otra máquina.

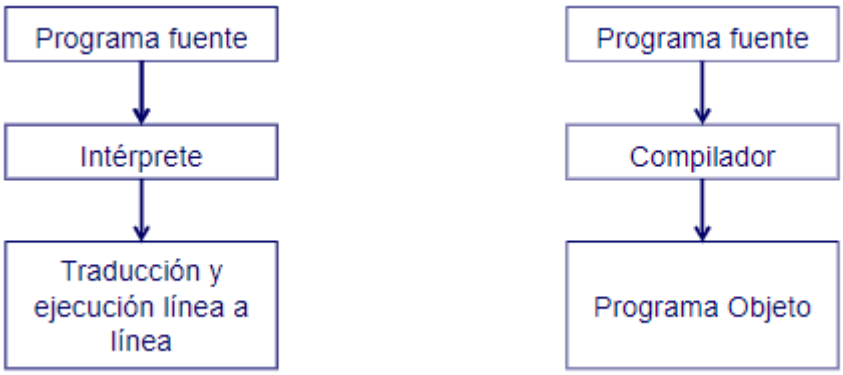
Los lenguajes de alto nivel son más fáciles de entender y de escribir, pero son más lentos que los lenguajes de bajo nivel. Están diseñados para ser legibles por los humanos, por lo que un programa escrito en un lenguaje de alto nivel se puede ejecutar en cualquier máquina, es decir, son portables.

La principal ventaja de los lenguajes de alto nivel es que permiten a los programadores escribir programas más rápidamente, ya que no tienen que preocuparse por los detalles de la máquina. Además, los lenguajes de alto nivel son más fáciles de depurar, ya que los errores son más fáciles de encontrar. Por contra, los lenguajes de alto nivel son más lentos que los lenguajes de bajo nivel, ya que tienen que ser traducidos a lenguaje máquina antes de ser ejecutados y no se aprovechan al máximo las características de la máquina.

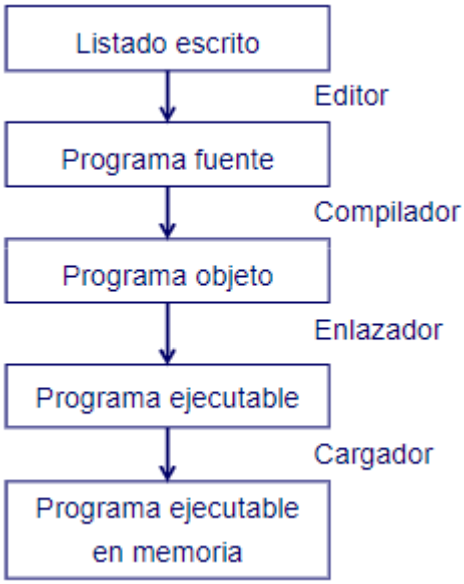
El traductor es un programa que se encarga de traducir un programa escrito en un lenguaje de programación a lenguaje máquina. Los traductores se dividen en varios tipos:

- Ensamblador: es un traductor que traduce un programa escrito en lenguaje ensamblador a lenguaje máquina.
- Compilador: es un traductor que traduce un programa escrito en un lenguaje de programación a lenguaje máquina de una sola vez.
- Intérprete: es un traductor que traduce un programa escrito en un lenguaje de programación a lenguaje máquina línea a línea.

La diferencia entre un compilador y un intérprete es que un compilador traduce un programa de una sola vez, mientras que un intérprete traduce un programa línea a línea. Los lenguajes compilados son más rápidos que los lenguajes interpretados, ya que no tienen que ser traducidos cada vez que se ejecutan.




La compilación es el proceso de traducir un programa escrito en un lenguaje de programación a lenguaje máquina o programa objeto. Para ello, debe usar o montador o enlazador.




# Programación estructurada

## Resolución de problemas y algoritmos

Un algoritmo es un conjunto de instrucciones que se utilizan para resolver un problema. Un algoritmo se puede representar de varias formas, como un diagrama de flujo, un pseudocódigo o un programa de computador. Las características de un algoritmo son:

- Finito: un algoritmo debe terminar en algún momento.
- Preciso: un algoritmo debe indicar el orden de realiazación de las instrucciones paso a paso.
- Determinista: un algoritmo debe producir el mismo resultado para los mismos datos de entrada. Es decir, siempre comportarse de la misma manera.

El diseño de un algortimo debe ser descendente, es decir, se debe empezar por el problema general y se debe ir descomponiendo en problemas más pequeños. Para ello, se utilizan técnicas de refinamiento, como la división y conquista, la recursividad o la programación dinámica.

```
Inicio
Leer a
Leer b
c = a + b
Escribir c
```

El desarrollo de algoritmos o desarrollo de software es el proceso de creación de un programa de computador. El desarrollo de software se divide en varias fases:

- Análisis: es la fase en la que se comprende el problema, identifican los requisitos del sistema y se definen los objetivos del proyecto.
- Diseño: es la fase en la que se definen las especificaciones del sistema, se diseñan los módulos del sistema y se planifica el desarrollo del sistema.
- Implementación: es la fase en la que se escribe el código del sistema, se prueban los módulos del sistema y se corrigen los errores del sistema.

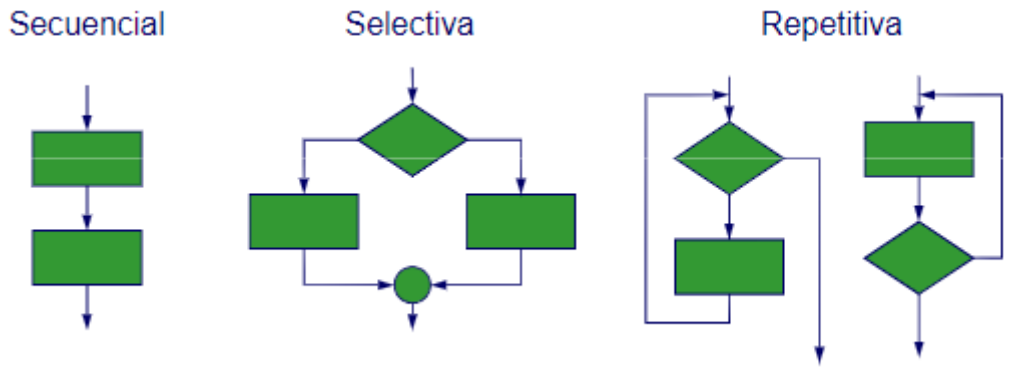
- Pruebas: es la fase en la que se prueban los módulos del sistema, se prueban los casos de prueba del sistema y se corrigen los errores del sistema.
- Mantenimiento: es la fase en la que se corrigen los errores del sistema, se mejoran las funcionalidades del sistema y se actualiza el sistema.

La programación modular o programación estructurada es una técnica de programación que consiste en dividir un programa en módulos o subprogramas. Los módulos son bloques de código que realizan una tarea específica y se pueden reutilizar en otros programas. La programación modular tiene varias ventajas:

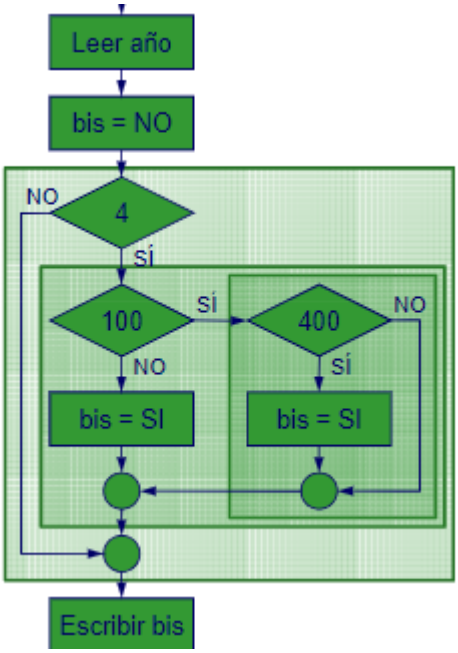
- Facilita la comprensión del programa.
- Facilita la depuración del programa.
- Facilita la reutilización del código.
- Facilita la modificación del programa.

La programación estructurada usa solamente tres estructuras de control: secuencia, selección e iteración.

- La estructura de control de secuencia es una secuencia de instrucciones que se ejecutan una detrás de otra.
- La estructura de control de selección es una estructura que permite ejecutar un bloque de instrucciones si se cumple una condición.
- La estructura de control de iteración es una estructura que permite ejecutar un bloque de instrucciones varias veces.




Veamos un ejemplo de cómo se puede resolver un problema con programación estructurada. Supongamos que queremos calcular si un año es bisiesto o no. Un año es bisiesto si es divisible por 4, pero no es divisible por 100, a menos que sea divisible por 400. Para resolver este problema, podemos utilizar la siguiente estructura de control:




```
Inicio
Leer año
Si año % 4 == 0 y año % 100 != 0 o año % 400 == 0
    Escribir "El año es bisiesto"
Sino
    Escribir "El año no es bisiesto"
```

# Codificación de la información

## Introducción

Los ordenadores almacenan información de forma diferente a la nuestra: mientra que nosotros manejamos conceptos como números enteros, números reales, alfabeto, colores, ... los ordenadores solo entienden secuencias de unos y ceros.

Es por ello que necesitamos un mecanismo que permita traducir la información que manejamos a secuencias de unos y ceros. A este proceso se le llama codificación de la información. A la información que se codifica se le llama datos, y a las diferentes formas de codificar la información se les llama tipos de datos.

Los sistemas posicionales son aquellos en los que el valor de un dígito depende de su posición dentro de la secuencia de dígitos. Un dígito tendrá menos valor conforme más a la derecha esté, y más valor conforme más a la izquierda se encuentre.

```
4532 = 4000 + 500 + 30 + 2 = 4 * 10^3 + 5 * 10^2 + 3 * 10^1 + 2 * 10^0
```

## Codificación de enteros sin signo

Los números enteros sin signo (o positivos) se condifican utilizando la codificación binaria. La codificación binaria es un sistema posicional en el que cada dígito puede tener dos valores: 0 o 1.

```
    Para encontrar la representación binaria de 25, debemos dividir 25
entre 2 sucesivamente y guardar el resto de cada división:
    25 / 2 = 12 resto 1
    12 / 2 = 6 resto 0
    6 / 2 = 3 resto 0
    3 / 2 = 1 resto 1
    1 / 2 = 0 resto 1
    Por lo que 25 en binario es 11001

    Si queremos encontrar la representación decimal de 11001. Debemos
multiplicar cada dígito por 2 elevado a la posición que ocupa:
    1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 16 + 8 + 0 + 0 + 1 =
25
```

Para operar con números binarios, se utilizan las mismas operaciones que con los números decimales, pero con las siguientes reglas.

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 10 |

| × | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Aunque los números enteros sin signo se pueden representar con cualquier número de bits, en la práctica se suelen utilizar la bases de numeración octal (base 8) y hexadecimal (base 16) para simplificar la representación de los números binarios.

El sistema de numeración octal es un sistema posicional en el que cada dígito puede tener ocho valores: 0, 1, 2, 3, 4, 5, 6, 7. Para convertir un número decimal a octal, se divide el número decimal entre 8 sucesivamente y se guardan los restos de cada división.

| Octal | Binario | Octal | Binario |
|---|---|---|---|
| 0 | 000 | 4 | 100 |
| 1 | 001 | 5 | 101 |
| 2 | 010 | 6 | 110 |
| 3 | 011 | 7 | 111 |

```
    Para encontrar la representación octal de 25, debemos dividir 25 entre
8 sucesivamente y guardar el resto de cada división:
    25 / 8 = 3 resto 1
    3 / 8 = 0 resto 3
    Por lo que 25 en octal es 31

    Para convertir un número binario a octal, se agrupan los dígitos
binarios de tres en tres y se convierten a octal:
    110 011 -> 6 3
```

El sistema de numeración hexadecimal es un sistema posicional en el que cada dígito puede tener dieciséis valores: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Para convertir un número decimal a hexadecimal, se divide el número decimal entre 16 sucesivamente y se guardan los restos de cada división.

| Hexadecimal | Decimal | Binario | Hexadecimal | Decimal | Binario |
|---|---|---|---|---|---|
| 0 | 0 | 0000 | 8 | 8 | 1000 |
| 1 | 1 | 0001 | 9 | 9 | 1001 |
| 2 | 2 | 0010 | A | 10 | 1010 |
| 3 | 3 | 0011 | B | 11 | 1011 |
| 4 | 4 | 0100 | C | 12 | 1100 |
| 5 | 5 | 0101 | D | 13 | 1101 |
| 6 | 6 | 0110 | E | 14 | 1110 |
| 7 | 7 | 0111 | F | 15 | 1111 |

```
    Para encontrar la representación hexadecimal de 25, debemos dividir 25
entre 16 sucesivamente y guardar el resto de cada división:
    25 / 16 = 1 resto 9
    1 / 16 = 0 resto 1
    Por lo que 25 en hexadecimal es 19

    Para convertir un número binario a hexadecimal, se agrupan los dígitos
binarios de cuatro en cuatro y se convierten a hexadecimal:
    1100 1101 -> C D
```

El desbordamiento de un número ocurre cuando el resultado de una operación aritmética es mayor que el rango de valores que se pueden representar con el número de bits disponibles. El desbordamiento de un número puede ser positivo o negativo, y puede ser detectado mediante la comparación de los bits de signo de los operandos y del resultado. Para evitarlo, es fundamental que el programador tenga en cuenta el rango de valores que se pueden representar con el número de bits disponibles, y seleccionar un tipo de dato adecuado para la operación que se va a realizar.

```
    Supongamos que queremos representar el mes del año (del 1 al 12),
evidentemente tendríamos suficiente con 4 bits.
    2^4 = 16 > 12

    Supongamos que queremos representar el día de la semana (del 1 al 7),
evidentemente tendríamos suficiente con 3 bits.
    2^3 = 8 > 7

    Sin embargo, si queremos representar el número de segundos en un día
(del 0 al 86400), necesitaríamos 17 bits.
    2^17 = 131072 > 86400
```

## Codificación de enteros con signo

Para codificar números enteros con signo se utiliza la codificación de complemento a 2. Esta codificación es un sistema posicional en el que el bit más significativo se utiliza para representar el signo del número: 0 para positivo y 1 para negativo. Presenta, además, varias ventajas sobre otras posibles codifications entre las que podemos destacar:

- Aprovecha al máximo el rango de valores que se pueden representar con el número de bits disponibles.

- Restar dos números equivale a sumar un número y su complemento a 2. Simplifica la implementación de la resta.

Para codificar un número en complemento a dos se realizan los siguientes pasos:

1. Se toma el valor absoluto del número (sin signo)
2. Se convierte el número a binario
3. Si el número es positivo, se añaden ceros a la izquierda hasta completar el número de bits
4. Si el número es negativo, se invierten todos los bits y se lse suma 1 al resultado

```
    Para encontrar la representación en complemento a 2 de -25, debemos
encontrar la representación en binario de 25 y sumarle 1:
    25 = 11001
    -25 = 00111 + 1 = 01000
```

Para convertir un número en complemento a 2 a decimal se realizan los siguientes pasos:

1. Si el bit más significativo es 0, el número es positivo y se convierte a decimal normalmente
2. Si el bit más significativo es 1, el número es negativo y se invierten todos los bits y se le suma 1 al resultado

```
    Para encontrar la representación decimal de 11001 en complemento a 2,
debemos invertir los bits y sumarle 1:
    11001 -> 00110 + 1 = 00111 = 7
```

Al igual que sucede en los enteros sin signo, los enteros con signo también pueden verse afectados por el problema del desbordamiento.

```
    Supongamos qun dato de ocho dígitos y con un valor almacenado de 127,
si le sumamos 1, el resultado sería 128, que no se puede representar con un
dato de ocho bits.
    Si ahora interpretamos el resultado, como el bit más significativo es
un 1, consideramos que el número es negativo, por lo que el resultado sería
-128. Es decir, el resultado de sumar 1 a 127 sería -128. Lo que es
incorrecto.
```

## Codificación de números decimales

Los números decimales se pueden representar en binario utilizando la codificación de punto fijo o la codificación de punto flotante. La codificación de punto fijo es un sistema posicional en el que se utiliza un número fijo de bits para representar la parte entera y la parte decimal de un número. La codificación de punto flotante es un sistema posicional en el que se utiliza un número variable de bits para representar la parte entera y la parte decimal de un número.

```

```

```
    Para codificar 175.8376 en punto fijo, se separa la parte entera de la
parte decimal y se convierte cada parte a binario. En un formato de 8 bits
con 4 bits para la parte entera y 4 bits para la parte decimal:
    175 = 10101111
    0.8376 = 1101
    Sería 10101111.1101

    Para codificar 175.8376 en punto flotante, se debe seguir la siguiente
estructura:
    1. Signo: 0 para positivo y 1 para negativo
    2. Exponente: se suma 127 al exponente y se convierte a binario
    3. Mantisa: se convierte la parte entera y la parte decimal a binario y
se concatenan

    Para encontrar la representación en punto flotante de 175.8376, debemos
seguir los siguientes pasos:
    1. Signo: 0
    2. Exponente: 127 + 7 = 134 = 10000110
    3. Mantisa: 10101111.1101
    Por lo que la representación en punto flotante de 175.8376 es 0
10000110 10101111110100000000000
```

# Tipos, constantes y variables

## Tipos de datos

Cada lenguaje de programación tiene un conjunto de tipos de datos que se pueden utilizar para representar la información. Cada tipo de dato permite determinar los siguientes aspectos:

- El rango de valores que se pueden representar con el tipo de dato.
- El tamaño en memoria que ocupa el tipo de dato.
- Las operaciones que se pueden realizar con el tipo de dato.
- La forma en la que se almacenan en memoria los diferentes valores.

Los tipos de datos se dividen en dos categorías: tipos de datos primitivos y tipos de datos compuestos.

Los tipos de datos primitivos son aquellos que representan un único valor y se utilizan para representar los datos más básicos. Los tipos de datos primitivos se dividen en varios tipos:

- Tipos de datos enteros: son aquellos que representan números enteros. Los tipos de datos enteros se dividen en varios tipos: *char*, *int*, *short*, *long*.
- Tipos de datos reales: son aquellos que representan números reales. Los tipos de datos reales se dividen en varios tipos: *float*, *double*.
- Tipos de datos lógicos: son aquellos que representan valores lógicos. Los tipos de datos lógicos se dividen verdaderos o falsos: *bool*.
- Tipos de datos carácter: son aquellos que representan caracteres. El tipo de dato carácter es: *char*.

La diferencia entre los diferentes tipos radica, como en el caso de los enteros sin signo, en la cantidad de memoria que ocupan y en el rango de valores que pueden representar. Su tamaño no está definido en el

estándar y puede variar de una máquina a otra. Lo que sí se garantiza es que el tamaño de un tipo de dato no será menor que el de otro tipo de dato de menor tamaño.

| Tipo de dato | Tamaño en bytes | Rango de valores |
|---|---|---|
| char | 1 | -128 a 127 |
| int | 2 | -32768 a 32767 |
| short | 2 | -32768 a 32767 |
| long | 4 | -2147483648 a 2147483647 |
| float | 4 | 3.4E-38 a 3.4E+38 |
| double | 8 | 1.7E-308 a 1.7E+308 |
| bool | 1 | true o false |

Los tipos de datos compuestos son aquellos que representan un conjunto de valores y se utilizan para representar los datos más complejos. Los tipos de datos compuestos se dividen en varios tipos:

- Tipos de datos estructurados: son aquellos que representan un conjunto de valores de diferentes tipos. Los tipos de datos estructurados se dividen en varios tipos: *struct*, *union*.
- Tipos de datos enumerados: son aquellos que representan un conjunto de valores que se pueden enumerar. El tipo de dato enumerado es el *enum*.
- Tipos de datos puntero: son aquellos que representan la dirección de memoria de un valor. El tipo de dato puntero es el *pointer*.

## Constantes

Decimos que son constantes auqellos datos que no son susceptibles de ser modificados durante la ejecución de un programa. Las constantes se utilizan para representar valores que no cambian, como el valor de *pi* o el número de días de la semana.

Estas constantes se pueden encajar en varias categorías:

- Constantes enteras: son aquellas que representan números enteros. Las constantes enteras se definen con la palabra clave *const* y se pueden inicializar con un valor entero.
- Constantes reales: son aquellas que representan números reales. Las constantes reales se definen con la palabra clave *const* y se pueden inicializar con un valor real.
- Constantes lógicas: son aquellas que representan valores lógicos. Las constantes lógicas se definen con la palabra clave *const* y se pueden inicializar con un valor lógico.
- Constantes carácter: son aquellas que representan caracteres. Las constantes carácter se definen con la palabra clave *const* y se pueden inicializar con un valor carácter.

```
const int DIAS_SEMANA = 7;
const float PI = 3.14159;
const bool VERDADERO = true;
const char INICIAL = 'J';
```

Algunos caracteres especiales se representan con secuencias de escape, que son secuencias de caracteres que se utilizan para representar caracteres especiales. Las secuencias de escape se representan con una barra invertida seguida de un carácter. Algunas secuencias de escape comunes son:

| Secuencia de escape | Carácter especial |
|---|---|
| \n | Salto de línea |
| \t | Tabulación |
| \b | Retroceso |
| \r | Retorno de carro |
| ' | Comilla simple |
| " | Comilla doble |
| \ | Barra invertida |

## Variables

Las variables se utilizan para almacenar valores que pueden cambiar durante la ejecución de un programa. Las variables se utilizan para representar datos que pueden variar, como el número de intentos de un usuario o el saldo de una cuenta bancaria.

Se designan mediante un nombre al que se llama identificador. Este identificador puede ser cualquier secuencia de letras, dígitos y guiones bajos, pero debe empezar por una letra. Además, no se pueden utilizar palabras reservadas del lenguaje de programación.

Es conveniente que el identificador de una variable sea descriptivo, para que se pueda entender fácilmente qué representa la variable. Además, es importante que el identificador de una variable sea único, para evitar confusiones.

```
int edad = 25;
float altura = 1.75;
bool casado = false;
char inicial = 'J';
```

Las operaciones básicas que pueden realizarse con una variable son declararla, utilizar el valor almacenado, o asignarle un nuevo valor.

Hay lenguajes en los que es necesario declarar una variable antes de poder utilizarla. La declaración de una variable consiste en indicar el tipo de dato que va a almacenar la variable y el nombre de la variable.

```
int edad;
float altura;
bool casado;
char inicial;
```

En otros lenguajes, como Python, no es necesario declarar una variable antes de poder utilizarla. En estos lenguajes, la variable se declara automáticamente cuando se le asigna un valor.

```
edad = 25
altura = 1.75
casado = False
inicial = 'J'
```

Hay que tener en cuenta que si una varibale se declara pero no se le asigna un valor, la variable contendrá un valor basura, es decir, un valor que no tiene sentido. Por ello, es importante inicializar las variables antes de utilizarlas.

Para realizar la asignación de un valor a una variable se utiliza el operador de asignación, que es un signo igual (=). El operador de asignación se utiliza para asignar un valor a una variable. El valor que se asigna a una variable puede ser una constante, una expresión o el valor de otra variable.

El tiempo de vida de una varibale determina cuándo se crea y cuándo se destruye. Las variables se crean cuando se declara y se destruyen cuando se sale del ámbito en el que se declararon. El ámbito de una variable es la parte del programa en la que se puede utilizar la variable. Esto trae consigo una serie de consecuencias:

- La primera, y más importante, es que el valor de una variable en sucesivas ejecuciones del programa puede ser diferente.
- La segunda es que el valor de inicialización especificado en la declaración de la variable se aplica en casa una de las ejecuciones del bloque de código en el que se declara la variable.
- La tercer es que las modificaciones que hagamos sobre la variable dentro de un bloque de código no afectarán a la variable fuera de ese bloque.

A estas variables se les conoce como variables automáticas o locales y son las que habitualmente se utilizarán en los programas.

Existe un procedimiento para evitar que las variables locales se destruyan al salir del ámbito en el que se declararon, y es declararlas como variables globales. Las variables globales son aquellas que se declaran fuera de cualquier función y se pueden utilizar en cualquier parte del programa. Esta especificación tiene un triple efecto sobre la variable:

- Hace que las variables se creen al inicio de la ejecución del programam, y no al declararla dentro de un bloque como las variables locales.
- Hace que las variables se destruyan al final de la ejecución del programa, y no al salir del bloque en el que se declararon.
- Hace que se inicialicen al comienzo de la ejecución del programa: al valor indicado en la declaración de la variable, o a cero si no se indica nada. Sólo pueden utilizarse expresiones constantes para la inicialización de las variables globales.

```
static int contador = 0;
```

Desde el punto de vista de la detección y corrección de errores, es muy conveniente que la visibilidad de las variables sea lo más reducida posible: podemos controlar el valor de dicha variable centrándonos en el contexto en el que es visible. Por esta razón es muy desaconsejable la utilización de variables globales en cualquier programa.

## Expresiones y operadores

En cualquier programa es necesario procesaro las datos de entrada para generar datos de salida. El elemento básico que tenemos para procesar los datos son las expresiones. Una expresión es una combinación de valores, variables y operadores que se evalúa para obtener un resultado. Las expresiones se utilizan para realizar cálculos, comparaciones y asignaciones.

Los paréntesis se utilizan para agrupar los elementos de una expresión y determinar el orden de evaluación de los operadores.

```
    Para evaluar la expresión 2 + 3 * 4, primero se evalúa la
multiplicación y luego la suma:
    2 + 3 * 4 = 2 + 12 = 14

    Para evaluar la expresión (2 + 3) * 4, primero se evalúa la suma y
luego la multiplicación:
    (2 + 3) * 4 = 5 * 4 = 20
```

### Operadores aritméticos

Toman como valores operandos numéricos y producen como resultado un valor numérico. Los operadores aritméticos en general son:

- Operadores unarios: son aquellos que toman un único operando. Los operadores unarios son el operador de incremento y el operador de decremento.
    - + no tiene efecto aparente
    - - cambia el signo del operando
- Operadores binarios: son aquellos que toman dos operandos. Los operadores binarios son el operador de suma, el operador de resta, el operador de multiplicación, el operador de división y el operador de módulo.
    - + suma dos operandos
    - - resta dos operandos
    - * multiplica dos operandos
    - / divide dos operandos. Devuelve el cociente de la división. Si los operandos son enteros, el resultado es un entero. Si los operandos son reales, el resultado es un real.
    - % calcula el resto de la división entera de dos operandos. Devuelve el resto de la división. Ambos operandos deben ser enteros.

Estos operadores aritméticos siguen las reglas de prioridad de las operaciones aritméticas. Si dos operadores tienen la misma prioridad, se evalúan de izquierda a derecha.

No hay que olvidar que los valores de tipo *char* se pueden tratar como enteros, por lo que se pueden utilizar en operaciones aritméticas. Sin embargo, hay dos oepraciones que resultan de especial interés:

- La suma de un valor de tipo *char* con un valor numérico, que produce un valor de tipo *char* y representa el carácter cuyo código se diferencia del primero tantas unidades como hayamos sumado o restado.

```
    char c = 'A';
    c = c + 1; // c = 'B'
```

- La resta de dos valores de tipo *char*, que produce un valor numérico y representa la diferencia entre los códigos de los dos caracteres. Esto es bastante útil para convertir un carácter en su posición en la tabla ASCII.

```
    char c = 'A';
    int n = c - 'A'; // n = 0
```

## Operadores relacionales

Toman como operando valores numéricos y producen como resultado un valor entero. Los operadores relacionales son todos binarios:

- == Devuelve verdadero si los dos operandos son iguales.
- != Devuelve verdadero si los dos operandos son diferentes.
- < Devuelve verdadero si el primer operando es menor que el segundo.
- <= Devuelve verdadero si el primer operando es menor o igual que el segundo.
- > Devuelve verdadero si el primer operando es mayor que el segundo.
- >= Devuelve verdadero si el primer operando es mayor o igual que el segundo.

```
    Si i,j,k son variables enteras, con valores 3, 4 y 5 respectivamente,
entonces:
    i == j es falso
    i != j es verdadero
    i < j es verdadero
    i <= j es verdadero
    i > j es falso
    i >= j es falso
```

Conviene recordar que las variables de tipo *char* también son tipos numéricos, por lo que se pueden utilizar en operaciones relacionales. En este caso, se comparan los códigos ASCII de los caracteres. Para el código ASCII, y en general para cualquier sistema de codificación de caracteres, se pueden establecer las siguientes relaciones:

- '0' < '1' < ... < '9'
- 'A' < 'B' < ... < 'Z'
- 'a' < 'b' < ... < 'z'

Cuando se utilizan los operadores de relación de igualdad o desigualdad con operandos con decimales, hay que tener cuidado con la precisión de estos, puesto que puede ser que el resultado esperado no sea exactamente el obtenido.

```
    Por ejemplo, si se compara 0.1 + 0.2 == 0.3, el resultado será falso,
ya que el resultado de la suma no es exactamente 0.3, sino un valor muy
próximo.
```

## Operadores lógicos

Tomando operandos numéricos, producen como resultado un valor lógico. Los operadores lógicos son todos binarios:

- *&&* Representa el Y de la lógica proposicional; es decir, devuelve verdadero si ambos operandos son verdaderos.
- // Representa el O de la lógica proposicional; es decir, devuelve verdadero si al menos uno de los operandos es verdadero.
- *!* Representa el NO de la lógica proposional; es decir, si el operando es verdadero, devuelve falso, y si el operando es falso, devuelve verdadero.

```
    Si i,j,k son variables enteras, con valores 3, 4 y 5 respectivamente,
entonces:
    i < j && j < k es verdadero
    i < j || j > k es verdadero
    !(i < j) es falso
```

## Operadores de manejo de bits

Sólo admiten operandos de tipo entero, y se aplican bit a bit. Los operadores de manejo de bits son:

- ~ Operador de negación bit a bit. Invierte los bits de un operando.
- *&* Operador de conjunción bit a bit. Realiza la operación lógica AND bit a bit.
- / Operador de disyunción bit a bit. Realiza la operación lógica OR bit a bit.
- ^ Operador de disyunción exclusiva bit a bit. Realiza la operación lógica XOR bit a bit.
- << Operador de desplazamiento a la izquierda. Desplaza los bits de un operando a la izquierda un número de posiciones determinado.
- >> Operador de desplazamiento a la derecha. Desplaza los bits de un operando a la derecha un número de posiciones determinado.

Lo mejor para utilizar correctamente estos operadores es obtener la representación binaria de los operandos y aplicar la operación bit a bit, y luego volver a convertir el resultado a decimal.

```
    ~126 = ~0111 1110 = 100 0001 = -127
    126 & 3 = 0111 1110 & 0000 0011 = 0000 0010 = 2
    126 | 3 = 0111 1110 | 0000 0011 = 0111 1111 = 127
    126 ^ 3 = 0111 1110 ^ 0000 0011 = 0111 1101 = 125
```

```
126 << 2 = 0111 1110 << 2 = 1111 1000 = 504
126 >> 2 = 0111 1110 >> 2 = 0001 1111 = 31
```

## Operadores de asignación

El operador de asignación admite dos operandos, el de la izquierda debe ser un identificador de variable, mientras que el de la derecha puede ser una constante, una variable o una expresión. El operador de asignación se representa con el símobolo =.

Una particularidad de este operador es que se evalúa de derecha a izquierda, por lo que las asignaciones de más a la derecha se realizan antes que las de más a la izquierda.

Cuando se realiza la asignación, el resultado de la expresión se convierte al tipo del operando de la izquierda. Si el tipo del operando de la izquierda es de mayor rango que el de la derecha, no hay ningún problema. Sin embargo, si es al revés sí pueden producirse alteraciones en los valores de los datos:

- Si un tipo entero se asigna a otro tipo entreo de menor rango, se produce un desbordamiento debido a que se copian los bits menos significativos del operando de la derecha en el operando de la izquierda.
- Si un valor flotante se asigna a una variable de tipo entero, se trunca el valor flotante, es decir, se pierde la parte decimal del valor flotante.
- Si un valor flotante se asigna a una variable flotante, se produce un redondeo que será tanto mayor cuanto menor sea la precisión del tipo de la variable.

```
int i = 3;
float f = 3.14;
i = f; // i = 3
f = i; // f = 3.0
```

Es importante destacar que las asignaciones que puedad dar lugar a pérdida de información (por redondeo o truncado) no son ilegales, y por lo tanto el compilador nos va a dejar realizarlas; pero deben realizarse con muchísimo cuidado.

Existen operadores que permiten simultáneamente realizar una operación con una variable, y asignar el resultado a la misma variable. Estos operadores son los operadores de asignación compacta. Los operadores de asignación compacta son:

- *+=* Operador de asignación de suma. Suma el operando de la derecha al operando de la izquierda y asigna el resultado al operando de la izquierda.
- *-=* Operador de asignación de resta. Resta el operando de la derecha al operando de la izquierda y asigna el resultado al operando de la izquierda.
- **=* Operador de asignación de multiplicación. Multiplica el operando de la derecha por el operando de la izquierda y asigna el resultado al operando de la izquierda.
- */* Operador de asignación de división. Divide el operando de la izquierda por el operando de la derecha y asigna el resultado al operando de la izquierda.
- *%=* Operador de asignación de módulo. Calcula el resto de la división entera del operando de la izquierda por el operando de la derecha y asigna el resultado al operando de la izquierda.

```
int i = 3;
i += 2; // i = 5
i -= 1; // i = 4
i *= 3; // i = 12
i /= 4; // i = 3
i %= 2; // i = 1
```

Es importante no perder la vista que en la asignación compacta, primero se evalúa la operación y luego se asigna el resultado a la variable. Por lo tanto, si se realiza una operación con una variable y se asigna el resultado a la misma variable, el resultado puede ser diferente al esperado.

```
La expresión x = x * y / z; es diferente de x *= y / z;
Ya que si desglosamos la primera expresión, primero se evalúa x * y, y luego se divide el resultado por z. En cambio, en la segunda expresión, primero se evalúa x * y / z, y luego se asigna el resultado a x.
```

Ya hemos visto que cuando se trata de una asignación, se realiza una conversión de tipos, de forma que el resultado de evaluar la expresión se convierte al tipo de la variable antes de realizar la asignación. Hay situaciones en las que es necesario realizar un cambio de tipo sin necesidad de realizar una asignación. Para ello, se utilizan los operadores de conversión de tipo. Suele aparecer en la literatura como *casting*.

```
int i = 3;
float f = 3.14;
i = (int)f; // i = 3
f = (float)i; // f = 3.0
```

## Operadores de condición

Los operadores de condición son operadores ternarios, es decir, que toman tres operandos. Los operadores de condición son:

- *?* Operador de condición. Evalúa el operando de la izquierda y, si es verdadero, devuelve el operando de la derecha; si es falso, devuelve el operando de la derecha.
- *:* Operador de condición. Se utiliza para separar los operandos del operador de condición.

```
int i = 3;
int j = 4;
int k = i < j ? i : j; // k = 3
```

## Reglas de prioridad

Los operadores tienen reglas de precedencia (o prioridad) y asociatividad que determinan exactamente la forma de evaluar las expresiones.

La precedencia de los operadores determina el orden en el que se evalúan los operadores en una expresión. Los operadores con mayor precedencia se evalúan antes que los operadores con menor precedencia. Si dos operadores tienen la misma precedencia, se evalúan de izquierda a derecha.

La asociatividad de los operadores determina el orden en el que se evalúan los operadores con la misma precedencia. Los operadores con asociatividad izquierda se evalúan de izquierda a derecha, mientras que los operadores con asociatividad derecha se evalúan de derecha a izquierda.

| Prioridad | Operador | Asociatividad | Descripción |
|---|---|---|---|
| 1 | () | Izquierda | Paréntesis |
| 2 | ++, -- | Izquierda | Operadores unarios postfijos |
| 3 | () | Izquierda | Operadores de llamadas a funciones |
| 4 | [], ., -> | Izquierda | Operadores de tablas y estructuras |
| 5 | +, -, !, ~, ++, -- | Derecha | Operadores unarios prefijos |
| 6 | *, /, % | Izquierda | Operadores de multiplicación y división |
| 7 | +, - | Izquierda | Operadores de suma y resta |
| 8 | <<, >> | Izquierda | Operadores de desplazamiento |
| 9 | <, <=, >, >= | Izquierda | Operadores de relación |
| 10 | ==, != | Izquierda | Operadores de igualdad |
| 11 | & | Izquierda | Operadores de conjunción |
| 12 | \| | Izquierda | Operadores de disyunción |
| 13 | ^ | Izquierda | Operadores de disyunción exclusiva |
| 14 | && | Izquierda | Operadores de conjunción lógica |
| 15 | \|\| | Izquierda | Operadores de disyunción lógica |
| 16 | ?: | Derecha | Operadores de condición |
| 17 | =, +=, -=, *=, /=, %= | Derecha | Operadores de asignación |

## Estructuras de control

Hasta ahora las instrucciones seguían una estructura secuencial: se ejecutaba una sentencia, tras la finalización se ejecutaba la siguiente, y así sucesivamente hasta alcanzar el final del programa. Sin embargo, en la mayoría de los programas es necesario tomar decisiones o repetir un conjunto de instrucciones. Para ello, se utilizan las estructuras de control.

Las estructuras de control se dividen en tres categorías: estructuras secuenciales, estructuras condicionales y estructuras repetitivas. Se debe evitar el uso de cualquier otra estructura ya que conduce a un código no estructurado. Esto no supone ninguna limitación a la hora de escribir un programa, ya que cualquier algoritmo se puede expresar utilizando únicamente estas tres estructuras.

En concreto se deben evitar las instrucciones de salto incondicional, como *goto*, ya que dificultan la comprensión del código y la detección de errores. Además, las instrucciones de salto incondicional pueden provocar la ejecución de instrucciones fuera de su contexto, lo que puede llevar a resultados inesperados. También se deben evitar las sentencias de salto condicional, como *break* o *continue*, ya que dificultan la comprensión del código y la detección de errores.

### Estructuras secuenciales

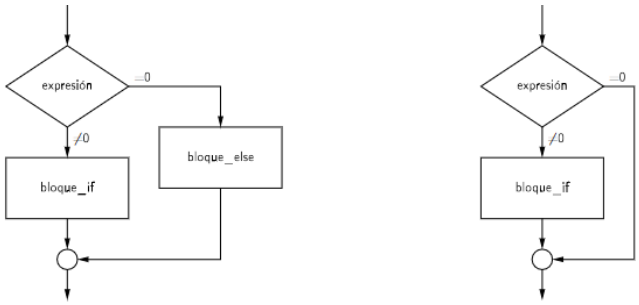
Las sentencias se ejecutan en el orden en el que aparecen en el programa. La ejecución de una sentencia no depende de la ejecución de otra sentencia. Las estructuras secuenciales se utilizan para ejecutar un conjunto de instrucciones en un orden determinado.

```
int i = 3;
int j = 4;
int k = i + j;
```

### Estructuras condicionales

También conocidas como estructuras de selección, se utilizan para tomar decisiones en un programa. Las estructuras condicionales se dividen en dos tipos: estructuras condicionales simples y estructuras condicionales compuestas.

Las estructuras condicionales simples se utilizan para ejecutar un conjunto de instrucciones si se cumple una condición. Las estructuras condicionales simples se dividen en dos tipos: estructuras condicionales simples con una sola alternativa y estructuras condicionales simples con dos alternativas.

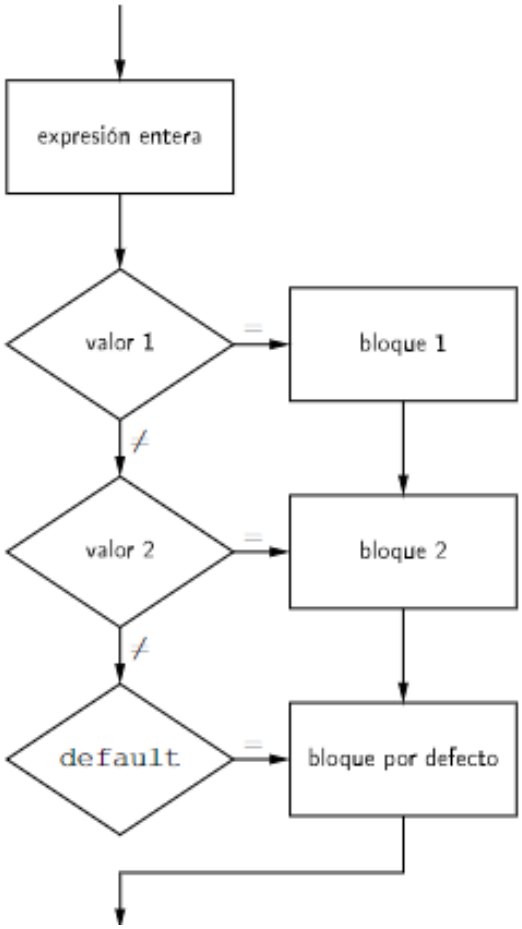



```
int i = 3;
int j = 4;
if (i < j) {
    System.out.println("i es menor que j");
}
```

Estas sentencias se pueden anidar para realizar una selección múltiple. En este caso, se evalúan las condiciones en orden y se ejecuta el bloque de instrucciones correspondiente a la primera condición que se cumpla.

```java
// Ejemplo de cálculo de año bisiesto
int año = 2020;
if (año % 4 == 0) {
    if (año % 100 != 0 || año % 400 == 0) {
        System.out.println("El año es bisiesto");
    } else {
        System.out.println("El año no es bisiesto");
    }
} else {
    System.out.println("El año no es bisiesto");
}
```

Las estructuras condicionales compuestas se utilizan para ejecutar un conjunto de instrucciones si se cumple una condición y otro conjunto de instrucciones si no se cumple la condición. Las estructuras condicionales compuestas se dividen en dos tipos: estructuras condicionales compuestas con una sola alternativa y estructuras condicionales compuestas con dos alternativas.




El funcionamiento de esta estructura es el siguiente:

1. Se evalúa la expresión que acompaña a la sentencia *switch*. Esta expresión debe ser de tipo entero, carácter o enumerado.
2. Se compara el valor de la expresión con el valor de cada una de las etiquetas *case*. Si se encuentra una coincidencia, se ejecutan las instrucciones asociadas a esa etiqueta. Estos valores deberán ser siempre expresiones constantes enteras.

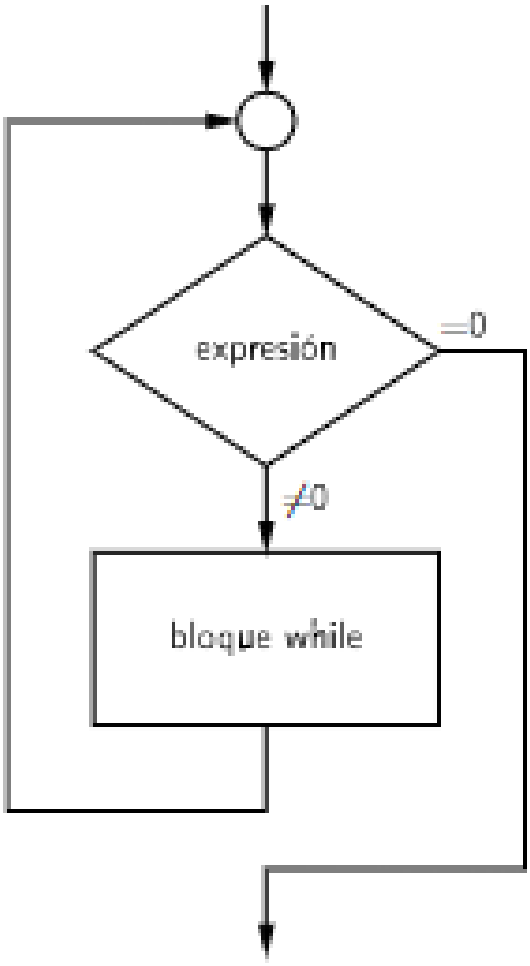
Existe una etiqueta especial, *default*, que se ejecuta si no se encuentra ninguna coincidencia. La etiqueta *default* es opcional y se puede colocar en cualquier posición dentro de la sentencia *switch*.

```
// Programa que lea un número de mes e indique cuántos días tiene
int mes = 2;
switch (mes) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        System.out.println("El mes tiene 31 días");
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        System.out.println("El mes tiene 30 días");
        break;
    case 2:
        System.out.println("El mes tiene 28 o 29 días");
        break;
    default:
        System.out.println("Mes incorrecto");
}
```

## Estructuras repetitivas

En este tipo de estructuras, se repite un conjunto de instrucciones en función de una condición. La principal diferencia entre las diferentes estructuras repetitivas consiste en qué punto se realiza la comprobación de la condición.

La sentencia *while* se utiliza para repetir un conjunto de instrucciones mientras se cumpla una condición. La condición se evalúa antes de ejecutar el bloque de instrucciones.




El funcionamiento es el siguiente:

1. Se evalúa la expresión que acompaña a la sentencia *while*. Si la expresión es verdadera, se ejecutan las instrucciones del bloque.
2. Se vuelve a evaluar la expresión. Si la expresión es verdadera, se ejecutan las instrucciones del bloque. Este proceso se repite hasta que la expresión sea falsa.

Es importante destacar que puede que el bloque que acompaña a la sentencia *while* no se ejecute nunca si la condición es falsa desde el principio. Además alguno de los valores que determinan la condición de salida del bucle deben cambiar en algún momento para que el bucle no sea infinito.

En la condición, es conveniente utilizar los operadores de rango (>, <, >=, <=) en lugar de los operadores de igualdad (==, !=) para evitar problemas de precisión.

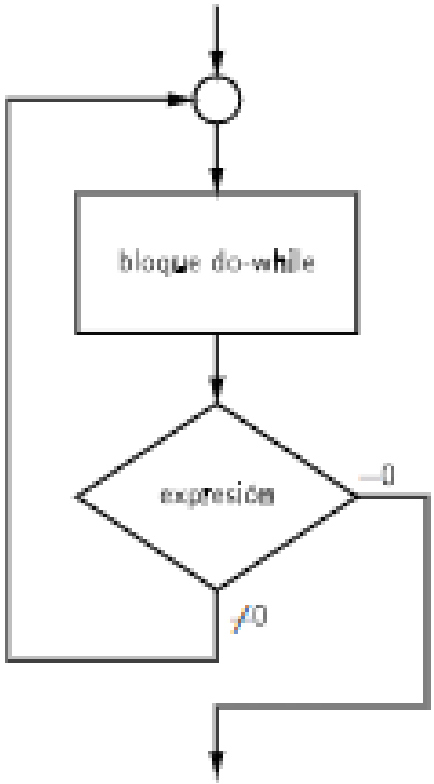
```
// Programa que copia la entrada estándar en la salida estándar
char c;
while ((c = System.in.read()) != -1) {
    System.out.write(c);
}
```

En las estructuras repetitivas en general, y como caso en particular en la sentencia *while*, es de especial importancia comprobar que los valores extermos de la condición de salida del bucle son correctos. Para ello

es de gran utilizad repasar mentalmente el bucle con los valores extremos y comprobar que el bucle se comporta como se espera.

```
Veamos un ejemplo en el que se pretende imprimir los números del 1 al 10.
Si la condición de salida del bucle es i < 10, el bucle se ejecutará 9
veces, ya que el valor 10 no se imprimirá. Por el contrario, si la
condición de salida del bucle es i <= 10, el bucle se ejecutará 10 veces, y
se imprimirán los números del 1 al 10.
```

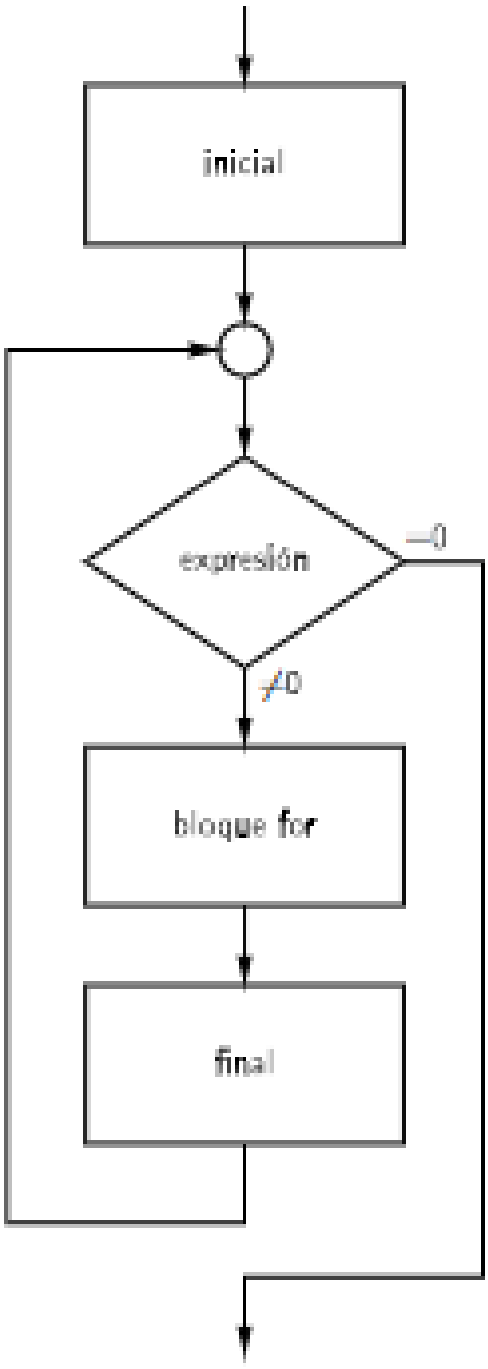
La sentencia *do-while* se utiliza para repetir un conjunto de instrucciones mientras se cumpla una condición. La condición se evalúa después de ejecutar el bloque de instrucciones.




A diferencia de la sentencia *while*, la sentencia *do-while* garantiza que el bloque de instrucciones se ejecuta al menos una vez. Esto es útil cuando se quiere ejecutar un bloque de instrucciones al menos una vez, independientemente de la condición.

```java
// Programa que imprime los números del 1 al 10
int i = 1;
do {
    System.out.println(i);
    i++;
} while (i <= 10);
```

La sentencia *for* se utiliza para repetir un conjunto de instrucciones un número determinado de veces. La sentencia *for* se compone de tres partes: la inicialización, la condición y la actualización.




El funcionamiento es el siguiente:

1. Se ejecuta la inicialización.
2. Se evalúa la condición. Si la condición es verdadera, se ejecutan las instrucciones del bloque.
3. Se ejecuta la actualización.
4. Se vuelve a evaluar la condición. Si la condición es verdadera, se ejecutan las instrucciones del bloque. Este proceso se repite hasta que la condición sea falsa.

La sentencia *for* es especialmente útil cuando se conoce el número de iteraciones que se van a realizar, ya que permite agrupar la inicialización, la condición y la actualización en un solo lugar.

En la sentencia *for*, la inicialización y la actualización son opcionales. Si no se especifican, se consideran vacías. La condición también es opcional. Si no se especifica, se considera verdadera.

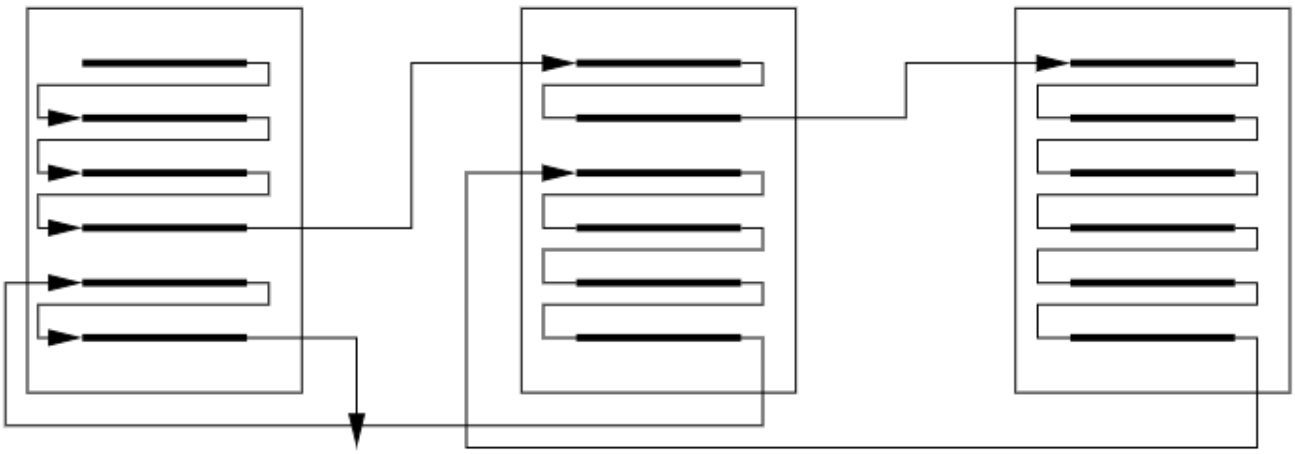
```java
// Programa que imprime los números del 1 al 10
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

## Funciones

Para resolver un programa es fundamental descomponerlo en problemas más pequeños, puesto que la resolución de cada uno de estos sería más fácil. También hay situaciones en las que una tarea tiene que realizarse más de una vez dentro de un programa; en estos casos, interesa escribir el código una sola vez, y poder ejecutarlo tantas veces como sea preciso.

Tanto una cosa como la otra se consiguen mediante el uso de funciones. Una función es un conjunto de instrucciones que se agrupan bajo un nombre y que se pueden invocar desde cualquier parte del programa. Las funciones se utilizan para dividir un programa en tareas más pequeñas y para reutilizar el código.

Un aspecto importante a cosinderar es que el hilo de ejecución no se pierde. Cuando desde un punto del programa llamamos a una función, la ejecuciñon sigue en el punto en que hemos definido la función; cuando esta última termina, la ejecución vuelve al punte dese el que se llamó a la función, continuando la ejecución normal. Este mecanismo puede repetise de forma reiterada, de forma que una función puede llamar a otra función, que a su vez puede llamar a otra función, y así sucesivamente... pero siempre manteniendo que tras la finalización de la ejecución de una función, el hilo de ejcución vuelve a la función que hizo la llamada.



Esto significa que todas las funciones están definidas al mismo nivel, en el más exterior dentro de un fichero, y por lo tanto su visibilidad y el alcance son los mismoq ue los de las variables definidas a ese nivel.

Un programa en general, es un conjunto de funciones que se llaman entre sí. En la mayoría de los lenguajes de programación, un programa tiene una función principal, que es la función que se ejecuta al inicio de la ejecución del programa. En algunos lenguajes como C y Java, la función principal se llama *main*. Un programa finalizará cuando se temrine la ejecución de la función *main*.

### Definición de funciones

Para poder utilizar funciones dentro de un programa es imprescindible que antes se especifique la tarea arealizar. Para ello se utiliza la definición de funciones.

La definición de una función consiste en especificar:

- Cómo se llama la función: el *identificador*.
- Los valores, y el tipo de dato al que pertenecen, que necesita para su ejecución: los *parámetros*.
- El tipo del valor que devuelve (si es que devuelve algún valor): el *tipo*.
- Qué es lo que realmente hace: el *cuerpo*.

```
int suma(int a, int b) {
    return a + b;
}
```

El identificador de una función permite identificar de forma única la función a la que estamos haciendo referencia dentro del programa. Para formar el identificador de una función se utilizan los mismos criterior que para formar los identificadores de las variables.

En un programa no puede haber dos funciones con el mismo identificador, a no ser que las funciones se encuentren en ficheros diferentes y tengan restrinciones de visibilidad. En este caso, se dice que las funciones tienen el mismo nombre pero son funciones diferentes. En algunos lenguajes esto se permite mediante la utilización de la cláusula *static*.

Cuando definimos una función para reutilizarla, es muy importante tener un mecanismo que nos permita modificar los valores utilziados por la función para sus cálculos. Esto se logra con los parámetros. Cada uno de los valores que necesita la función para realizar su tarea se denomina parámetro. Los parámetros se utilizan para pasar valores a la función. Los parámetros se definen en la cabecera de la función, entre paréntesis, separados por comas. Los parámetros se componen de un tipo y un identificador.

El cuerpo de una función lo constituye el bloque en el que se incluyen las sentencias necesarias para realizar el objetivo encomendado. Dependiendo del lengauje de programación, el cuerpo de la función puede estar formado por una o varias sentencias. En el caso de que la función devuelva un valor, la última sentencia del cuerpo de la función debe ser una sentencia de retorno.

El tipo de una función es el tipo del valor que devuelve la función. Si una función no devuelve ningún valor, se dice que la función es de tipo *void*. En este caso, la función no debe contener una sentencia de retorno. Si una función devuelve un valor, la función debe contener una sentencia de retorno. La sentencia de retorno se utiliza para devolver un valor desde la función al punto en el que se llamó a la función.

## Declaración de funciones

Para poder utilizar una función en un programa, es necesario que la función esté definida antes de ser utilizada. Esto significa que la definición de la función debe aparecer antes de la llamada a la función. En algunos lenguajes de programación, como C, es necesario que la definición de la función aparezca antes de la llamada a la función. En otros lenguajes de programación, como Java, no es necesario que la definición de la función aparezca antes de la llamada a la función.

A la declaración de una función se le llama prototipo de la función. El prototipo de una función se compone del tipo de la función, el identificador de la función y los parámetros de la función. El prototipo de una función se utiliza para indicar al compilador que la función existe y que se va a utilizar en el programa.

```
int suma(int a, int b);
```

## Utilización de funciones

Para utilizar una función (o hacer una llamada), escribiremos el identificador de la función, y colocaremos entre paréntesis los valores que deseamos para los parámetros. En general, podremos utilizar como prarámetro cualquier expresión del mismo tipo que el parámetro formal correspondiente.

```
int a = 3;
int b = 4;
int c = suma(a, b);
```

Cuando se realiza una llamada a una función, antes de comenzar su ejecución se evalúan las expresiones que forman los parámetros reales, y los valores obtenidos se asignan a las variables que representan los parámetros formales. Sólo entonces comienza la ejecución de la función.

La asignación entre parámetros reales y formales se relaiza siempre según el orden en el que aparecen tanto en la llamada como en la definición: el primer parámetro real se asigna al primer parámetro formal, el segundo parámetro real se asigna al segundo parámetro formal, y así sucesivamente. A esto se le llama correspondencia posicional.

Por tanto, es fundamental que el número de parámetros reales coincida con el de parámetros formales, debiendo también coincidir en el orden y en el tipo de ambas clases de parámetros. En algunos lengaujes de programación, es posible que una función tenga un número variable de parámetros. En este caso, se dice que la función tiene parámetros variables o está sobrecargada. Pero esto no es lo habitual.

```
int suma(int a, int b) {
    return a + b;
}
int a = 3;
int b = 4;
int c = suma(a, b);
```

Es importante destacar que los parámetros formales actúan como variables locales, los cambios que se realicen en los parámetros formales no afectan a los parámetros reales. Por tanto, los parámetros reales actúan como constantes, y no se pueden modificar dentro de la función. Además, puesto que vamos a utilizar los parámetros reales para realizar una asignación sobre los parámetros formales, sólo podremos utilizr como parámetros aquellos elementos que puedan aparecer en el lado derecho de una asignación.

## Recursividad

Un caso particular de llamada a una función es cuando una función se llama a sí misma. A este mecanismo se le llama recursividad. La recursividad es un concepto muy importante en programación, ya que permite resolver problemas de forma más sencilla y elegante.

Esta técnica se aplica problemas que pueden definirse de modo natural de forma recusriva. No todos los problemas admiten soluciones recursivas, y si la admiten muchas veces no son inmediatas. Sí podemos asegurar que toda solución recusriva tiene una solución iterativa equivalente aunque no siempre es inmediato obtenerla.

En las definiciones de funciones recursivas es imprescindible que exista un caso base, es decir, un caso en el que la función no se llame a sí misma. Si no se cumple esta condición, la función se llamará a sí misma de forma indefinida, lo que provocará un error en tiempo de ejecución.

Aunque el código de una función que utiliza técnicas recursivas pueda ser más corto, habitualmente consume más memoria durante la ejecución, y puede llegar a ser más lento en la ejecución que una solución iterativa, debido a que en cada llamada que la función se hace a sí misma es necesario copiar valores en variables y estas ocupan espacio en memoria, tanto más cuanto mayor sea el número de llamadas.

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

// Si se llama a factorial(5), se realizarán las siguientes llamadas:
// factorial(5) = 5 * factorial(4)
// factorial(4) = 4 * factorial(3)
// factorial(3) = 3 * factorial(2)
// factorial(2) = 2 * factorial(1)
// factorial(1) = 1 * factorial(0)
// factorial(0) = 1 (caso base)
```

## Compilación separada

Cuando un programa es muy grade y tiene muchas funciones, suele ser conveniente dividir el programa en varios ficheros. Cada fichero contendrá un conjunto de funciones relacionadas entre sí. A esta técnica se le llama compilación separada.

La compilación separada es un concepto distinto al de la modularidad, aunque muy relacionado: la compilación separada implica necesariamente modularidad, pero un programa modular puede estar compuesto por un único fichero, y por lo tanto no le es de aplicación la compilación separada.

La división del programa en varios ficheros tiene dos considerables ventajas:

- El programa queda más estructurado y organizado.
- Si se necesita modificar un fichero, sólo se compilaría el modificado, para posteriormente enlazarlo con el resto de ficheros.

Cada fichero en los que se descompone un porgrmaa fuente conforma lo que se denomina *unidad de compilación*. Cada unidad de compilación se compila de forma independiente, generando un fichero objeto. Posteriormente, los ficheros objeto se enlazan para formar el programa ejecutable.

# Tablas

Hasta ahora, para representar una magnitud del mundo real en un programa utilizamos las variables. ¿Qué ocurre si necesitamos representar un conjunto de magnitudes? En este caso, utilizaremos las tablas.

Una tabla es un conjunto finito y ordenado de elementos del mismo tipo agrupados bajo le mismo nombre. Su principal caracterísitca es el modo de almacenamiento en memoria: se realiza en posiciones consecutivas. Una característica diferenciadora de las tablas con respecto a otros tipos de datos es que no soportan el operador de asignación ne la mayoría de lenguajes; para asignar una tabla a otra será necesario asignar cada uno de los elementos de la tabla.

## Declaración de tablas

La declaración de una tabla se hace indicando el tipo de todos los elementos de la tabla, seguido del nombre de la variable, seguido del número de elementos encerrados entre corchetes.

Se puede aprovechar la declaración de una tabla para inicializarla. Para ello, se encierran entre llaves los valores de los elementos de la tabla, separados por comas.

```
int tabla[5] = {1, 2, 3, 4, 5};
```

La asignación se realiza en orden ascendente del índice, de forma que si no hay suficientes valores para inicializar la tabla, se asignarán valores a los índices más bajos; los elementos para los que no haya valores se inicializarán a 0.

Un caso especial es el de las tablas de caracteres, que se utilizan para almacenar cadenas de caracteres. En este caso, la inicialización se realiza con una cadena de caracteres entre comillas dobles.

```
char cadena[] = "Hola";
```

En algunos lengiajes es necesario incluir un carácter especial al final de la cadena, que se llama carácter nulo, y que se representa con el carácter *\0*. Este carácter se utiliza para indicar el final de la cadena.

Es importante especificar que cuando se utilizan funciones de escaneo y de impresión de cadenas, es necesario que la cadena esté terminada con el carácter nulo. Si no se incluye este carácter, la función de escaneo o de impresión de cadenas no funcionará correctamente. Es por ello, que al leer hay que tener en cuenta que el carácter nulo ocupa una posición en la tabla.

## Acceso a elementos de una tabla

Para acceder a un elemento de una tabla, se utiliza el nombre de la tabla, y, encerrado entre corchetes, el índice del elemento al que queremos acceder, sabien que el primer elemento de la tabla tiene índice 0.

Este índice siempre debe ser un valor entero, y puede ser una constante, una variable o una expresión.

```
    int tabla[5] = {1, 2, 3, 4, 5};
    int a = tabla[2]; // a = 3
```

Puesto que el primer elemento de la tabla tiene índice 0, el último elemento de la tabla tiene índice *n - 1*, siendo *n* el número de elementos de la tabla.

### Recorrido de tablas

Para recorrer los elementos de una tabla, la estructura repetitiva for es la ideal. Nos permite realizar una asignación inicial (que el valor del índice sea cero), una operación tras cada iteración (incrementar el índice una unidad), y una comparación de control (que no se haya superado el límite de la tabla).

```
    // Veamos un ejemplo que calcula la media de las calificaciones de diez
alumnos, que se introducirán por teclado.

    int calificaciones[10];
    float media = 0;
    for (int i = 0; i < 10; i++) {
        System.out.println("Introduce la calificación del alumno " + (i +
1));
        calificaciones[i] = System.in.read();
        media += calificaciones[i];
    }
    media /= 10;
    System.out.println("La media de las calificaciones es " + media);

    // Obsérvese que la función System.in.read() devuelve un valor entero,
por lo que la calificación se almacena en un entero y que en la condición
de control del bucle se compara con 10, que es el número de elementos de la
tabla, y no i <= 10, puesto que el índice del último elemento de la tabla
es 9.
```

### Utilización de constantes simbólicas

En ocasiones, la dimensión de una tabla es un valor que se utiliza más de una vez: tenemos que declarar la tabla, inicializarla, recorrerla... En estos casos, es conveniente utilizar una constante simbólica, que es un identificador que representa un valor constante.

También es importante que el nombre de la constante simbólica sea significativo, de forma que se pueda identificar fácilmente el valor que representa en el programa.

```
    // El siguiente programa calcula las calificaciones ponderadas de un
curso.

    int calificaciones[TAM_CALIFICACIONES];
    float ponderaciones[TAM_CALIFICACIONES] = {0.1, 0.2, 0.3, 0.4};
    float media = 0;
```

```
    for (int i = 0; i < TAM_CALIFICACIONES; i++) {
        System.out.println("Introduce la calificación del alumno " + (i +
1));
        calificaciones[i] = System.in.read();
        media += calificaciones[i] * ponderaciones[i];
    }
```

## Tablas multidimensionales

Hasta ahora, hemos visto tablas de una dimensión, es decir, tablas que tienen un único índice. Sin embargo, en ocasiones necesitamos representar información que tiene más de una dimensión. Para ello, utilizamos tablas multidimensionales.

La principal característica de las tablas multidimensionales es que se almacenan en memoria de forma bidimensional, es decir, en filas y columnas.

Para declarar una tabla multidimensional, se indica el tipo de todos los elementos de la tabla, seguido del nombre de la variable, seguido de los números de elementos de cada dimensión encerrados entre corchetes.

```
    int tabla[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
```

Es importante recordar que en las tablas multidimensionales, la primera dimensión se corresponde con las filas, y la segunda dimensión se corresponde con las columnas. Por tanto, el primer índice se refiere a la fila, y el segundo índice se refiere a la columna. Además, al inicializar una tabla multidimensional, se pueden omitir los corchetes interiores, es decir, únicamente puede suprimirse el primero de los índices, y sólo ése.

```
    int tabla[3][4] = {
        1, 2, 3, 4,
        5, 6, 7, 8,
        9, 10, 11, 12
    };

    // En este caso, la tabla se inicializa de la misma forma que en el
ejemplo anterior.
```

Las tablas pueden pasarse como parámetros a las funciones. Cuandos e hace esto, lo que realmente se pasa como parámetro es la dirección al primer elemento de la tabla, y no la tabla en sí misma, esto es consecuencia de la no existencia del operador asignación para las tablas.

Puesto que el tamaño de la tabla no va implíctio en la misma, siempre que se pase una tabla como parámetro a una fucnión será necesario pasar también el tamaño de la tabla.

```

```

```
void imprime_tabla(int tabla[][4], int filas) {
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < 4; j++) {
            System.out.print(tabla[i][j] + " ");
        }
        System.out.println();
    }
}
```

# Tipos agregados

## Estructuras de datos

Una estructura es un conjunto finito de elementos de cualquier tipo agrupados bajo un mismo nombre. Las estructuras se utilizan para agrupar datos relacionados entre sí. Las estructuras se utilizan para representar un objeto del mundo real en un programa.

A cada elemento de la estructura se le denomina campo. Al contrario de lo que sucede con las tablas, no puede asegurarse en ningún caso que los campos de una estructura se almacenen en posiciones consecutivas de memoria.

Para declarar una estructura, se utiliza la palabra reservada *struct*, seguida del nombre de la estructura, y entre llaves, los campos de la estructura.

En lenguajes como Java, las estructuras se conocen como clases, y se definen de forma distinta. En Java, las clases se utilizan para agrupar datos y métodos relacionados entre sí.

```
class alumno {
    char nombre[20];
    int edad;
    float nota;
};
```

## Enumeraciones

En muchas situaciones, necesitamos que una variable pueda tener un valor dentro de un conjunto finito de valores. Para representar este tipo de variables, utilizamos las enumeraciones.

Dicho tipo nos permite definir un conjunto de valores que representarán algún parámetro cuyos valores sean discretos y finitos. Por ejemplo, los días de la semana, los meses del año, los colores, etc.

Para declarar una enumeración, se utiliza la palabra reservada *enum*, seguida del nombre de la enumeración, y entre llaves, los valores de la enumeración.

```
enum dias_semana {
    lunes, martes, miercoles, jueves, viernes, sabado, domingo
};
```

## Entrada y salida

Muchas aplicaciones requieren escribir o leer información de un dispositivo de entrada/salida. En Java, la entrada y salida se realiza a través de flujos de datos. Un flujo de datos es una secuencia de datos que se transfiere de un lugar a otro. En Java, los flujos de datos se utilizan para leer y escribir datos en un dispositivo de entrada/salida.

### Entrada de datos

La entrada de datos se realiza a través de la clase *Scanner*. La clase *Scanner* se utiliza para leer datos de la entrada estándar. Para utilizar la clase *Scanner*, es necesario importar la clase *Scanner* del paquete *java.util*.

```
import java.util.Scanner;

Scanner sc = new Scanner(System.in);

int a = sc.nextInt();
float b = sc.nextFloat();
String c = sc.nextLine();
```

### Salida de datos

La salida de datos se realiza a través de la clase *System.out*. La clase *System.out* se utiliza para escribir datos en la salida estándar. Para utilizar la clase *System.out*, es necesario importar la clase *System* del paquete *java.lang*.

```
System.out.println("Hola, mundo!");
```

### Formateo de datos

El formateo de datos se realiza a través de la clase *String*. La clase *String* se utiliza para formatear datos en una cadena de caracteres. Para utilizar la clase *String*, es necesario importar la clase *String* del paquete *java.lang*.

```
int a = 3;
float b = 4.5;
String c = "Hola";
String d = String.format("a = %d, b = %f, c = %s", a, b, c);
```

### Archivos

La entrada y salida de archivos se realiza a través de la clase *File*. La clase *File* se utiliza para leer y escribir archivos en un dispositivo de entrada/salida. Para utilizar la clase *File*, es necesario importar la clase *File* del

paquete *java.io*.

```
import java.io.File;

File f = new File("archivo.txt");
```

## El sistema operativo UNIX

A continuación, se describen los conceptos básicos del sistema operativo UNIX. Aunque las explicaciones de los comandos se refieren al sistema operativo linux, casi en su totalidad son aplicables a cualquier sistema operativo UNIX.

### Estructura de directorios y sistema de ficheros

Los ficheros sirven para guardar informaciñon, y normalmente están almacenados en disco. Todo el sistema UNIX está basado en ficheros. Cada fichero tiene un nombre, un contenido, un lugar para almacenarlo y cierta información de tipo administrativo, como por ejemplo, quién es el dueño y cuál es su tamaño.

El nombre de un fichero puede tener hasta 255 caracteres, y puede contener cualquier carácter, excepto el carácter barra inclinada (/) y el carácter nulo. Los nombres de los ficheros son sensibles a mayúsculas y minúsculas.

La estructura de directorios de UNIX es jerárquica, y se representa mediante un árbol. El directorio raíz del sistema se representa por una barra inclinada (/). Cada directorio puede contener ficheros y otros directorios. Los directorios se utilizan para organizar los ficheros en grupos lógicos.

Cada usuario tiene un directorio de trabajo, que es el directorio en el que se encuentra cuando inicia sesión en el sistema. El directorio de trabajo se representa por un punto (.) y el directorio padre se representa por dos puntos (..).

Para seleccionar un fichero dentro de una estructura de directorios, se utiliza una ruta. La ruta de un fichero se compone de una secuencia de nombres de directorios separados por barras inclinadas (/). La ruta de un fichero puede ser absoluta o relativa. Podemos utilizar cualquiera de los tres métodos siguientes:

- Ruta absoluta: se inicia en el directorio raíz del sistema.
- Ruta relativa: se inicia en el directorio de trabajo del usuario.
- Ruta relativa: se inicia en el directorio actual.

```
# Ejemplo de rutas absolutas
/home/usuario/fichero
# Ejemplo de rutas relativas
fichero
# Ejemplo de rutas relativas
./fichero
```

En los sistemas operativos de tipo UNIX los ficheros y directorios presentan tres propiedades especiales: el propietario, el grupo y los permisos. El propietario es la persona que creó el fichero o directorio. El grupo es

un conjunto de usuarios que tienen permisos para acceder al fichero o directorio. Los permisos son las acciones que se pueden realizar sobre el fichero o directorio.

Los tipos de permisos son:

- Lectura (r): permite leer el contenido del fichero o directorio.
- Escritura (w): permite modificar el contenido del fichero o directorio.
- Ejecución (x): permite ejecutar el fichero o acceder al directorio.

Y los conjuntos de permisos son:

- Propietario: el propietario del fichero o directorio.
- Grupo: el grupo al que pertenece el fichero o directorio.
- Otros: el resto de usuarios del sistema.

Dentro de cada conjunto, el primer carácater indica el tipo de fichero o directorio, y los tres siguientes caracteres indican los permisos del propietario, del grupo y de los otros usuarios.

```
    # Ejemplo de permisos. En este caso, el fichero tiene permisos de
lectura y escritura para el propietario, y permisos de lectura para el
grupo y para los otros usuarios.
    -rw-r--r-- 1 usuario grupo 0 ene 1 00:00 fichero
```

## Formato general de las órdenes

El procesador de órdenes (también conocido como *shell* o intérprete de comandos) es un programa que permite al usuario interactuar con el sistema operativo. El procesador de órdenes se utiliza para ejecutar comandos y programas, y para realizar tareas de administración del sistema.

El formato general de las órdenes es el siguiente:

- Comando: es el nombre del comando que se desea ejecutar.
- Opciones: son modificadores que se utilizan para cambiar el comportamiento del comando.
- Argumentos: son los valores que se pasan al comando para que realice su tarea.

```
    # Ejemplo de formato general de las órdenes
    comando -opciones argumentos
```

Los corchetes angulares (<>) se utilizan para indicar que el usuario debe sustituir el valor por el valor real. Los corchetes cuadrados ([]) se utilizan para indicar que el argumento es opcional.

```
    # Ejemplo de formato con corchetes
    comando [-opciones] <argumentos>
```

Las órdenes se pueden ejecutar de forma interactiva o en lotes. En la ejecución interactiva, el usuario introduce las órdenes una a una, y el sistema responde inmediatamente. En la ejecución en lotes, el usuario

introduce las órdenes en un fichero, y el sistema las ejecuta secuencialmente.

### Ayuda en línea

El sistema operativo UNIX dispone de un sistema de ayuda en línea que permite al usuario obtener información sobre los comandos y programas del sistema. La ayuda en línea se utiliza para obtener información sobre el uso de los comandos, las opciones disponibles y los argumentos que se pueden utilizar.

La ayuda en línea se obtiene a través del comando `man`. El comando `man` se utiliza para mostrar las páginas del manual del sistema. Para obtener información sobre un comando, se utiliza el comando `man` seguido del nombre del comando.

```
# Ejemplo de ayuda en línea
man pwd
```

### Comandos básicos

El sistema operativo UNIX dispone de un conjunto de comandos básicos que se utilizan para realizar tareas de administración del sistema. Los comandos básicos se utilizan para gestionar los ficheros y directorios del sistema, y para realizar tareas de administración del sistema.

Los comandos básicos se utilizan para realizar tareas como:

- Navegar por el sistema de ficheros.
- Crear, modificar y eliminar ficheros y directorios.
- Cambiar los permisos de los ficheros y directorios.
- Visualizar el contenido de los ficheros.
- Comprimir y descomprimir ficheros.
- Copiar, mover y renombrar ficheros y directorios.
- Buscar ficheros y directorios.
- Mostrar información del sistema.

### Navegación por el sistema de ficheros

El comando `pwd` se utiliza para mostrar el directorio de trabajo actual. El directorio de trabajo es el directorio en el que se encuentra el usuario cuando inicia sesión en el sistema.

```
pwd
```

El comando `cd` se utiliza para cambiar de directorio. El comando `cd` se utiliza para moverse por el sistema de ficheros. Para cambiar al directorio raíz del sistema, se utiliza el comando `cd` seguido de una barra inclinada (/).

```
# Cambiar al directorio raíz del sistema
cd /
```

El comando `ls` se utiliza para mostrar el contenido de un directorio. El comando `ls` se utiliza para listar los ficheros y directorios de un directorio. Para mostrar los ficheros y directorios ocultos, se utiliza el comando `ls` seguido de la opción `-a`.

```
    # Mostrar el contenido del directorio actual
    ls
    # Mostrar el contenido del directorio actual, incluyendo los ficheros y
directorios ocultos
    ls -a
```

## Creación, modificación y eliminación de ficheros y directorios

El comando `touch` se utiliza para crear un fichero vacío. El comando `touch` se utiliza para crear un fichero vacío en el directorio actual. Si el fichero ya existe, se actualiza la fecha de modificación del fichero.

```
    touch fichero
```

El comando `mkdir` se utiliza para crear un directorio. El comando `mkdir` se utiliza para crear un directorio en el directorio actual. Si el directorio ya existe, se muestra un mensaje de error.

```
    mkdir directorio
```

El comando `rm` se utiliza para eliminar un fichero o directorio. El comando `rm` se utiliza para eliminar un fichero o directorio en el directorio actual. Si el fichero o directorio no existe, se muestra un mensaje de error.

```
    # Eliminar un fichero
    rm fichero
    # Eliminar un directorio
    rm -r directorio
```

## Cambio de permisos de ficheros y directorios

El comando `chmod` se utiliza para cambiar los permisos de un fichero o directorio. El comando `chmod` se utiliza para cambiar los permisos de un fichero o directorio en el directorio actual. Los permisos se especifican mediante un número octal.

El número octal se compone de tres dígitos, que representan los permisos del propietario, del grupo y de los otros usuarios. Cada dígito se compone de tres bits, que representan los permisos de lectura, escritura y ejecución.

```

```

```
    # Cambiar los permisos de un fichero a lectura y escritura para el
propietario, y a lectura para el grupo y para los otros usuarios
    chmod 644 fichero
    # Cambiar los permisos de un directorio a lectura, escritura y
ejecución para el propietario, y a lectura y ejecución para el grupo y para
los otros usuarios
    chmod 755 directorio
```

### Visualización del contenido de los ficheros

El comando `cat` se utiliza para mostrar el contenido de un fichero. El comando `cat` se utiliza para mostrar el contenido de un fichero en la salida estándar. Si el fichero es muy grande, el comando `cat` mostrará el contenido del fichero por pantalla.

```
    cat fichero
```

El comando `more` se utiliza para mostrar el contenido de un fichero por páginas. El comando `more` se utiliza para mostrar el contenido de un fichero por páginas en la salida estándar. Si el fichero es muy grande, el comando `more` mostrará el contenido del fichero por páginas.

```
    more fichero
```

El comando `less` se utiliza para mostrar el contenido de un fichero por páginas. El comando `less` se utiliza para mostrar el contenido de un fichero por páginas en la salida estándar. Si el fichero es muy grande, el comando `less` mostrará el contenido del fichero por páginas.

```
    less fichero
```

### Comprimir y descomprimir ficheros

El comando `gzip` se utiliza para comprimir un fichero. El comando `gzip` se utiliza para comprimir un fichero en el directorio actual. El comando `gzip` crea un fichero comprimido con la extensión `.gz`.

```
    gzip fichero
```

El comando `gunzip` se utiliza para descomprimir un fichero. El comando `gunzip` se utiliza para descomprimir un fichero en el directorio actual. El comando `gunzip` crea un fichero descomprimido sin la extensión `.gz`.

```
    gunzip fichero.gz
```

El comando `tar` se utiliza para crear un archivo tar. El comando `tar` se utiliza para crear un archivo tar en el directorio actual. El comando `tar` crea un archivo tar con los ficheros y directorios especificados.

```
tar -cvf archivo.tar fichero1 fichero2 directorio
```

El comando `tar` se utiliza para extraer un archivo tar. El comando `tar` se utiliza para extraer un archivo tar en el directorio actual. El comando `tar` extrae un archivo tar con los ficheros y directorios especificados.

```
tar -xvf archivo.tar
```

### Copia, movimiento y renombrado de ficheros y directorios

El comando `cp` se utiliza para copiar un fichero o directorio. El comando `cp` se utiliza para copiar un fichero o directorio en el directorio actual. El comando `cp` crea una copia del fichero o directorio con el nombre especificado.

```
# Copiar un fichero
cp fichero1 fichero2
# Copiar un directorio
cp -r directorio1 directorio2
```

El comando `mv` se utiliza para mover un fichero o directorio. El comando `mv` se utiliza para mover un fichero o directorio en el directorio actual. El comando `mv` mueve el fichero o directorio con el nombre especificado.

```
# Mover un fichero
mv fichero1 fichero2
# Mover un directorio
mv directorio1 directorio2
```

El comando `mv` se utiliza para renombrar un fichero o directorio. El comando `mv` se utiliza para renombrar un fichero o directorio en el directorio actual. El comando `mv` renombra el fichero o directorio con el nombre especificado.

```
# Renombrar un fichero
mv fichero1 fichero2
# Renombrar un directorio
mv directorio1 directorio2
```

### Búsqueda de ficheros y directorios

El comando `find` se utiliza para buscar ficheros y directorios. El comando `find` se utiliza para buscar ficheros y directorios en el sistema de ficheros. El comando `find` busca los ficheros y directorios que coinciden con los criterios especificados.

```
    # Buscar ficheros y directorios en el directorio actual
    find . -name fichero
    # Buscar ficheros y directorios en el directorio actual y en los
subdirectorios
    find . -name fichero -type f
```

El comando `grep` se utiliza para buscar texto en los ficheros. El comando `grep` se utiliza para buscar texto en los ficheros del sistema de ficheros. El comando `grep` busca el texto que coincide con los criterios especificados.

```
    # Buscar texto en los ficheros del directorio actual
    grep texto fichero
    # Buscar texto en los ficheros del directorio actual y en los
subdirectorios
    grep -r texto directorio
```

## Información del sistema

El comando `date` se utiliza para mostrar la fecha y la hora del sistema. El comando `date` se utiliza para mostrar la fecha y la hora del sistema en la salida estándar. El comando `date` muestra la fecha y la hora en el formato especificado.

```
    date
```

El comando `who` se utiliza para mostrar los usuarios conectados al sistema. El comando `who` se utiliza para mostrar los usuarios conectados al sistema en la salida estándar. El comando `who` muestra los usuarios conectados al sistema con la información especificada.

```
    who
```

El comando `ps` se utiliza para mostrar los procesos en ejecución en el sistema. El comando `ps` se utiliza para mostrar los procesos en ejecución en el sistema en la salida estándar. El comando `ps` muestra los procesos en ejecución en el sistema con la información especificada.

```
    ps
```

## Redirección y tuberías

La redirección se utiliza para cambiar la entrada y la salida de los comandos. La redirección se utiliza para enviar la salida de un comando a un fichero, y para enviar la entrada de un comando desde un fichero.

```
# Redirigir la salida de un comando a un fichero
comando > fichero
# Redirigir la entrada de un comando desde un fichero
comando < fichero
```

Las tuberías se utilizan para combinar la salida de un comando con la entrada de otro comando. Las tuberías se utilizan para enviar la salida de un comando a la entrada de otro comando.

```
# Combinar la salida de un comando con la entrada de otro comando
comando1 | comando2
```

### Variables de entorno

Las variables de entorno se utilizan para almacenar información sobre el entorno del sistema. Las variables de entorno se utilizan para almacenar información sobre el sistema, como el nombre del usuario, el directorio de trabajo y la configuración del sistema.

```
# Mostrar el valor de una variable de entorno
echo $variable
# Asignar un valor a una variable de entorno
variable=valor
```

## Control de versiones

### Desarrollo colaborativo

Los programadores desarrollan código de forma colaborativa. Las herramientas populares como Dropbox, Google Drive o OneDrive no son adecuadas para el desarrollo de software. Las herramientas de control de versiones permiten a los programadores trabajar juntos en un proyecto de software.

Estas herramientas tienen muchos nombres: sistema de control de versiones (VCS), sistema de control de código fuente (SCM), sistema de control de revisiones (RCS), repositorios de código, etc. Pero todos hacen lo mismo: rastrean los cambios en el código fuente.

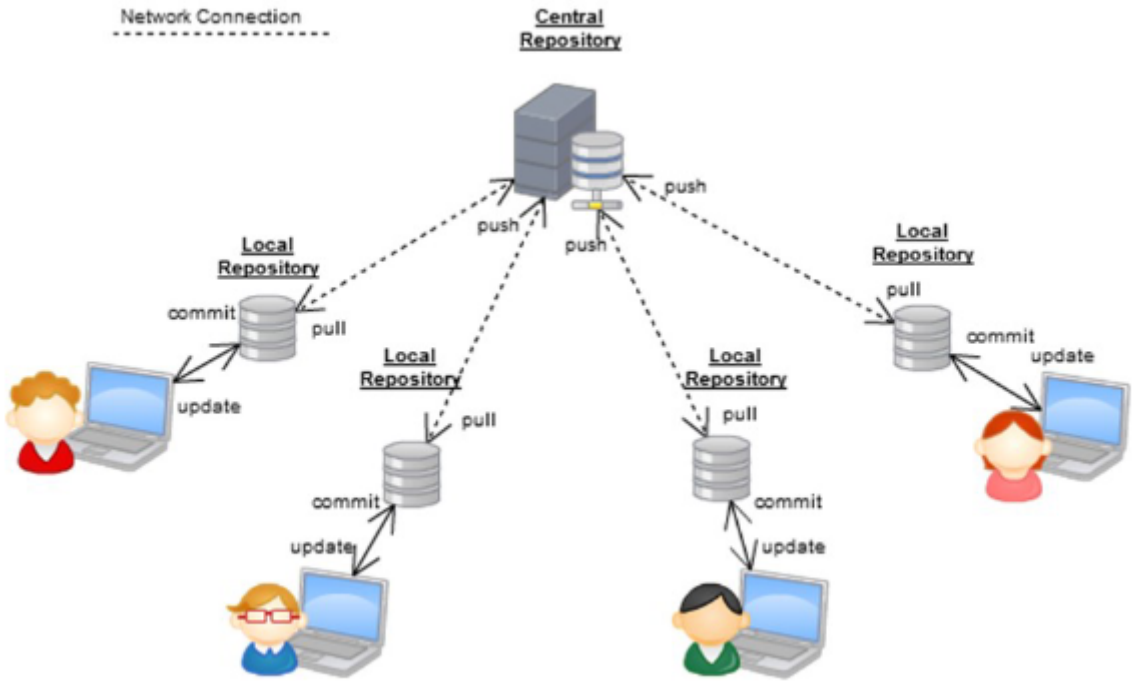





En esta ocasión, vamos a introducir Git. Git es un sistema de control de versiones con diversas caracterísitcas que lo hacen ideal para el desarrollo colaborativo:

- Es software libre
- Es muy popular
- Existen múltiples herramientas para trabajar con él
- Es muy rápido en relizar las operaciones
- Permite trabajar offline y luego sincronizar (distribuido)
- Posee funcionalidades que facilitan el trabajo colaborativo

Fué desarrollado por Linus Torvalds en 2005 para el desarrollo del kernel de Linux. Git es un sistema de control de versiones distribuido, lo que significa que cada desarrollador tiene una copia local del repositorio, lo que permite trabajar offline y luego sincronizar.




Posee varios clientes como TortoiseGit, GitKraken, RabbitVCS, etc. Que permiten integrarlo en le explorador de ficheros.

Cabe destacar que está integrado en editores de código como Visual Studio Code, Atom, Sublime Text, etc.





Además, existen múltiples aplicaciones para trabajar con los repositorios locales (clientes) mediante línea de comandos o interfaz gráfica. Aunque los clientes gráficos es posible que no ofrezcan todas las funcionalidades de Git se puede combinar sin problemas con las operaciones por líneas de comandos.

Git en Linux / Mac

Git for Windows

Por último, existen servicios de alojamiento de repositorios remotos como GitHub, GitLab, Bitbucket, etc. Que permiten alojar repositorios de forma gratuita o de pago.





## Conceptos básicos

¿Qué es un repositorio? Es una carpeta en disco que contiene los ficheros de un proyecto y la información necesaria para controlar las versiones de los mismos. Tiene una subcarpeta oculta llamada `.git` que contiene la información de control de versiones.

¿Cómo se crea un repositorio?

- Local: se crea un directorio y se inicializa como repositorio con el comando `git init`.
- Remoto: se crea un repositorio en un servidor y se clona en la máquina del desarrollador con el comando `git clone`.

¿Qué se guarda en un repositorio?

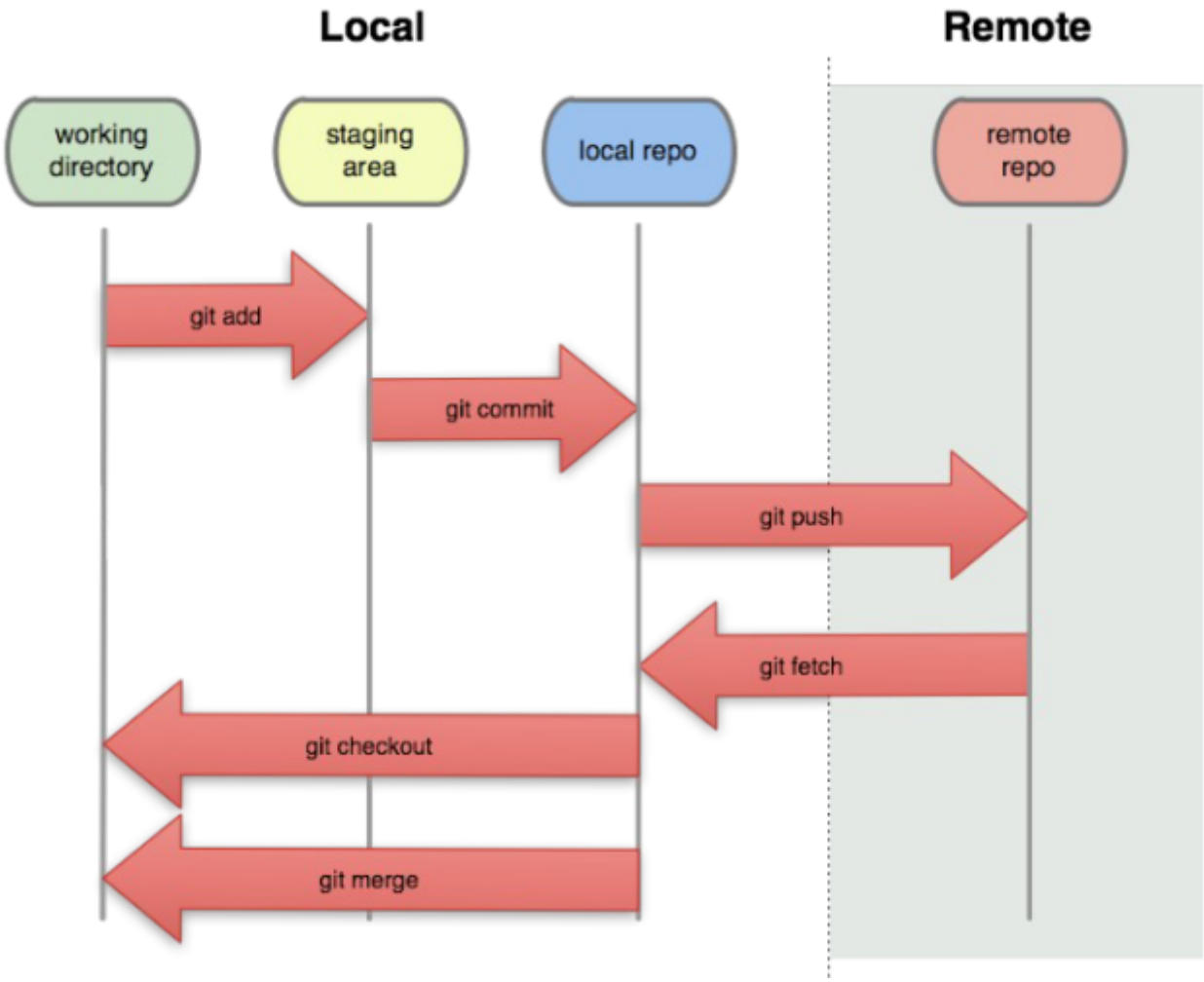
- Ficheros: los ficheros del proyecto.
- Metadatos: información sobre los ficheros y las versiones. Quién modificó cada fichero, cuándo y por qué.

¿Cómo se trabaja con un repositorio?

1. El desarrollador crea y edita los ficheros de forma habitual con editores de código.
2. Cuando se completa una funcionalidad, se indican los ficheros nuevos o los que se han modificado con el comando `git add` y se crea una nueva versión con el comando `git commit`.
3. Cuando se desea compartir los cambios con otros desarrolladores, se suben al repositorio remoto con el comando `git push`.
4. Cuando se desea obtener los cambios de otros desarrolladores, se descargan del repositorio remoto con el comando `git pull`.

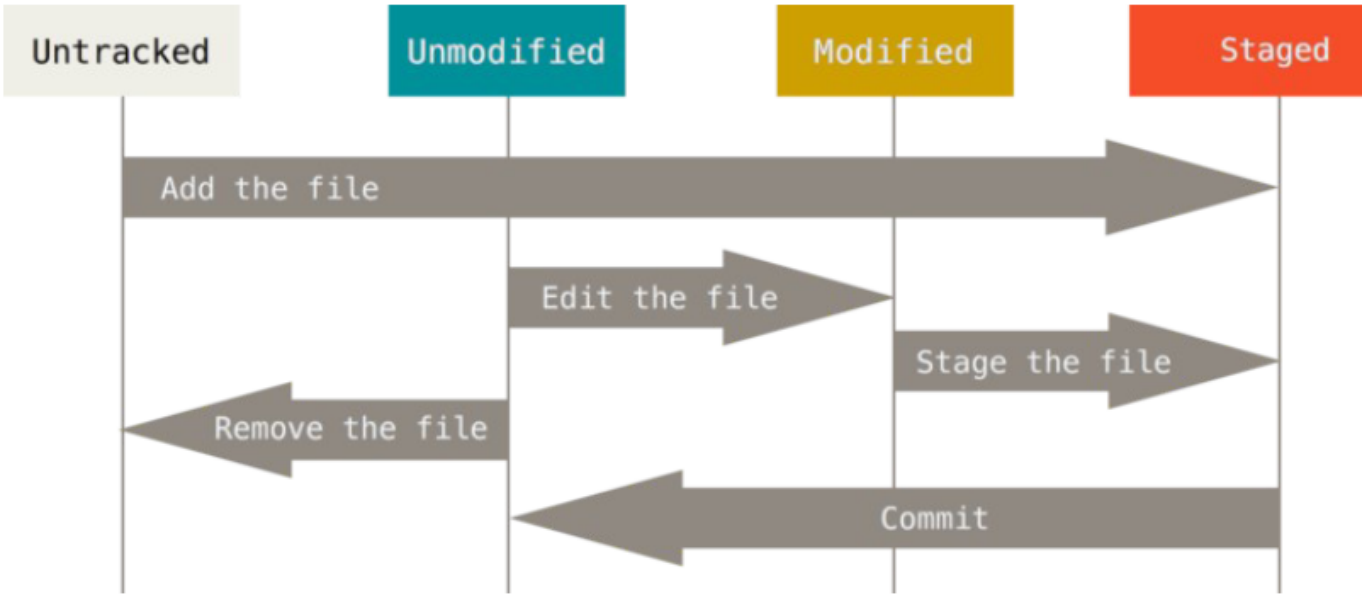
Veamos las zonas en las que está un fichero:

- Working directory: es el directorio de trabajo, donde se encuentran los ficheros del proyecto. Este es el espacio donde el desarrollador crea y modifica los ficheros. Si se va a trabajar con un repositorio existente, los ficheros se obtienen de una versión (*commit*) alohada en el directorio git usando el comando `git checkout`.
- Local repo: es el directorio `.git` que contiene la información de control de versiones. En este directorio se almacenan los metadatos de los ficheros y las versiones. Si se va a trabajar con un repositorio existente, se obtiene la información de control de versiones del repositorio remoto con el comando `git clone`.
- Staging area: no es una carpeta real, es una lista de ficheros, digamos que un área intermedia entre el directorio de trabajo y el repositorio local. En esta área se preparan los ficheros que se van a añadir al repositorio local. Si se va a trabajar con un repositorio existente, se añaden los ficheros al área de preparación con el comando `git add`.




Los cuatro estados de los ficheros son:

- Untracked: ficheros que no están en el repositorio. Por ejemplo, ficheros que se acaban de crear y todavía no se han añadido al repositorio.
- Unmodified: ficheros que no han sido modificados desde la última versión.
- Modified: ficheros que han sido modificados desde la última versión. Por ejemplo, ficheros que se han editado y todavía no se han añadido al repositorio.
- Staged: ficheros que han sido añadidos al área de preparación. Por ejemplo, ficheros que se han añadido al área de preparación y todavía no se han añadido al repositorio.




Los *commits* son las versiones de los ficheros que se guardan en el repositorio. Cada *commit* tiene un identificador único que se genera automáticamente. Los *commits* se crean con el comando `git commit`. Generalmente se añade un mensaje que describe los cambios realizados en el *commit*. Además cada commit se identifica con un hash que es un número hexadecimal de 40 caracteres que se genera automáticamente. Se suele usar la expresión "hash del commit" en lugar del "número de commit" o "id del commit" para referirse a este identificador.

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test
```

Los *snapshots* son las versiones de los ficheros que se guardan en el repositorio. Cada *snapshot* tiene un identificador único que se genera automáticamente. Los *snapshots* se crean con el comando `git commit`.

La identidad del desarrollador se configura con el comando `git config`. La identidad del desarrollador se compone de un nombre de usuario y una dirección de correo electrónico. La identidad del desarrollador se utiliza para identificar al desarrollador en los *commits*.

```
# Configurar el nombre de usuario
git config --global user.name "Nombre Apellido"
# Configurar la dirección de correo electrónico
git config --global user.email "Email"
# Ver la configuración de la identidad del desarrollador
git config --list
cat $HOME/.gitconfig
```

Además estos datos se configuran a nivel de ususario en el sistema operativo, en el fichero `.gitconfig` que se encuentra en el directorio de usuario.

## Git: trabajando en local

Para instalar Git en Windows, se descarga el instalador de la página web oficial de Git y se ejecuta el instalador. El instalador de Git instala Git en el sistema y configura las variables de entorno.

Para instalar Git en Linux, se utiliza el gestor de paquetes de la distribución. En Ubuntu, se instala Git con el comando `sudo apt-get install git`.

```
sudo apt-get install git
```

Para comprobar rápidamente si Git está instalado, se ejecuta el comando `git --version`. Si Git está instalado, se muestra la versión de Git.

```
git --version
```

A la hora de crear un repositorio, se crea un directorio y se inicializa como repositorio con el comando `git init`. El comando `git init` crea un directorio `.git` en el directorio actual.

```
# Crear un directorio
mkdir proyecto
# Cambiar al directorio
cd proyecto
# Inicializar como repositorio
git init
```

Veamos un ejemplo para crear un fichero y añadirlo al repositorio. Se crea un fichero con el comando `touch` y se añade al área de preparación con el comando `git add`. Se crea una nueva versión con el comando `git commit`.

```
# Crear un fichero
touch fichero
# Añadir al área de preparación
git add fichero
# Crear una nueva versión
git commit -m "Mensaje del commit"
```

Para subir los cambios al repositorio remoto, se añade el repositorio remoto con el comando `git remote add` y se suben los cambios al repositorio remoto con el comando `git push`.

```
# Añadir el repositorio remoto
git remote add origin URL_REPOSITORIO
# Subir los cambios al repositorio remoto
git push -u origin master
```

A la hora de consultar el estado del repositorio, se muestra el estado de los ficheros con el comando `git status`. El comando `git status` muestra los ficheros que han sido modificados y los ficheros que han sido añadidos al área de preparación.

```
git status
```

Existen una serie de buenas prácticas en el uso de los commits:

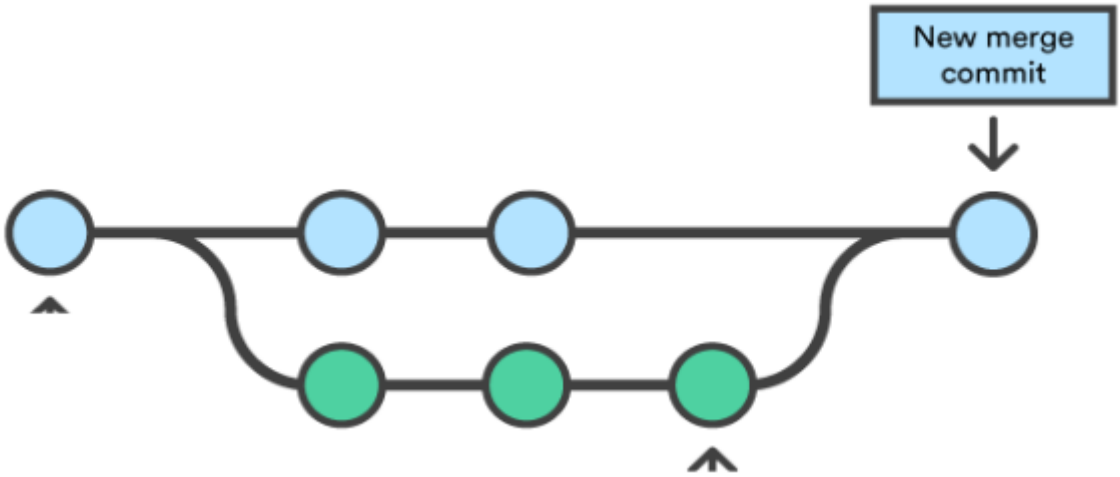
- Hacer commits pequeños y frecuentes
- Hacer commits atómicos, es decir, que contengan un solo cambio
- Añadir un mensaje descriptivo al commit
- Separar el asunto del cuerpo del mensaje con una línea en blanco
- Limitar la longitud del asunto a 50 caracteres
- Limitar la longitud del cuerpo a 72 caracteres
- Utilizar el imperativo en el asunto del mensaje
- Utilizar el presente en el asunto del mensaje
- Utilizar el cuerpo del mensaje para explicar el por qué del cambio

```
commit eb0b56b19017ab5c16c745e6da39c53126924ed6
Author: Pieter Wuille <pieter.wuille@gmail.com>
Date:   Fri Aug 1 22:57:55 2014 +0200

    Simplify serialize.h's exception handling

    Remove the 'state' and 'exceptmask' from serialize.h's stream
    implementations, as well as related methods.

    As exceptmask always included 'failbit', and setstate was always
    called with bits = failbit, all it did was immediately raise an
    exception. Get rid of those variables, and replace the setstate
    with direct exception throwing (which also removes some dead
    code).

    As a result, good() is never reached after a failure (there are
    only 2 calls, one of which is in tests), and can just be replaced
    by !eof().

    fail(), clear(n) and exceptions() are just never called. Delete
    them.
```

Las ramas o *branches* se utilizan para trabajar en paralelo en el mismo repositorio. Las ramas se utilizan para desarrollar nuevas funcionalidades o corregir errores sin afectar a la rama principal. Las ramas se crean con el comando `git branch`.

Una rama se puede crear partiendo de cualquier commit del repositorio. Es como hacer una copia de un documento para poder trabajas en dos "líneas de trabajo" en paralelo.

En cualquier momento se pueden mezclar el código de dos ramas diferentes. Cuando el código de la rama master está estbale, es cuando se publicará una nueva release.




```
# Crear una rama
git branch rama
# Cambiar a la rama
git checkout rama
# Hacer cambios en la rama y añadir al área de preparación
git add fichero
# Crear una nueva versión
git commit -m "Mensaje del commit"
# Cambiar a la rama master
git checkout master
# Mezclar la rama
git merge rama
```

Existe una estrategia de desarrollo bastante común que consiste en que cada nueva funcionalidad se desarrolla en una rama diferente. Una vez que la funcionalidad está terminada y probada, se mezcla con la rama principal. Esta técnica se conoce como *feature branch*.

Para borrar commits o ramas, se borra el commit con el comando `git reset` y se borra la rama con el comando `git branch -d`.

```
# Borrar el último commit
git reset --hard HEAD~1
# Borrar una rama
git branch -d rama
```

Hay que tener cuidado de no borrar commits o ramas que contengan cambios importantes. Si se borra un commit o una rama por error, se pueden recuperar los cambios con el comando `git reflog`. Sin embargo, es mejor prevenir que curar, por lo que es recomendable hacer una copia de seguridad manual de los cambios importantes antes de borrar.

Para borrar el contenido de un área de trabajo y retroceder al último commit, se utiliza el comando `git reset`. El comando `git reset` se utiliza para borrar el contenido del área de trabajo y retroceder al último commit. Es importante destacar que esta operación es irreversible, por lo que se recomienda hacer una copia de seguridad de los cambios importantes antes de borrar.

```
git reset --hard HEAD
```

Si por el contrario, queremos hacer un reset suave de manera que los cambios se mantengan en el área de trabajo, se utiliza el comando `git reset --soft HEAD~1`. El comando `git reset --soft HEAD~1` se utiliza para hacer un reset suave y retroceder al último commit. En este caso, los cambios se mantienen en el área de trabajo.

```
git reset --soft HEAD~1
```

Cuando en dos ramas se han modificado el mismo fichero, se produce un conflicto. Los conflictos se producen cuando dos ramas han modificado el mismo fichero y se intenta mezclar las ramas. Los conflictos se resuelven manualmente, editando el fichero y eliminando las líneas que causan el conflicto.

Es necesario que el desarrollador decide qué cambios se mantienen y cuáles se eliminan. Para resolver un conflicto, se edita el fichero y se eliminan las líneas que causan el conflicto. Una vez resuelto el conflicto, se añade el fichero al área de preparación con el comando `git add` y se crea una nueva versión con el comando `git commit`.

## Git y GitHub: trabajando en equipo

La forma más recomendada de trabajar con repositorios remotos en git es usar una clave SSH. Técnicamente se deben generar dos claves, una pública y una privada. La clave pública se añade a la cuenta de GitHub y la clave privada se guarda en el sistema local. Para generar las claves se utiliza el comando `ssh-keygen`.

```
ssh-keygen -t rsa -b 4096 -C "Email"
```

Luego, se copia la clave pública al portapapeles con el comando `cat` y se añade a la cuenta de GitHub en la sección de claves SSH.

```
cat $HOME/.ssh/id_rsa.pub
```

Hay dos formas de comenzar a trabajar en GitHub:

- Crear un repositorio en GitHub y clonarlo en local. Normalmente se incluye un fichero README.md con información sobre el proyecto y el fichero de licencia.
- Crear un repositorio en local y subirlo a GitHub. Esto nos permite subir nuestros cambios si ya tenemos contenido. Es un poco más engorroso pero es útil si ya tenemos un proyecto en local.

El repositorio local tiene automáticamente configurado el repositorio de GitHub como un repositorio remoto. Lo habitual es tener un único repositorio remoto, pero se pueden tener varios. Cada uno tiene un nombre, el repositorio de GitHub se llama `origin`.

```
# Crear un repositorio en GitHub
git remote add origin URL_REPOSITORIO
# Subir los cambios al repositorio remoto
git push -u origin master
```

Para el desarrollo colaborativo normalmente se utiliza GitHub como repositorio centralizado de confianza donde cada desarrollador tendrña una copia de este repositorio enlocal. Los cambios ademñas de comitearlos a nuestro repositorio local, se subirán al repositorio remoto empujando los cambios con el comando `git push`.

Periódicamente tendremos que traer los cambios del repositorio remoto a nuestro repositorio local. Para ello utilizamos el comando `git pull`.

```
# Descargar los cambios del repositorio remoto
git pull
# Subir los cambios al repositorio remoto
git push
```

Para poder habilitar que otros usuarios colaboren con nuestro repositorio, debemos añadirlos como colaboradores en GitHub. Para ello, en la página del repositorio, en la pestaña `Settings`, en la sección `Manage access`, se añade el nombre de usuario del colaborador.

Cuando se trabaja sobre master, los miembros del equipo sólo pueden conocer lo que hace un compañero cuando ya está integrado. Sería interesante que otros miembros del equipo pudieran revisar los cambios antes de que finalmente se integren para, si es necesario, solicitar mejoras. Esto es especialmente útil para desarrolladores con poca experiencia y para proyectos de software libre en los que los desarrolladores ajenos al proyecto hacen contribuciones al mismo. Esto es lo que se conoce como *pull request*.

Un pull request es una petición que hace un colaborador al propietario del repositorio para que incorpore los cambios que ha hecho en su repositorio. El propietario del repositorio revisa los cambios y decide si los incorpora o no.

El funcionamiento es el siguiente: 1. El colaborador hace un fork del repositorio del propietario. Creando una rama en local, generalmente asociada a una funcionalidad o *branch-per-feature*. 2. El colaborador hace los cambios en su rama y los sube a su repositorio en GitHub. 3. Desde la interfaz web, solicita integrar la rama en master del repositorio del propietario (pull request). 4. El pull request tiene un hilo de comentarios asociado para dar feedback sobre sus commits. Se pueden hacer más commits sobre la rama y se añadirán al pull request. 5. Cuando el código está listo, se acepta el pull request y se integra en master.

Para que los cambios en el servidor se refleren en el repositorio local, se utiliza el comando `git fetch`. El comando `git fetch` se utiliza para descargar los cambios del servidor y actualizar el repositorio local. El comando `git fetch` descarga los cambios del servidor y actualiza el repositorio local.

```
git fetch
```

Los tags o etiquetas se utilizan para marcar los *commits* importantes. Los tags se utilizan para marcar los *commits* importantes en el repositorio o para marcar releases e hitos del proyecto. Los tags se crean con el comando `git tag`.

```
git tag -a v1.0 -m "Versión 1.0"
```