

# UT-00 INTRODUCCIÓN A LA PROGRAMACIÓN

¿Qué vamos a ver?

- Datos, algoritmos y programas.
- Paradigmas de la programación.
- Lenguajes de programación.
- Herramientas y entornos de desarrollo de programas.
- Errores y calidad de los programas.
- Introducción al Lenguaje Java. Entorno de programación.

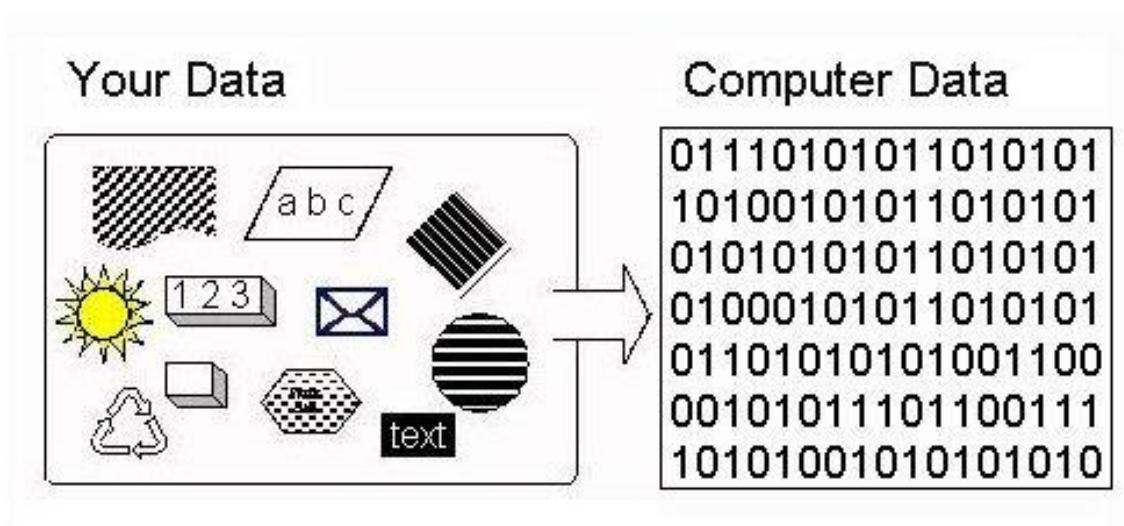
## 1. DATOS, ALGORITMOS Y PROGRAMAS

### 1.1. Datos

Según la RAE, dato, relacionado con la informática, se define como: “*Información dispuesta de manera adecuada para su tratamiento por una computadora*”.

Una computadora es una máquina que procesa, memoriza y transmite información. Esta información se da en forma de datos, que son conjuntos de símbolos utilizados para expresar o representar un valor numérico, un hecho, un objeto o una idea; en la forma adecuada para ser objeto de tratamiento.

Dentro de la computadora los datos son simples representación de número, textos, sonidos, vídeos, ... que en realidad consisten en ristras de unos y ceros, puesto que el lenguaje binario es el único que entiende.



Simplificando, la ejecución de un programa de ordenador se reduce a la realización de una serie de tratamientos sobre unos datos determinados.

#### 1.1.1. Tipos de datos

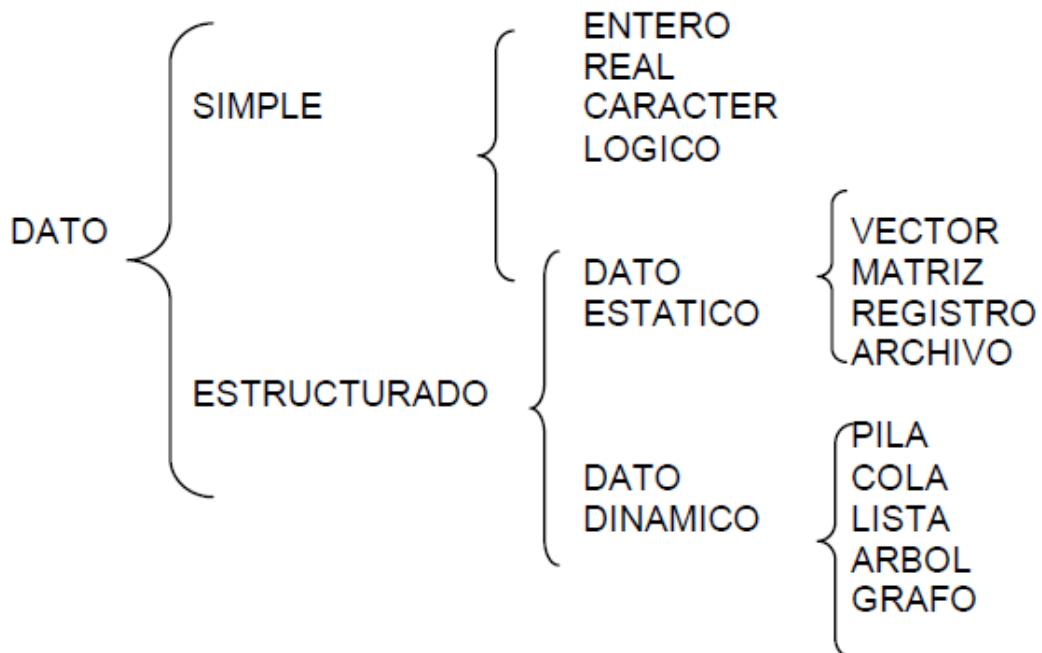
Podemos distinguir dos tipos de datos en función de su complejidad:

- **Tipos de datos simples:** a los cuales únicamente les corresponde un valor. Por ejemplo, un número o un carácter.

- **Tipos de datos compuestos o estructuras de datos:** se componen de varios valores de tipos simples. Su objetivo es el de ser más eficiente a la hora de almacenar y procesar la información. Un ejemplo son las cadenas de caracteres.

Dentro de las estructuras de datos podemos distinguir dos tipos, en función de la forma en la que realizan la reserva de memoria donde se van a almacenar los datos:

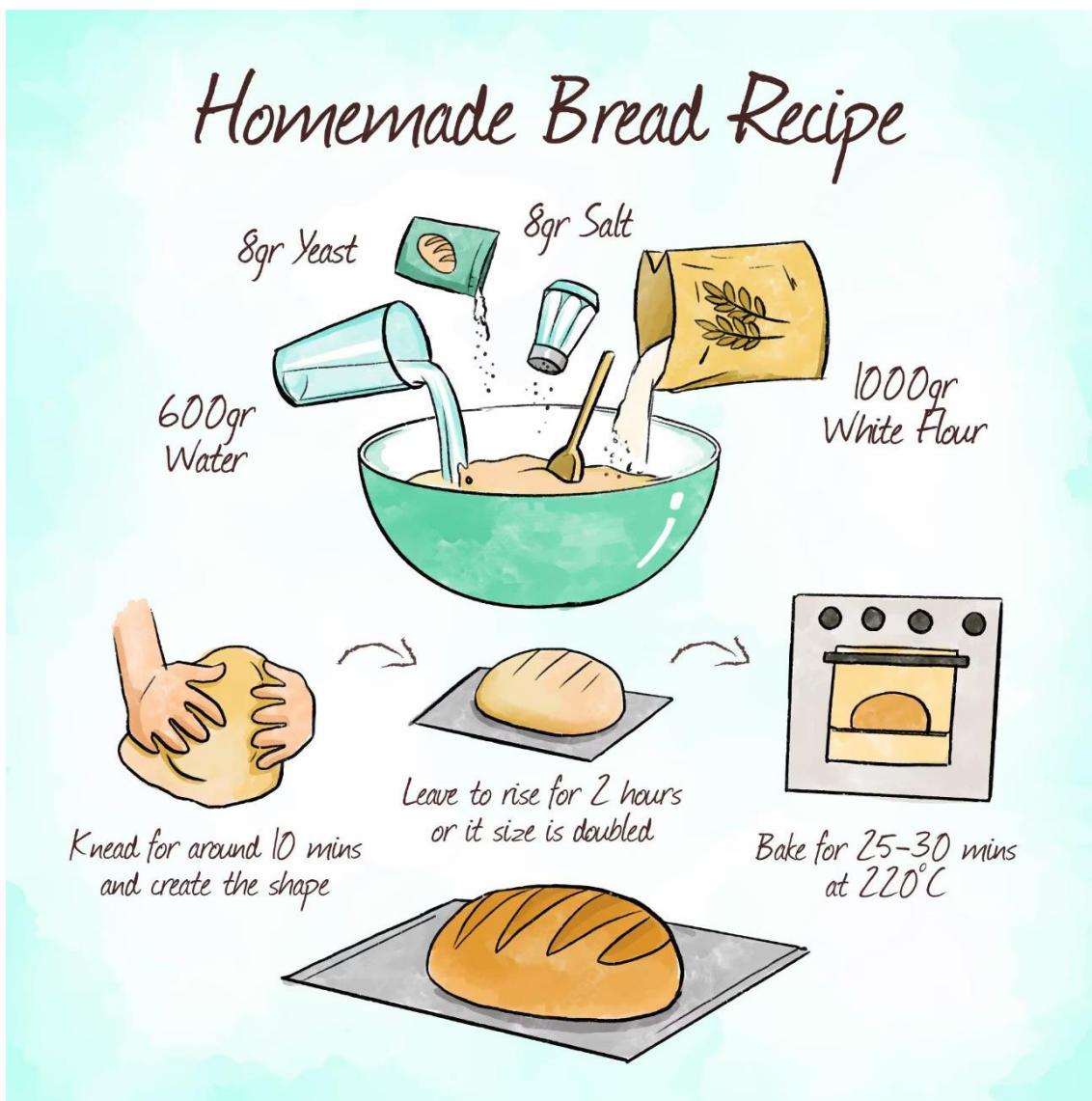
- **Estáticas:** la reserva del espacio de memoria que va a ocupar se realiza antes de la ejecución del programa y no se puede modificar. Como ejemplo principal tenemos los arrays o matrices.
- **Dinámicas:** la reserva del espacio de memoria que a utilizar se puede realizar tanto antes como durante la ejecución del programa y es posible modificarla. Aquí tenemos una subclasiﬁcación:
  - Lineales: a cada elemento le sigue y/o precede como máximo un único elemento. Como ejemplos tenemos las listas, las pilas y las colas.
  - No lineales: a cada elemento le puede seguir y/o preceder más de un elemento. Como ejemplos tenemos los árboles y los grafos.



## 1.2. Algoritmos

Un algoritmo es una secuencia ordenada de pasos, descrita sin ambigüedades, que lleva a la resolución de un problema dado, es decir, que un algoritmo nos describe paso a paso el proceso que debemos de llevar a cabo para resolver dicho problema.

Nos encontramos con algoritmos en nuestro día a día. Un tipo de algoritmo muy habitual es una receta.



En internet estamos rodeados de algoritmos, ... Google los usa para buscar información, YouTube para recomendarte vídeos, Spotify para recomendarte música, las redes sociales los usan para mostrarte personas que tiene gustos

similares a ti, mientras que navegas por internet un algoritmo determina los anuncios que se te van mostrando por pantalla, ...

Los algoritmos son conocidos y usados por el nombre desde tiempos inmemoriales, y ha sido uno de los factores por los que es la especie dominante en el planeta tierra, pero ... ¿Sabéis en qué siglo se data el primer algoritmo aplicado a la informática? ¿Y a quién se le atribuye?



A Ada Lovelace, hija de Lord Byron, se le atribuye el título de la primera programadora de la historia. Y en sus anotaciones sobre los números de Benoulli para la máquina analítica de Charles Babbage, se identifica el que se denomina el primer algoritmo informático de la historia. Esto ocurrió en el siglo XIX. En honor a ella se puso su nombre al lenguaje de programación Ada 95.

No debemos confundir un algoritmo con un programa, sobre los cuales hablaremos en el siguiente apartado, puesto que el algoritmo es el que determina cómo se resuelve el problema, mientras que el programa es la traducción del algoritmo a un lenguaje que entiende una computadora. Es por ello que en la resolución de un problema utilizando medios informáticos se exigen al menos los siguientes pasos y en este orden:

1. Definición y análisis del problema.
2. Diseño del algoritmo.
3. Transformación del algoritmo en un programa.
4. Ejecución y validación del programa.

La etapa del diseño del algoritmo es una tarea crucial y necesitará de creatividad y conocimientos de las técnicas de programación. El correcto diseño del algoritmo que dé solución a un problema ayuda al programador a evitar errores innecesarios en la fase de implementación.

Estilos distintos, de distintos programadores a la hora de obtener la solución de un problema, darán lugar a algoritmos diferentes, igualmente válidos.

#### *1.2.1. Características de un algoritmo*

Algunas de las características más importantes de un algoritmo son:

- **Preciso:** debe indicar perfectamente el orden de realización de cada paso.
- **Bien definido:** si se ejecuta dos veces con las mismas entradas se debe obtener el mismo resultado.
- **Finito:** el algoritmo debe acabar en un tiempo finito.
- **Eficiente:** utiliza de forma óptima los recursos del ordenador.
- **Robusto:** tiene prevista una respuesta clara, sean cuales sean los datos de entrada y las circunstancias bajo las que se ejecute.
- **Legible:** fácil de leer, comprensible, etc.
- **Repetitivo:** debe poder repetirse tantas veces como sea necesario.

#### *1.2.2. Eficiencia de un algoritmo*

Como hemos dicho antes, pueden definirse infinidad de algoritmos que resuelvan un mismo problema, pero interesa encontrar el que sea más eficiente en cuanto a dos factores fundamentales:

- **Tiempo de ejecución**
- **Recursos** necesarios para llevar a cabo el algoritmo

Existen dos formas de averiguar la eficiencia de un algoritmo:

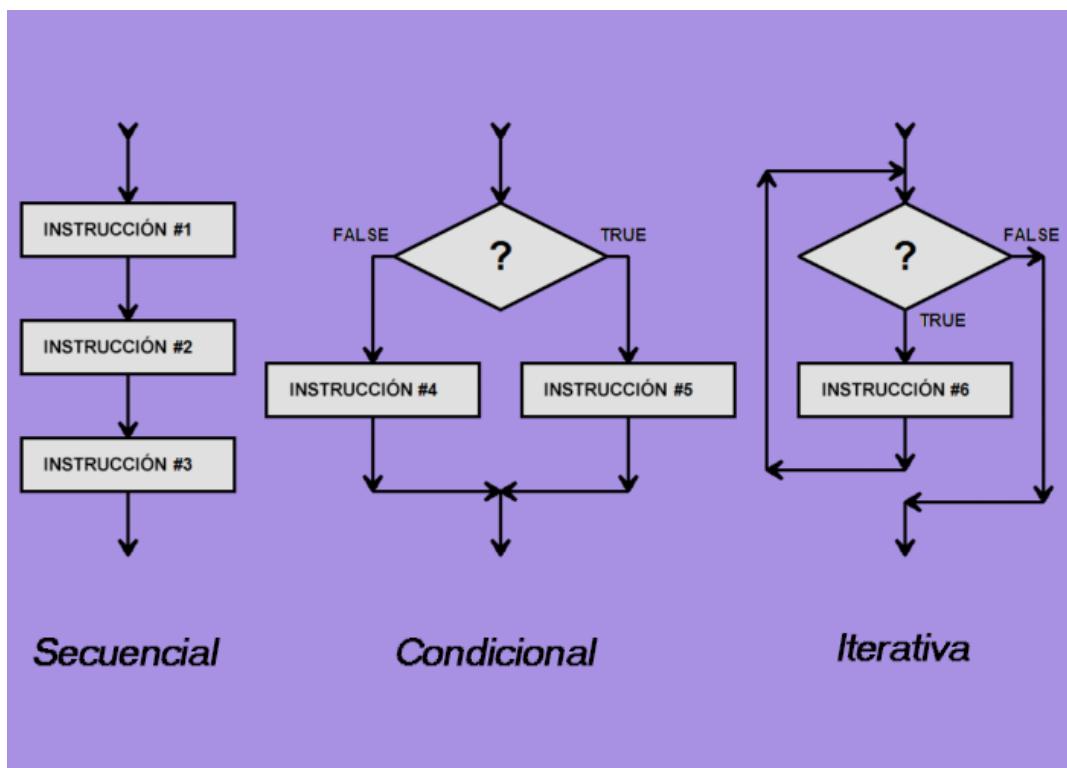
- **Empírica**: consiste en ejecutar un algoritmo dado con distintas entradas.
- **Teórica**: consiste en determinar matemáticamente la cantidad de recursos que necesita un algoritmo mediante una función.

### 1.2.3. Elementos de un algoritmo

En general, cualquier algoritmo está formado por:

- **Datos**: ya hemos hablado de ellos antes. Son los elementos sobre los que actúa el ordenador a partir de las instrucciones recibidas.
- **Operadores**: actúa sobre uno o más datos, a los que denominaremos operandos, para realizar una operación y obtener un resultado. Los más utilizados son:
  - Aritméticos: +, -, \*, /, ...
  - Lógicos: AND, OR, NOT, ...
  - Relacionales: ==, >, <, ...
  - Asignación: =
- **Operandos**: son los datos que intervienen en las operaciones. Tenemos dos tipos:
  - Constantes: su valor no puede cambiar nunca durante la ejecución del algoritmo.
  - Variables: su valor puede variar durante la ejecución del algoritmo.
- **Expresiones**: viene representada por la combinación de uno o varios operandos, unidos entre sí por operadores que realizan una acción entre ellos. Por ejemplo:  $5 + 2 * 4$
- **Instrucciones**: Son las acciones que en su momento se solicitará que lleve a cabo el procesador. Las instrucciones se pueden clasificar, según la función que desempeñan, en:
  - Instrucciones de declaración: se utilizan para declarar los objetos que se van a usar en el algoritmo.

- **Instrucciones primitivas:** se ejecutan de forma inmediata por el procesador. Pueden ser de tres tipos:
  - *Entrada:* para tomar datos del exterior.
  - *Salida:* para presentar información a través de algún dispositivo de salida.
  - *Asignación:* para asignar valores a las variables.
- **Instrucciones de control:** son las instrucciones encargadas de controlar el orden en que las instrucciones de un programa se ejecutarán. Pueden ser de tres tipos:
  - Estructuras secuenciales: se caracterizan porque se ejecuta una acción a continuación de otra y así sucesivamente, en el mismo orden que están escritas.
  - Estructuras selectivas: se caracterizan porque se ejecuta una acción u otra según se cumpla o no una determinada condición. Pueden ser simples, dobles o múltiples.
  - Estructuras repetitivas: se caracterizan porque se ejecutan las acciones del cuerpo del bucle mientras o hasta que se cumpla una determinada condición. Pueden ser de varios tipos.



#### 1.2.4. Técnicas descriptivas

Para representar gráficamente los algoritmos existen diferentes técnicas descriptivas que ayudan a describir su comportamiento de una forma precisa y genérica, para luego poder codificarlos con el lenguaje de programación que corresponda.

Destacan las siguientes técnicas:

- **Pseudocódigo**
- **Diagramas de flujo**
- **Diagramas N-S**
- **Tablas de decisión**

##### 1.2.4.1. Pseudocódigo

Esta técnica se basa en el uso de palabras clave del lenguaje natural, constantes, variables, otros objetos, instrucciones y estructuras de programación que expresan de forma escrita la solución del problema. Es una técnica muy utilizada. Se puede considerar como un paso intermedio entre la solución de un problema y su codificación en el lenguaje de programación seleccionado.

Ejemplo: Pseudocódigo para el algoritmo DetectarNegativo.

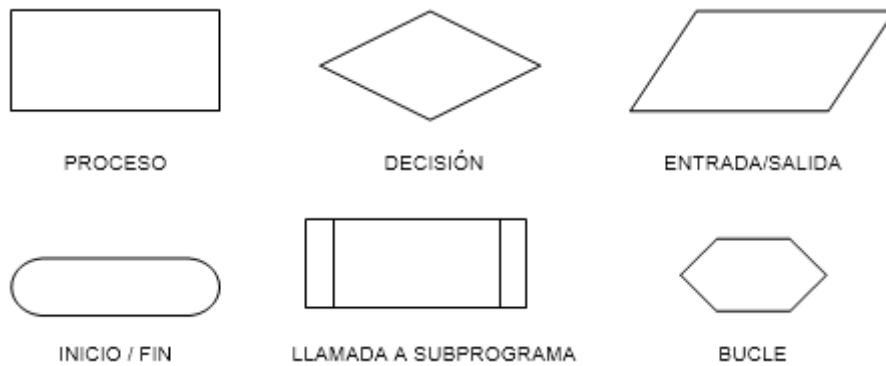
```
Proceso DetectarNegativo
    Escribir "Introduzca un número entero: "
    Leer número
    Si número < 0 Entonces
        Escribir "Es negativo"
    Sino
        Escribir "No es negativo"
    Fin Si
FinProceso
```

Herramientas software para crear pseudocódigo: PSeInt e Inter P.

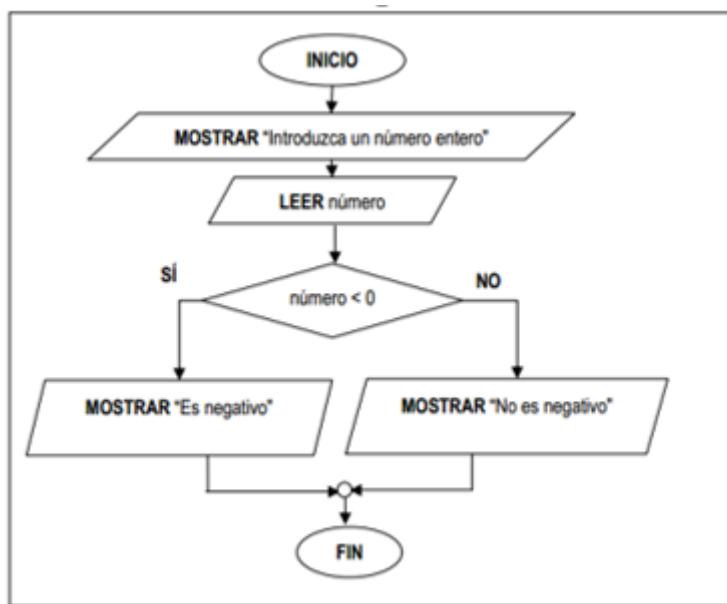
### 1.2.4.2. Diagramas de flujo

Los diagramas de flujo o flujogramas consisten en representar gráficamente las distintas operaciones que componen un procedimiento o parte de este, estableciendo su secuencia cronológica. Para ello se dispone de grafismos para representar el diagrama.

Los símbolos más utilizados en los diagramas de flujo son:



Ejemplo: Diagrama de flujo (ordinograma) para el algoritmo DetectarNegativo.

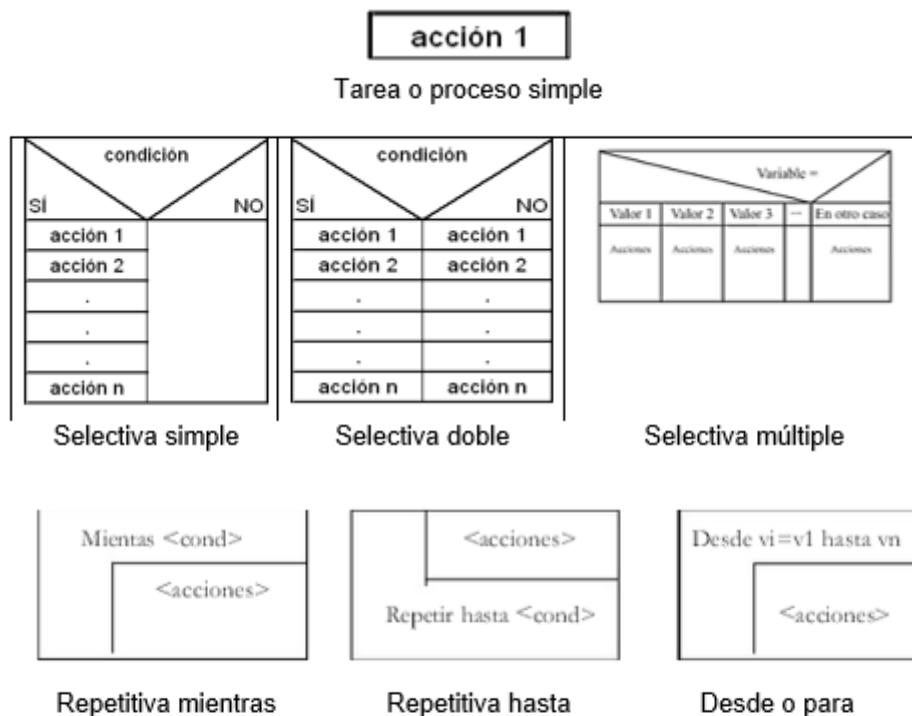


Herramientas software para crear diagramas de flujo: DFD, DIA, draw.io.

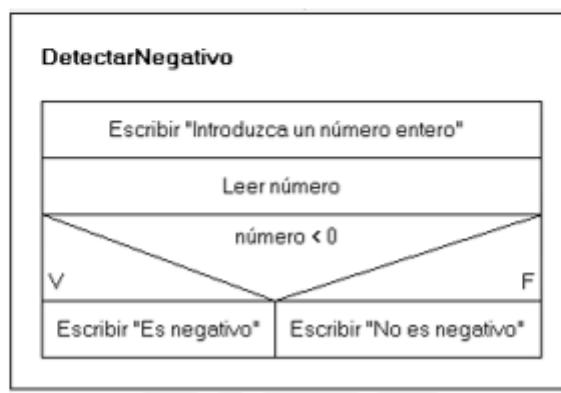
### 1.2.4.3. Diagramas N-S

Los diagramas N-S (Nassi-Shneiderman) permiten representar gráficamente algoritmos, utilizando una representación de cajas o bloques contiguos.

Los símbolos más utilizados en los diagramas N-S son:



Ejemplo: Diagrama N-S para el algoritmo DetectarNegativo



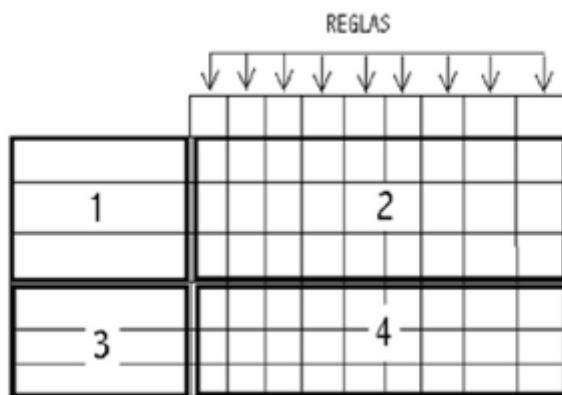
Herramientas software para crear diagramas N-S: Edraw y Structorizer.

#### 1.2.4.4. Tablas de decisión

En una tabla son representadas las posibles condiciones del problema con sus respectivas acciones. Suelen ser una técnica de apoyo al pseudocódigo cuando existen situaciones condicionales complejas.

La estructura de la tabla es la siguiente:

- **Zona 1:** Representa las condiciones que se examinan.
- **Zona 2:** Especifica los valores que pueden tomar las condiciones de la zona 1.
- **Zona 3:** Especifica las acciones a realizar.
- **Zona 4:** Especifica si se realiza la acción, en base a los valores de las condiciones.



### 1.3. Programas

Una computadora está compuesta por dos partes principales y bien diferenciadas: el hardware y el software. Gracias a la interacción de ambas es como los ordenadores pueden llevar a cabo desde simples operaciones matemáticas hasta tareas de gran complejidad.

El hardware hace referencia a la parte tangible del ordenador, es decir, la parte física, como puede ser la carcasa, la placa base, el procesador, los discos duros, etc.

El software es la parte no tangible del ordenador, es decir, la parte lógica, compuesta por programas de distintos tipos y naturalezas. Los programas son los encargados de indicar a las distintas partes del ordenador qué deben hacer a cada momento, y así llevar a cabo las tareas que se le han asignado.

Podríamos hacer el símil de que el hardware es como el cuerpo de una persona, con sus distintos órganos, y el software es la información que se almacena en el cerebro y que se utiliza para dar órdenes al organismo para realizar distintas tareas como hablar, andar, comer, etc.

Para que la computadora realice una tarea asignada de forma correcta, deberá contener un programa específico que le indique paso a paso cómo debe llevarla a cabo, siendo de una gran importancia que este conjunto de pasos, también llamado algoritmo, esté definido de una forma correcta. Debemos tener en cuenta que una computadora solo hace lo que se le ordena y, por tanto, si el algoritmo no es correcto, la computadora realizará la tarea de una forma incorrecta tantas veces como se le ordene hacerla y nunca corregirá sus fallos, hasta que una persona reprograme el programa y corrija el error.

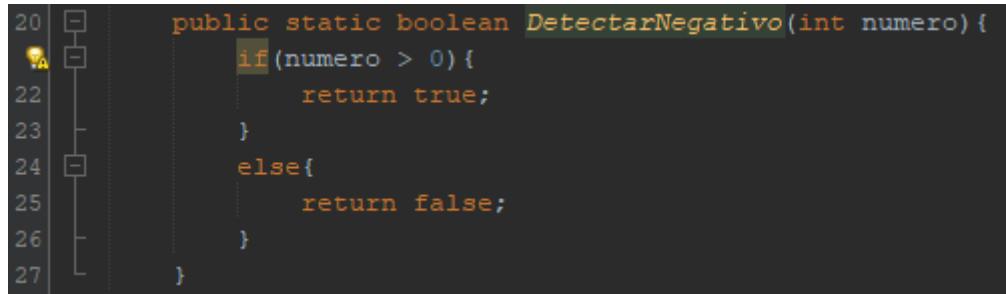


#### 1.3.1. Características de un programa

Un algoritmo, posteriormente convertido en programa, debe cumplir una serie de características:

- **Finito:** debe estar formado por un conjunto finto de líneas de forma que en algún punto ve alcanzado su fin.

- **Legible:** es importante crear códigos “límpios” y fáciles de leer, con tabulaciones y espacios que diferencien las partes del programa. Si desarrollamos código bien estructurado nos será más fácil la corrección de errores y modificación del mismo.



```
20  public static boolean DetectarNegativo(int numero){  
21      if(numero > 0){  
22          return true;  
23      }  
24      else{  
25          return false;  
26      }  
27  }
```

A screenshot of a Java code editor showing a simple method named 'DetectarNegativo'. The code uses standard Java syntax with an if-else conditional statement. The code is numbered from 20 to 27 on the left side. A small icon of a person is visible in the top-left corner of the code area.

- **Modificable:** debe estar diseñado de forma que debe ser sencillo el proceso de actualización o modificación del mismo ante nuevas necesidades del usuario final.
- **Eficiente:** debemos crear programas que ocupen poco espacio en memoria y se ejecuten rápidamente.
- **Modulares:** esto ayuda a que el programa sea más legible y fácil de entender. Debemos perseguir realizar algoritmos que se encuentren divididos en subalgoritmos de forma que se disponga de un grupo principal desde el que llamaremos al resto. Al usar subprogramas además propiciamos la reutilización de código y evitamos la repetición de este.
- **Estructurado:** Esta característica engloba de alguna forma las anteriores, ya que un programa estructurado será fácil de leer, de modificar y estará compuesto de subprogramas que permitirán la reutilización de código.

## 2. PARADIGMAS DE PROGRAMACIÓN

### 2.1. Definición de paradigma de programación

Un paradigma de programación consiste en un método para llevar a cabo cómputos y la forma en la que deben estructurarse y organizarse las tareas que debe realizar un programa. Un paradigma está delimitado en el tiempo en cuanto a aceptación y uso, porque nuevos paradigmas aportan nuevas o mejores soluciones que los sustituyen parcial o totalmente.

### 2.2. Paradigmas imperativo y declarativo



En general, la mayoría de los paradigmas son variantes de dos tipos principales de programación, la imperativa y la declarativa.

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

Algunos de los paradigmas más destacados que provienen del imperativo son:

- Programación estructurada: orientada a mejorar la claridad, calidad y tiempo de desarrollo de un programa recurriendo a subrutinas y a tres estructuras básicas: secuenciales, condicionales y repetitivas.
- Programación modular: consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y

manejable. Se presenta históricamente como una evolución de la programación estructurada para solucionar problemas de programación más grandes y complejos.

- Programación orientada a objetos: el núcleo central de este paradigma es la unión de datos y procedimientos en una entidad llamada “objeto”, relacionable a su vez con otras entidades “objeto”.
- **Programación declarativa**: en este paradigma las sentencias que se utilizan describen el problema que se quiere solucionar; se programa diciendo lo que se quiere resolver a nivel de usuario, pero no las instrucciones necesarias para solucionarlo. Esto último se realizará mediante mecanismos internos de inferencia de información a partir de la descripción realizada.
- Algunos de los paradigmas más destacados que provienen del declarativo son:
  - Programación funcional: basada en la definición de predicados y es de corte más matemático.
  - Programación lógica: basada en la definición de relaciones lógicas. Son lenguajes en los que se especifican un conjunto de hechos y una serie de reglas que permiten la deducción de los hechos. El sistema utiliza esa información para encontrar la solución.

### 2.3. Programación orientada a objetos

La programación orientada a objetos permite una representación más directa del modelo del mundo real en el código. El resultado es que se reduce considerablemente la transformación de los requisitos del sistema (definido en términos de usuario) a la especificación del sistema (definido en términos de computador). Es un paradigma de programación que usa objetos en sus interacciones, para diseñar aplicaciones y programas informáticos. Está basada en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

Fue a principios de la década de los 90 cuando su uso se popularizó y en la actualidad es el paradigma de programación más utilizado existiendo gran variedad de lenguajes de programación que soportan la orientación a objetos.

Las principales ventajas de la programación orientada a objetos son:

- **Comprendión:** al representar los problemas en términos del mundo real se facilita la comprensión. Es una representación más cercana a nuestra forma de pensar.
- **Modularidad:** al estar las definiciones de los objetos en módulos, hace que las aplicaciones estén mejor organizadas y sean más fáciles de entender.
- **Fácil mantenimiento:** gracias a la claridad de código que se consigue es muy sencillo aplicar modificaciones o añadir nuevos elementos.
- **Seguridad:** no se pueden modificar los datos de un objeto directamente, sino que se debe hacer mediante las acciones definidas para ese objeto.
- **Reutilización del código:** facilita la reutilización mediante el uso de bibliotecas de clases.

Las principales características de la programación orientada a objetos, sobre las que profundizaremos en futuras unidades, son:

- **Abstracción**
- **Encapsulación**
- **Herencia**
- **Polimorfismo**

Sobre estas características hablaremos más a fondo en el bloque dedicado a la programación orientada a objeto.

### 3. LENGUAJES DE PROGRAMACIÓN



Un lenguaje de programación puede definirse como un idioma artificial diseñado para que sea fácilmente entendible por un humano e interpretable por una máquina. Consta de una serie de reglas y de un conjunto de órdenes o instrucciones. Cada una de estas instrucciones realiza una tarea determinada. A través de la secuencia de instrucciones podemos indicar a una computadora el algoritmo que debe seguir para solucionar un problema dado. A un algoritmo escrito utilizando las instrucciones de un lenguaje de programación se le denomina programa.

#### 3.1. Características de un lenguaje de programación

- **Facilidad de lectura y comprensión:** Los programas deben ser corregidos y modificados, por lo tanto, el lenguaje debe facilitar la comprensión de los programas una vez escritos.
- **Facilidad de codificación:** Debe ser simple utilizar el lenguaje para desarrollar programas que resuelvan el tipo de problemas al que está orientado.
- **Fiabilidad:** Es fiable si funciona bien en cualquier condición. Depende de la comprobación de tipos de datos, del control de excepciones y del manejo de la memoria.
- **Posibilidad de usar un entorno de programación:** Se refiere a la existencia de herramientas para el desarrollo.

- **Portabilidad:** Un mismo programa debe funcionar en ordenadores diferentes.
- **Coste:** Relacionado con el aprendizaje, codificación, ejecución, fiabilidad y mantenimiento.
- **Detallabilidad:** Define el número de pasos que es necesario definir en un programa para dar solución a un determinado problema.
- **Generalidad:** Indica las opciones de uso que tiene un lenguaje para resolver problemas de distinto tipo. Cuantos más problemas diferentes pueda solucionar más general es el lenguaje.

## 3.2. Clasificaciones de lenguajes de programación

### *3.2.1. Clasificación en función de su nivel de abstracción*

Esta clasificación se realiza atendiendo a la forma en que son validados por el ordenador. Nos referimos a lo próximo que está el lenguaje de programación del lenguaje máquina. Tenemos los siguientes tipos:

- **Lenguaje máquina:** Está íntimamente ligado a la máquina. La arquitectura concreta del ordenador determinará el repertorio de instrucciones máquina que el ordenador es capaz de ejecutar, así como el formato de estas. Estas instrucciones se representan internamente como secuencias de ceros y unos.
- **Lenguaje ensamblador o de bajo nivel:** En lugar de las pesadas series de 1 y 0, el lenguaje ensamblador utiliza símbolos reconocibles, llamados mnemotécnicos, para representar las instrucciones. Son populares los lenguajes en ensamblador de los microprocesadores Z-80, 68000, 8086/88, 80386, todos ellos desarrollados en la década de los 80.

Estos dos lenguajes están muy próximos a la máquina. Necesitan muchas instrucciones para realizar una tarea específica y son únicos para el procesador particular.

- **Lenguajes de alto nivel:** Estos lenguajes permiten programar sin necesidad de conocer el funcionamiento interno de la máquina. Normalmente una instrucción en lenguaje de alto nivel equivale a varias

en lenguaje máquina. Están diseñados para que las personas escriban y entiendan los programas de un modo mucho más fácil que los lenguajes máquina y ensambladores. Además, un lenguaje de alto nivel es independiente de la máquina, es decir, las instrucciones del programa no dependen del diseño del hardware. Ejemplos de lenguajes de alto nivel son Cobol, Fortran, Pascal, Prolog, Smalltalk, Modula, Delphi, Ada, C, C++, C#, Java, Lisp, SQL, etc. A su vez, los lenguajes de alto nivel se pueden clasificar en:

- **Lenguajes compilados:** Para ser ejecutado es necesario traducirlo previamente a código máquina, mediante la operación de compilación, la cual da como resultado un programa ejecutable. Una vez compilado la ejecución es más rápida que la de un lenguaje interpretado, pero la misma compilación no se puede utilizar en distintas plataformas. Ejemplos: C, C++, Pascal.
- **Lenguajes interpretados:** Estos lenguajes son traducidos, o interpretados, a código máquina en tiempo de ejecución. La ejecución es más lenta que un lenguaje compilado, pero se puede ejecutar en varias plataformas. Ejemplo: PHP, JavaScript.
- **Lenguajes parcialmente compilados o interpretados:** Realizan una compilación previa a un lenguaje intermedio, como bytecode, y ese es el código que se traduce, o interpreta, a código máquina en tiempo de ejecución. La ejecución es más lenta que un lenguaje compilado, pero se puede ejecutar en varias plataformas. Ejemplos: Java, Python.



### 3.2.2. Clasificación cronológica

Esta clasificación se realiza atendiendo a la fecha de creación de los lenguajes. De esta forma, podemos diferenciar:

- **Lenguajes de primera generación:** Fueron los primeros lenguajes en aparecer. Los primeros ordenadores se programaban directamente en código de máquina (basado en sistema binario), que puede representarse mediante secuencias de 0 y 1. No obstante, cada modelo de ordenador tiene su propia estructura interna a la hora de programarse. Por ejemplo, se utiliza este tipo de lenguajes para programar tareas críticas de los sistemas operativos, de aplicaciones en tiempo real o controladores de dispositivos.
- **Lenguajes de segunda generación:** Los lenguajes simbólicos, asimismo propios de la máquina, simplifican la escritura de las instrucciones y las

hacen más legibles. Se refiere al lenguaje ensamblado a través de un macroensamblador. Es el lenguaje de máquina combinado con poderosas macros que permiten declarar estructuras de datos y de control complejas.

- **Lenguajes de tercera generación:** Aparecen técnicas de programación (estructurada, concurrente, orientada a objetos) y con ellas nuevos lenguajes como Fortran, Ada, C, C++, Java, Pascal y Prolog, etc. Son usados en ámbitos computacionales donde se logra un alto rendimiento con respecto a los lenguajes de generaciones anteriores.
- **Lenguajes de cuarta generación:** Aparecen herramientas orientadas al desarrollo de aplicaciones de gestión que permiten automatizar operaciones como, definir bases de datos, realizar informes, consultas, ... Hoy se piensa que estas herramientas no son, propiamente hablando, lenguajes. Algunas de sus características son: acceso a base de datos, generación de código automáticamente, así como poder programar visualmente (como por ejemplo Visual Basic o SQL). Entre sus ventajas se cuenta una mayor productividad y menor agotamiento del programador. Por ejemplo: NATURAL y PL/SQL.
- **Lenguajes de quinta generación:** En ocasiones se llama así a los lenguajes asociados con sistemas expertos y la inteligencia artificial. Principalmente utilizan programación declarativa y lógica. Por ejemplo: Mercury, Haskell, etc.

### 3.2.3. Clasificación según el tipo de aplicación

Esta clasificación se realiza atendiendo a la funcionalidad o tipo de aplicación a la que va dirigido el lenguaje de programación:

- **Comercial:** Son lenguajes que deben dotar al programador de facilidades en cuanto a gestión de ficheros, registros y campos. Debe ofrecer al proceso una integridad total de datos. Ejemplos: Cobol o PowerBuilder.
- **Para problemas de cálculo científico:** Son lenguajes que se caracterizan por las pocas operaciones de entrada/salida que necesitan. Por el contrario, realizará una gran cantidad de complejos cálculos. Ejemplos: Fortran.

- **Multipropósito:** Son lenguajes que pueden resolver problemas de diferentes áreas. Por ejemplo, el lenguaje Pascal resuelve problemas tanto de tipo científico como comercial.
- **Especializados:** Son lenguajes que nacen en las empresas que tienen un hardware tan específico, que además han de construir el propio software para la utilización de esos equipos.
- **Orientados a Internet:** Son lenguajes que han aparecido debido al auge de Internet en los últimos años y están orientados a trabajar en red. Ejemplos: Visual J, PHP, etc.

### 3.2.4. Clasificación por paradigmas

Esta clasificación se basa en el tipo de paradigma, o paradigmas, que implementa el lenguaje de programación:

- **Programación estructurada:** Orientada a mejorar la claridad, calidad y tiempo de desarrollo de un programa recurriendo a subrutinas y tres estructuras básicas: secuencia, selección e iteración.
- **Programación modular:** Consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable. Se presenta históricamente como una evolución de la programación estructurada para solucionar problemas de programación más grandes y complejos.
- **Programación orientada a objetos:** Está basada en el imperativo, pero el núcleo central de este paradigma es la unión de datos y procedimientos en una entidad llamada “objeto”, relacionable a su vez con otras entidades “objeto”. Está representado por C++, C#, Java, Python o Smalltalk.
- **Programación dinámica:** Está definida como el proceso de romper problemas en partes pequeñas para analizarlos y resolverlos de forma lo más cercana al óptimo. Este paradigma está más basado en el modo de realizar los algoritmos, por lo que se puede usar con cualquier lenguaje imperativo.
- **Programación funcional:** Basada en la definición de los predicados y es de corte más matemático, está representado por Scheme o Haskell. Python también representa este paradigma.

- **Programación lógica:** Basado en la definición de relaciones lógicas, son lenguajes en los que se especifican un conjunto de hechos y una serie de reglas que permiten la deducción de otros hechos. El sistema utiliza esa información para encontrar la solución. El lenguaje lógico más representativo es PROLOG, basado en la lógica de predicados.
- **Programación con restricciones:** Similar a la lógica usando ecuaciones. Casi todos los lenguajes son variantes del Prolog.
- **Programación reactiva:** Se basa en la declaración de una serie de objetos emisores de eventos asíncronos y otra serie de objetos que se “suscriben” a los primeros (es decir, quedan a la escucha de la emisión de eventos de estos) y reaccionan a los valores que reciben.
- **Lenguaje específico del dominio o DSL:** Se denomina así a los lenguajes desarrollados para resolver un problema específico, pudiendo entrar dentro de cualquier grupo anterior. Ejemplo: SQL para el manejo de las BBDD.
- **Programación multiparadigma:** Es el uso de dos o más paradigmas dentro de un programa. Por ejemplo, lenguajes de programación como C++, Visual Basic o PHP combinan el paradigma imperativo con la orientación a objetos.

### 3.2.5. Otras clasificaciones

Veamos brevemente otras posibles clasificaciones:

- Según el lugar de ejecución:
  - **Cliente:** El código fuente se obtiene de un servidor en el que reside y se ejecuta en el lado cliente. Ejemplos: JavaScript.
  - **Servidor:** El código fuente reside en el servidor y el cliente no puede acceder a él, pero sí puede pedir que se ejecute para obtener el resultado a través de la red. Ejemplo: PHP.
- Según la concurrencia:
  - **No concurrente:** Los programas que generan se ejecutan, dando lugar a un proceso en el que se las instrucciones se ejecutan unas

detrás de otras, por lotes, sin interactuar con otros procesos.  
Ejemplos: Batch, Bash.

- **Concurrentes:** Tienen mecanismos que permiten compartir información entre procesos distintos. También, a partir de un proceso se pueden crear otros subprocesos y ejecutarse a la vez, de forma concurrente. Ejemplos: Java y C.

### 3.3. Lenguajes de programación más utilizados

Según la consultora Tiobe Software y de acuerdo al índice TIOBE de septiembre de 2023, los lenguajes de programación más utilizados por los desarrolladores son:

Posición	Lenguaje de Programación
1	 Python
2	 C
3	 C++
4	 Java
5	 C#
6	 JavaScript
7	 Visual Basic
8	 PHP
9	 Assembly language
10	 SQL

- **Python:** Fue creado a finales de los 80. Es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma. Se trata de un lenguaje ideal para generar scripts y es muy personalizable gracias a la multitud de módulos existentes.
- **C:** Creado en 1972 por Dennis Ritchie con la colaboración de Kernigham a partir de un lenguaje desarrollado por Ken Thompson denominado B, que a su vez fue elaborado a partir de otro lenguaje denominado BCPL. Con la popularidad de las microcomputadoras se crearon muchas implementaciones de C, así que el Instituto Nacional de Estándares Americano (ANSI) estableció un comité en 1983 para definir un estándar. Finalmente, en 1990 se crea ANSI C como estándar del lenguaje C.
- **C++:** Fue creado en 1983 por Bjarne Stroustrup. La intención de su creación fue el extender el exitoso lenguaje C a los mecanismos que permitían la manipulación de objetos.
- **Java:** Fue creado en 1995 por Sun Microsystems. Es orientado a objetos, El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple, y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria. Con respecto a la memoria, su gestión no es un problema ya que ésta es gestionada por el propio lenguaje y no por el programador.
- **C#:** Fue creado en 2001 por Microsoft. Es un lenguaje de programación orientado a objetos. Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .Net, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes.
- **Javascript:** Fue creado en 1995 por Netscape. Es un lenguaje de programación interpretado y diseñado para complementar las capacidades de HTML. El código de JavaScript es enviado al cliente como parte del código HTML de una página.

- **Basic:** Fue creado en 1965 con el fin de crear un lenguaje fácil de aprender, lo que le dio gran popularidad. Este lenguaje ha evolucionado mucho y actualmente existen versiones orientadas a objetos, Visual Basic.
- **PHP:** Fue creado en 1995. Es un lenguaje de programación interpretado, diseñado originalmente para la creación de páginas web dinámicas. Se usa principalmente para la interpretación del lado del servidor.
- **SQL:** Es un lenguaje de dominio específico utilizado en programación, diseñado para administrar, y recuperar información de sistemas de gestión de bases de datos relacionales.

### 3.4. Nuevos lenguajes de programación

Además de todos los lenguajes de programación estudiados anteriormente existen otros lenguajes que son ya una realidad y que hay que tener en cuenta, ya que en el futuro pueden ser muy importantes. Algunos de ellos son:

- **Kotlin:** Fue el lenguaje de programación de moda de 2019, no por ser el más fácil, sino porque Google lo ha “marcado” como el mejor lenguaje de programación para Android. Es un lenguaje multiplataforma y está indexado 100% con Java pero tiene sus propias reglas.
- **Rust:** Se trata del lenguaje de programación que Mozilla inició su desarrollo en 2009 y cuya primera versión estable se publicó en mayo de 2015. Muchas empresas y desarrolladores apuestan por él. Es un lenguaje de sistemas enfocado principalmente a la seguridad, la velocidad y la concurrencia.
- **Go (Golang):** Es un lenguaje de programación diseñado por Google. Está inspirado en C, pero es bastante más complicado. Está enfocado a procesos muy concretos, pero el objetivo principal es la seguridad. Por eso, Go no tiene aritmética de punteros. Siendo de Google, en cuanto se empiece a popularizar, seguro que gana un importante hueco en el desarrollo de Android.
- **Swift:** Es un lenguaje de programación diseñado por Apple. Swift es uno de los lenguajes clave para programar en iOS con iOS Development Bootcamp.

## 4. HERRAMIENTAS Y ENTORNOS DE DESARROLLO DE PROGRAMAS



¿Qué es un IDE?  
Integrated development environment (IDE)



NetBeans



Entorno de desarrollo integrado

Los entornos de desarrollo integrados (IDE) son una combinación de herramientas que permiten automatizar el proceso de desarrollo y prueba de los programas.

Los primeros entornos de desarrollo que se crearon estaban dedicados exclusivamente a un lenguaje de programación, pero con el tiempo han ido apareciendo otros con los que se puede programar en distintos lenguajes.

Algunas de sus características son:

- Facilitan las labores de programación.
- Aportan herramientas automáticas de ayuda.
- Son independientes respecto del entorno final de ejecución de la aplicación.
- Facilitan la compatibilidad y la integración en familias de productos.

### 4.1. Funciones de un entorno de desarrollo

Las principales funciones de los IDE son:

- **Editor de código:** coloración de la sintaxis.
- **Autocompletado de código,** atributos y métodos de clases.
- **Identificación automática de código.**

- **Herramientas de concepción visual** para crear y manipular componentes visuales.
- **Asistentes y utilidades de gestión y generación de código.**
- **Compilación** de proyectos complejos en un solo paso.
- **Control de versiones.**
- **Generador de documentación** integrado.
- **Detección de errores** de sintaxis en tiempo real.
- **Refactorización de código:** cambios menores en el código que facilitan su legibilidad sin alterar su funcionalidad (por ejemplo, cambiar el nombre a una variable).
- **Permite introducir automáticamente tabulaciones y espaciados** para aumentar la legibilidad.
- **Depuración:** seguimiento de variables, puntos de ruptura y mensajes de error del intérprete.
- Aumento de funcionalidades a través de la gestión de sus **módulos y plugins**.
- **Administración de las interfaces de usuario** (menús y barras de herramientas).
- **Administración de las configuraciones del usuario.**

#### 4.2. Componentes de un entorno de desarrollo

Los entornos de desarrollo están formados por una serie de componentes software que determinan sus funciones.

- **Editor de texto**
- **Compilador / Intérprete**
- **Depurador**
- **Generador de interfaces gráficas**

#### 4.3. Entornos de desarrollo libres y propietarios

Los entornos de desarrollo integrados de desarrollo más relevantes en la actualidad son:

- **Entornos de desarrollo libres:** son aquellos con licencia de uso público.

Tipos de entornos de desarrollo libres más relevantes en la actualidad		
IDE	Lenguajes que soporta	Sistema Operativo
NetBeans	C/C++, Java, JavaScript, PHP, Python	Windows, Linux, Mac OS X
Eclipse	Ada, C/C++, Java, JavaScript, PHP	Windows, Linux, Mac OS X
IntelliJ IDEA Community Edition	Java y Kotlin	Windows, Linux, Mac OS X
Microsoft Visual Studio Community Edition	Visual Basic, C/C++, C#, Python, Javascript	Windows, Mac OS X
Anjuta	C/C++, Python, Javascript	Linux
Geany	C/C++, Java	Windows, Linux, Mac OS X

- **Entornos de desarrollo propietario:** son aquellos entornos de desarrollo integrado que tiene una licencia de pago.

Tipos de entornos de desarrollo propietarios más relevantes en la actualidad		
IDE	Lenguajes que soporta	Sistema Operativo
Microsoft Visual Studio Enterprise Edition	Visual Basic, C/C++, C#, Python, Javascript, Java	Windows, Mac OS X
IntelliJ IDEA Ultimate Edition	Java y Kotlin	Windows, Linux, Mac OS X
Xcode	C/C++, Java	Mac OS X
JBuilder	Java	Windows, Linux, Mac OS X
JCreator	Java	Windows
C++ Builder	C/C++	Windows
Turbo C++ Profesional	C/C++	Windows

## 5. ERRORES Y CALIDAD DE LOS PROGRAMAS

### 5.1. Errores de los programas

Durante el proceso de desarrollo de software, se pueden producir dos tipos de errores:

- **Errores de compilación:** se corresponde con el incumplimiento de reglas sintácticas del lenguaje de programación utilizado. Ocasiona que el programa no puede ejecutarse hasta que el programador no corrija ese error. Ejemplos de errores de compilación:
  - Palabras reservadas mal escritas.
  - Expresiones erróneas o incompletas.
  - Variables no declaradas.

Son detectados por el propio entorno de desarrollo en tiempo real durante su escritura o durante la compilación del código.

- **Errores lógicos o de ejecución:** comúnmente llamados bugs. Estos no evitan que el programa se pueda compilar con éxito, ya que no hay errores sintácticos y permiten su ejecución. Sin embargo, pueden provocar que el programa devuelva resultados erróneos, que no sean los esperados o pueden provocar que el programa termine antes de tiempo o no termine nunca. Ejemplos de errores lógicos o de ejecución son:
  - Dividir un número por cero.
  - Exceder un rango de valores posible en una variable o una estructura de datos.
  - Crear un bucle infinito.

Para la localización y corrección de estos errores se usa una herramienta de gran utilidad, como es el depurador.

#### 5.1.2. El depurador de código

El depurador permite supervisar la ejecución de los programas, para localizar y eliminar los errores lógicos. Un programa debe compilarse con éxito para posteriormente poder utilizarlo en el depurador. El depurador nos permite

analizar todo el programa, mientras éste se ejecuta. Permite suspender la ejecución de un programa, examinar y establecer los valores de las variables, comprobar los valores devueltos por un determinado método, el resultado de una comparación lógica o relacional, etc.

De este modo, el depurador permite analizar el flujo de ejecución del código y el estado de los datos conforme van siendo manipulados por el programa.

Para poder depurar un programa, podemos ejecutar el programa de diferentes formas, de manera que en función del problema que se quiera solucionar, resulte más sencillo un método u otro.

Existen los siguientes tipos de ejecución:

- **Paso a paso por instrucción:** algunas veces es necesario ejecutar un programa línea por línea para buscar y corregir errores lógicos.
- **Paso a paso por procedimiento:** permite introducir los parámetros que queremos a un método o función de nuestro programa, pero en vez de ejecutar instrucción por instrucción ese método, nos devuelve su resultado. Es útil, cuando hemos comprobado que un procedimiento funciona correctamente, y no nos interesa volver a depurarlo, solo nos interesa el valor que devuelve.
- **Ejecución hasta una instrucción:** el depurador ejecuta el programa y se detiene en la instrucción donde se encuentra el cursor, a partir de ese punto podemos hacer una depuración paso a paso o por procedimiento.
- **Ejecutar hasta el final del programa:** se ejecuta las instrucciones de un programa hasta el final, sin detenerse en las instrucciones intermedias.

```

1 package Depurar;
2
3 import java.util.Random;
4
5 /* @author (Dario José Zubayar) */
6
7 public class Depurar {
8     public static void main(String[] args) {
9         System.out.println(sumar(5, 6));
10    }
11    public static int sumar(int num1, int num2){
12        int aux = num1 - num2;
13        return aux;
14    }
15 }

```

Name	Type	Value
<Enter new watch>		
+ Static		
num1	int	5
num2	int	6
aux	int	-1

### 5.3. Calidad de los programas

Para escribir programas que proporcionen los mejores resultados, cabe tener en cuenta una serie de detalles.

- **Corrección:** un programa es correcto si hace lo que debe hacer tal y como se estableció en las fases previas a su desarrollo. Por ello se deben especificar claramente qué debe hacer el programa.
- **Claridad:** es muy importante que el programa sea lo más claro y legible posible, para facilitar así su desarrollo y posterior mantenimiento. Al elaborar un programa se debe intentar que su estructura sea sencilla y coherente, así como cuidar el estilo de la edición.
- **Eficiencia:** se trata de que el programa, además de realizar aquello para lo que fue creado (es decir, que sea correcto), lo haga gestionando de la mejor forma posible los recursos con los que cuenta.

- **Portabilidad:** un programa es portable cuando tiene la capacidad de poder ejecutarse en una plataforma, ya sea hardware o software, diferente a aquella en la que se elaboró.

Para obtener un programa de buena calidad es muy importante seguir las recomendaciones de buenas prácticas tanto generales de desarrollo de software, como específicas del lenguaje programación con el que estemos programando. Existen una gran cantidad fuentes que nos deber servir como referencia a la hora de desarrollar nuestro código, como pueden ser:

- El libro “Clean Code” (Código limpio) de Robert C. Martin.
- Los principios S.O.L.I.D.
- El uso de patrones de diseño.
- Guías de buenas prácticas para el desarrollo de código en Java.

## 6. INTRODUCCIÓN AL LENGUAJE JAVA. ENTORNO DE PROGRAMACIÓN

### 6.1. Lenguaje de programación Java



El lenguaje de programación Java fue desarrollado por James Gosling y Sun Microsystems en mayo de 1995 como lenguaje de desarrollo de internet, aunque ya anteriormente, en 1991 había sido presentado como lenguaje de programación para componentes eléctricos. Su sintaxis es parecida a los

lenguajes de C y C++, aunque no posee tantas facilidades de acceso a bajo nivel como estos.

En 1996 aparece la versión del kit de desarrollo para java, JDK (Java Development Kit) 1.0. Actualmente, la última versión liberada del JDK es la 20, aunque nosotros vamos a utilizar la versión JDK 17, puesto que es la última versión LTS (Long Term Support), es decir, una versión estable que tiene una larga duración de soporte.

Enlace donde se pueden descargar las últimas versiones JDK:  
<https://www.oracle.com/es/java/technologies/downloads/#java17>

Las principales características de Java son:

- **Orientado a objetos:** Es un lenguaje orientado a objetos, lo que significa que se basa en el concepto de objetos y clases para organizar y estructurar el código.
- **Multiplataforma:** Es independiente de la plataforma, lo que significa que el código Java se puede ejecutar en cualquier sistema operativo o dispositivo compatible con Java sin necesidad de recompilar el código.
- **Seguro:** Tiene características de seguridad integradas que ayudan a proteger los sistemas contra amenazas externas.
- **Manejo automático de memoria:** Libera automáticamente la memoria utilizada por objetos que ya no se utilizan.
- **Multithreading:** Permite la ejecución de varios hilos de ejecución en paralelo, lo que permite a las aplicaciones realizar varias tareas al mismo tiempo.
- **API's integradas:** Dispone de una gran cantidad de API's (interfaz de programación de aplicaciones) integradas, lo que permite a los desarrolladores utilizar una variedad de funciones y herramientas para crear aplicaciones.
- **Popular:** Es uno de los lenguajes de programación más populares y utilizados en el mundo. En su mayor parte, para el desarrollo de aplicaciones empresariales y el desarrollo de aplicaciones para Android.

## 6.2. Entorno de desarrollo Apache NetBeans



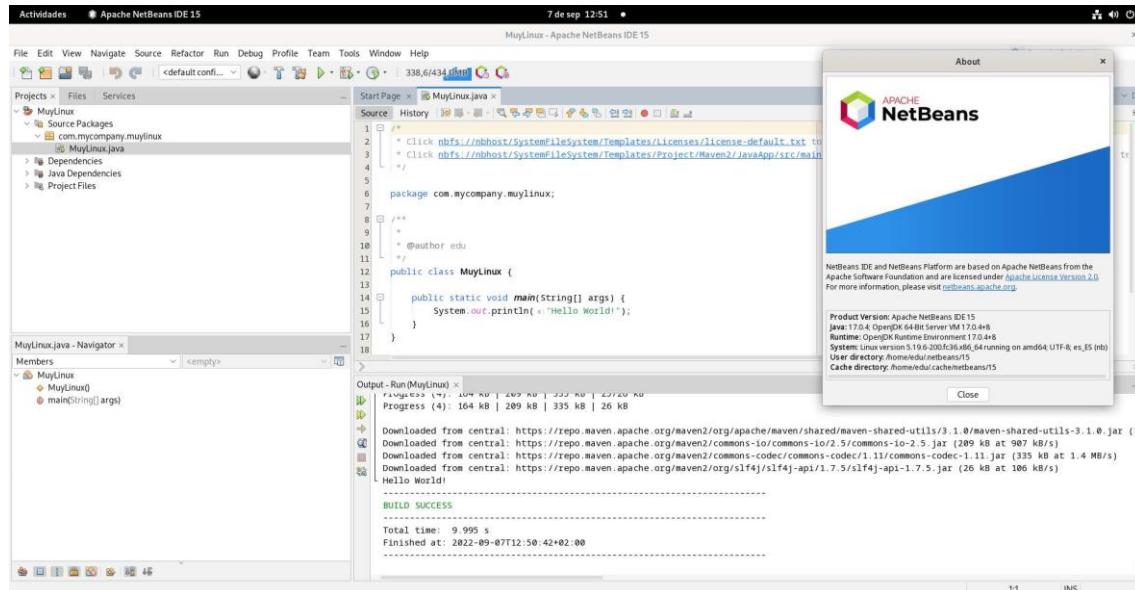
Apache NetBeans es un IDE gratuito, que se distribuye para diferentes plataformas, es decir, puede usarse en Windows, Linux y Mac OS X. Actualmente está disponible la versión 19 (septiembre de 2023), y puede utilizarse para el desarrollo en distintos lenguajes como Java, JavaScript, PHP, HTML5, CSS y más.

En el siguiente enlace es posible descargarlo: <https://netbeans.apache.org/>

Es recomendable realizar en primer lugar la instalación del JDK de Java y a continuación instalar el Apache NetBeans IDE.

Con NetBeans el programador dispone de multitud de herramientas para llevar a cabo su tarea. Lo más básico es el editor de texto donde escribir las instrucciones y un compilador que transforme el fichero de texto, con las sentencias Java, en un fichero escrito en un lenguaje intermedio (bytecode), capaz de ser interpretado por la Máquina Virtual de Java (JVM).

Además cuenta con reconocimiento automático de código, autocompletado de código, tabulación autómática, creación de elementos visuales para interfaces gráficas, reconocimiento de errores sintácticos en tiempo real, posibilidad de añadir plugins, etc.



# **UT-03: LECTURA Y ESCRITURA DE LA INFORMACIÓN. STREAMS.**

¿Qué vamos a ver?

- Lectura y escritura de la información
- Flujos (streams)
- Tipos de flujos. Flujos de bytes y de caracteres.
- Clases relativas a flujos. Jerarquías de clases
- Utilización de flujos
  - Entrada/salida estándar
  - Entrada desde teclado

## 1. LECTURA Y ESCRITURA DE INFORMACIÓN

### 1.1. Salida de datos

La salida de datos por pantalla es una de las funcionalidades fundamentales al momento de desarrollar programas para usuarios, ya que, como programadores, probablemente necesitemos que nuestro programa se comunique con el usuario y le informe de todo lo que está sucediendo. Además, la salida de datos nos permite solicitar información al usuario.

### 1.2. Entrada de datos

Por otro lado, la entrada de datos también será una funcionalidad de gran utilidad, puesto que gracias a ella conseguiremos recolectar los datos con los que va a trabajar nuestro programa, solicitándolos al usuario, un soporte de almacenamiento u otros sistemas.

A todas las operaciones que constituyen un flujo de información del programa al exterior se les conoce como Entrada/Salida (E/S). Durante el curso veremos dos tipos de E/S:

- **E/S estándar** que se realiza con el terminal del usuario
- **E/S a través de fichero**, en la que se trabaja con fichero de disco.

En esta unidad de trabajo nos centraremos en la primera de ellas, y la segunda la veremos más adelante.

## 2. FLUJOS DE DATOS. STREAMS

Los flujos de datos son flujos de información entre el programa y el origen o destino de la información. Estos flujos son tratados en Java con objetos denominados streams.

Un stream (flujo en inglés) es un intermediario entre el programa y el origen o destino de la información.

Al utilizar los streams llevaremos a cabo los siguientes pasos:

- Lectura de la información
  - 1. Abrir un flujo de entrada
  - 2. Leer la información que contiene
  - 3. Cerrar el flujo de entrada
- Escritura de información
  - 1. Abrir un flujo de salida
  - 2. Escribir la información en el flujo de salida
  - 3. Cerrar el flujo de salida

### **3. TIPOS DE FLUJOS. FLUJOS DE BYTES Y DE CARACTERES**

A la hora de utilizar los flujos podremos optar por trabajar con flujos de bytes o con flujos de caracteres.

#### 3.1. Flujos de bytes

Los flujos de bytes nos permitirán tanto leer como escribir información en formato de bytes, sin necesidad de traducir dicha información. Estos nos serán de gran utilidad a la hora de leer o escribir información referente a texto, vídeo, audio, imágenes, etc...

#### 3.2. Flujos de caracteres

A diferencia de los flujos de bytes, estos únicamente están enfocados a la entrada y salida de datos de tipo texto, aunque en realidad siguen siendo secuencias de bytes.

## 4. CLASES RELATIVAS A FLUJOS. JERARQUÍAS DE CLASES

El paquete `java.io` contiene todas las clases relacionadas con las funciones de entrada (input) y salida (output).

Las clases de este paquete permiten deducir su función a partir de las palabras que componen sus nombres. Véase la siguiente tabla:

Palabra	Significado
<code>InputStream/OutputStream</code>	Lectura/Escritura de byte
<code>Reader/Writer</code>	Lectura/Escritura de caracteres
<code>File</code>	Archivos
<code>String, CharArray, ByteArray, StringBuffer</code>	Memoria
<code>Piped</code>	Tubo de datos
<code>Buffered</code>	Buffer
<code>Filter</code>	Filtro
<code>Data</code>	Intercambio de datos en formato Java
<code>Object</code>	Persistencia de objetos
<code>Print</code>	Imprimir

### 4.1. Jerarquías de clases Entrada/Salida de bytes

#### *4.1.1. Entrada/salida con `InputStream` y `OutputStream`*

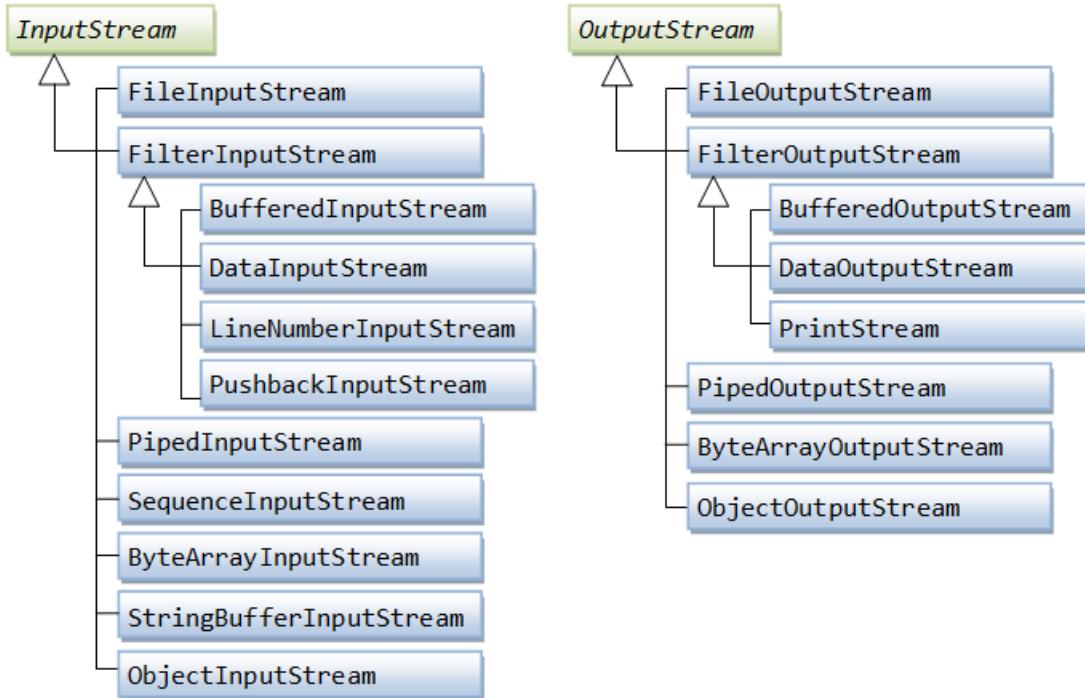
`java.lang.System` es una clase que posee multitud de pequeñas clases relacionadas con la configuración del sistema. Entre ellas están:

- `.in` que es un `InputStream` que representa la entrada estándar (normalmente el teclado). Puede utilizar los métodos:

- **read()**: devuelve el código correspondiente al carácter que se ha introducido por el teclado leyendo del buffer de entrada o “-1” si no se ha introducido ningún carácter.
- **skip(n)**: ignora los n caracteres siguientes a la entrada.
- **.out** que es un OutputStream que representa a la salida estándar (normalmente la pantalla). Puede utilizar los métodos:
  - **print(a)**: imprime a en la salida, donde a puede ser cualquier tipo primitivo de Java, ya que Java hace su conversión automática a cadena.
  - **println(a)**: es idéntico a print(a) salvo que con println() se imprime un salto de línea al final de la impresión de a.
- **.err** que es un OutputStream que representa a la salida en caso de error.

```
public static void main(String[] args) {  
    try{  
        System.out.print("Dame un carácter");  
        int caracter = System.in.read();  
        System.out.println((char)caracter);  
    } catch (Exception ex){  
        System.  
    }  
}
```

A continuación, tenemos un esquema en el que podemos ver las distintas clases con las que se puede trabajar con entrada y salida de bytes.



## 4.2. Jerarquías de clases Entrada/Salida de caracteres

### 4.2.1. Entrada/salida con InputStreamReader y OutputStreamWriter

El hecho de que las clases *InputStream* y *OutputStream* usen el tipo byte para la lectura, complica mucho su uso.

Las soluciones que se dieron fueron las clases *InputStreamReader* y *OutputStreamWriter*.

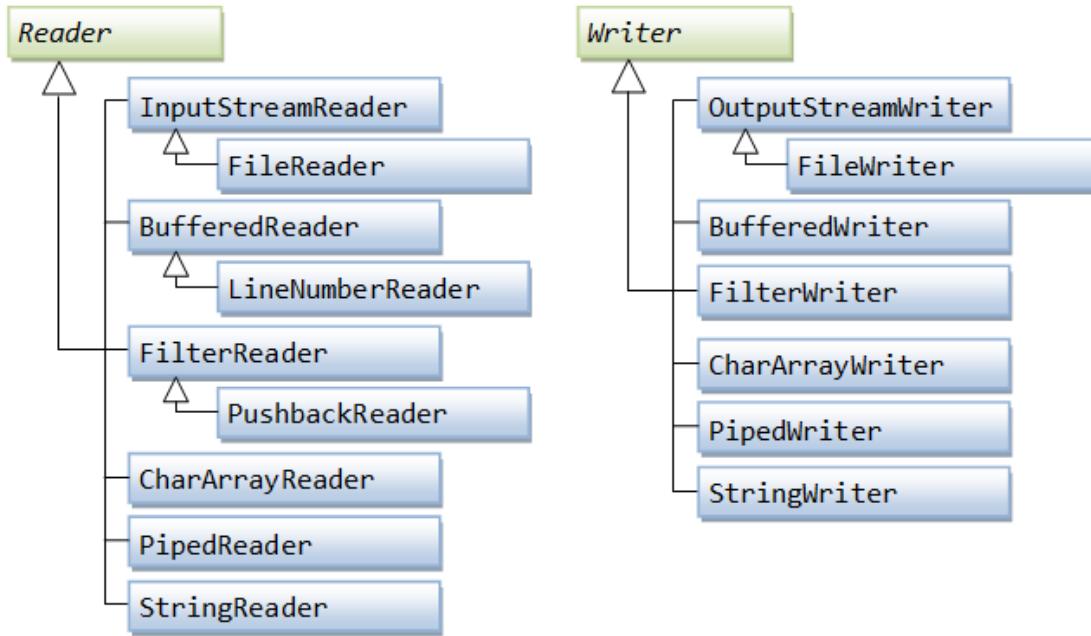
Estas clases se utilizan para convertir secuencias de bytes en secuencias de caracteres y permiten construir objetos de este tipo a partir de objetos *InputStream* u *OutputStream*.

El constructor de la clase *InputStreamReader* requiere un objeto *InputStream*.

```
InputStreamReader stdin=new InputStreamReader(System.in);
```

Estas clases trabajan con el formato de carácter UNICODE, de 16 bits.

A continuación, tenemos un esquema en el que podemos ver las distintas clases con las que se puede trabajar con entrada y salida de caracteres.



### Ejemplo de lectura usando InputStreamReader

```

//Importamos la librería de InputStreamReader para poder utilizarla
import java.io.InputStreamReader;

public class EntradaSalida {

    public static void main(String[] args) {
        try{
            //Creamos el lector escuchando de System.in
            InputStreamReader stdin = new InputStreamReader(System.in);
            //Creamos una variable capaz de almacenar hasta 1024
            caracteres
            char mensaje[] = new char[1024];
            int tamMensaje;

            //Solicitamos al usuario un mensaje
            System.out.print("Dame un mensaje: ");

            //Leemos el mensaje y almacenamos su tamaño
            tamMensaje = stdin.read(mensaje);

            //Escribimos el mensaje si la lectura no ha devuelto un error
            if (tamMensaje > -1){
                System.out.print(mensaje);
            }
        } catch (Exception ex){
    
```

```
    }  
  
}  
  
}
```

#### 4.2.2. Entrada salida con clases buffered

La palabra buffered hace referencia a la capacidad de almacenamiento temporal en la lectura y la escritura.

Los datos se almacenan en una memoria temporal antes de ser realmente leídos o escrito.

Se trata de cuatro clases:

- **BufferedInputStream**
- **BufferedOutputStream**
- **BufferedReader**
- **BufferedWriter**

Trabajan con métodos distintos, pero suelen utilizar los mismos flujos de entrada que podrán ser bytes (InputStream/OutputStream) o caracteres (Reader/Writer).

La clase BufferedReader aporta el método readLine() que permite leer caracteres hasta la presencia de null o del salto de línea y los devuelve como un String sin incluir '\n';

Este método puede lanzar una excepción java.io.Exception.

Esta clase usa un constructor que acepta objetos Reader (y por lo tanto InputStreamReader, ya que desciende de esta).

```
BufferedReader in=new BufferedReader(new InputStreamReader(System.in));  
String frase=in.readLine();
```

Ejemplo de lectura escritura con BufferedReader:

```
//Importamos la librerías BufferedReader e InputStreamReader para poder
utilizarlas
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class EntradaSalida {

    public static void main(String[] args) {

        try{
            String texto="";

            //Obtención del objeto lector
            InputStreamReader stdin = new InputStreamReader(System.in);

            //Obtención del BufferedReader
            BufferedReader bufferReader = new BufferedReader(stdin);

            //Solicitamos un texto al usuario
            System.out.print("Dame un texto: ");
            texto = bufferReader.readLine();

            //Escribe la línea de texto por pantalla
            System.out.println(texto);

        }catch (Exception ex){

        }

    }
}
```

#### 4.2.3. Entrada con clase Scanner

La clase Scanner está disponible desde Java 5, en el paquete java.util.

<https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html>

Dispone de métodos para leer valores de entrada de diferentes tipos de datos  
(nextInt(), nextDouble(), nextLine(),...)

Para utilizar la clase tenemos que instanciarla indicando el medio de

entrada de información. Si es la entrada estándar se indicará System.in o si es desde un fichero en disco, la ruta de directorios del fichero.

```
Scanner sc = new Scanner(System.in);
```

Con la instancia sc creada, podemos invocar sus métodos.

```
sc.nextLine(); Lee una línea de texto de teclado;
```

A continuación tenemos un cuadro con los distintos métodos que podemos utilizar con esta clase:

Método	Significado	Ejemplo
nextByte()	Lectura de un dato byte	byte b = sc.nextByte();
nextDouble()	Lectura de número real	double d = sc.nextDouble();
nextFloat()	Lectura de número real	float f = sc.nextFloat();
nextInt()	Lectura de número entero	int i = sc.nextInt();
nextLong()	Lectura de entero largo	long l = sc.nextLong();
nextShort()	Lectura de entero corto	short s = sc.nextShort();
next()	Lectura de String	String p = sc.next();
nextLine()	Devuelve la línea entera como un String. Lee el salto de línea \n	String o = sc.nextLine();

#### 4.2.3.1. Funcionamiento de la clase Scanner

Cuando se introducen caracteres por teclado, el objeto Scanner toma toda la cadena introducida y la divide en elementos llamados tokens.

Los tokens resultantes pueden convertirse en valores de diferentes tipos utilizando los métodos apropiados para cada tipo.

El carácter predeterminado que sirve de separador de tokens es el espacio en blanco. Para usar otro carácter diferente tendremos que indicarlo a la hora de instanciar la clase Scanner usando `useDelimiter("carácter")`;

Por ejemplo, si introducimos la cadena: 12 20.001 Lucas w

Los tokens que se crean son:

- 12
- 20.001
- Lucas
- W

Cuando en un programa se leen por teclado datos numéricos y datos de tipo carácter o String debemos tener en cuenta que al introducir los datos y pulsar intro estamos también introduciendo en el buffer de entrada el intro.

Este carácter no es leído por `nextInt`, `nextDouble`,... por lo que permanecerá en el buffer y podría ser leído por `nextLine()` en una siguiente instrucción.

La solución es poner una línea `sc.nextLine();` que limpie el buffer.

#### 4.3. Conversión a otros tipos de datos

En algunas ocasiones es conveniente leer datos por teclado como una cadena de caracteres String y realizar la conversión al tipo de dato adecuado. Para ello Java dispone de métodos en las siguientes clases:

- Clase Integer de la librería java.lang
  - `public static int parseInt(String s);` //recibe String y retorna entero
  - Ej. `int num = Integer.parseInt(sc.nextLine());`
- Clase Float de la librería java.lang
  - `public static float parseFloat(String s);` //recibe String y retorna float.
  - Ej. `float numf = Float.parseFloat(sc.nextLine());`
- Clase Double de la librería java.lang

- public static long parseDouble(String s); //recibe String y retorna double
- Ej. double numReal = Double.parseDouble(sc.nextLine());
- Clase Boolean de la librería java.lang
  - public static boolean parseBoolean(String s); //recibe String y retorna boolean
  - Ej. boolean boleano = Boolean.parseBoolean(sc.nextLine());

## 5. SALIDA DE DATOS ESTÁNDAR (POR CONSOLA)

Como ya hemos comentado anteriormente, para devolver una cadena de caracteres al flujo de datos estándar, es decir, que salga por consola debemos utilizar la clase `System.out` y sus funciones “`println()`” o “`print()`”, según necesitemos escribir con un salto de línea tras la cadena de caracteres o no.

Estas funciones reciben como parámetro un `String`, pero nos está permitido introducir un valor de un tipo de datos primitivo y Java se encargará de convertirlo a `String` mediante una conversión implícita. Por ejemplo:

```
int numeroEntero = 567;  
  
System.out.print(numeroEntero);
```

Se realizará una conversión implícita, de tal forma que `númeroEntero` se convertirá a un `String` y la consola nos devolverá “567”.

Además de la conversión implícita de datos primitivos a `String`, vamos a poder utilizar el operador “`+`” para encadenar, o concatenar, cadenas de caracteres, pudiendo construir mensajes a través de la unión de varios `Strings`, ya sean variables, constantes o literales. Por ejemplo:

```
final String inicioTexto = "Tengo ";  
  
int numAnyos = 21;
```

```
System.out.println(inicioTexto + numAnyos + " años");
```

Por consola devolverá: “Tengo 21 años”;

## 5.2. Aplicar formato a la salida estándar

En ocasiones desearemos aplicar un formato a la salida por consola. Esto suele ocurrir principalmente con salida de numéricas.

Para esto haremos uso de la función “format” de la clases System.out.

```
System.out.format(String formato, Object... args)
```

**formato**: es una cadena que especifica el formato que debe utilizarse.

**args**: es una lista de las variables que desea imprimir utilizando ese formato. Por ejemplo:

```
System.out.format("El valor de la variable float es %f, mientras que el  
valor de la variable entera es %d y la cadena es %s", VarReal, VarEntera,  
VarCadena);
```

La cadena de formato contiene unos caracteres especiales que dan formato a los argumentos de Object... args. Son **especificadores de formato** que comienzan con un signo de porcentaje (%) y un carácter que indica el tipo de argumento para ser formateado.

- **%d**: formatea el valor entero (int) como un valor decimal.
- **%f**: da formato a un valor en coma flotante(float, double) como valor decimal.
- **%n**: da salida a un final de línea específica de la plataforma.
- **%x**: formatea un entero como un valor hexadecimal.
- **%s**: formatea cualquier valor como una cadena.

Además, puede presentar unos elementos adicionales de manera opcional. Estos elementos deben aparecer en el orden que se indica. Trabajando desde la derecha, los elementos opcionales son:

- **%**: Inicio de especificador de formato
- **argumentIndex\$**: indica el argumento que estamos utilizando mediante su posición en el listado de argumentos.
- **+**: muestra el signo positivo en número positivos.
- **0**: rellena con “0” a la izquierda hasta alcanzar la longitud que se le ha indicado.
- **numeroAncho**: indica la longitud de dígitos que debe tener.
- **.numeroDecimales**: indica el número de decimales que debe mostrarse a la derecha del punto.
- **especificadorFormato**: indica el formato final que debe darse (listado ya visto anteriormente)

```
long n = 461012;  
System.out.format("%d%n", n); // --> "461012"  
System.out.format("%08d%n", n); // --> "00461012"  
System.out.format("%+8d%n", n); // --> "+461012"  
double pi = Math.PI;  
System.out.format("%f%n", pi); // --> "3.141593"  
System.out.format("%.3f%n", pi); // --> "3.142"  
System.out.format("%10.3f%n", pi); // --> " 3.142"
```

Ejemplos con varios argumentos:

```
int edad = 20;  
String nombre = Lola;  
System.out.format("La persona se llama %s y tiene %d años\n", nombre,  
edad);
```

Resultado: “La persona se llama Lola y tiene 20 años”

```
int cantidad = 10;
String producto = "mesas";
double precio = 125.55;
double total = cantidad * precio;
System.out.format("Para una cantidad de %d %s el total de la compra es
%.2f \n", cantidad, producto, total);
```

Resultado: “Para una cantidad de 10 mesas el total de la compra es 1255,50”

### 5.3. Caracteres de barra invertida

Existen una serie de caracteres especiales denominados de barra invertida o literales especiales que se componen del símbolo de barra invertida “\” y otro carácter adicional al que este da un significado especial al escribir un String. A continuación, podemos observar de cuales se trata y para qué se utilizan.

## Caracteres literales especiales

Códigos	Significado
'\n'	Nueva línea
'\r'	Retorno de carro
'\t'	Tabulación
'\"	Comilla simple
'\\'	Comilla doble
'\\'	Barra inclinada inversa

Ejemplo:

```
System.out.print("Hola me llamo Javier, pero me puedes llamar  
\"Javi\"\n");  
System.out.print("¿Cuál es tu nombre?\n");
```

Resultado:

Hola me llamo Javier, pero me puedes llamar “Javi”

¿Cuál es tu nombre?

# UT-04: Estructuras de control

¿Qué vamos a ver?

- Sentencias condicionales
  - Condicional simple: IF
  - Condicional doble: IF – ELSE
  - Anidación de sentencias condicionales
  - Condicional múltiple SWITCH-CASE
- Sentencias iterativas
  - WHILE
  - DO-WHILE
  - FOR
- Sentencias de salida de un bucle
  - Break
  - Continue
- Prueba, depuración y documentación de la aplicación

## 1. SENTENCIAS CONDICIONALES

Las sentencias condicionales determinan si se ejecuta una instrucción, o un conjunto de instrucciones, en función de si se cumple, o no se cumple, una condición.

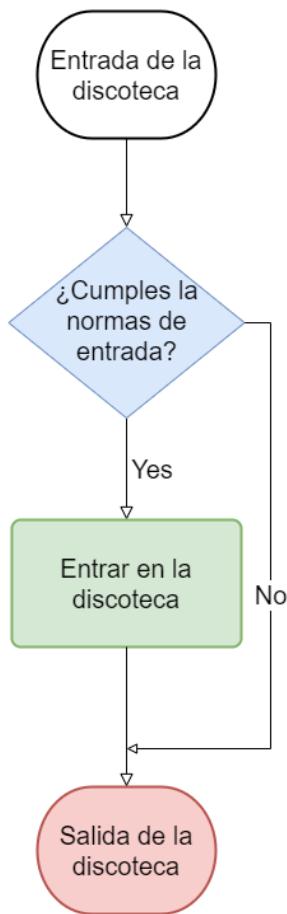
Existen con tres tipos de sentencias condicionales, en función del número de posibles valores que pueda tomar la condición:

- Simples
- Dobles
- Múltiple

### 1.1. Sentencia condicional simple (if)

Las sentencias condicionales simples son las que ejecutan una instrucción o un conjunto de instrucciones únicamente si se cumple una determinada condición.





En el lenguaje de programación Java, la sentencia condicional simple está representada por la instrucción “if”.

```
if (condición) {
```

instrucción o bloque de instrucciones a ejecutar

```
}
```

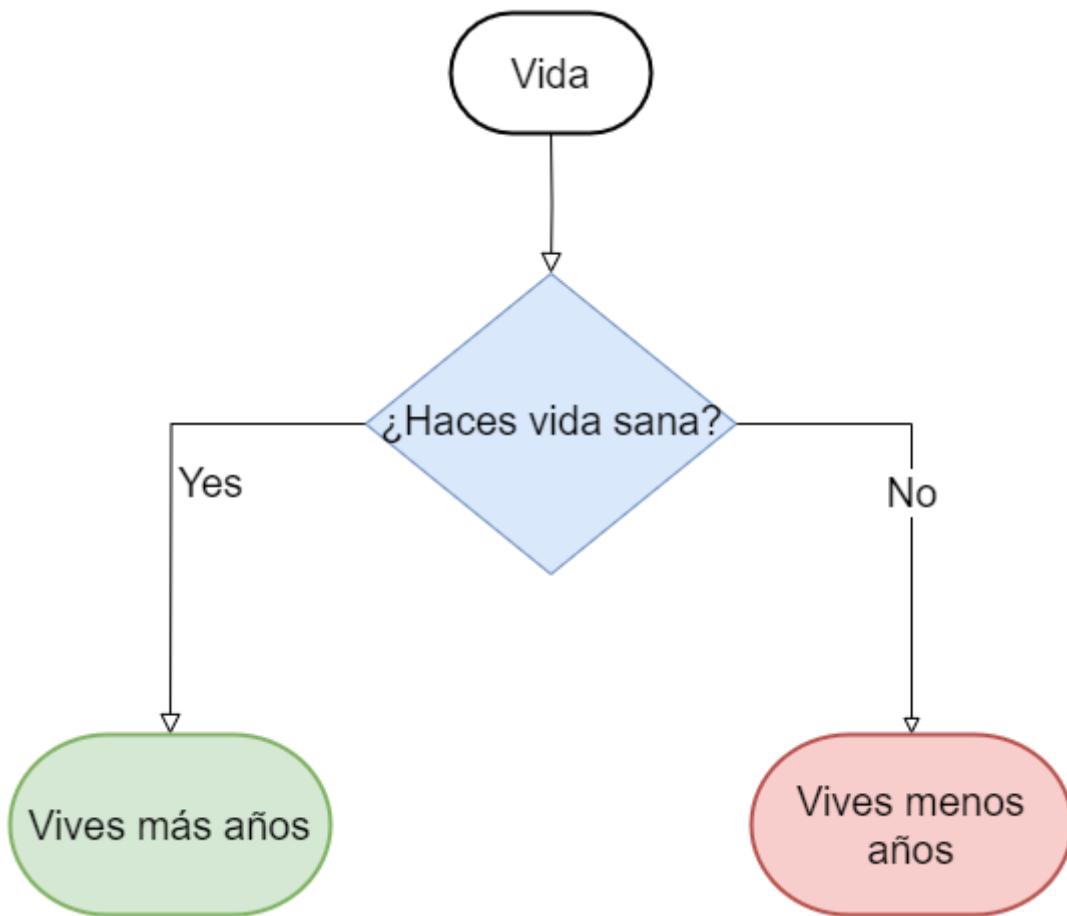
```
if (numDiaSemana == 6) {  
    System.out.println("¡Es sábado!");  
}
```

Cuando sólo vamos a ejecutar una única instrucción dentro del if, podemos prescindir de los corchetes.

```
if (numDiaSemana == 6)  
    System.out.println("¡Es sábado!");
```

### 1.1. Sentencia condicional simple (if-else)

Las sentencias condicionales dobles son las que ejecutan una instrucción o un bloque de instrucciones si se cumple una determinada condición, y si no se cumple se ejecuta otra instrucción u otro bloque de instrucciones.



Vamos a verlo ejemplificado en un vídeo...

Para realizar esta acción en Java vamos a añadir un complemento a la instrucción if, el else. De tal forma, que si se cumple la condición se ejecutará la parte dentro del if y si no se cumple se ejecutará la parte del else.

```
if (condición) {  
    bloque de instrucciones a ejecutar si se cumple la condición  
}  
  
else {  
    bloque de instrucciones a ejecutar si no se cumple la condición  
}
```

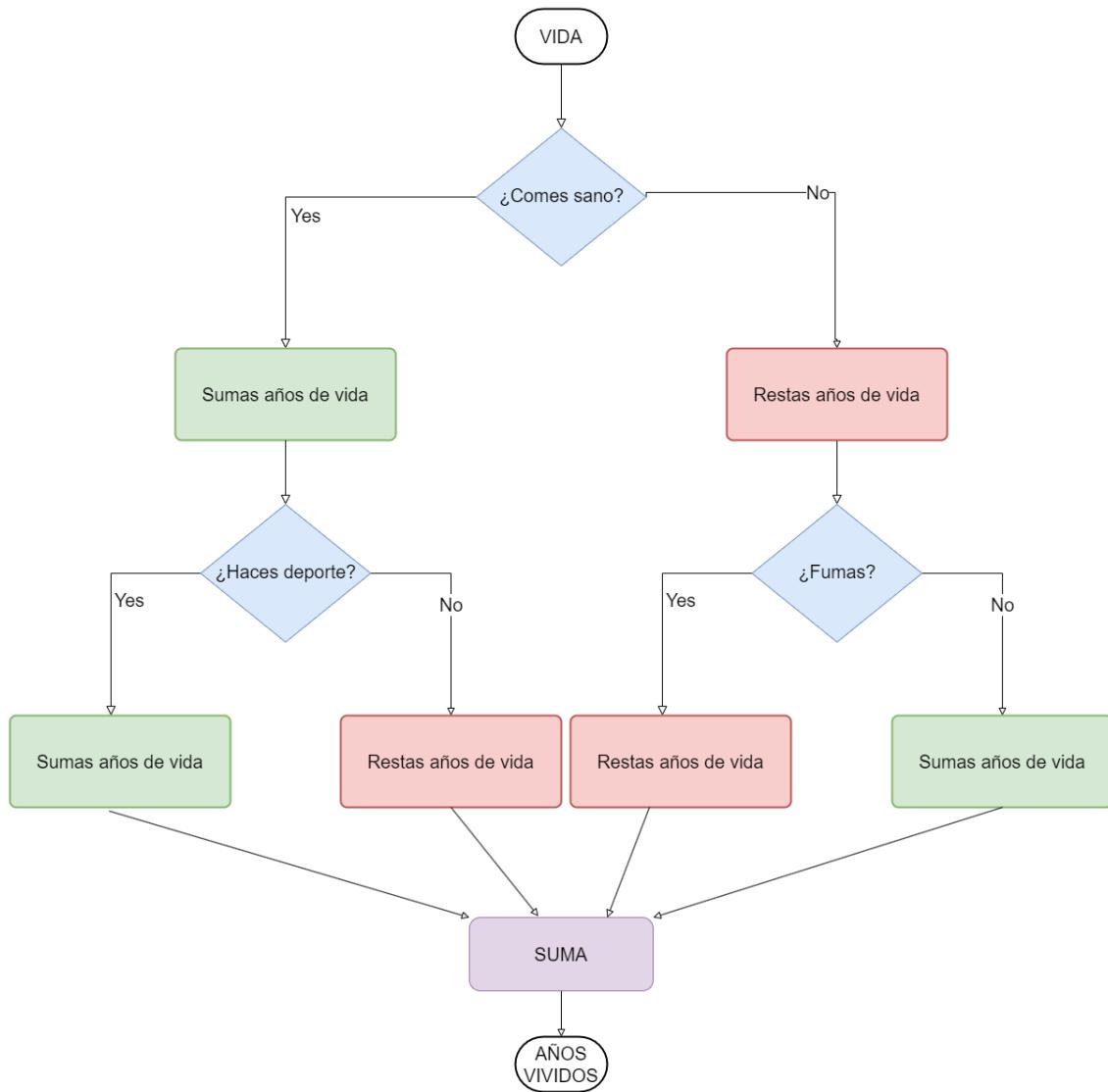
```
if (numero % 2 == 0){  
    System.out.println(numero + " es par");  
}  
else {  
    System.out.println(numero + " es impar");  
}
```

Cuando sólo vamos a ejecutar una única instrucción dentro del if, podemos prescindir de los corchetes.

```
if (numero % 2 == 0)  
    System.out.println(numero + " es par");  
else  
    System.out.println(numero + " es impar");
```

## 1.2. Anidamiento de sentencias condicionales

En ocasiones necesitaremos utilizar sentencias condicionales dentro de otras sentencias condicionales.



Para representar este anidamiento en Java solo tendremos que introducir una instrucción if dentro de otra instrucción if o else:

```

if (num1!=num2){
    if (num1 > num2){
        System.out.println(num1 + " es mayor que " + num2);
    }
    else {
        System.out.println(num1 + " es menor que " + num2);
    }
} else {
    System.out.println("Los dos números son iguales");
}
  
```

```
if (num1==num2){  
    System.out.println("Los dos números son iguales");  
} else {  
    if (num1 > num2){  
        System.out.println(num1 + " es mayor que " + num2);  
    }  
    else {  
        System.out.println(num1 + " es menor que " + num2);  
    }  
}
```

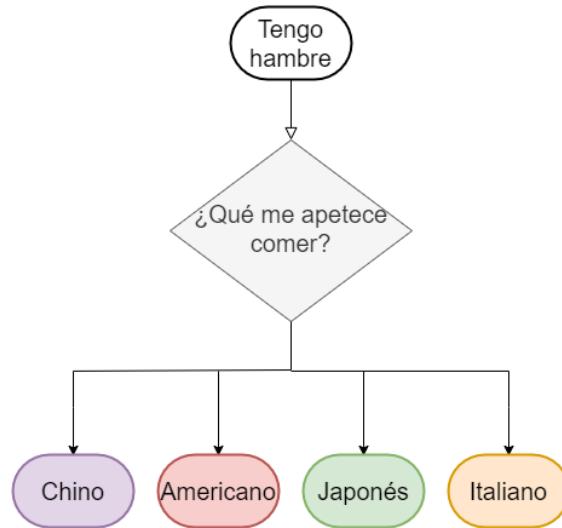
En el caso de incluir dentro de un else un nuevo if, podremos hacerlo de la siguiente manera:

```
if (num1==num2){  
    System.out.println("Los dos números son iguales");  
} else if (num1 > num2){  
    System.out.println(num1 + " es mayor que " + num2);  
}  
else {  
    System.out.println(num1 + " es menor que " + num2);  
}
```



### 1.3. Condicional múltiple SWITCH-CASE

En determinadas ocasiones tendremos varias opciones, no solo dos, como resultado de una expresión o condición.



En estos casos podríamos utilizar anidamientos de expresiones condicionales, pero el código quedaría bastante engorroso, por ejemplo:

```

System.out.print("Dame un número de 0 a 3: ");
num = scan.nextInt();

if (num==0){
    System.out.println("El número es 0");
}
else {
    if (num==1) {
        System.out.println("El número es 1");
    }
    else {
        if (num==2) {
            System.out.println("El número es 2");
        }
        else {
            if (num==3) {
                System.out.println("El número es 3");
            }
        }
    }
}
  
```

Para evitar esta situación, contamos una sentencia condicional múltiple en Java denominada SWITCH-CASE, de tal forma que la traducción del anterior código a este sería el siguiente:

```
switch (num){  
    case 0:  
        System.out.println("El número es 0");  
        break;  
    case 1:  
        System.out.println("El número es 1");  
        break;  
    case 2:  
        System.out.println("El número es 2");  
        break;  
    case 3:  
        System.out.println("El número es 3");  
        break;  
}
```

Esto nos permitirá tener un código mucho más claro.

La instrucción SWITCH-CASE consiste en proporcionar la expresión que queremos evaluar en el paréntesis después de la palabra reservada “switch” y, a continuación, le pondremos por cada opción la palabra reservada “case” seguido del valor al que corresponde dicha opción. Seguiremos con un signo de dos puntos (“：“) y escribiremos el bloque de código que necesitemos ejecutar cuando obtengamos dicha opción. Finalizaremos dicho bloque con la palabra reservada “break”.

Además, esta instrucción nos permite contemplar la opción de que no se cumpla ninguna de las que hemos definido mediante la palabra reservada “default”, la cual utilizaremos como una opción más y donde introduciremos el bloque de código que queremos que se ejecute si no se cumple ninguna de las opciones anteriores.

```
switch (num){  
    case 0:  
        System.out.println("El número es 0");  
        break;  
    case 1:  
        System.out.println("El número es 1");  
        break;  
    case 2:  
        System.out.println("El número es 2");  
        break;  
    case 3:  
        System.out.println("El número es 3");  
        break;  
    default:  
        System.out.println("El número proporcionado no es correcto");  
        break;  
}
```

Veamos un pequeño vídeo para entender mejor cómo funciona la sentencia switch case.

La sentencia switch case nos permite ejecutar un bloque de instrucciones para varias opciones. Para hacer esto, tenemos las dos siguientes opciones:

```
switch (piso){  
    case 1, 2, 3:  
        System.out.println("El piso es bajo");  
        break;  
    case 4, 5, 6:  
        System.out.println("El piso es medio");  
        break;  
    case 7, 8, 9:  
        System.out.println("El piso es alto");  
        break;  
}
```

```
switch (piso){  
    case 1:  
    case 2:  
    case 3:  
        System.out.println("El piso es bajo");  
        break;  
    case 4:  
    case 5:  
    case 6:  
        System.out.println("El piso es medio");  
        break;  
    case 7:  
    case 8:  
    case 9:  
        System.out.println("El piso es alto");  
        break;  
}
```

También podemos hacer que una opción ejecute varios bloques de instrucciones seguido retirando la palabra reservada “break” entre ellas.

```
switch (opcion){  
    case 0:  
        System.out.println("Hola ¿qué tal?");  
    case 1:  
        System.out.println("¿Cómo estás?");  
        break;  
    case 2:  
        System.out.println("Yo estoy bien");  
    case 3:  
        System.out.println("¿Y tú?");  
        break;  
    default:  
        System.out.println("Nada que decir");  
        break;  
}
```

En este ejemplo las salidas, según el valor de opción, serían:

- 0 → “Hola ¿Qué tal?  
“¿Cómo estás?”
- 1 → “¿Cómo estás?
- 2 → “Yo estoy bien”  
“¿Y tú?”

- 3 → “¿Y tú?”
- Default → “Nada que decir”

## 2. SENTENCIAS ITERATIVAS

Las sentencias iterativas nos permitirán repetir bloques de instrucciones mientras, o hasta, que se cumpla una condición un número determinado o indeterminado de veces.

Son instrucciones con una gran potencia y que nos permitirán proporcionar gran flexibilidad a nuestros programas combinadas con las sentencias condicionales, pero también debemos tener cuidado a la hora de definirlos, puesto que podemos no contemplar todas las posibles condiciones y provocar que entremos en un bucle infinito. Esto provocará que nuestro programa no salga nunca del bucle y por tanto no llegue a finalizar.

Mira el vídeo para entender a qué nos referimos...

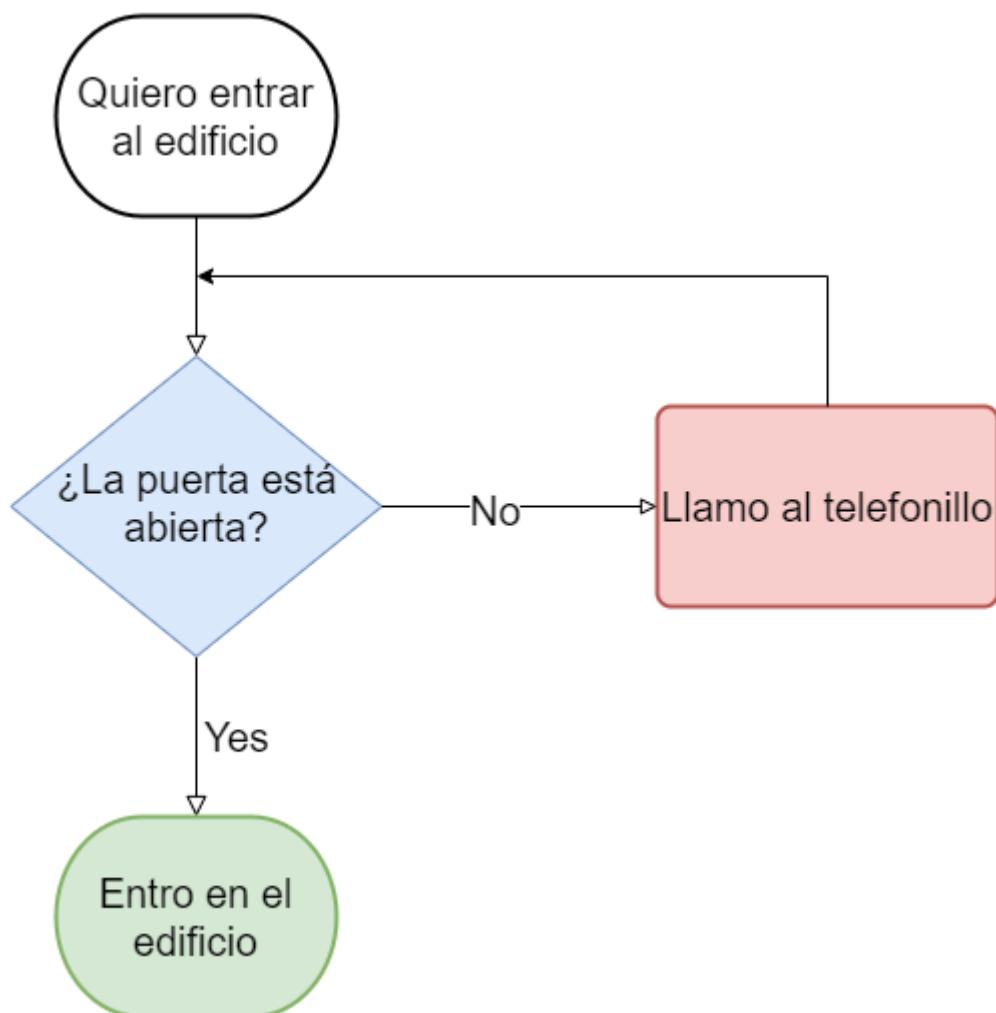
Dentro de este tipo de sentencias Java disponemos de tres instrucciones:

- While
- Do ... While
- For

### 2.1. While

La instrucción while consiste en realizar la comprobación de una condición y en el caso de cumplirse se ejecuta el bloque de código que tiene en su interior. Una vez ejecutado el bloque vuelve a comprobarse la condición, de tal forma que si se vuelve a cumplir, vuelve a ejecutar el bloque de código para luego volver a comprobar la condición, y así hasta que no se cumpla dicha condición y saltaremos a la siguiente instrucción.

Debemos tener en cuenta que durante la primera comprobación de la condición esta no se cumple, nunca se ejecutará el bloque de instrucción dentro del while.



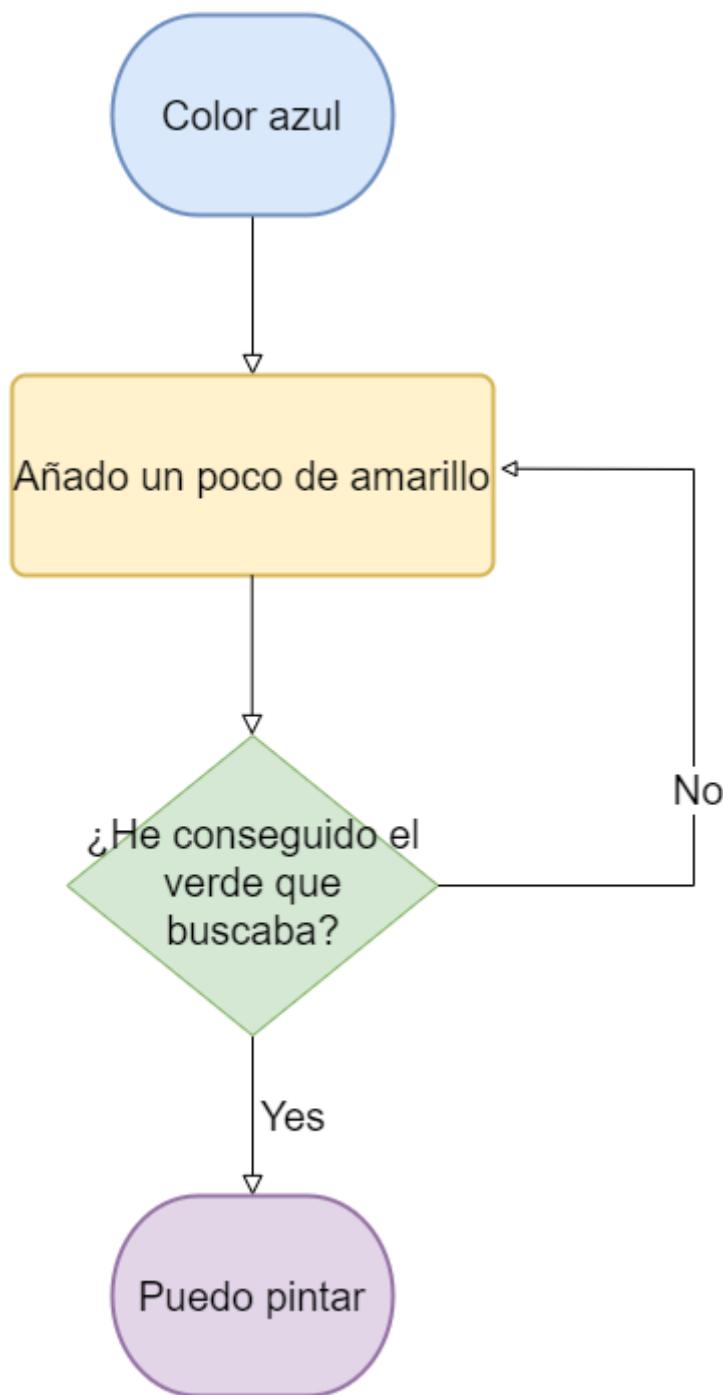
La estructura de instrucción while es la siguiente:

```
while (condición) {  
    bloque de instrucciones  
}
```

```
int num = 5;  
while (num < 10){  
    System.out.println(num);  
    num++;  
}
```

## 2.2. Do ... While

La instrucción do ... while es similar a la instrucción while, con la única diferencia de que la comprobación de la condición se realiza al finalizar la ejecución del bloque de código que contiene. De esta forma siempre se ejecutará al menos una vez el bloque de instrucción dentro del do ... while.



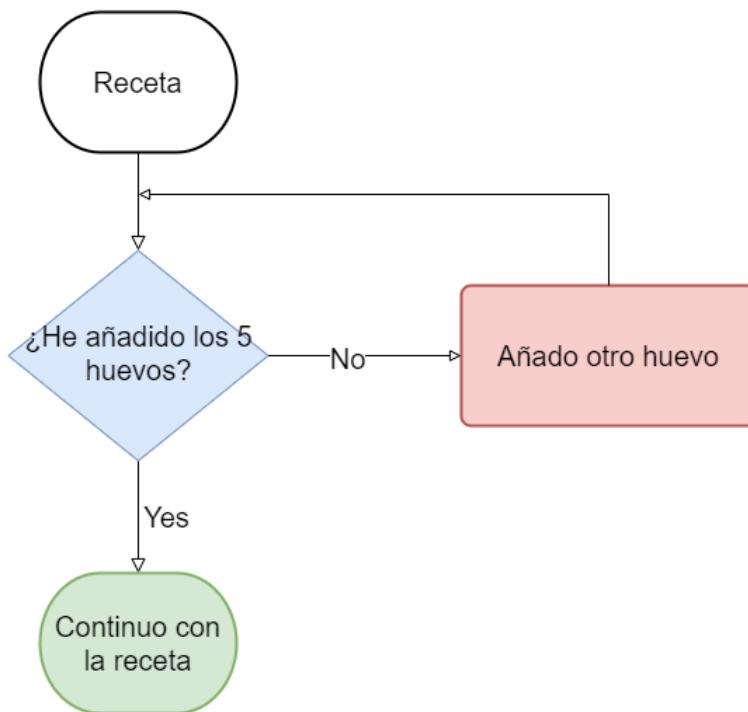
La instrucción do ... while se estructura de la siguiente manera:

```
do {
    bloque de código
} while (condición);
```

```
num = 100;
do {
    System.out.println(num);
    num--;
} while (num >= 0);
```

### 2.3. For

La instrucción for es utilizada, normalmente, cuando conocemos las veces que se va a ejecutar el bucle. Esta instrucción es particular puesto que debe de una variable que inicializaremos y que utilizaremos como contador. Al igual que la instrucción while, la comprobación de la condición se realiza previamente a la ejecución del bloque de instrucciones que contiene y, por tanto, puede ocurrir que dicho bloque no se ejecute nunca.



La instrucción for tiene la siguiente estructura:

```
for (int contador = valorInicial; condición; modificaciónContador){
```

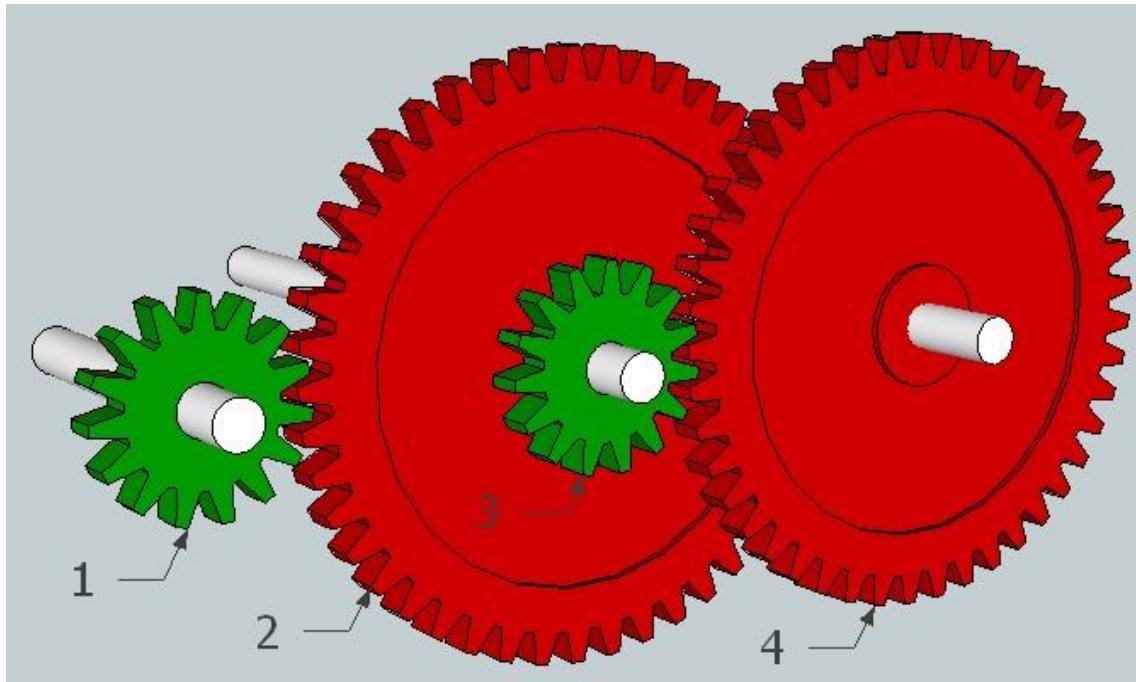
bloque de código

```
}
```

```
for (int i = 1; i <= 10; i++){  
    System.out.println("Escrito " + i + " veces");  
}
```

## 2.4. Bucles anidados

Al igual que las sentencias condicionales también podemos anidar bucles, del mismo tipo o de distinto tipo.



Estos bucles pueden ser de dos tipos:

#### 2.4.1. Bucles independientes

En esta modalidad los bucles no están relacionados entre sí, es decir, que el funcionamiento de uno de afecte al otro.



```
int contWhile = 0;
int contDoWhile = 5;
while (contWhile <= 50){
    do{
        System.out.println(contDoWhile + " ");
        contDoWhile--;
    } while (contDoWhile >= 0);
    System.out.println();
    contWhile++;
}
```

#### 2.4.2. Bucles dependientes

En esta modalidad los bucles están relacionados, de tal forma que el funcionamiento de uno afecta al otro.

TABLA MULTIPLICAR				
1 x 1 = 1	2 x 1 = 2	3 x 1 = 3	4 x 1 = 4	5 x 1 = 5
1 x 2 = 2	2 x 2 = 4	3 x 2 = 6	4 x 2 = 8	5 x 2 = 10
1 x 3 = 3	2 x 3 = 6	3 x 3 = 9	4 x 3 = 12	5 x 3 = 15
1 x 4 = 4	2 x 4 = 8	3 x 4 = 12	4 x 4 = 16	5 x 4 = 20
1 x 5 = 5	2 x 5 = 10	3 x 5 = 15	4 x 5 = 20	5 x 5 = 25
1 x 6 = 6	2 x 6 = 12	3 x 6 = 18	4 x 6 = 24	5 x 6 = 30
1 x 7 = 7	2 x 7 = 14	3 x 7 = 21	4 x 7 = 28	5 x 7 = 35
1 x 8 = 8	2 x 8 = 16	3 x 8 = 24	4 x 8 = 32	5 x 8 = 40
1 x 9 = 9	2 x 9 = 18	3 x 9 = 27	4 x 9 = 36	5 x 9 = 45
1 x 10 = 10	2 x 10 = 20	3 x 10 = 30	4 x 10 = 40	5 x 10 = 50
6 x 1 = 6	7 x 1 = 7	8 x 1 = 8	9 x 1 = 9	10 x 1 = 10
6 x 2 = 12	7 x 2 = 14	8 x 2 = 16	9 x 2 = 18	10 x 2 = 20
6 x 3 = 18	7 x 3 = 21	8 x 3 = 24	9 x 3 = 27	10 x 3 = 30
6 x 4 = 24	7 x 4 = 28	8 x 4 = 32	9 x 4 = 36	10 x 4 = 40
6 x 5 = 30	7 x 5 = 35	8 x 5 = 40	9 x 5 = 45	10 x 5 = 50
6 x 6 = 36	7 x 6 = 42	8 x 6 = 48	9 x 6 = 54	10 x 6 = 60
6 x 7 = 42	7 x 7 = 49	8 x 7 = 56	9 x 7 = 63	10 x 7 = 70
6 x 8 = 48	7 x 8 = 56	8 x 8 = 64	9 x 8 = 72	10 x 8 = 80
6 x 9 = 54	7 x 9 = 63	8 x 9 = 72	9 x 9 = 81	10 x 9 = 90
6 x 10 = 60	7 x 10 = 70	8 x 10 = 80	9 x 10 = 90	10 x 10 = 100

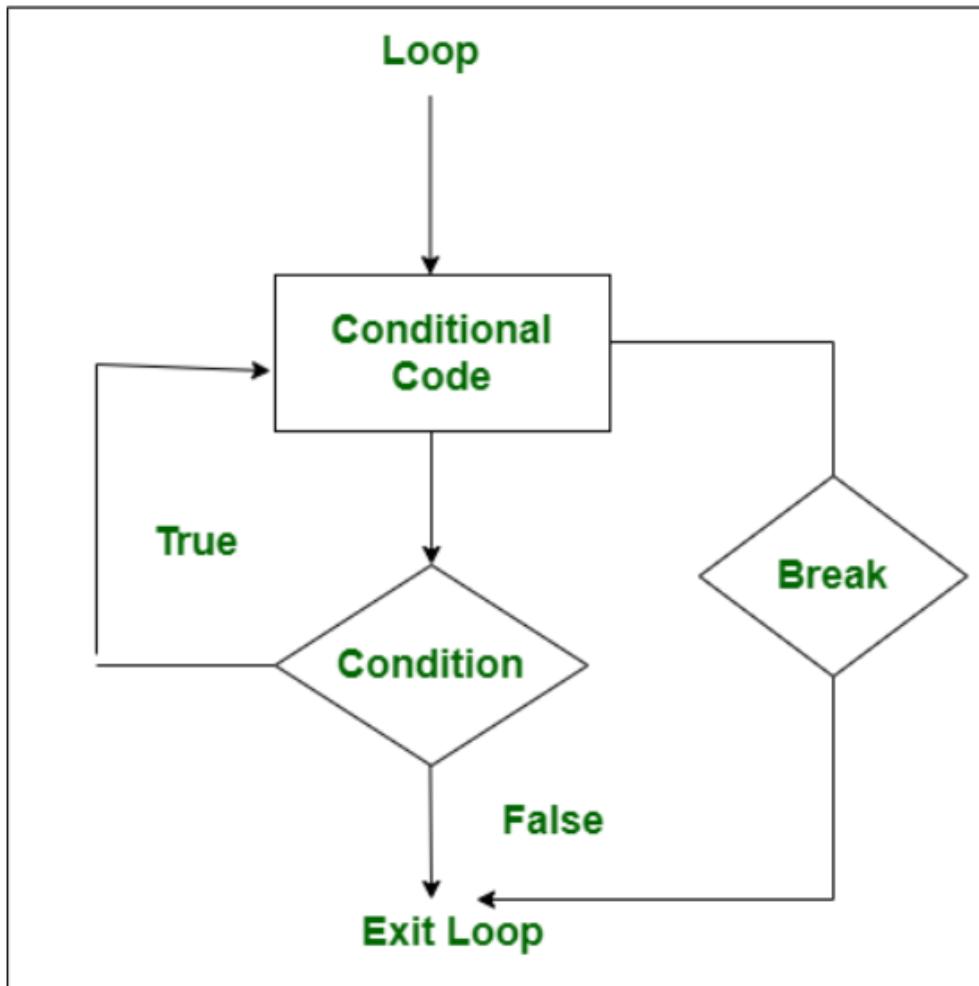
```
for(int i = 10; i >= 0 ; i--){
    for (int j = i; i <= 10; j++){
        System.out.print(j + " ");
    }
    System.out.println();
}
```

## 4. SENTENCIAS DE SALTO

Las sentencias de salto nos serán muy útiles cuando necesitemos interrumpir la ejecución de un bucle o saltar parte de su código para volver a comprobar la condición. En java disponemos de las instrucciones break y continue. Estas dos instrucciones podremos utilizarlas en cualquiera de los tres tipos de sentencias iterativas que hemos visto anteriormente.

#### 4.1. Break

Con esta instrucción finalizaremos inmediatamente el bucle y continuaremos en la línea siguiente a la que finaliza este.

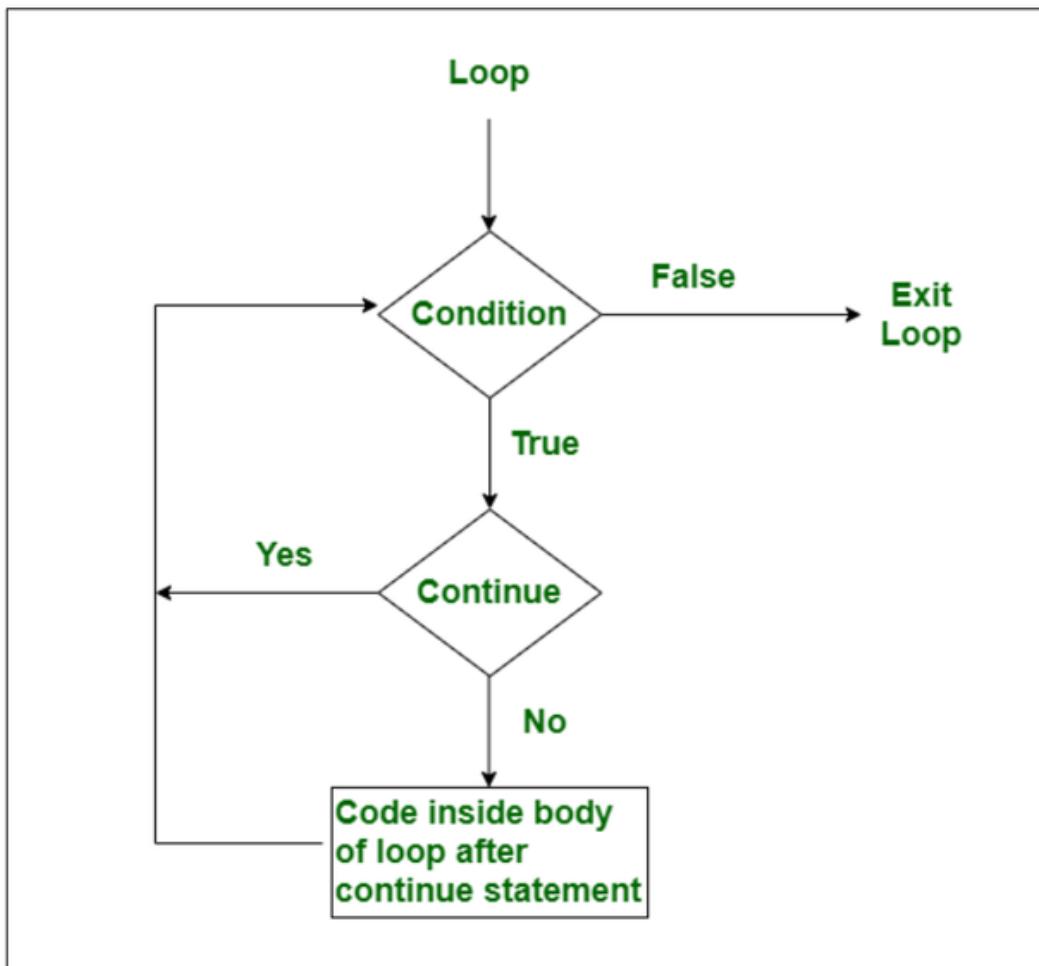


La forma de utilizarlo es simplemente escribiendo la instrucción “break;”.

```
num = 5;
while (num < 10){
    System.out.println(num);
    if (num == 8) {
        break;
    }
    num++;
}
System.out.println("He llegado hasta el número " + num);
```

## 4.2. Continue

Con la instrucción continue saltaremos el resto del contenido del bloque de instrucciones dentro del bucle y volveremos a comprobar la condición. En el caso de la instrucción for también realizará la modificación del contador correspondiente a la iteración.



La forma de utilizarlo es simplemente escribiendo la instrucción “continue;”.

```

for(int i=0; i<=10; i++){
    if (i%2 == 0){
        continue;
    }
    System.out.println("Voy por el " + i);
}
  
```

## 5. PRUEBAS, DEPURACIÓN Y DOCUMENTACIÓN DE LA APLICACIÓN

### 5.1. Pruebas de programas

Cuando se habla de pruebas de software o de programa es común utilizar los términos “prueba” y “caso de prueba”. Su definición es la siguiente:

- **Prueba:** Es el proceso de ejecución de un programa con el intento deliberado de encontrar errores.
- **Caso de prueba:** Conjunto de entradas, condiciones de ejecución y resultados esperados diseñados para un objetivo particular de condición de prueba.

### 5.2. Errores de los programas

Durante el proceso de desarrollo de software, se pueden producir dos tipos de errores:

- **Errores de compilación:** se corresponde con el incumplimiento de reglas sintácticas del lenguaje de programación utilizado. Ocasiona que el programa no puede ejecutarse hasta que el programador no corrija ese error. Ejemplos de errores de compilación:
  - Palabras reservadas mal escritas.
  - Expresiones erróneas o incompletas.
  - Variables no declaradas.

Son detectados por el propio entorno de desarrollo en tiempo real durante su escritura o durante la compilación del código.

- **Errores lógicos o de ejecución:** comúnmente llamados bugs. Estos no evitan que el programa se pueda compilar con éxito, ya que no hay errores sintácticos y permiten su ejecución. Sin embargo, pueden provocar que el programa devuelva resultados erróneos, que no sean los esperados o pueden provocar que el programa termine antes de tiempo o no termine nunca. Ejemplos de errores lógicos o de ejecución son:

- Dividir un número por cero.
- Exceder un rango de valores posible en una variable o una estructura de datos.
- Crear un bucle infinito.

Para la localización y corrección de estos errores se usa una herramienta de gran utilidad, como es el depurador.

### 5.3. El depurador de código

El depurador permite supervisar la ejecución de los programas, para localizar y eliminar los errores lógicos. Un programa debe compilarse con éxito para posteriormente poder utilizarlo en el depurador. El depurador nos permite analizar todo el programa, mientras éste se ejecuta. Permite suspender la ejecución de un programa, examinar y establecer los valores de las variables, comprobar los valores devueltos por un determinado método, el resultado de una comparación lógica o relacional, etc.

De este modo, el depurador permite analizar el flujo de ejecución del código y el estado de los datos conforme van siendo manipulados por el programa.

Para poder depurar un programa, podemos ejecutar el programa de diferentes formas, de manera que en función del problema que se quiera solucionar, resulte más sencillo un método u otro.

Existen los siguientes tipos de ejecución:

- **Paso a paso por instrucción:** algunas veces es necesario ejecutar un programa línea por línea para buscar y corregir errores lógicos.
- **Paso a paso por procedimiento:** permite introducir los parámetros que queremos a un método o función de nuestro programa, pero en vez de ejecutar instrucción por instrucción ese método, nos devuelve su resultado. Es útil, cuando hemos comprobado que un procedimiento funciona correctamente, y no nos interesa volver a depurarlo, solo nos interesa el valor que devuelve.

- **Ejecución hasta una instrucción:** el depurador ejecuta el programa y se detiene en la instrucción donde se encuentra el cursor, a partir de ese punto podemos hacer una depuración paso a paso o por procedimiento.
- **Ejecutar hasta el final del programa:** se ejecuta las instrucciones de un programa hasta el final, sin detenerse en las instrucciones intermedias.

```
1 package Depurar;
2
3 import java.util.Random;
4
5 /* @author (Dario José Zubarray) */
6
7 public class Depurar {
8     public static void main(String[] args) {
9         System.out.println(sumar(5, 6));
10    }
11    public static int sumar(int num1, int num2) {
12        int aux = num1 - num2;
13        return aux;
14    }
15 }
```

Name	Type	Value
<Enter new watch>		
+ Static		
num1	int	5
num2	int	6
aux	int	-1

## 5.4. Documentación

Para documentar nuestros programas lo realizaremos mediante comentarios, siguiendo las mismas indicaciones que dimos en su momento en la unidad de trabajo número 2.

# **UT-04: ESTRUCTURAS DE DATOS. ARRAYS Y CADENAS DE CARACTERES**

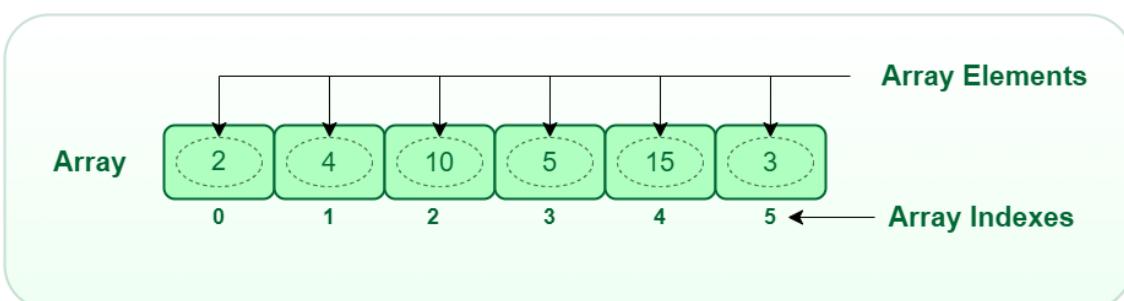
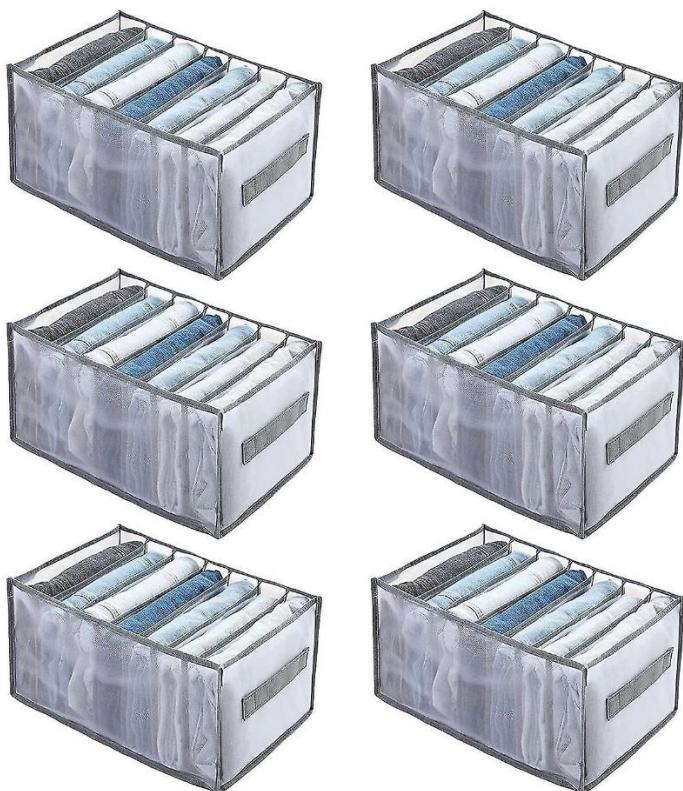
¿Qué vamos a ver?

- Arrays unidimensionales
  - Declarar y crear un array
  - Inicializar un array
  - Acceder a los elementos de un array
- Operaciones
  - Recorrido
  - Copia
  - Ordenación
  - Comparación
  - Búsqueda
  - Inserción de elementos
  - Eliminación de elementos
  - La clase Arrays
- Arrays bidimensionales o matrices
  - Declarar y crear una matriz
  - Inicializar una matriz
  - Acceder a los elementos de una matriz
- Cadenas de caracteres. String
  - Métodos de la clase string
  - Formato de cadenas en java
  - Operaciones
    - Acceso a elementos
    - Conversiones
    - Concatenaciones
  - Expresiones regulares

## 1. ARRAYS UNIDIMENSIONALES

Los arrays unidimensionales son tipos de datos lineales, compuestos, ordenados y homogéneos, es decir, que una variable de tipos array almacena varios valores, no solo uno, como habíamos visto hasta el momento y todos son del mismo tipo (int, float, char, String, ...). Además, cada uno de estos valores ocupa una determinada posición dentro del array, identificada por un índice y al ser lineales cada elemento tiene como máximo otro elemento que le preceda y otro que le siga.

Con una imagen lo vemos de una forma más sencilla:



### 1.1. Declaración e inicialización

En java podemos declarar e inicializar los arrays unidimensionales de dos maneras:

1. Solo declarar un array vacío de la longitud que deseamos

```
int edades[] = new int[3];  
int[] edades = new int[3];
```

En este caso inicializaremos los valores más adelante. Para ello, haremos uso de los índices que apunta a cada una de las posiciones del array. Algo MUY IMPORTANTE que debemos tener en cuenta es que los índices de LOS ARRAYS SIEMPRE EMPIEZAN A CONTAR DESDE 0, de tal forma que el índice del último elemento será el de la longitud del arrays menos 1.

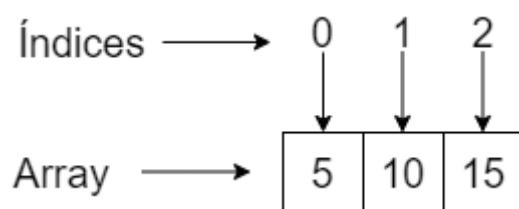
Para inicializar el valor a cada una de las posiciones del array haremos lo siguiente:

```
int edades[];  
edades[0] = 5;  
edades[1] = 10;  
edades[2] = 15;
```

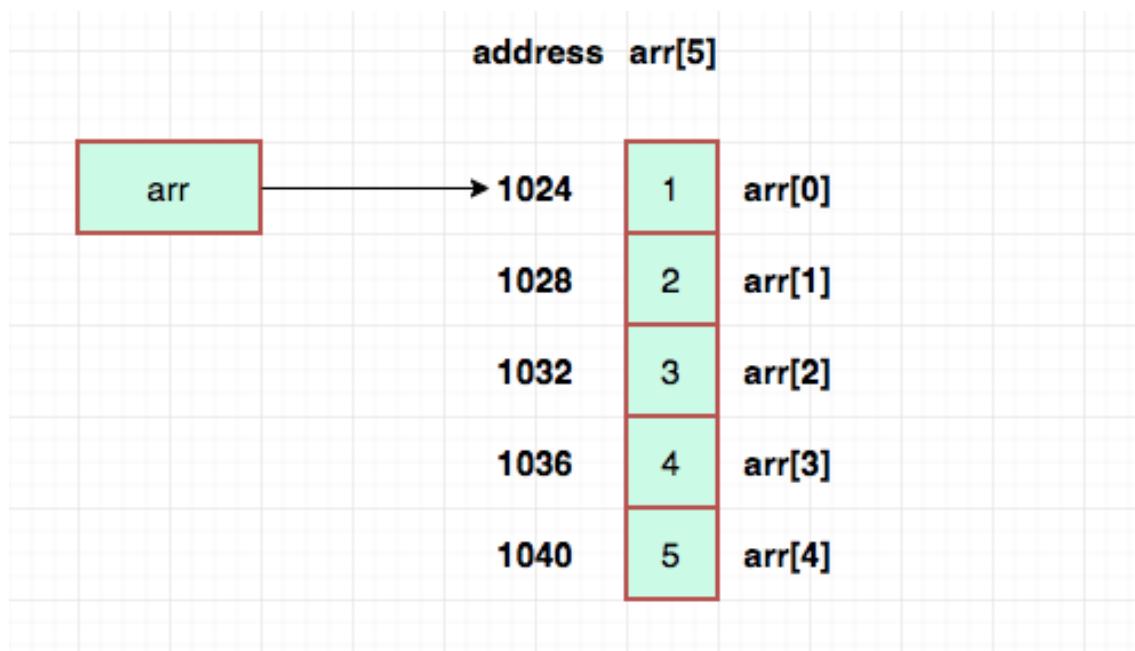
2. Declarar un array e inicializando su contenido directamente y de esta forma le damos de una forma implícita el tamaño que deseamos.

```
int edades[] = {5, 10, 15}  
int[] edades = {5, 10, 15};
```

Ambos casos se pueden representar de la siguiente manera:



Cuando nosotros creamos una array, ya sea por medio de la instrucción new o inicializando directamente, nuestra variable lo que almacena es una referencia, o puntero, a la dirección donde se almacena primera posición del array en la memoria:



*How we perceive an array:*

*Array:*

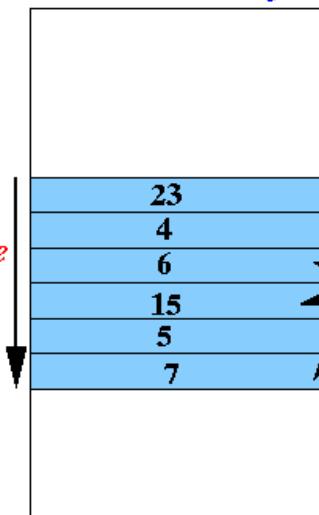
23	4	6	15	5	7
0	1	2	3	4	5

*How it is stored in memory*

*RAM memory*

*Consecutive  
memory  
locations*

*Same data type*



## 1.2. Acceder a los elementos de un array

Para acceder individualmente a cada elemento del array solo debemos utilizar su índice, de tal forma que escribimos el nombre del array seguido del índice al que queremos acceder, poniéndolo entre corchetes (nombreArray[índice]).

Retomando el ejemplo anterior imaginemos que queremos almacenar el contenido de las tres posiciones del array en tres variables distintas. Haríamos lo siguiente:

```
int edadPosicion0 = edades[0];
int edadPosicion1 = edades[1];
int edadPosicion2 = edades[2];
```

De tal forma que:

- edadPosicion1 tendrá ahora el valor 5
- edadPosicion2 tendrá ahora el valor 10
- edadPosicion3 tendrá ahora el valor 15

Este modo de acceso se puede utilizar tanto para conocer el valor del elemento indicado, como para cambiar su valor:

```
numeros[1] = 20;
```

Para cualquier array unidimensional el índice del primer elemento será 0.

```
int edadPrimero = edades[0];
```

Además, para cualquier array unidimensional el índice del último elemento será el de la longitud del array menos 1.

En el caso de no conocer la longitud del array, podemos utilizar la siguiente instrucción:

```
nombreArray.length;
```

De tal forma que si queremos acceder al último elemento de un array podremos hacer lo siguiente:

```
int edadUltimo = edades[nombreArray.length - 1];
```

Que en este caso sería lo mismo que:

```
int edadUltimo = edades[2];
```

Ya que la longitud del array es 3, el último índice sería  $3 - 1 = 2$ .

Debemos tener cuidado a la hora de acceder a las posiciones de los arrays, puesto que, si accedemos a un índice que no exista, ya sea porque es menor que 0 o mayor o igual a la longitud del array nos dará un error que finalizará la aplicación. En el ejemplo anterior, no debemos intentar accesos como los siguientes:

```
int edadPosicion0 = edades[-1];
int edadPosicion1 = edades[3];
int edadPosicion2 = edades[20];
```

Ya que en cualquiera de estos casos nos dará error.

### 1.3. Referencia a un array, null y recolector de basura

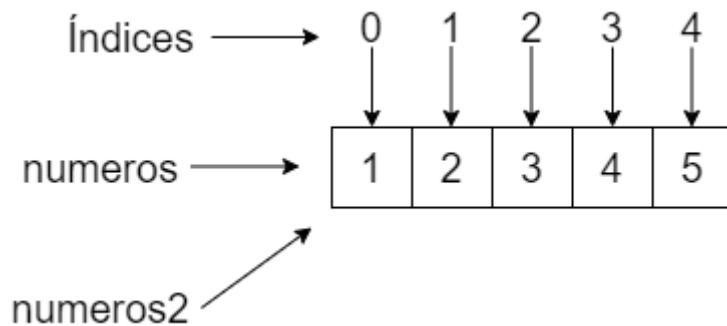
Tenemos la posibilidad de que varias variables hagan referencia a un mismo array, para ellos solo tenemos que asignar dicho array a una variable de tipo array del mismo tipo que le original.

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

int numeros2[];

numeros2 = numeros;
```

Esto se representaría gráficamente de la siguiente manera:



De esta forma ahora podremos acceder a la misma parcela de memoria desde las variables numeros y numeros2. Por ejemplo, ahora mismos numeros[4] y numeros2[4] estarían haciendo referencia al mismo valor, es decir, 5.

Debemos tener en cuenta que cualquier modificación que hagamos en el contenido del array desde cualquiera de las dos variables también afectará el otro, puesto que están apuntando a la misma posición de memoria, es decir, a los mismos datos.

En el caso de que queramos que una variable array deje de hacer referencia una parcela de memoria, debemos utilizar el valor null, y dicha referencia desaparecerá.

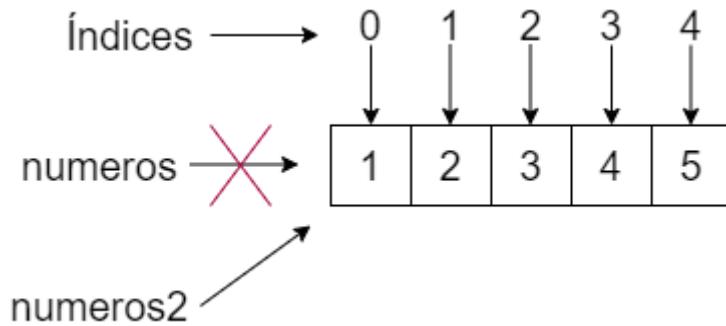
```
int numeros[] = {1, 2, 3, 4, 5};

int numeros2[];

numeros2 = numeros;

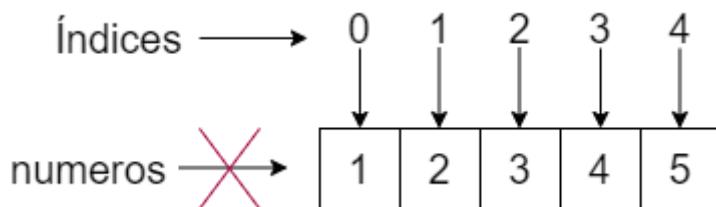
numeros = null;
```

Continuando con el ejemplo anterior si después de copiar la referencia de numeros en numeros2 asignásemos el valor null a numeros, este ya no podría acceder al array, puesto que habría desaparecido la referencia que almacenaba.



Es decir, podrímos seguir accediendo a la región de memoria desde numeros2, pero ya no tendríamos acceso desde numeros.

En el ejemplo que estamos viendo, los datos se mantienen puesto que sigue habiendo una variable que hace referencia a ellos, pero... ¿Qué ocurriría si a la única variable que tiene acceso a esa zona de la memoria le asignamos el valor null?



Pues que esa región de memoria ya sería inaccesible y por tanto no puede utilizarse. En esta situación entraría en marcha lo que denomina “Recolector de basura”, que es el proceso encargado de limpiar la memoria de datos que no pueden utilizarse, para que ese espacio pueda ser reutilizado.

## 2. RECORRIDO DE ARRAYS

Recorrer el contenido de los arrays es muy sencillo utilizando los bucles, en especial el bucle for. Vamos a ver cómo recorrer un array unidimensional con cualquiera de ellos:

```

public static void main(String[] args) {
    int numeros[] = {1, 2, 3, 4, 5};

    //Recorrido con for
    for(int indice=0; indice<numeros.length; indice++){
        System.out.print(numeros[indice] + " ");
    }
    System.out.println();

    //Recorrido con while
    int indice = 0;
    while (indice < numeros.length){
        System.out.print(numeros[indice] + " ");
        indice++;
    }
    System.out.println();

    //Recorrido con do while
    indice = 0;
    do{
        System.out.print(numeros[indice] + " ");
        indice++;
    } while (indice < numeros.length);
    System.out.println();
}

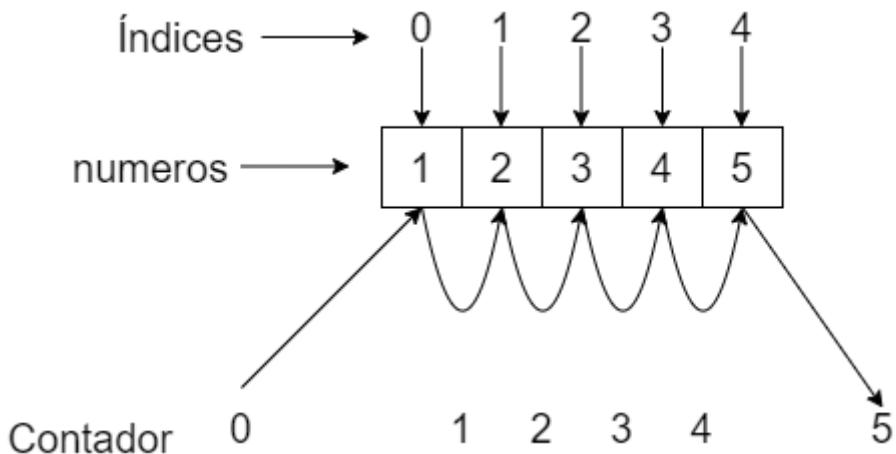
```

Esto daría como resultado por consola:

```

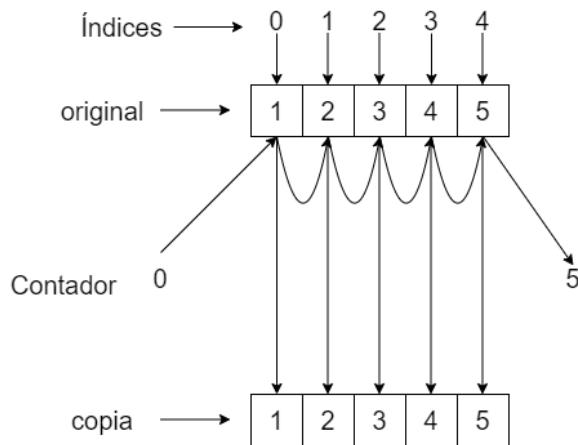
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5

```



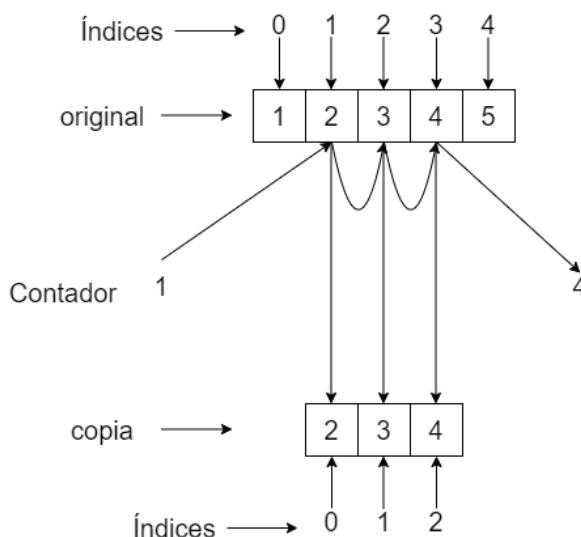
### 3. COPIA DE ARRAYS UNIDIMENSIONALES

Para realizar la copia completa de un array únicamente tendremos que crear un nuevo array con el mismo tamaño que el que original, e ir copiando los distintos elementos mientras que lo vamos recorriendo.



También podremos realizar copias parciales de un array marcando el elemento inicial y final entre los que queremos copiar.

Por supuesto, podremos realizar tantas variaciones como nos permita la imaginación.



Por supuesto, podremos realizar tantas variaciones como nos permita la imaginación.

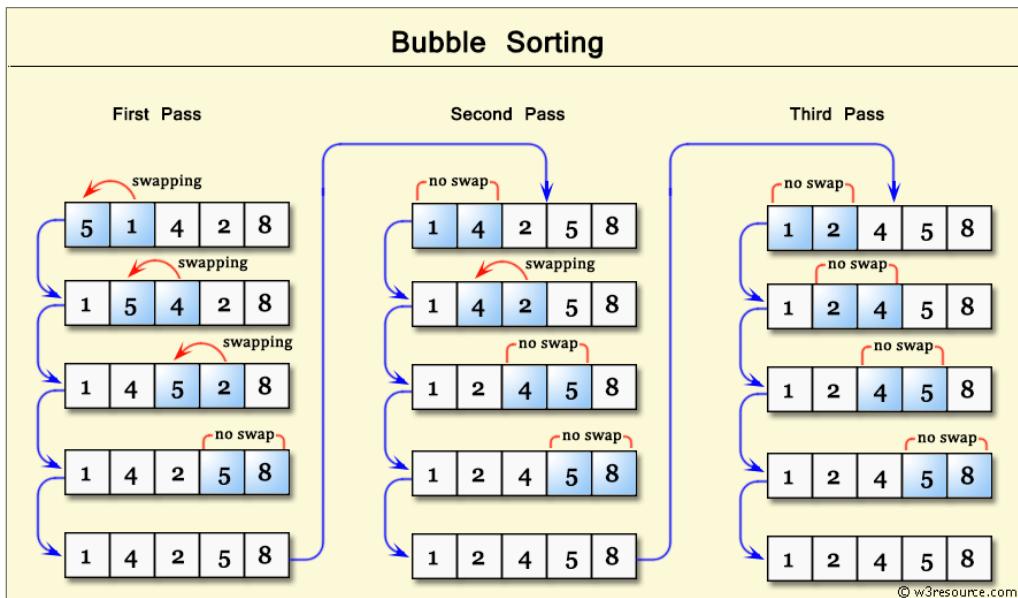
## 4. ORDENACIÓN DE ARRAYS UNIDIMENSIONALES

Existen una gran cantidad de algoritmos mediante los que podemos ordenar el contenido de un array unidimensional. La gran mayoría están orientados a arrays que contienen valores numéricos.

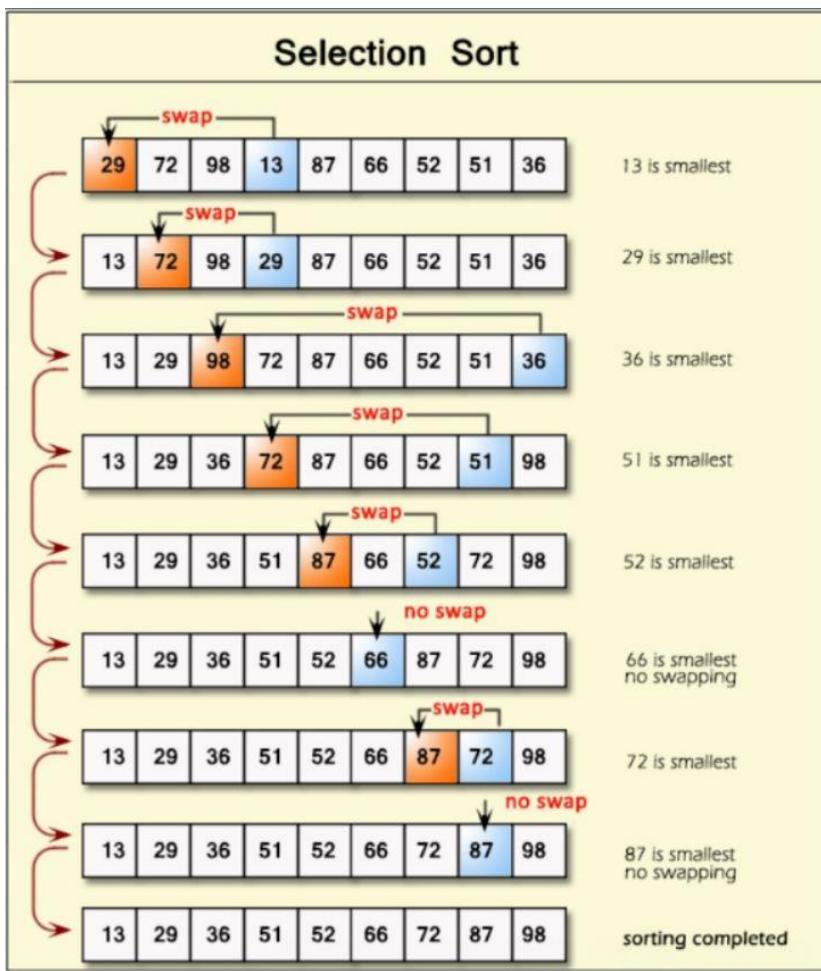
En esta unidad vamos a centrarnos en tres principales. Primero veremos cómo funcionan en una serie de vídeos y después veremos que se pueden implementar muy fácilmente.

Estos tres algoritmos de ordenación son:

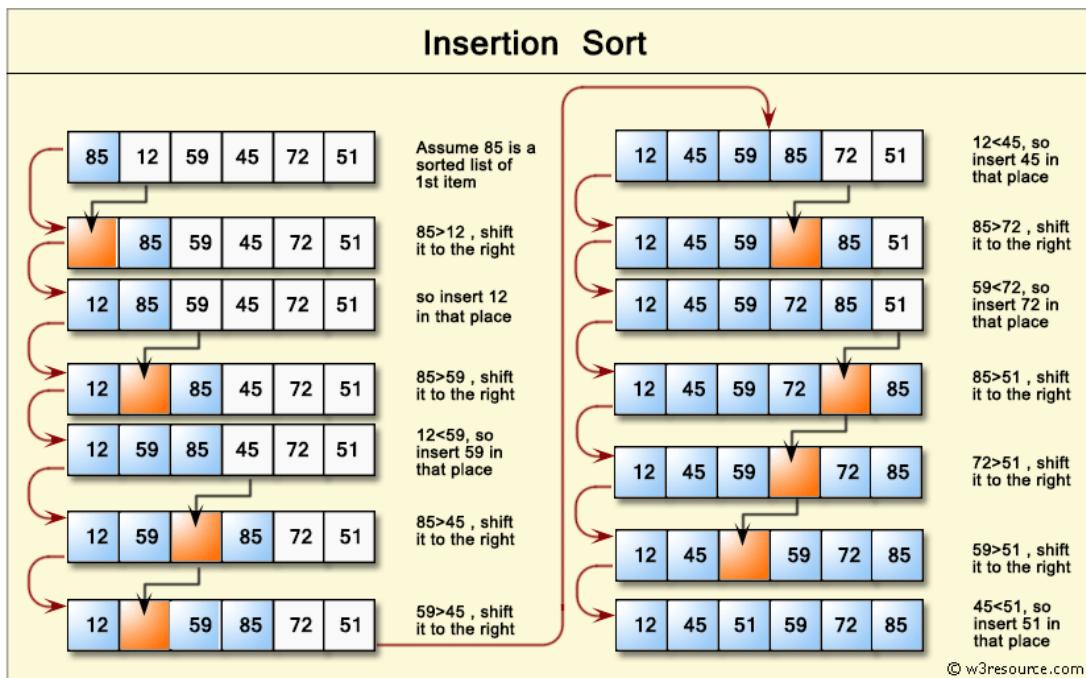
- Algoritmo de la burbuja



- Algoritmo de selección



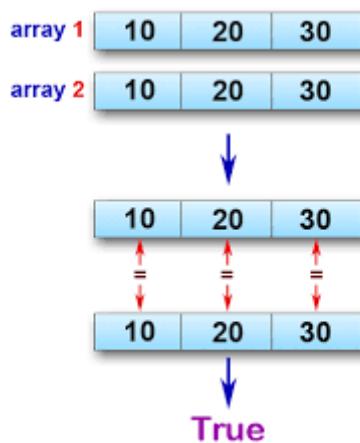
- Algoritmo de inserción



## 5. COMPARACIÓN DE ARRAYS UNIDIMENSIONALES

Para conocer si dos arrays son iguales, seguiremos los siguientes pasos:

1. Comprobar que tienen la misma longitud. Si tienen distinta longitud son distintos.
2. Si tienen la misma longitud, ordenamos los arrays.
3. Recorremos ambos arrays a la vez y vamos comparando el contenido de cada una de las posiciones con el mismo índice. En el caso de que los valores de la misma posición de los dos arrays sean distintos, los arrays son distintos.
4. Si ninguno de los casos anteriores nos ha dado como resultado que los arrays son distintos, los arrays son iguales.

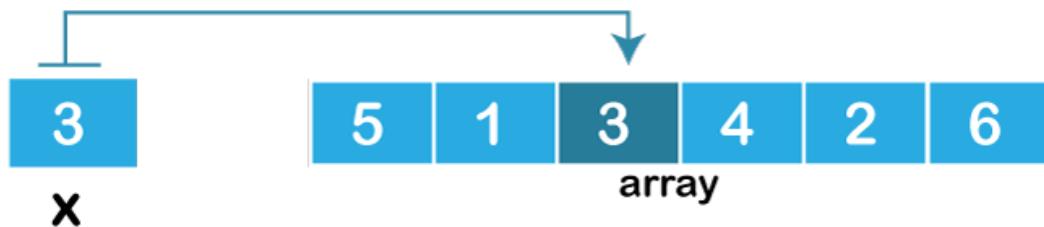


## 6. BÚSQUEDA DE ELEMENTOS ARRAYS UNIDIMENSIONALES

Para buscar de forma elementos en arrays unidimensionales tenemos dos opciones:

1. Si el array no está ordenado lo recorreremos de forma secuencial desde el principio hasta el final. De tal forma que si lo encontramos interrumpimos la búsqueda y si no devolveremos que el elemento no se encuentra en el array.

## Search Operation in Unsorted Array



2. En el caso de que el array sea ordenado tenemos posibilidades:

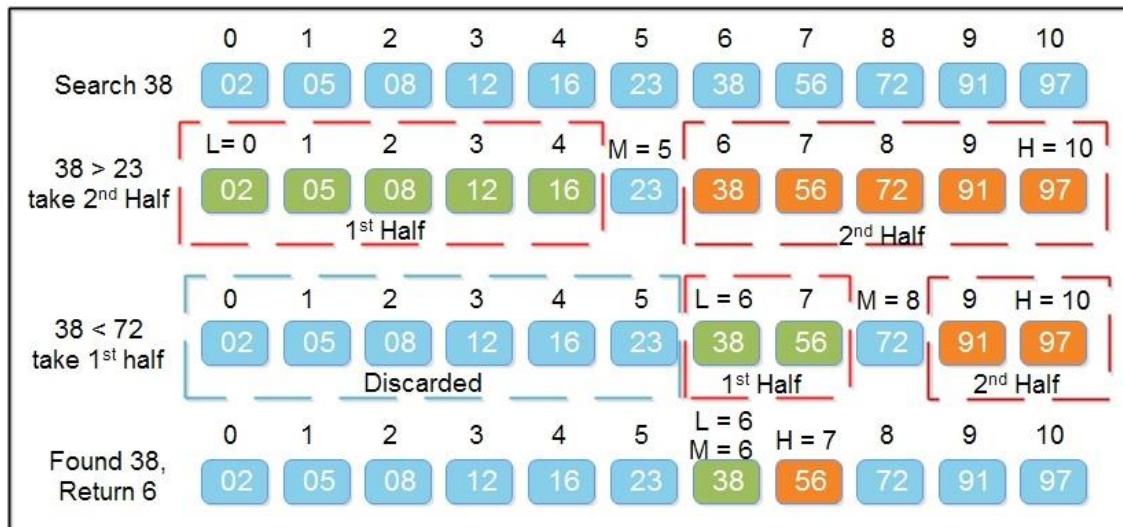
- Secuencial:** También lo recorreremos de principio a fin, pero en este caso interrumpiremos la búsqueda si encontramos el valor o si encontramos uno más alto que él. En el primer caso habríamos encontrado el elemento que buscábamos, y en el segundo sabríamos que no existe en el array.

Linear Search



- Binaria:** Algoritmo mediante el cual vamos dividiendo el array en dos partes y nos quedamos con la parte en donde puede estar ese valor.

### Binary Search



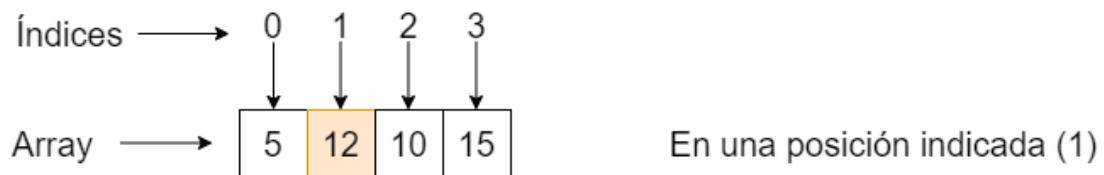
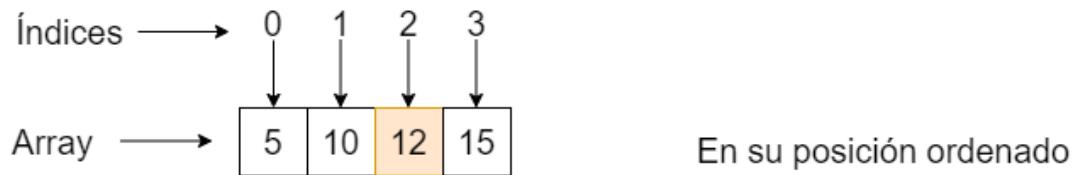
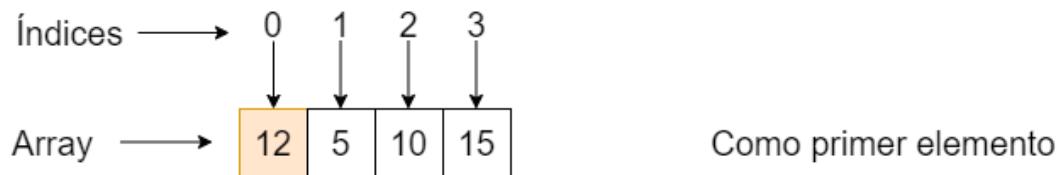
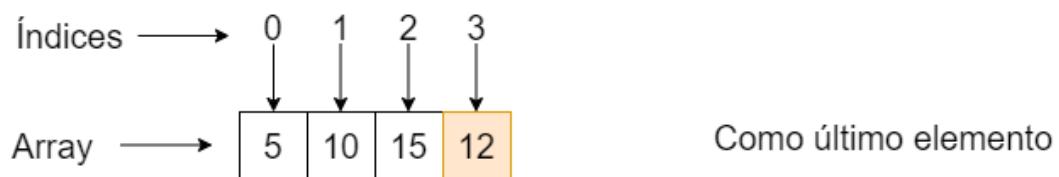
## 7. INSERCIÓN DE ELEMENTOS EN UN ARRAY

A la hora de insertar un nuevo elemento en un array disponemos de varias posibilidades, pero en todos los casos necesitaremos de un nuevo array con una longitud igual a la longitud del array original + 1.

Las distintas opciones son las siguientes:

1. Insertar un elemento al final de un array.
2. Insertar un elemento al inicio de un array.
3. Insertar un elemento donde le corresponde en un array ordenado.
4. Insertar un elemento en una posición determinada.

## TIPOS DE INSERCIÓN



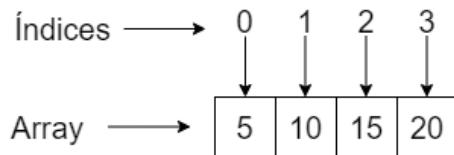
## 8. ELIMINACIÓN DE ELEMENTOS EN UN ARRAY

A la hora de insertar un nuevo elemento en un array disponemos de varias posibilidades, pero en todos los casos necesitaremos de un nuevo array con una longitud igual a la longitud del array original - 1.

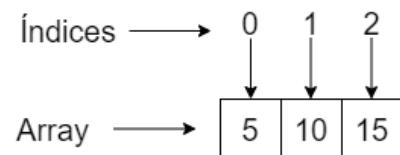
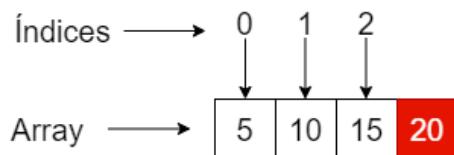
Las distintas opciones son las siguientes:

1. Eliminar un elemento al final de un array.
2. Eliminar un elemento al inicio de un array.
3. Eliminar un elemento determinado dentro de un array, ordenado o no ordenado.
4. Eliminar un elemento en una posición determinada.

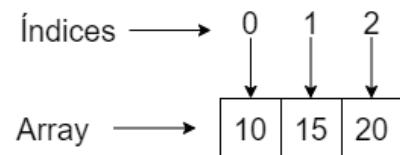
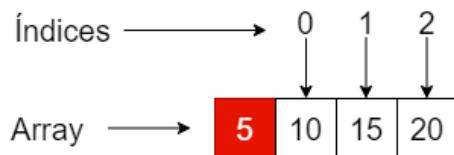
## TIPOS DE ELIMINACIÓN



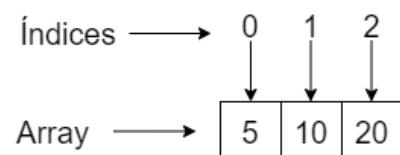
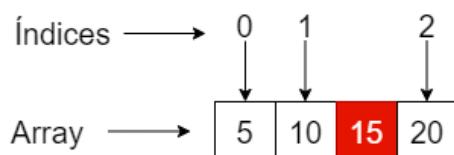
Como último elemento



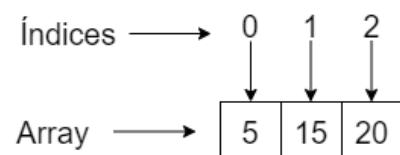
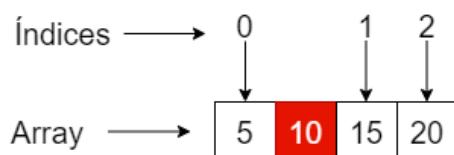
Como primer elemento



Valor indicado (15)



En una posición indicada (1)



## 9. LA CLASE ARRAYS

Después de realizar de forma “manual” cada una de las operaciones que hemos visto: copia, comparación, búsqueda, inserción y eliminación de arrays unidimensionales, debemos saber que existe una clase (Arrays) que nos proporciona estas operaciones de una forma “automática” a través de métodos. Para poder utilizar esta clase debemos añadir la siguiente importación a nuestros programas:

```
import java.util.Arrays;
```

A continuación, veamos en qué nos puede ayudar la clase Arrays:

### 9.1. Copia de arrays

La clase Arrays nos proporciona dos métodos mediante los cuales podemos realizar copias de arrays:

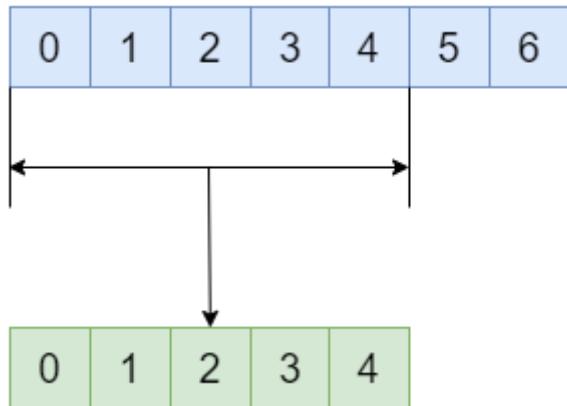
- El método **Arrays.copyOf(array, nuevaLongitud)** copia un array construyendo uno nuevo que se ajustará a la nueva longitud indicada. Veamos tres ejemplos:

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

int[] copia1 = Arrays.copyOf(numeros, numeros.length);
// devolverá {1,2,3,4,5}

int[] copia2 = Arrays.copyOf(numeros, 3);
// devolverá {1,2,3}

int[] copia3 = Arrays.copyOf(numeros, 7);
// devolverá {1,2,3,4,5,0,0}
```



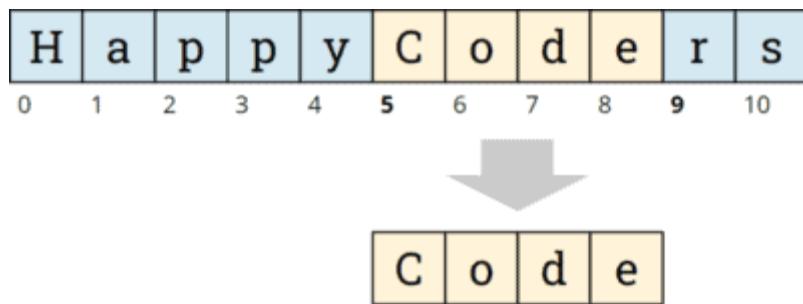
- El método **Arrays.copyOfRange(array, posIni, posFin)** realiza una copia parcial del array, empezando por la posición de inicio y finalizando en la posición anterior a fin. En este caso, al utilizar posiciones, debemos tener cuidado de movernos en los rangos del propio array original.

Veamos dos ejemplos:

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

int[] copia1 = Arrays.copyOfRange(numeros, 0, numeros.length);
//devolverá {1,2,3,4,5}

int[] copia2 = Arrays.copyOfRange(numeros, 1, 4);
//devolverá {2,3,4}
```



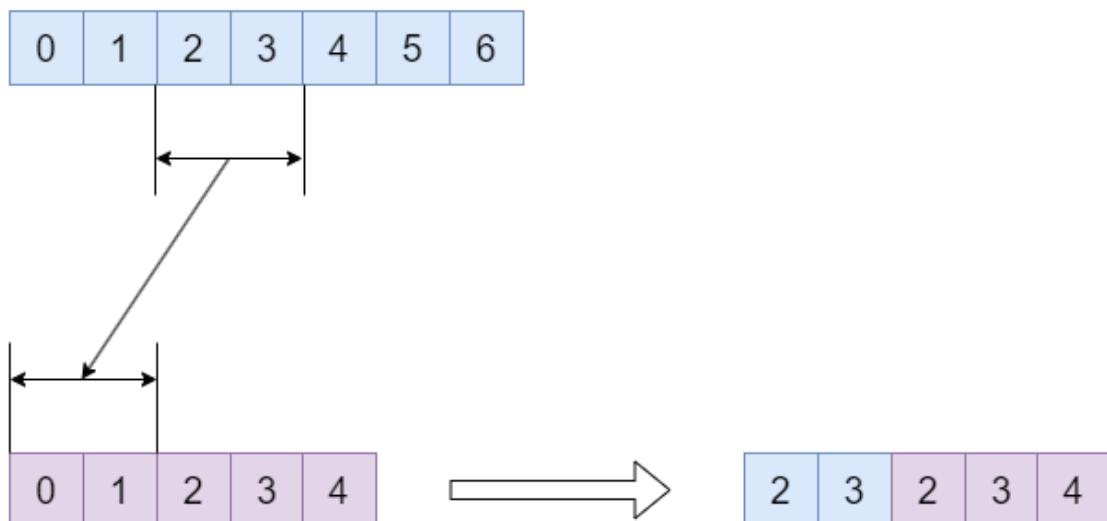
Además de estos dos métodos de la clase **Arrays** contamos con un tercer método, pero en este caso de la clase System, que tiene el nombre de **System.arraycopy(arrayOrigen, posOrig, arrayDestino, posDest, longitud)**. Con este método realizaremos copias en el arrayDestino, a partir de la posición de destino, los datos del arrayOrigen, comenzando en la posición origen. El

parámetro longitud determina el número de elementos que se copiarán entre los dos arrays. Veamos algún ejemplo:

```
int arr1[] = {0, 1, 2, 3, 4, 5};
int arr2[] = {5, 10, 20, 30, 40, 50};

System.arraycopy(arr1, 0, arr2, 0, 1);
// arr2 pasará a ser {0, 10, 20, 30, 40, 50}

System.arraycopy(arr1, 1, arr2, 3, 2);
// arr2 pasará a ser {5, 10, 20, 1, 2, 50}
```



## 9.2. Comparación de arrays

Para comparar dos arrays podemos utilizar el método **Arrays.equals(array1, array2)**, la cual nos devolverá un booleano true si los dos arrays son iguales, y false si no lo son. Pero debemos tener en cuenta que solo tomará dos arrays como iguales si tiene los mismos datos y en el mismo orden.

```
if(Arrays.equals(arrayNumeros1, arrayNumeros2)){
    System.out.println("Son iguales");
}
else{
    System.out.println("No son iguales");
}
```

### 9.3. Búsqueda en un array

También podemos buscar la primera aparición de un valor dentro de un array utilizando el método **Arrays.binarySearch(array1, valor)**. Para poder realizar esta búsqueda binaria, al igual que en el método “manual”, el array debe estar ordenado. Este método nos devolverá la posición en la que se encuentra el elemento que buscamos, o no devolverá un valor negativo en caso contrario.

```
int pos = Arrays.binarySearch(numeros, numero);
if(pos >= 0){
    System.out.println("Está en el array");
}
else{
    System.out.println("No está en el array");
}
```

### 9.4. Rellenar array

Para inicializar los valores de un array de una forma cómoda podemos utilizar el método **Arrays.fill(array, valor)**. De esta forma todos los elementos del array tomarán el valor que se ha proporcionado.

```
int arr1[] = {0, 1, 2, 3, 4, 5};

Arrays.fill(arr1, 6);
// arr1 ahora será {6,6,6,6,6,6}
```

### 9.5. Convertir un array en un String

También tenemos la posibilidad de convertir el contenido de un array en un String, de tal forma que nos sea más sencillo escribirlo pantalla. Para ello debemos utilizar el método **Arrays.toString(array)**.

```
int arr1[] = {0, 1, 2, 3, 4, 5};

System.out.println(Arrays.toString(arr1));
//la salida por pantalla será "[0, 1, 2, 3, 4, 5]"
```

## 9.6. Ordenación de un array

Para ordenar un array con la clase Arrays, solo necesitaremos usar el método Arrays.sort(array).

```
int[] numeros = {2, 5, 3, 56, 8, 40, 32};  
Arrays.sort(numeros);  
//Ahora numeros será {2, 3, 5, 8, 32, 40, 56}
```

## 9.7. Inserción en un array

Para insertar elementos en un array no tenemos métodos específicos, pero podemos hacer uso de los métodos de copia para llevarlos a cabo.

## 9.8. Eliminación de elementos en un array

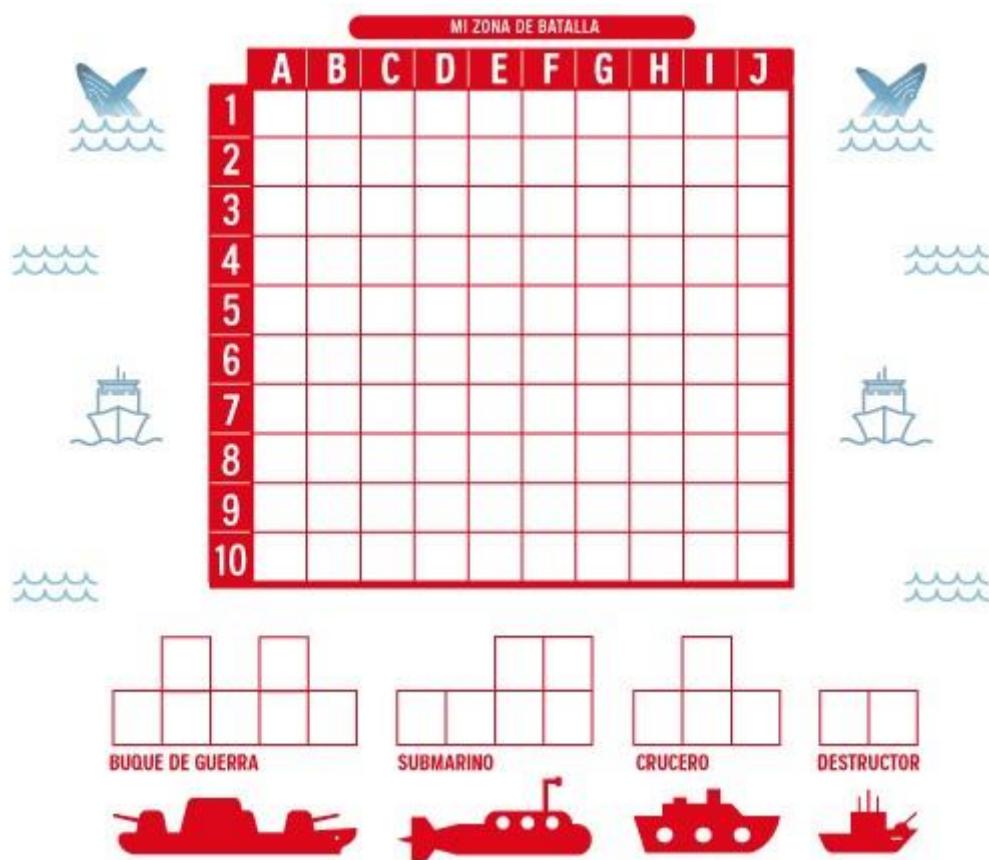
Para la eliminación de elementos en un array, al igual que ocurre con las eliminaciones, no disponemos de métodos específicos, pero podremos llevarlas a cabo utilizando los métodos de copia.

# 10. ARRAYS BIDIMENSIONALES O MATRICES

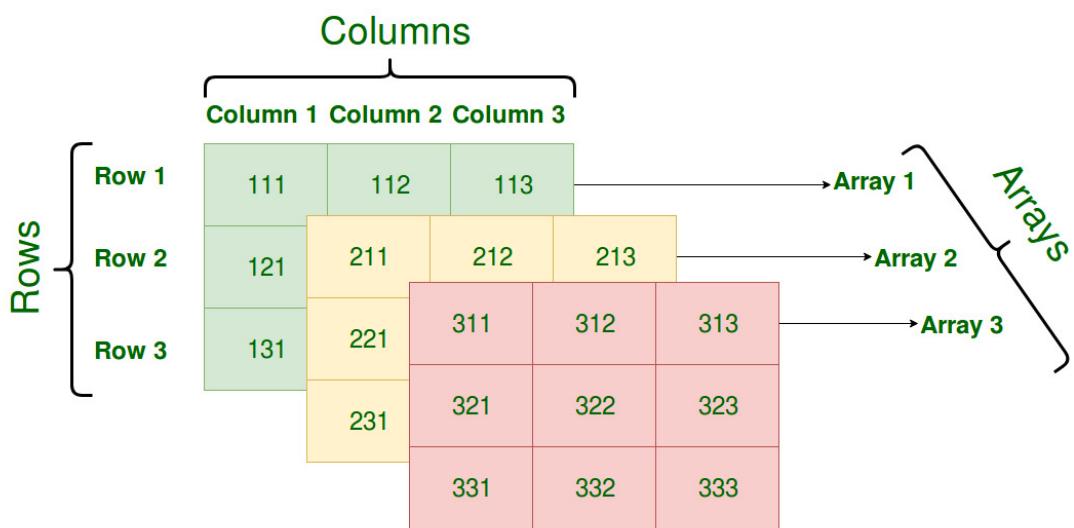
Dentro de cada posición de un array no solo podemos almacenar elemento de tipos primitivos o Strings, sino que podemos almacenar cualquier tipo de objetos, entre ellos otros arrays, de esta forma obtenemos lo que se denominan arrays multidimensionales, lo más conocidos son los arrays bidimensionales, también conocidos como matrices o tablas, y que será en los que nos centraremos en los siguientes apartados...

	0	1	2	.....	n-1
0	a[0][0]	a[0][1]	a[0][2]	.....	a[0][n-1]
1	a[1][0]	a[1][1]	a[1][2]	.....	a[1][n-1]
2	a[2][0]	a[2][1]	a[2][2]	.....	a[2][n-1]
3	a[3][0]	a[3][1]	a[3][2]	.....	a[3][n-1]
4	a[4][0]	a[4][1]	a[4][2]	.....	a[4][n-1]
.	.	.	.	.....	.
.	.	.	.	.....	.
n-1	a[n-1][0]	a[n-1][1]	a[n-1][2]	.....	a[n-1][n-1]

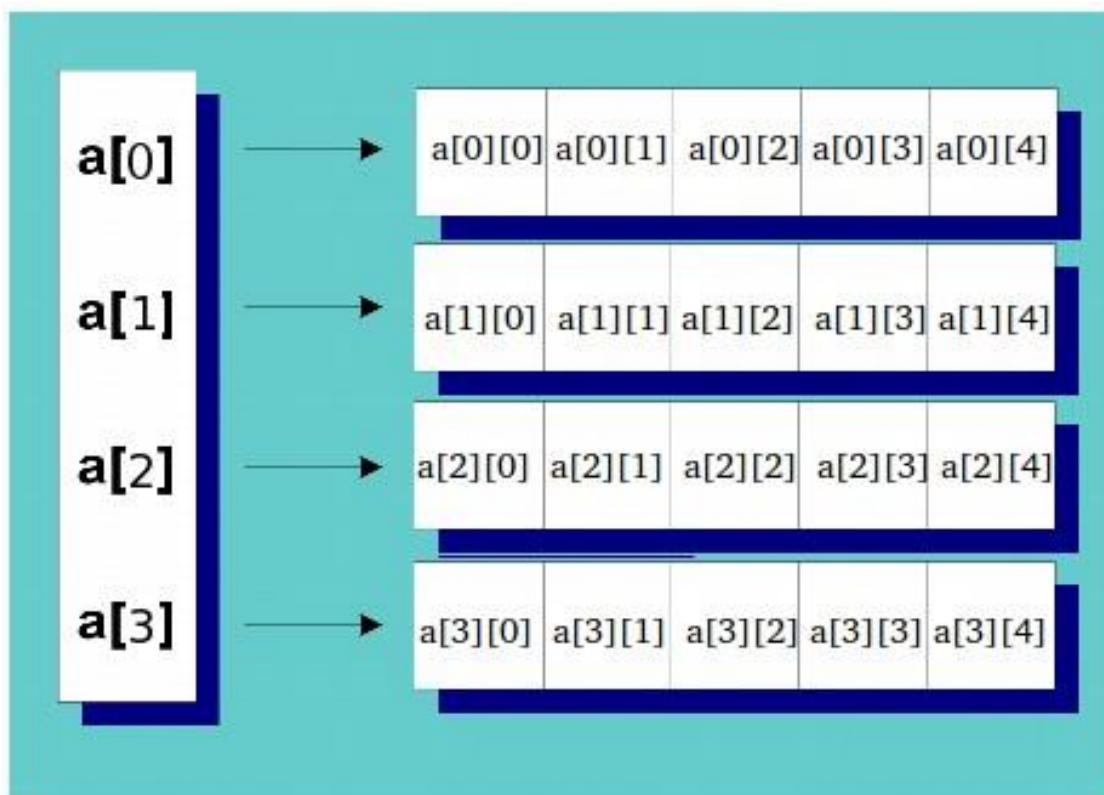
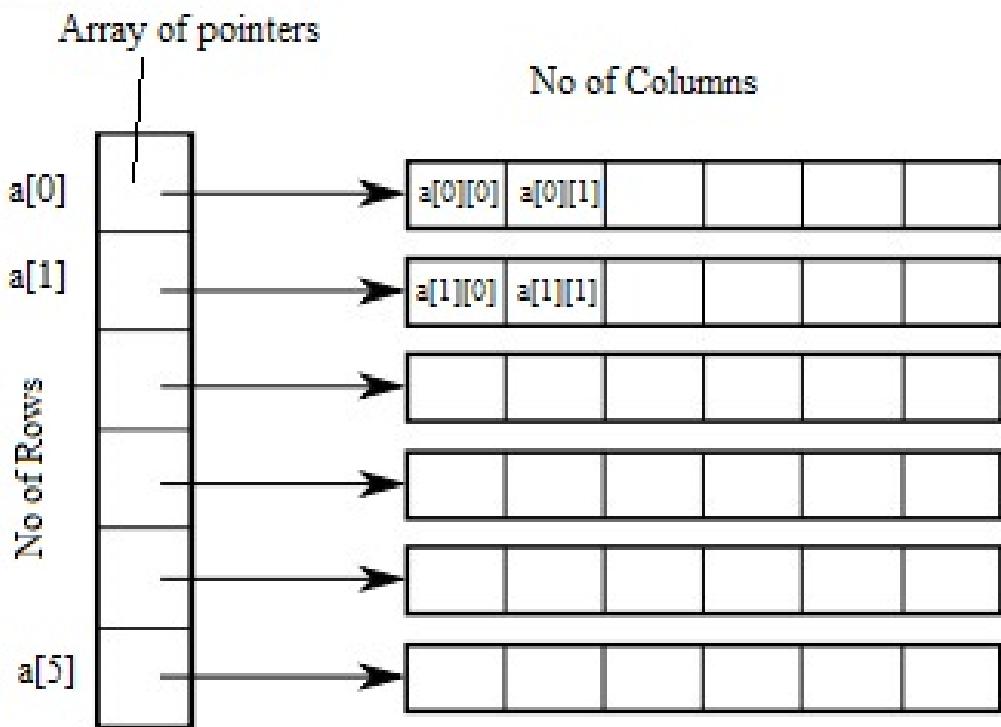
**a[n][n]**

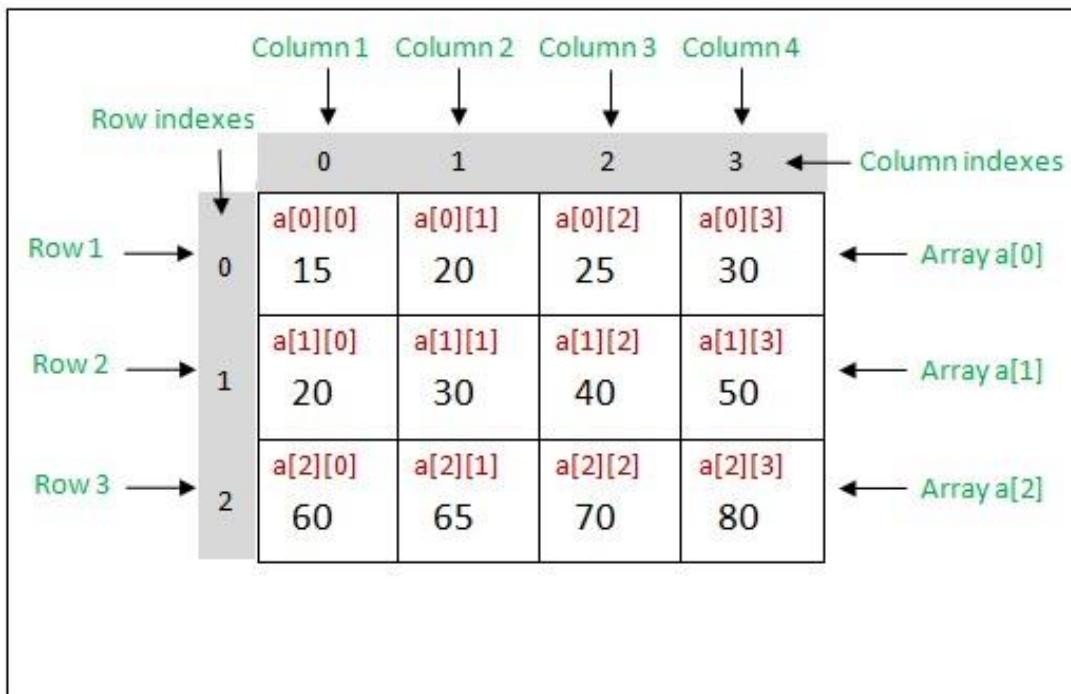


O los arrays de tres dimensiones...



### 10.1. Declaración, creación e inicialización de matrices





A la hora de declarar un array multidimensional lo haremos de forma similar que a los unidimensionales, con la única diferencia de añadir una pareja “[]” por cada dimensión. Ejemplo de matriz bidimensional:

```
int[][] matriz;
```

Para crear las matrices bidimensionales también tenemos tres formas distintas:

Construcción tras haberlo declarado:

```
matriz = new int[3][4];
```

Declaración y construcción a la vez:

```
int[][] matriz = new int[3][4];
```

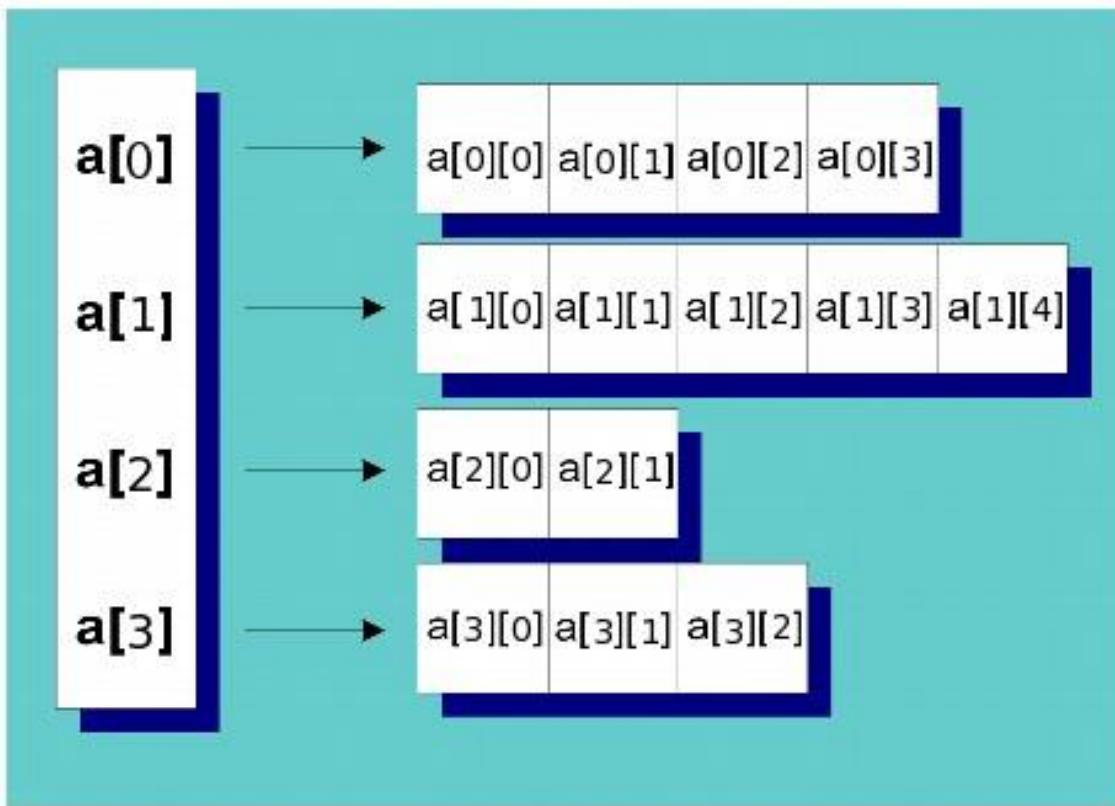
O declaración, construcción e inicializando directa de la matriz:

```
int[][] matriz = {{15, 20, 25, 30},  
                 {20, 30, 40, 50},  
                 {60, 65, 70, 80}};
```

Si solo hemos declarado y construido la matriz, para inicializarla utilizaremos bucles anidados, tantos como dimensiones tenga. En el caso de las bidimensionales, 2.

```
for (int i=0; i < matriz.length; i++){
    for(int j=0; j < matriz[i].length; j++){
        matriz[i][j] = i + j;
    }
}
```

Habrá ocasiones en las que no todos los arrays de la matriz tengan la misma longitud...



En estas situaciones primer definiríamos el array que representa la primera dimensión, y después iríamos asociando los arrays de la segunda dimensión, cada uno en su posición. Como es habitual, tenemos distintas opciones:

Creando el array base, después los arrays de las filas y finalmente asignando cada fila a su posición...

```
int[][] matriz = new int[4][];
int[] matrizFila0 = new int[4];
int[] matrizFila1 = new int[5];
int[] matrizFila2 = new int[2];
int[] matrizFila3 = new int[3];

matriz[0] = matrizFila0;
matriz[1] = matrizFila1;
matriz[2] = matrizFila2;
matriz[3] = matrizFila3;
```

Creando directamente un array en cada posición...

```
int[][] matriz2 = {new int[4],
                  new int[5],
                  new int[2],
                  new int[3]};
```

Creando y asignando directamente...

```
int[][] matriz3 = {{00, 01, 02, 03},
                  {10, 11, 12, 13, 14},
                  {20, 21},
                  {30, 31, 32}};
```

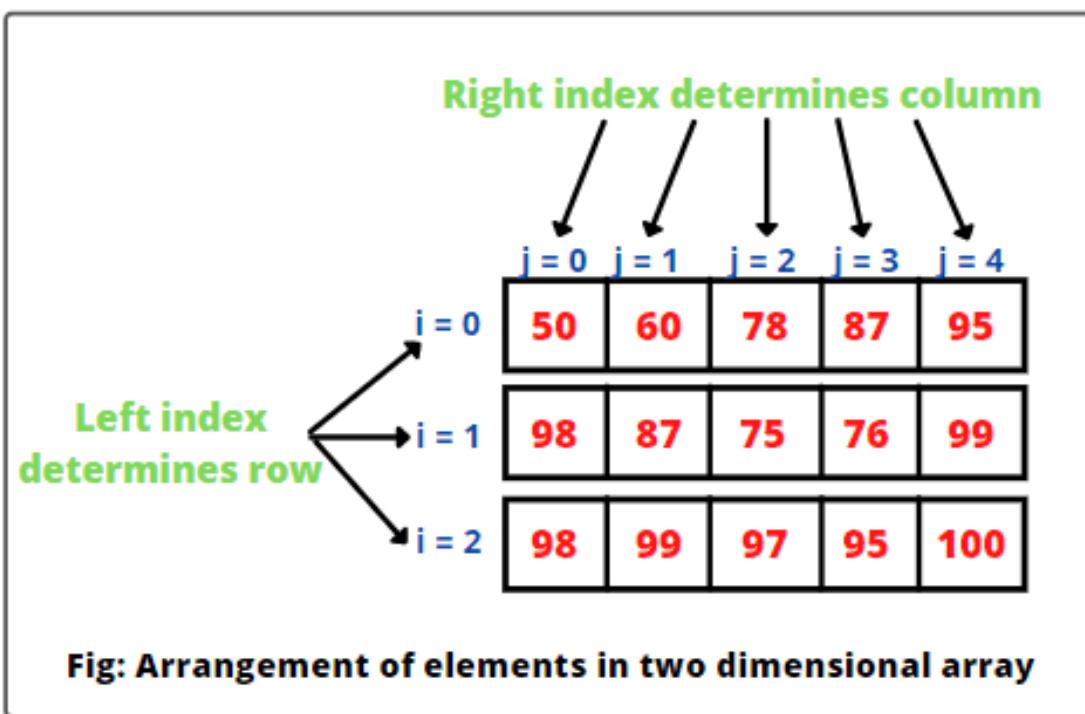
En el caso de no utilizar última opción, la inicialización la haríamos de la misma forma que hemos visto anteriormente:

```
for (int i=0; i < matriz.length; i++){
    for(int j=0; j < matriz[i].length; j++){
        matriz[i][j] = i + j;
    }
}
```

## 10.2. Acceso a los datos de un matriz

Como ya hemos visto en la inicialización mediante bucles, para acceder a uno de los elementos de una matriz lo haremos de forma similar a cómo lo hacíamos con un array unidimensional, pero añadiremos otro grupo de corchetes con la posición de la segunda dimensión.

	Col1	Col2	Col3	Col4	....
Row1	Arr[0][0]	Arr[0][1]	Arr[0][2]	Arr[0][3]	
Row2	Arr[1][0]	Arr[1][1]	Arr[1][2]	Arr[1][3]	
Row3	Arr[2][0]	Arr[2][1]	Arr[2][2]	Arr[2][3]	
Row4	Arr[3][0]	Arr[3][1]	Arr[3][2]	Arr[3][3]	
:					



Para insertar un valor en una posición de una matriz...

```
matriz[2][3] = 95;
```

Para obtener el valor de una posición de una matriz...

```
int numero = matriz[2][3];
```

## 11. CADENAS DE CARACTERES. STRING.

### 11.1. Tipo primitivo char

Este tipo primitivo ya lo hemos visto anteriormente, y sabemos que los valores corresponden a códigos numéricos (code point) que son representados mediante letras, números, símbolos, caracteres especiales, etc... que nos encontramos en la codificación Unicode.

A la hora de seleccionar un carácter es posible usar su codificación Unicode o el propio carácter si es posible escribirlo mediante teclado.

```
char c;
c = 'a'; //directamente mediante el teclado
c = 97; //usando el code point en decimal
c = '\u0061'; //o con el code point en hexadecimal
```

En realidad, para codificar cualquier code point de Unicode necesitamos 3 bytes, mientras que el tipo char solo utiliza 2 bytes, de tal forma que no todos los code points pueden representarse en este tipo primitivo, concretamente los que tengan un valor inferior o igual a 65535 (\uFFF en hexadecimal). En el caso de querer almacenar el code point de un carácter Unicode superior a este valor debemos utilizar una variable de int que se almacena en 4 bytes.

Algunos caracteres especiales ya los vimos en la unidad de entrada/salida,... son las secuencias de escape, que tienen un significado especial a la hora de representarse por pantalla.

Carácter	Nombre
\b	Borrado a la izquierda
\n	Nueva línea
\r	Retorno de carro
\t	Tabulador
\f	Nueva página
\'	Comilla simple
\"	Comilla doble
\\\	Barra invertida

#### 11.1.1. Conversión `char` ↔ `int`

Como hemos visto antes los code points no son más que números enteros, por tanto, la conversión del tipo char a int es prácticamente directa.

Aprovechando los mecanismos de conversión podemos realizar asignaciones de la forma:

```
int e = 'a'; //asigna un carácter a una variable int
System.out.println(e); // muestra 97
e = '\u0061'; //asigna un carácter a una variable int
System.out.println(e); // muestra 97
char c = 97; //asigna un entero a una variable char
System.out.println(c); // muestra 'a'
```

También podemos forzar la conversión por medio de un cast:

```
char c = 'a';
System.out.println((int) c); //muestra 97
int e = 97;
System.out.println((char) e); //muestra 'a'
```

Se pueden realizar asignaciones de variables del tipo int a char con un cast (realizando una conversión por estrechamiento):

```
int e = 97;
char c = (char) e; //c vale 'a'
```

O también se pueden realizar asignaciones del tipo char a int de forma implícita (realizando una conversión por ensanchamiento):

```
char c = 'a';
int e = c; //e vale 97
```

Tanta es la estrecha relación que hay entre el tipo char y el int, que se nos permite realizar operaciones aritméticas con sus valores.

```
char c = 'a' + 1; //c tendrá valor 98, es decir, el carácter 'b'
```

## 11.2. Clase Character

La clase Character es un tipo de clase a la que se le denomina wrapper (envoltorio), las cuales nos proporcionan una extensión de un carácter primitivo, char en este caso, añadiendo nuevas funcionalidades para operar con él.

### *11.2.1. Clasificación de caracteres*

Para la clase Character existen 4 tipos de caracteres:

- **Dígitos**: formado por los números del 0 al 9.
- **Letras**: formado por todos los elementos del alfabeto tanto minúsculas como mayúsculas.
- **Caracteres blancos**: como el espacio o la tabulación, entre otros.
- **Otros caracteres**: signos de puntuación, matemáticos, ...

Esto nos sirve porque la clase Character nos proporciona una serie de métodos que nos permite determinar de qué tipo es un carácter.

- **boolean isDigit(char c)**: devuelve true si c es un dígito, false en caso contrario.
- **boolean isLetter(char c)**: devuelve true si c es una letra, false en caso contrario.

- **boolean isLetterorDigit(char c):** devuelve true si c es una letra o un dígito (carácter alfanumérico), false en caso contrario.
- **boolean isLowerCase(char c):** devuelve true si c es una letra en minúscula, false en caso contrario.
- **boolean isUpperCase(char c):** devuelve true si c es una letra en mayúscula, false en caso contrario.
- **boolean isSpaceChar(char c):** devuelve true si c es específicamente un espacio en blanco, false en caso contrario.
- **boolean isWhiteSpace(char c):** devuelve true si c es un carácter blanco, false en caso contrario. Estos caracteres son:
  - Espacio en blanco
  - Retorno de carro
  - Nueva línea
  - Tabulador
  - Otros (están en desuso)

Así se utilizarían:

```
if(Character.isDigit(c)){  
    ...  
}  
  
if(Character.isLetter(c)){  
    ...  
}  
  
if(Character.isLetterorDigit(c)){  
    ...  
}  
  
if(Character.isLowerCase(c)){  
    ...  
}  
  
if(Character.isUpperCase(c)){  
    ...  
}
```

```
if(Character.isSpaceChar(c)){  
    ...  
}  
  
if(Character.isWhiteSpace(c)){  
    ...  
}
```

### 11.2.2. Conversión entre caracteres

Dentro del grupo de caracteres de tipo letra podremos hacer dos tipos de conversiones mediante los siguientes métodos:

- `char toLowerCase(char c)`: convierte `c` en la misma letra pero en minúscula.

```
char c = 'A';  
c = Character.toLowerCase(c);  
//Ahora c es 'a'
```

- `char toUpperCase(char c)`: convierte `c` en la misma letra pero en mayúscula.

```
char c = 'a';  
c = Character.toUpperCase(c);  
//Ahora c es 'A'
```

También tenemos la opción de convertir un carácter que corresponda a un dígito al tipo `int` mediante el método `int getNumericValue(char carácter)`.

```
char c = '9';  
int num = Character.getNumericValue(c); //num es 9
```

## 11.2. Cadenas de caracteres



Aunque llevamos mucho tiempo utilizando la clase String (cadena de caracteres) apenas hemos aprovechado su potencial. Como ya sabemos los String son conjuntos secuenciales de caracteres y como tales, vamos a poder manipularlos según nuestros intereses. A continuación veremos una gran variedad de métodos para poder trabajar con Strings.

### *11.2.1. Valores de otros tipos*

Vamos a poder convertir valores de otros tipos en Strings y a la inversa Strings a valores de otros tipos.

Para la primera opción contamos con el método **valueOf()** de la clase String:

```
String cad;
cad = String.valueOf(1234); //cad ahora es "1234"
cad = String.valueOf(-12.34); //cad ahora es "-12.34"
cad = String.valueOf('C'); //cad ahora es "C"
cad = String.valueOf(true); //cad ahora es "true"
```

En el caso contrario disponemos del mismo método **valueOf()** de las clases wrapper de otros tipos de datos, como Integer, Double, Float y Boolean.

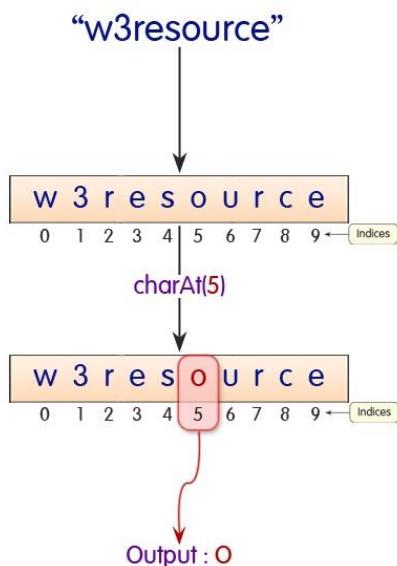
```
String cad;
cad = "2";
int numInt = Integer.valueOf(cad);
cad = "2.5";
double numDouble = Double.valueOf(cad);
cad = "5.2";
float numFloat = Float.valueOf(cad);
```

```
cad = "true";
boolean bool = Boolean.valueOf(cad);
```

En el caso del tipo char vamos a hacer uso una función propia de los Strings denominada `charAt` que nos permitirá obtener el carácter que ocupa una posición indicada en un String.

```
String letra = "H";
c = letra.charAt(0); // c ahora es 'H'

String palabra = "Hola";
c = letra.charAt(2); // c ahora es 'l'
```



### 11.2.2. Comparación de Strings

Para comparar dos strings disponemos de varios métodos distintos:

- **boolean equals(String cadena):** devuelve true si la cadena invocante y la que se pasa por parámetro son exactamente iguales, o false en caso contrario.

```
String myStr1 = "Hello";
String myStr2 = "Hello";
String myStr3 = "Another String";
System.out.println(myStr1.equals(myStr2)); // Returns true because
they are equal
System.out.println(myStr1.equals(myStr3)); // false
```

- **boolean equalsIgnoreCase(String cadena):** similar al método equals, pero sin distinguir entre mayúsculas y minúsculas.
- **boolean regionMatches(int inicio, String otraCad, int inicioOtra, int longitud):** compara dos fragmentos de cadenas: el primero corresponde a la cadena invocante y comienza en el carácter con índice inicio; y el segundo corresponde a la cadena otraCad y comienza en el carácter con índice inicioOtra. Ambos fragmentos tendrán la longitud indicada. El método devuelve true o false para indicar si las regiones coinciden.

```
String Str1 = new String("Welcome to Tutorialspoint.com");
String Str2 = new String("Tutorials");
String Str3 = new String("TUTORIALS");

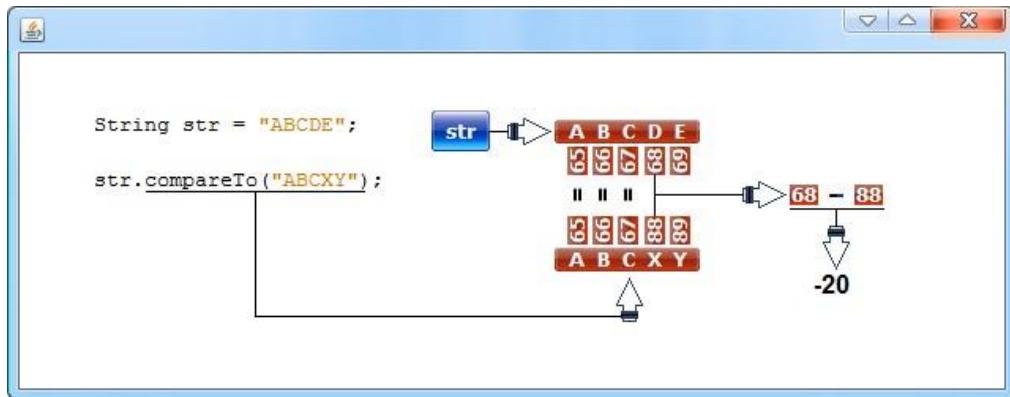
System.out.print("Return Value :");
System.out.println(Str1.regionMatches(11, Str2, 0, 9));

System.out.print("Return Value :");
System.out.println(Str1.regionMatches(11, Str3, 0, 9));

Return Value :true
Return Value :false
```

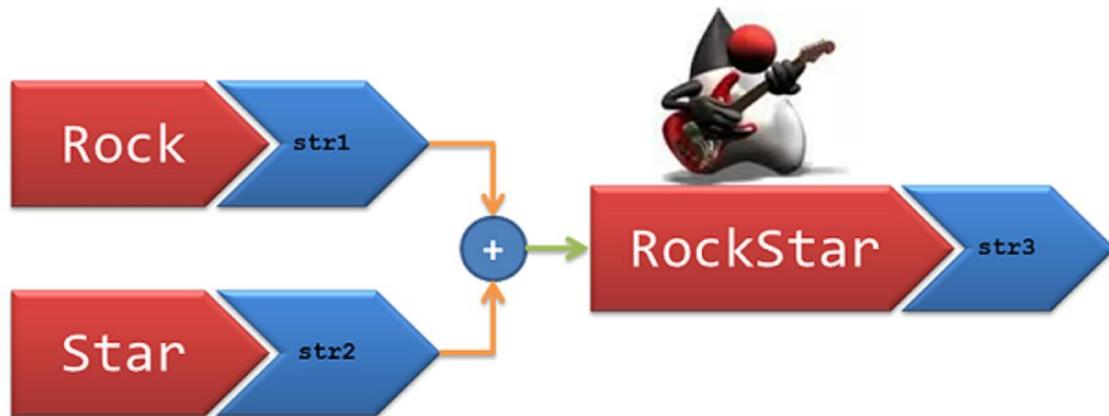
- **boolean regionMatches(boolean ignora, int inicio, String otraCad, int inicioOtra, int longitud):** hace lo mismo que el método anterior, pero si el parámetro ignora está a true, no distingue entre mayúsculas y minúsculas.
- **int compareTo(String cadena):** compara alfabéticamente la cadena que invoca el método y la que se pasa por parámetro, de tal forma que devolverá un entero cuyo valor determina el orden de las cadenas de la forma:
  - 0: si las cadenas son exactamente iguales
  - Número negativo: si la cadena invocante es menor alfabéticamente que la pasada por parámetro, es decir, va antes en orden alfabético.
  - Número positivo: si la cadena invocante es mayor alfabéticamente que la pasada por parámetro, es decir, va después en orden alfabético.

```
String myStr1 = "Hello";
String myStr2 = "Hello";
System.out.println(myStr1.compareTo(myStr2)); // Returns 0
because they are equal
```



- `int compareToIgnoreCase(String cadena)`: funciona igual que `compareTo`, pero sin distinguir entre mayúsculas y minúsculas.

### 11.2.3. Concatenación



Conocemos concatenación como la unión de dos cadenas de caracteres, de tal forma que una queda detrás de la otra.

Para realizar esta operación utilizaremos los operadores "+" y "+=". El método `concat()` también se puede utilizar, pero no suele ser habitual.

```
String first = "Hello";
String second = "World";

String third = first + second;
```

```
System.out.println(third); //HelloWorld

// yet another way to concatenate strings
first += second;
System.out.println(first); //HelloWorld
```

```
String first = "Hello";
String second = "World";

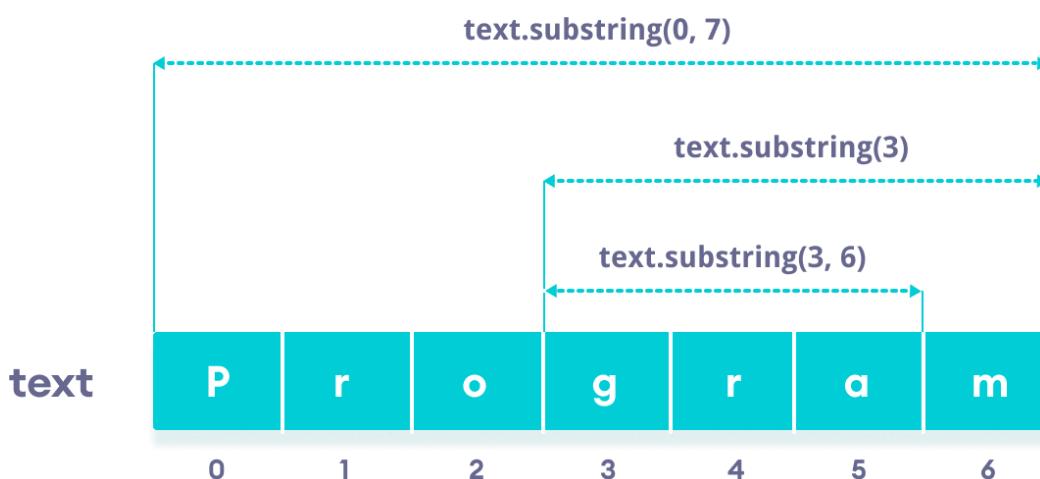
String third = first.concat(second);
System.out.println(third); //HelloWorld
```

#### 11.2.4. Obtención de subcadenas

Otra posibilidad que tenemos con los Strings es extraer secciones de ellos, creando otros Strings. Para ello tenemos los siguientes métodos:

- **String substring(int inicio)**: devuelve una subcadena formada desde la posición de inicio hasta el final de la cadena.
- **String substring(int inicio, int fin)**: devuelve una subcadena formada desde la posición de inicio hasta la anterior posición fin indicadas.

```
String s1="javatpoint";
System.out.println(s1.substring(2,4));//returns va
System.out.println(s1.substring(2));//returns vatpoint
```



### 11.2.5. Longitud de una cadena

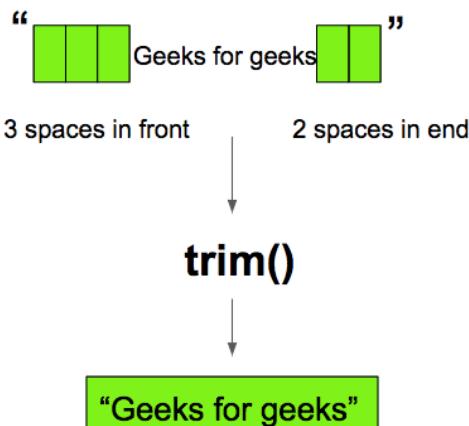
Podremos conocer la longitud de una cadena mediante el método **int length()**.

```
String s1="java";
int longitud = s1.length(); //longitud es igual a 4
```

### 11.2.6. Eliminación de espacios vacíos

También será posible eliminar los espacios en blanco que tengamos en un String, tanto al comienzo como al final del mismo. Para ello utilizaremos el método **String trim()**.

```
String myStr = "      Hello World!      ";
System.out.println(myStr); // "      Hello World!      "
System.out.println(myStr.trim()); // "Hello World!"
```



### 11.2.7. Conversión de cadenas de caracteres

Disponemos de distintos métodos para modificar el contenido de una cadena de caracteres, como, por ejemplo:

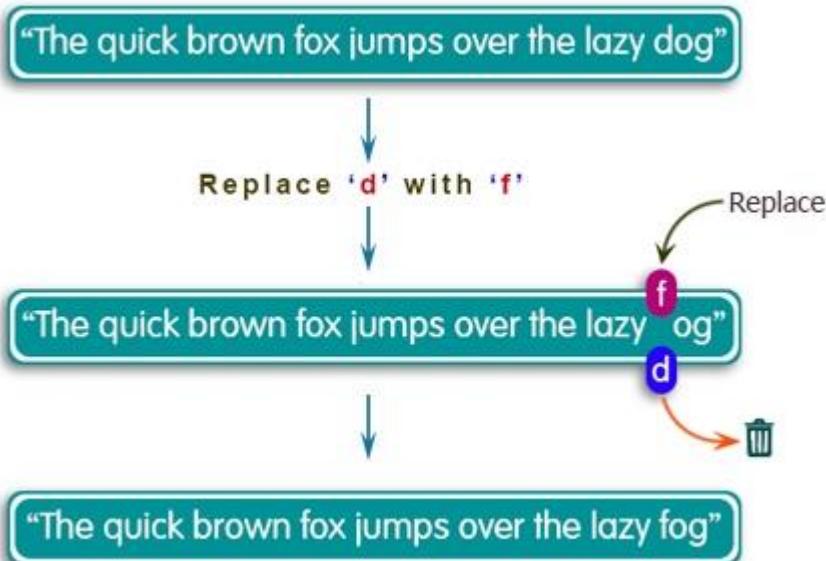
- **String toLowerCase()**: devuelve una copia de la cadena donde se han convertido las letras a minúsculas.
- **String toUpperCase()**: devuelve una copia de la cadena donde se han convertido las letras a mayúsculas.

```
String txt = "Hello World";
System.out.println(txt.toUpperCase()); // "HELLO WORLD"
```

```
System.out.println(txt.toLowerCase()); // "hello world"
```

- **String replace(char original, char otro)**: devuelve una copia de la cadena invocante donde se han sustituido todas las ocurrencias del carácter original por otro.
- **String replace(CharSequence original, CharSequence otro)**: igual que el anterior, pero en lugar de sustituir una carácter por otro, sustituye una cadena de caracteres por otra.

```
String myStr = "Hello";  
  
System.out.println(myStr.replace('l', 'p')); // "Heppo"  
System.out.println(myStr.replace("llo", "re")); // "Here"
```

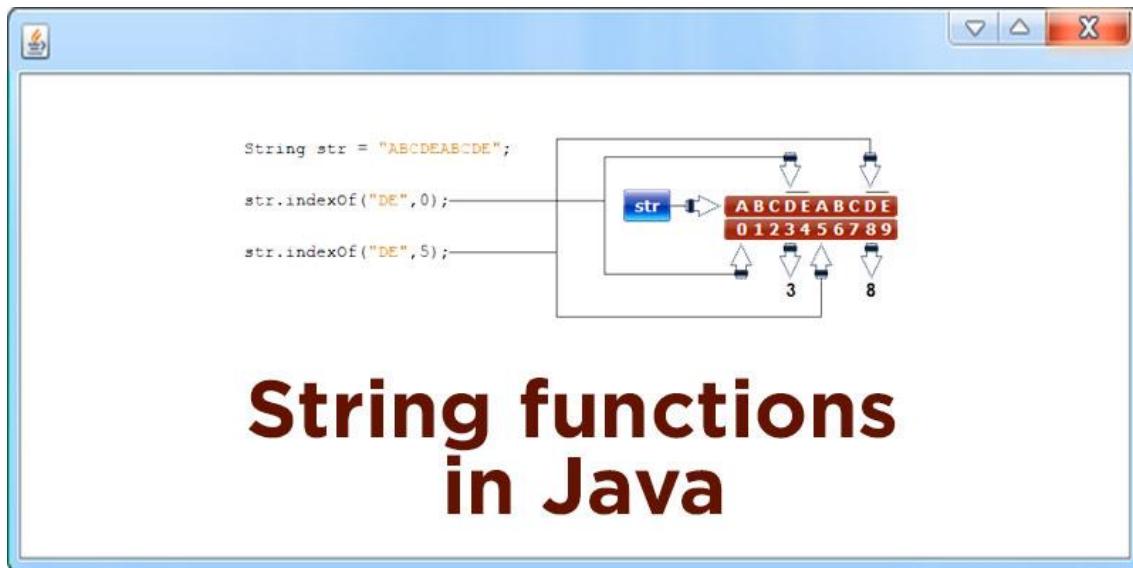


#### 11.2.8. Búsqueda

Dentro de un String tendremos la posibilidad de localizar tanto caracteres como subcadenas de caracteres con los siguientes métodos:

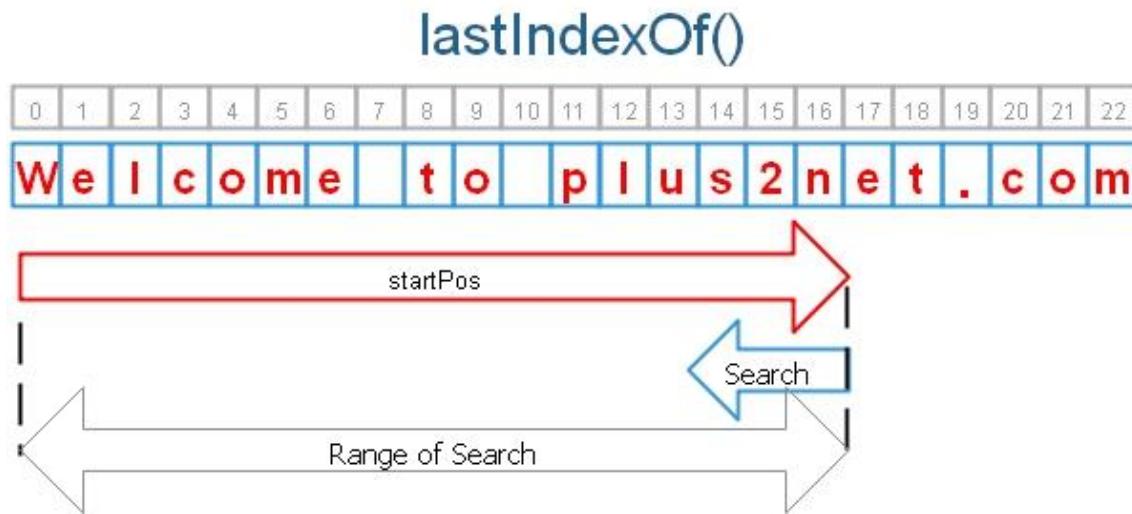
- **int indexOf(int c)**: nos devuelve la posición de la primera aparición del carácter c en una cadena de caracteres. Si no lo encuentra devuelve -1.
- **int indexOf(String cadena)**: igual que el anterior, pero busca una cadena de caracteres, en lugar de un único carácter.

- **int indexOf(int c, int inicio)**: igual que la primera versión, pero comienza a buscar desde la posición de inicio.
- **int indexOf(String cadena, int inicio)**: igual que la segunda versión, pero comienza a buscar desde la posición de inicio.



Los anteriores métodos buscaban la primera aparición de un carácter o una subcadena de caracteres. Las siguientes buscan la última aparición dentro de la cadena de caracteres.

- **int lastIndexOf(int c)**: nos devuelve la posición de la última aparición del carácter c en una cadena de caracteres. Si no lo encuentra devuelve -1.
- **int lastIndexOf (String cadena)**: igual que el anterior, pero busca una cadena de caracteres, en lugar de un único carácter.
- **int lastIndexOf (int c, int inicio)**: igual que la primera versión, pero comienza a buscar desde la posición de inicio hacia el principio de la cadena de caracteres.
- **int lastIndexOf (String cadena, int inicio)**: igual que la segunda versión, pero comienza a buscar desde la posición de inicio hacia el principio de la cadena de caracteres.



```
String myStr = "Hello planet earth, you are a great planet.";
System.out.println(myStr.lastIndexOf("planet")); //36
System.out.println(myStr.lastIndexOf("planet", 20)); //6
```

#### 11.2.9. Comprobaciones

Tenemos la posibilidad de comprobar si una cadena de caracteres es vacía mediante el método **isEmpty()**.

```
String cadenaVacia = "";
String cadenaHola = "Hola";

if(cadenaVacia.isEmpty()){ //true
    ...
}

if(cadenaHola.isEmpty()){ //false
    ...
}
```

Por otra parte, también nos es posible conocer si una subcadena está contenido dentro de otra cadena de caracteres. Esto lo haremos a través del método **contains()**.

```
String cadena = "En un lugar de la Mancha";
String palabra = "lugar";

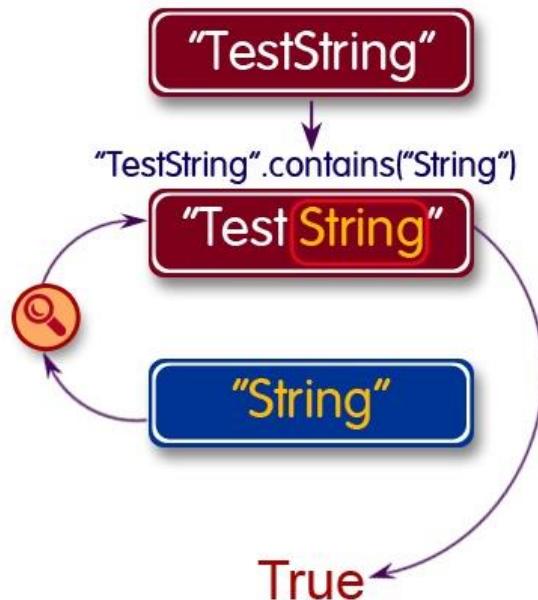
if(cadena.contains(palabra)){ //true
    ...
}
```

```

}

if(cadena.contains("Hola")){ //false
    ...
}

```



Finalmente, podemos comprobar si una cadena de caracteres comienza o termina con una subcadena determinada. Para ello haremos uso de:

- **boolean startsWith(String prefijo)**: comprueba si la cadena que invoca el método comienza con la subcadena prefijo.
- **boolean startsWith(String prefijo, int inicio)**: hace lo mismo que la anterior pero empieza a buscar desde la posición de inicio.
- **boolean endsWith(String sufijo)**: indica si la cadena termina con el sufijo que le pasamos por parámetro.

```

String cadena = "En un lugar de la Mancha";
String palabra = "lugar";

if(cadena.startsWith(palabra)){ //false
    ...
}

if(cadena.startsWith(palabra, 6)){ //true
    ...
}

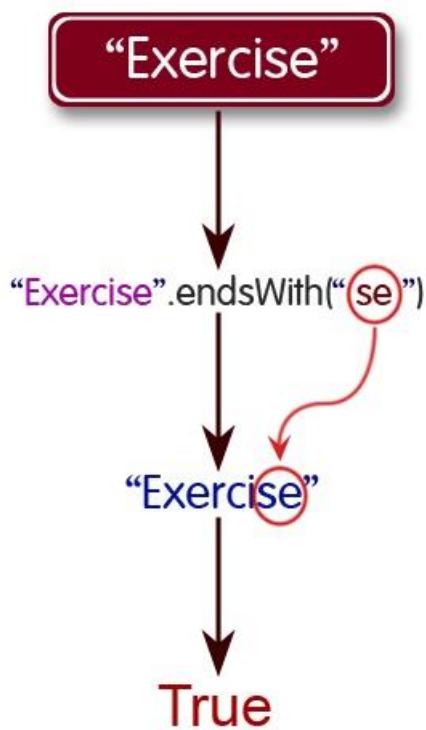
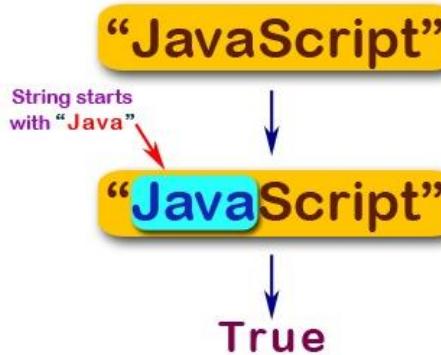
```

```

}

if(cadena.endsWith("Mancha")){ //true
    ...
}

```



#### 11.2.10. Separación en partes

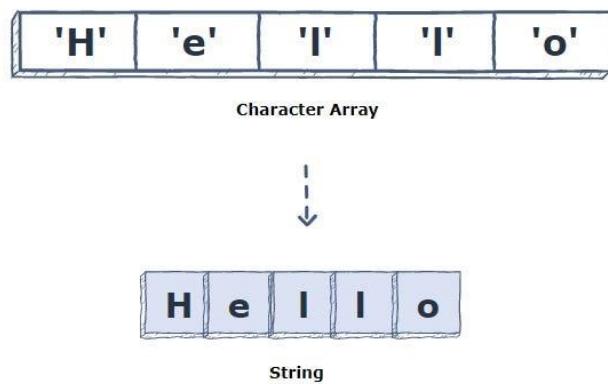
Otra funcionalidad muy útil con los Strings es la dividirlo en partes separadas por una subcadena. Por ejemplo “ ” o “;” suelen ser de los más habituales, aunque es posible utilizar subcadenas de mayor tamaño. Estas subcadenas no podrán contener ninguno de los siguientes caracteres (., +, \*, \$ o ?).

El método que utilizaremos para separar distintas partes será **String[] Split(String separador)**, que nos devolverá un array de Strings con las partes separadas y en el que habrán desaparecido todas las ocurrencias de la subcadena separador.

```
String cadena = "En un lugar de la Mancha";
String[] cadenaDividida = cadena.split(" ");
//cadenaDividida contiene {"En", "un", "lugar", "de", "la", "Mancha"}
```



### 11.3. CADENAS Y ARRAYS DE CARACTERES



Existe una innegable relación entre las cadenas de caracteres, clase String, y los arrays de caracteres char[]. Tal es así que nos va a ser posible convertir arrays de caracteres a Strings, ya sea de forma parcial o completa. Para ello usaremos el método valueOf() que vimos anteriormente.

- **static String valueOf(char[] charArray):** devuelve un String con el contenido del array que se pasa por parámetro.
- **static String valueOf(char[] charArray, int inicio, int longitud):** también devuelve un String, pero en este caso el contenido abarcará la longitud indicada a partir de la posición de inicio.

```
String cadena;

char charArray[] = {'H', 'o', 'l', 'a'};
cadena = String.valueOf(charArray);
//cadena tiene el valor "Hola"

charArray[] = {'a', 'b', 'c', 'd', 'e', 'f'};
cadena = String.valueOf(charArray, 2, 3);
//cadena tiene el valor "cde"
```

También podremos realizar la operación inversa, es decir, convertir un String en un array de caracteres mediante el método **char[] toCharArray()**.

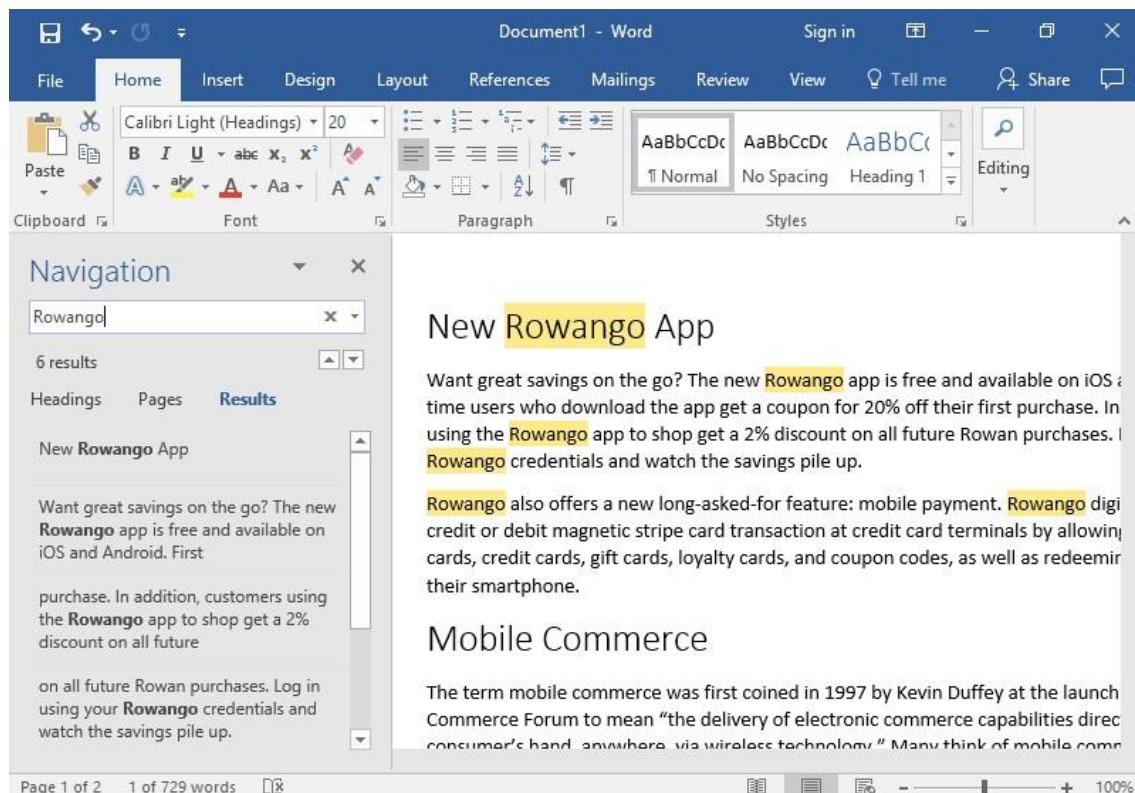
```
char[] charArray;
String cadena = "Hola mundo";

charArray = cadena.toCharArray();
//el valor de charArray es {'H','o','l','a',' ','m','u','n','d','o'}
```

## 12. EXPRESIONES REGULARES

Las expresiones regulares (comúnmente denominadas como *regex*) son secuencias de caracteres que conforman un patrón de búsqueda. Por lo general, estas búsquedas se realizan sobre cadenas de caracteres, es decir, sobre texto plano.

Actualmente las expresiones regulares tienen una gran utilidad puesto que se utilizar en una gran cantidad de ámbitos, desde el buscador de texto que incluyen las aplicaciones ofimáticas como Word o buscadores web, hasta la detección de expresiones clave en con correos electrónicos y mensajes de texto por parte de las fuerzas de seguridad, pasando por los validadores en la creación de cuentas de usuarios y contraseñas.



Las expresiones tienen una gran complejidad y su dominio requiere de un profundo conocimiento de la materia, es por ello que en esta unidad únicamente vamos a conocer cómo utilizarlas de una forma superficial.

Para hacer uso de las expresiones regulares debemos conocer los cuantificadores y metacaracteres con los que se construyen.

## 12.1. Cuantificadores para una expresión regular

Tenemos caracteres especiales que nos van a indicar el número de repeticiones de la expresión, la siguiente tabla muestra los caracteres:

Cuantificador	Descripción
n+	Encuentra cualquier string con al menos un «n»
n*	Encuentra cero o más ocurrencias de n
n?	Encuentra en el string la aparición de n cero o una vez
n{x}	Encuentra la secuencia de n tantas veces como indica x.
n{x,}	Encuentra una secuencia de n al menos tantas veces como indica n.

## 12.2. Metacaracteres en una expresión regular

Metacaracter	Descripción
	Símbolo para indicar OR.
.	Encuentra cualquier carácter
^	Sirve para hacer match al principio del string
\$	Hace match al final de un String
\d	Encuentra dígitos
\s	Busca un espacio
\b	Hace match al principio de una palabra.

### 12.3. Metacaracteres y ejemplos con expresiones regulares

Expresión regular	Descripción
.	Hace match con cualquier carácter
^regex	Encuentra cualquier expresión que coincide al principio de la línea.
regex\$	Encuentra la expresión que haga match al final de la línea.
[abc]	Establece la definición de la expresión, por ejemplo, la expresión escrita haría match con a, b o c.
[abc][vz]	Establece una definición en la que se hace match con a, b o c y a continuación va seguido por v o por z.
[^abc]	Cuando el símbolo ^ aparece al principio de una expresión después de [, negaría el patrón definido. Por ejemplo, el patrón anterior negaría el patrón, es decir, hace match para todo menos para la a, la b o la c.
[e-f]	Cuando hacemos uso de -, definimos rangos. Por ejemplo, en la expresión anterior buscamos hacer match de una letra entre la e y la f.
Y X	Establece un OR, encuentra la Y o la X.
HO	Encuentra HO (texto literal)
\$	Verifica si el final de una línea sigue.

## 12.4. Ejemplo simples de expresiones regulares

- Número entero de exactamente 6 dígitos: “\d{6}”
- Palabras en minúscula compuestas por 6 letras como máximo, con un espacio en blanco delante y otro detrás: “\b[a-z]{1,6}\b”
- Una línea compuesta por una única palabra compuesta por letras en mayúsculas y/o minúsculas: “^\*[a-zA-Z]+\$”
- Palabra que empieza por una letra mayúscula y el resto minúsculas: “[A-Z]{1}[a-z]”
- Contraseña compuesta por entre 8 y 15 caracteres alfanuméricos: “[a-zA-Z0-9]{8,15}”

En las siguientes páginas web podéis encontrar información sobre cómo crear expresiones regulares, ejemplos de algunas de las más usadas y simuladores para probarlas:

- <https://www.freeformatter.com/java-regex-tester.html#before-output>
- <https://geekebrains.com/code-brains/expresiones-regulares>
- <https://ihateregex.io/playground>

## 12.5. Uso de expresiones regulares en Java

En Java utilizaremos dos clases específicas para aplicar esta herramienta tan potente: **Matcher** y **Pattern**.

En primer lugar, necesitaremos importarlas:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

Y a continuación crearemos la instancia de cada una de ellas:

```
String prueba = "123456";
Pattern pattern = Pattern.compile("\\d{6}");
//cuando tengamos que poner una barra invertidas será necesaria poner dos juntas
Matcher matcher = pattern.matcher(prueba);
```

Finalmente utilizaremos uno de los métodos que nos proporciona la clase Matcher para comprobar la expresión regular con el texto:

- **boolean find():** comprueba si existe al menos una coincidencia de la expresión en todo el texto proporcionado.
- **boolean find(int ini):** comprueba si existe al menos una coincidencia de la expresión en el texto proporcionado a partir de la posición ini.
- **boolean lookingAt():** comprueba si existe una coincidencia de la expresión regular justo al inicio del texto.
- **boolean matches():** comprueba si el texto coincide exactamente con la expresión regular.

```
if(matcher.find()){
    System.out.println("Encontrado");
}
else{
    System.out.println("No encontrado");
}

//contar cuantas veces se ha encontrado coincidencia
contador = 0;
while(matcher.find()){
    contador++;
}

if(matcher.lookingAt()){
    System.out.println("Encontrado");
}
else{
    System.out.println("No encontrado");
}

if(matcher.matches()){
    System.out.println("Encontrado");
}
else{
    System.out.println("No encontrado");
}
```

# UT-05: INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

¿Qué vamos a ver?

- PROGRAMACIÓN ORIENTADA A OBJETOS
  - CONCEPTOS: CLASE - OBJETO.
  - PROPIEDADES DE LA P.O.O.
- CLASES.
  - CARÁCTERÍSTICAS.
  - CREAR UNA CLASE EN JAVA. FORMATO GENERAL.
  - DEFINICIÓN DE ATRIBUTOS O PROPIEDADES.
  - MODIFICADORES DE ACCESO.
- MÉTODOS.
  - COMUNICACIÓN ENTRE OBJETOS. ENVÍO DE MENSAJES.
  - CONSTRUCTORES.
  - MÉTODOS SETTER Y GETTER
  - EL MÉTODO MAIN.
- OBJETOS.
  - CREAR UN OBJETO.
  - ACCEDER A LOS ATRIBUTOS DE UN OBJETO.
- ATRIBUTOS Y MÉTODOS STATIC.
- CLASES DE INTERÉS

## 1. PROGRAMACIÓN ORIENTAD A OBJETOS

La programación orientada a objetos permite una representación más directa del modelo del mundo real en el código. El resultado es que se reduce considerablemente la transformación de los requisitos del sistema (definido en términos de usuario) a la especificación del sistema (definido en términos de computador). Es un paradigma de programación que usa objetos en sus interacciones, para diseñar aplicaciones y programas informáticos. Está basada en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

Fue a principios de la década de los 90 cuando su uso se popularizó y en la actualidad probablemente sea uno de los paradigmas de programación más utilizados existiendo gran variedad de lenguajes de programación que soportan la orientación a objetos.

### 2.1. Ventajas de la programación orientada a objetos

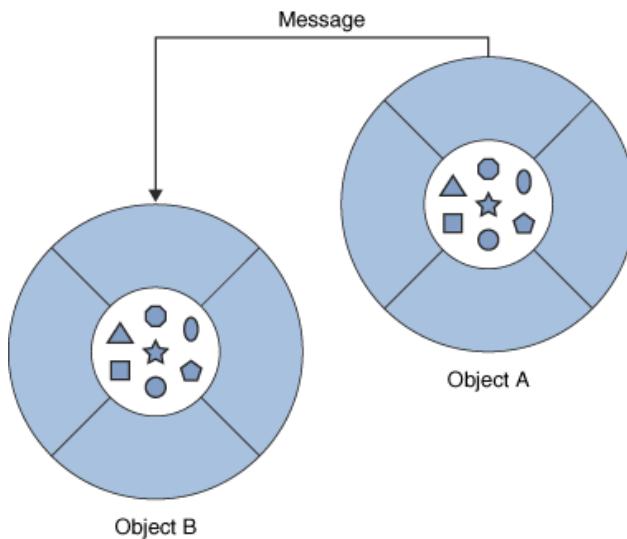
- **Comprensión:** Facilita la comprensión al representar los problemas en términos del mundo real. Es una representación más cercana a nuestra forma de pensar.
- **Modularidad:** Al estar las definiciones de objetos en módulos, hace que las aplicaciones estén mejor organizadas y sean más fáciles de entender.
- **Fácil mantenimiento:** Cualquier modificación en las acciones queda automáticamente reflejada en los datos.
- **Seguridad:** No se pueden modificar los datos de un objeto directamente, sino que se debe hacer mediante las acciones definidas para ese objeto.
- **Reutilización de código:** Facilita la reutilización mediante el uso de bibliotecas de clases.

### 2.2. Objetos

Un objeto se corresponde con una entidad del mundo real que se caracteriza por un estado (datos) y un comportamiento (operaciones).

- **Estado:** El estado de un objeto viene determinado por los valores que toman sus datos. Los datos se denominan también atributos o propiedades y componen la estructura del objeto. Por ejemplo, un objeto Coche su estado podría estar definido por atributos como Marca, Modelo, Color, etc.
- **Comportamiento:** El comportamiento de un objeto viene determinado por las operaciones (métodos) que se pueden realizar sobre el objeto. Por ejemplo, un objeto Coche su comportamiento podría estar definido por acciones como arrancar, parar, acelerar, etc.

Los objetos se comunican unos con otros llamando a sus métodos. Estos métodos determinan cómo actúan los objetos cuando reciben un mensaje. Un mensaje es la acción que hace un objeto. Y un método especifica cómo se ejecuta un mensaje.



Los objetos son entidades dinámicas, que existen en el tiempo; por ello deben ser creados o instanciados, normalmente a través de otros objetos. Esta operación se hace a través de los métodos especiales llamados constructores, que pueden ser ejecutados implícitamente por el compilador o explícitamente por el programador mediante la invocación a los citados constructores.

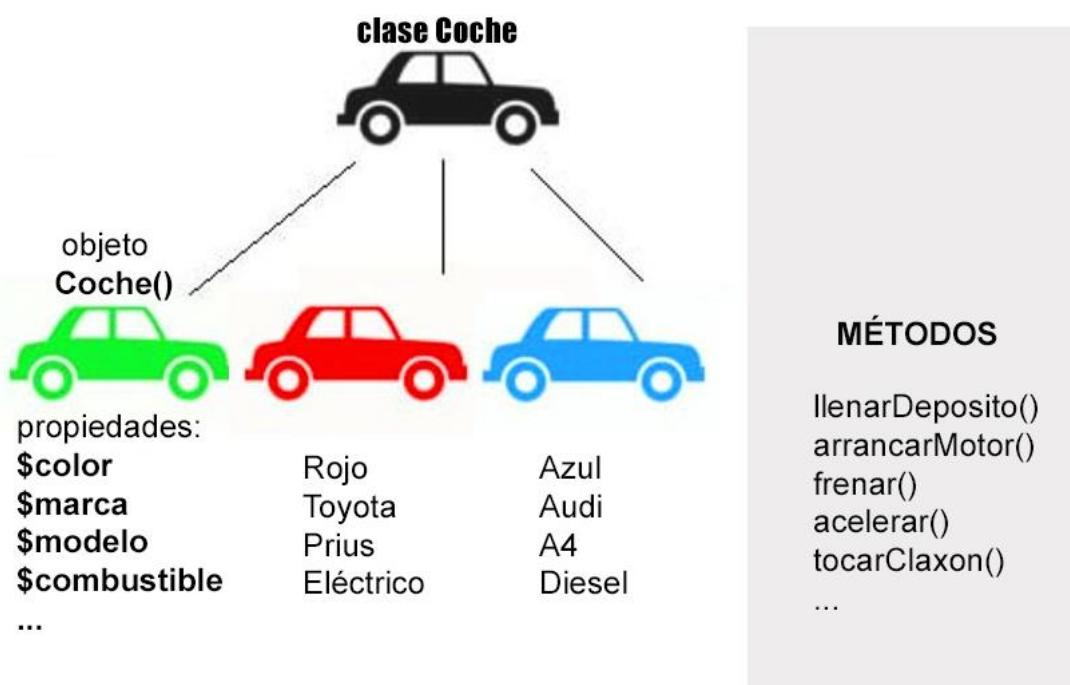
### 2.3. Clases

Una clase es la caracterización abstracta de un conjunto de objetos y define los datos que se utilizan para representar un objeto y las operaciones que se pueden realizar sobre esos datos. Se pueden definir muchos objetos de la misma clase. Es decir, una clase es la declaración de un tipo objeto.

Las clases son similares a los tipos de datos y equivalen a modelos o plantillas que describen:

- **Los atributos o propiedades comunes** a todos los objetos de la clase.
- **Los métodos o funciones** que son las operaciones que pueden utilizarse para manejar esos objetos.

Cada vez que se construye un objeto a partir de una clase, estamos creando lo que se llama una instancia de esa clase. En general, instancia de una clase y objeto son términos intercambiables. Veamos un ejemplo:



## 2.4. Propiedades de la programación orientada a objetos

- Abstracción
- Encapsulación
- Herencia
- Polimorfismo

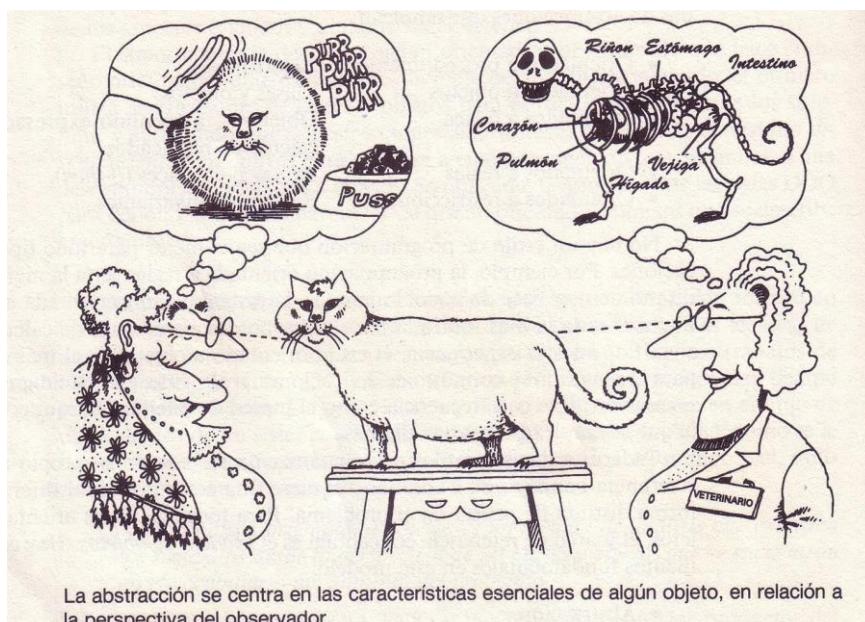
### *2.4.1. Abstracción*

La abstracción es la propiedad que permite representar las características esenciales de un objeto desde un punto de vista determinado y no tener en cuenta las restantes características, reduciendo así la complejidad. Es decir, permite no preocuparse por los detalles no esenciales.

Por ejemplo, diferentes modelos de abstracción del objeto Coche podrían ser:

- Un coche es la combinación de diferentes partes, tales como motor, carrocería, número de puertas, etc.
- Un coche puede clasificarse por el nombre del fabricante, por su categoría (turismo, deportivo, todoterreno ...), por el carburante que utilizan (gasolina, gasoil, híbrido ...).

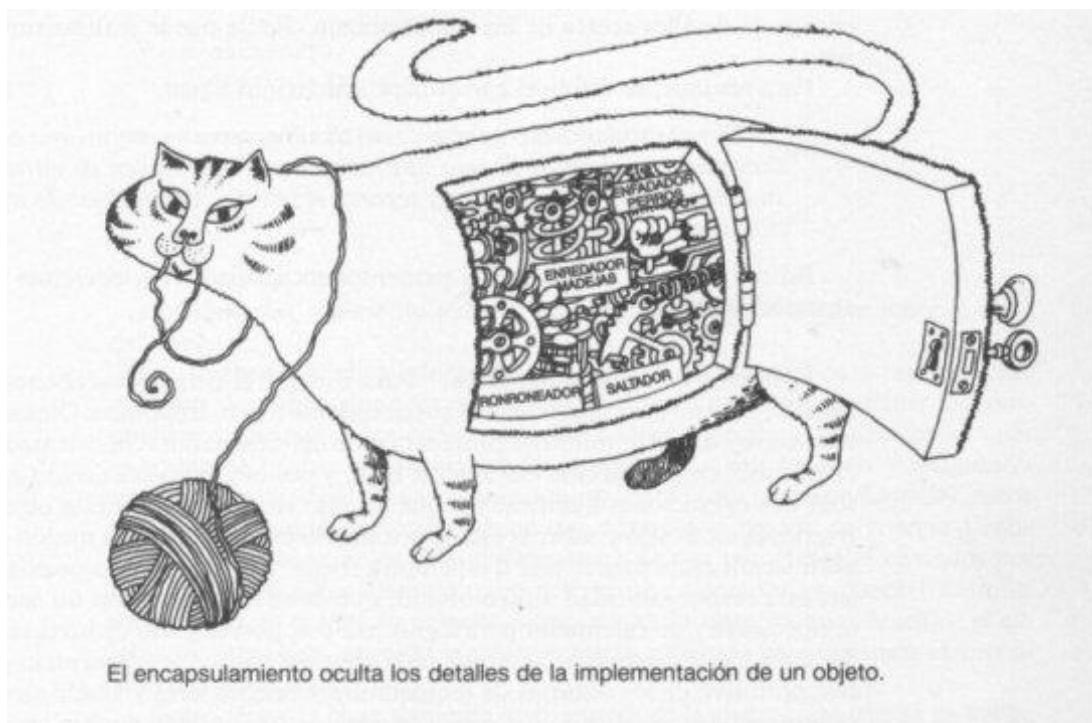
Por ejemplo, la segunda abstracción vista anteriormente se utilizará siempre que la marca, la categoría o el carburante sean significativos. Por tanto, la abstracción a utilizar dependerá de la naturaleza del problema a resolver.



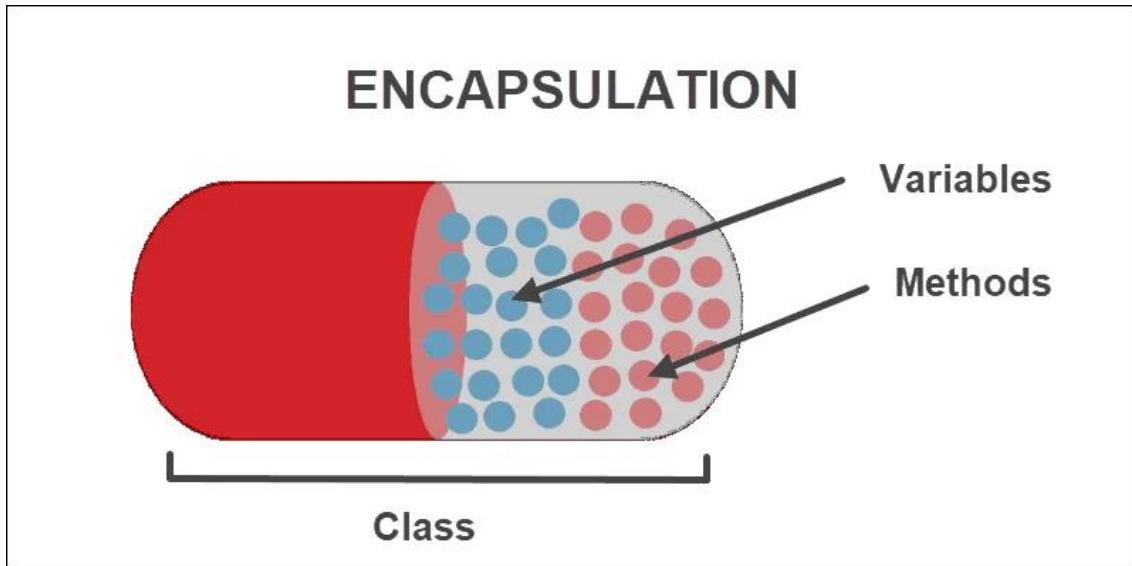
#### 2.4.2. Encapsulación de datos

La encapsulación es la propiedad que permite asegurar que el contenido de la información de un objeto está oculto. La encapsulación (también se conoce como ocultación de la información), en esencia, es el proceso que oculta todos los secretos de un objeto que contribuyen a sus características esenciales.

La encapsulación permite la división de un programa en módulos. Estos módulos se implementan mediante clases, de forma que una clase representa la encapsulación de una abstracción. En la práctica, esto significa que cada clase debe tener dos partes: una interfaz y una implementación. La interfaz de una clase captura sólo su vista externa y la implementación contiene la representación de la abstracción, así como los mecanismos que realizan el comportamiento deseado.



El encapsulamiento oculta los detalles de la implementación de un objeto.

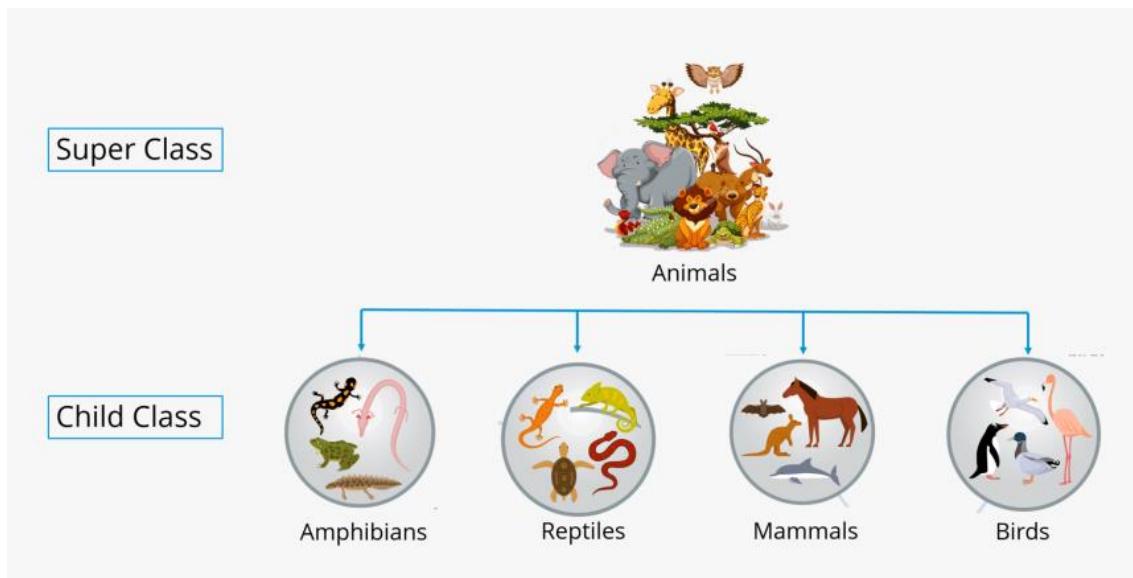


### 2.4.3. Herencia

La herencia es la propiedad que permite a los objetos ser construidos a partir de otros objetos. La idea fundamental es permitir crear nuevas clases aprovechando las características (atributos y métodos) de otras clases ya creadas evitando así tener que volver a definir esas características (reutilización).

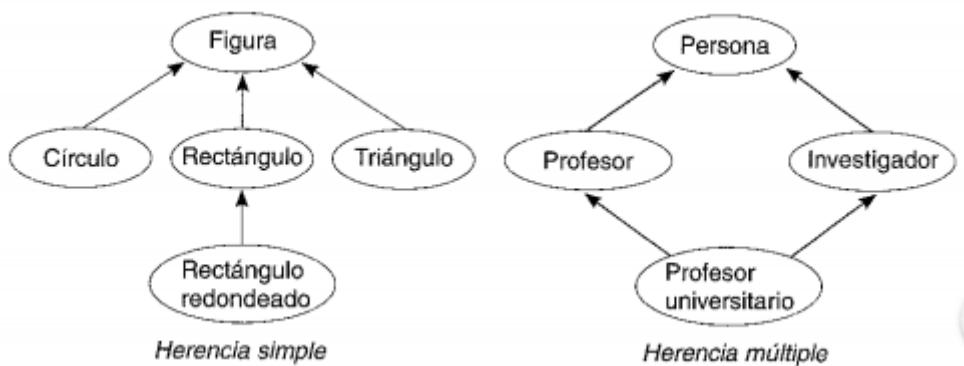
A una clase que hereda de otra se le llama subclase (clase derivada, clase hija) y aquella de la que se hereda es conocida como superclase (clase base, clase padre).

Las clases derivadas heredan el código y datos de su clase base, añadiendo su propio código y datos, incluso puede cambiar aquellos elementos de la clase base que necesita que sean diferentes. Por ejemplo, la clase animal se puede dividir en las subclases anfibios, mamíferos, aves, etc., y la clase vehículo en coches, camiones, etc.



Existen dos tipos de herencia:

- **Herencia simple:** Una subclase puede heredar datos y métodos de una única clase. Java admite solo herencia simple.
- **Herencia múltiple:** Una subclase puede heredar datos y métodos de más de una clase. C++ admite herencia simple y múltiple.



La herencia tiene como ventajas principales la reutilización y compartición de código (evita repetir código ahorrando mucho tiempo), la consistencia de la interfaz (el comportamiento que heredan será el mismo en todos los casos) y la ocultación de información. Pero también tiene inconvenientes como son la velocidad de ejecución (los métodos heredados suelen ser más lentos) y la curva

de aprendizaje (al principio el aprendizaje de la programación orientada a objetos suele ser más lento).

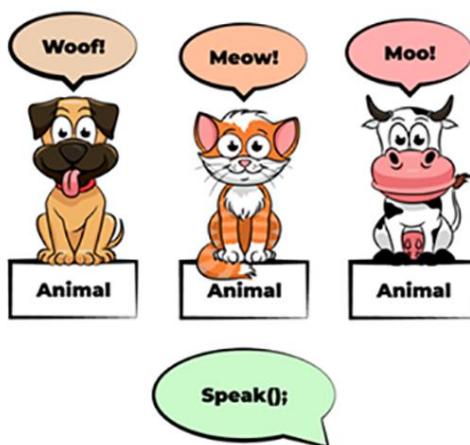
#### 2.4.4. Polimorfismo

El polimorfismo es la propiedad que permite que un operador o un método actúen de modo diferente en función del objeto sobre el que se aplican. Cuando un operador existente en el lenguaje tal como +, = o \* se le asigna la posibilidad de operar sobre un nuevo tipo de datos, se dice que está sobrecargado. La sobrecarga es una clase polimorfismo.

En un sentido más general, el polimorfismo supone que un mismo mensaje puede producir acciones totalmente diferentes cuando se reciben por objetos diferentes.

El polimorfismo adquiere su máxima expresión en la herencia de clases, es decir, cuando se obtiene una clase a partir de una clase ya existente.

Por ejemplo, cuando se describe una clase base Mamífero se puede observar que la operación comer es una operación fundamental en su vida, de modo que cada tipo de mamífero (vaca, humano, león) debe poder realizar la operación comer, aunque cada una de las clases derivadas que representan los distintos tipos de mamíferos la realizará de un modo diferente (polimorfismo).



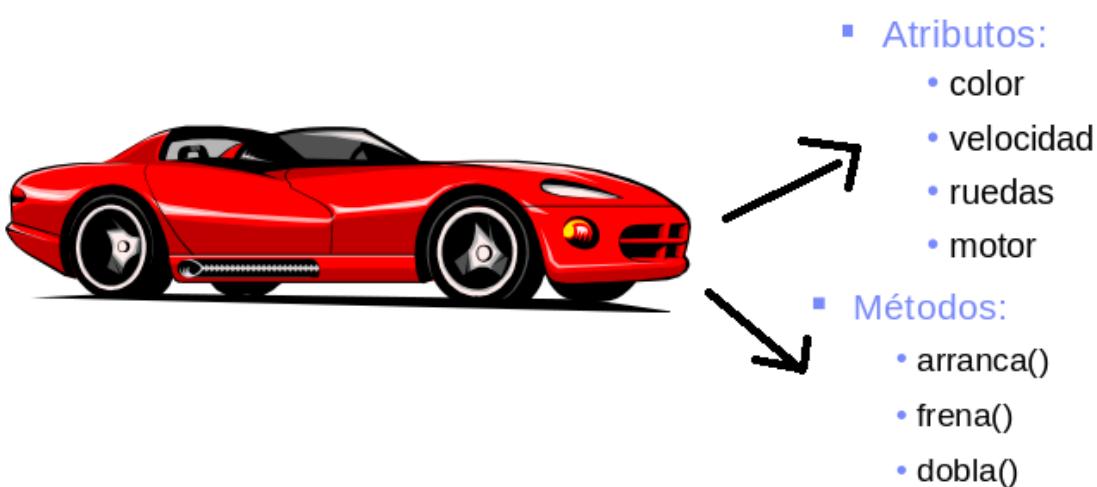
## 2. CLASES

### 2.1. Características

Como hemos visto anteriormente una clase se caracteriza por sus atributos (estado) y sus métodos (comportamiento).

Por un lado, los atributos almacenarán datos que consideremos necesarios para la clase y los objetos que te van a crear a partir de ella.

Por otro lado, los métodos serán acciones que la clase podrá llevar a cabo sobre ella o sobre otros elementos.



### 2.2. Crear un clase en java. Formato general.

Aunque será a partir de la siguiente unidad en la que nos centraremos en la creación de clases, veamos a grandes rasgos cual es estructura general que suelen tener.

Podemos distinguir dos tipos de clases, las principales y las secundarias.

Las principales son las primeras que arranca el programa y contienen el método main.

```
package ut06;

import java.util.Scanner;

public class UT06 {

    private static int numero;

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String nombre;

        System.out.print("Dame tu nombre: ");
        nombre = scan.nextLine();
        saludar(nombre);
    }

    public static void saludar(String nombre){
        System.out.println("¡Hola " + nombre + "!");
    }

    public class SubUT06 {
        ...
    }
}
```

Las clases principales suelen tener la siguiente estructura:

- Paquete al que pertenecen
- Importaciones de clases de las que se hace uso
- Declaración de la clase
- Variables y/o constantes globales (no aconsejables)
- Método main
- Métodos auxiliares que realizan diversas acciones
- Definición de subclases

Las clases secundarias se utilizan para modularizar el código y hacerlo mucho más legible y organizado. Las vamos a utilizar directamente o a través de creación de objetos para interactuar con la clase principal u otras clases secundarias.

Las clases secundarias suelen tener la siguiente estructura:

```
package ut06;

public class Employee {
    //Atributos
    private String name;
    private int employeeNum;
    private final int COMISIONPERCENTAGE = 10;

    //Método constructor
    public Employee(String name, int employeeNum) {
        this.name = name;
        this.employeeNum = employeeNum;
    }

    //Métodos getters
    public String getName() {
        return name;
    }

    public int getEmployeeNum() {
        return employeeNum;
    }

    //Métodos setters
    public void setName(String name) {
        this.name = name;
    }

    public void setEmployeeNum(int employeeNum) {
        this.employeeNum = employeeNum;
    }

    //Otros métodos
    public double calculateComision(double saleValue) {
        return saleValue * (COMISIONPERCENTAGE/100);
    }
}
```

- Paquete al que pertenecen
- Declaración de la clase
- Atributos
- Método constructor
- Métodos getters

- Métodos setters
- Otros métodos

### 2.3. Definición de atributos o propiedades

Los atributos son los encargados de conservar el estado de las clases, es decir, almacena información acerca de ella, que tendrá diferentes valores para cada objeto que se cree por medio de esa clase.



Nombre:	Fernanda	→	str
Edad:	28	→	int
Dirección:	Los Jacintos 1329	→	str
Estatura:	1.65	→	float
Usa lentes?	Sí	→	bool



Nombre:	Héctor	→	str
Edad:	20	→	int
Dirección:	Maratón 94 Dp. 31	→	str
Estatura:	1.76	→	float
Usa lentes?	No	→	bool

Podemos definir los atributos como variables internas de la clase, con la particularidad de que normalmente no se suele permitir el acceso directo a estas variables desde fuera de la clase, sino utilizamos unos métodos especiales tanto para conocer sus valores como para asignarles unos nuevos.

Como variables, o constantes, que son las definiremos de la misma manera que hemos hecho hasta el momento, es decir, el nombre de la variable irá precedido del tipo de dato que va a almacenar.

Para acceder a un atributo de un objeto, en el caso de que nos esté permitido, solo tenemos que poner un punto después de la clase u objeto del tipo clase, según el tipo de clase, y el nombre del atributo. Ejemplo:

```
String valor = objeto.atributo;
```

## 2.4. Modificadores de acceso

Como hemos comentado en el apartado anterior, normalmente no se permite el acceso a los atributos desde fuera de la clase, para esto utilizamos lo que se denomina modificadores de acceso.

Un modificador de acceso es una palabra clave que antepondremos al tipo de datos del atributo y con el que indicaremos si solo se puede acceder a él desde la propia clase o también se podrá acceder desde fuera.

Por ahora vamos a utilizar tres modificadores de acceso:

- **private**: solo permite que accedamos a la variable, o constante, desde la propia clase que lo ha declarado, ni siquiera desde el propio paquete en el que se encuentra la clase. Es decir, que cuando utilicemos un objeto con un atributo private, no vamos a poder llamar a este como habíamos visto en el apartado anterior. Va a ser solo de uso interno.

```
private String mensaje;
```

- **default**: es el modificador de acceso que se asigna por defecto al atributo si no especificamos ningún otro modificador, es decir, si no ponemos nada delante del tipo de dato del atributo. Este modificador nos permitirá acceder al atributo desde la misma clase o desde alguna otra clase que se encuentre en el mismo paquete que ella.

```
String mensaje;
```

- **public**: normalmente es el menos aconsejable, puesto que permite el acceso al atributo desde cualquier sitio.

```
public String mensaje;
```

Modificador (palabra reservada)	Misma clase	Mismo paquete	Subclase de otro paquete	Resto
<i>private</i>	✓	✗	✗	✗
<i>(defecto)</i>	✓	✓	✗	✗
<i>public</i>	✓	✓	✓	✓

### 3. MÉTODOS

#### 3.1. Comunicación entre objetos. Envío de mensajes.

La programación orientada a objetos resuelve los problemas mediante la interacción de los objetos. Esta interacción, o comunicación, se lleva a cabo mediante intercambio de mensajes, esos mensajes son recibidos por los métodos y estos son los encargados de dar respuesta o actuar en consecuencia al mensaje. Por tanto, los métodos son los encargados de gestionar el comportamiento de la clase u objeto.

Podríamos decir que los métodos son subprogramas dentro de los programas, al que le asignamos a cada uno de ellos una labor que realizar, y al que llamamos cada vez que necesitemos llevar a cabo dicha labor.

La codificación de los métodos consiste en dos partes, la cabecera y el cuerpo.

- La cabecera está compuesta por un modificador de acceso, el tipo de dato que devuelve, el nombre del método y, entre paréntesis, los argumentos que debemos pasarle con su tipo de dato correspondiente.

```
public double calculateComision(double saleValue)
```

- El cuerpo es el código que va a determinar qué hace el método, y se escribe entre llaves {}. Además, si se trata de una función, es decir, un método que devuelve un resultado, debemos hacer uso de la palabra

reservada “return” cuando queramos finalizar el método y devolver dicho resultado.

```
{  
    return saleValue * (COMISIONPERCENTAGE/100);  
}
```

El método de ejemplo quedaría de la siguiente manera:

```
public double calculateComision(double saleValue) {  
    return saleValue * (COMISIONPERCENTAGE/100);  
}
```

Para llamar a un método utilizamos la clase o un objeto de esa clase, en función del tipo clase, colocamos un punto a continuación, después el nombre del método y entre paréntesis los argumentos que sean necesarios para realizar la llamada.

```
objeto.metodo(argumento1, argumento2);
```

### 3.2. Constructores

Los constructores son unos métodos especiales de las clases, a través de los cuales creamos los objetos de esas clases. Un constructor tendrá el mismo nombre que la clase y recibirá una serie de parámetros o ninguno, en función de las necesidades de la clase. Normalmente, además de crear en memoria el objeto, se suelen utilizar para inicializar los valores de los atributos de la clase.

```
public class Employee {  
    //Atributos  
    private String name;  
    private int employeeNum;  
    private final int COMISIONPERCENTAGE = 10;  
  
    //Método constructor  
    public Employee(String name, int employeeNum) {  
        this.name = name;  
        this.employeeNum = employeeNum;  
    }  
}
```

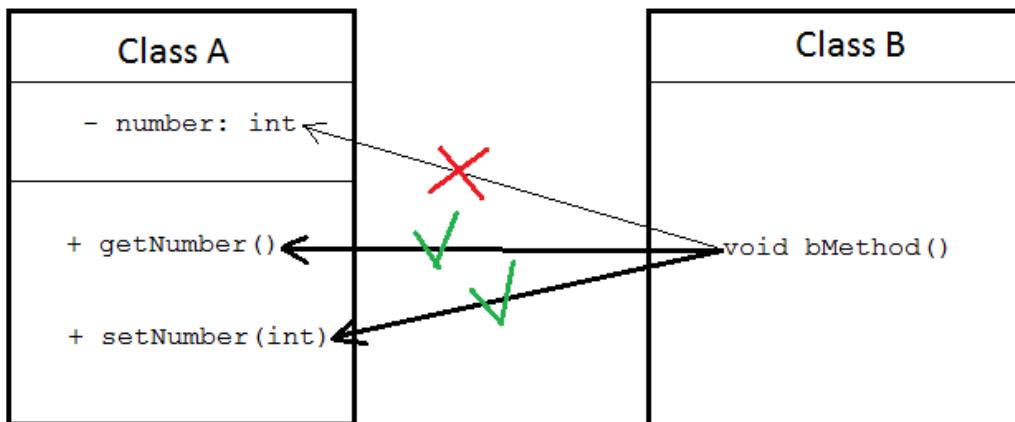
Habitualmente una clase suele tener un único constructor, pero en ocasiones puede tener varios, que se diferencian por el número de parámetros que se le pasan. Incluso hay un tipo especial de clases, sobre las que hablaremos más adelante, que no utilizan constructores, es decir, no se pueden crear objetos a partir de esa clase.

A excepción de la clase principal, los constructores son llamados desde otras clases. Estas llamadas se realizan con una estructura similar a la de la declaración de una variable, ya que los objetos, en realidad, son variables del tipo de la clase, pero utilizando la palabra clave “new”.

```
ClaseAlumno alumno = new ClaseAlumno(argumento1);
```

### 3.3. Métodos getters y setters

Son otro tipo especial de métodos que utilizamos para acceder a los atributos desde fuera de la clase, en el caso de que sean “private”.



Los métodos getters nos devolverán el valor de un atributo determinado.

Internamente están construidos como un método normal, pero habitualmente se le da como nombre “getNombreAtributo”, no recibe argumentos y devuelve datos del tipo del atributo.

```
public String getName() {  
    return name;  
}
```

A la hora de llamarlos, se hace igual que un método igual, poniendo “.” Después del objeto, el nombre del método, seguido de “()”.

```
objeto.getNombre();
```

Los métodos setters nos servirán para asignar un valor al atributo correspondiente.

Al igual que los getters, están estructurados como un método normal, pero en este caso no devuelve valor, es decir, pondremos “void” como tipo de dato que devuelve, y como argumento tendremos el valor que quiere asignar con el tipo de dato correspondiente.

```
public void setName(String name) {  
    this.name = name;  
}
```

Para llamar a un método setter, lo haremos igual que otro tipo de método, y le pasaremos como parámetro el valor que queremos asignar al atributo.

```
objeto.setNombre("Juan");
```

Tanto los getters como los setters son herramientas imprescindibles para mantener la encapsulación y el principio de ocultación de la información.

Es importante tener en cuenta que no es necesario crear getters o setters para todos los atributos:

- Podemos tener atributos solo con método getter, es decir, no se le puede modificar el valor que almacenan de forma externa.
- Podemos tener atributos solo con método setter, es decir, solo podemos modificar su valor, pero no leerlo, aunque es menos común.

- Podemos tener atributos sin getter ni setter, ya sea por que es público y no lo necesita, o por que es privado y no queremos que se acceda a él de ninguna manera desde fuera de la clase.

### 3.4. Método main

El método main es el más importante de un programa. Únicamente se encuentra en la clase principal y siempre es el primer método que se ejecuta, es decir, que es el encargado de llamar al resto cuando sea necesario.

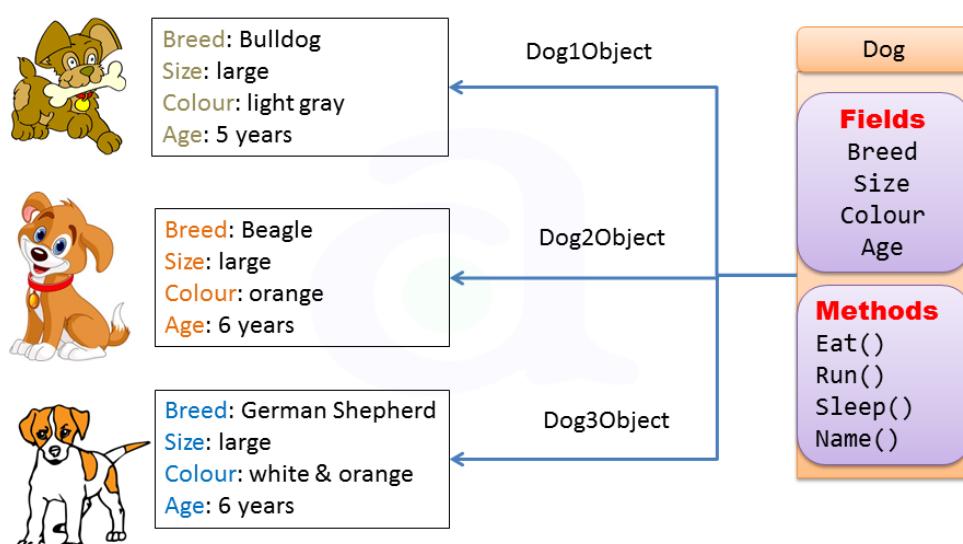
El encabezado del método main siempre es el mismo:

```
public static void main(String[] args)
```

## 4. OBJETOS

Los objetos los obtenemos gracias la instanciación de las clases, es por ello que también son llamados instancias de las clases. Esto se lleva a cabo por medio de los métodos constructores.

Podríamos decir que las clases son plantillas mediante las cuales creamos objetos similares del mismo tipo.



Cada objeto derivado de una clase contará con los mismos atributos y métodos que esta, pero los valores de los atributos serán distintos.

A continuación, tenemos un ejemplo de cómo crearíamos los tres objetos de clase Dog, y cómo utilizar sus métodos y atributos.

```
Dog Dog1Object = new Dog ("Bulldog", "large", "light gray", 5);
Dog Dog2Object = new Dog ("Beagle", "large", "orange", 6);
Dog Dog3Object = new Dog ("German Shepherd", "large", "white & orange",
6);

Dog1Object.eat();
Dog2Object.run();
Dog3Object.sleep();

Dog2Object.setBreed("Chiguagua");

System.out.println("The breed of my dog is " + Dog2Object.getBreed());
```

## 5. ATRIBUTOS Y MÉTODOS ESTÁTICOS

Existen clases que contienen una serie de atributos y métodos especiales a los que denominamos estáticos o de clase.

### 5.1. Atributos estáticos

Los atributos estáticos son atributos que tienen el mismo valor para todas las instancias, u objetos, de una clase. Este atributo puede ser una constante, o puede ser variable, de tal forma que si se modifica se modificará para todos los objetos de esa clase.

La forma de acceder a un atributo estático, o de clase, es igual que para un atributo de instancia.

Aunque la siguiente unidad aprenderemos a crearlos y el sentido que tienen a la hora de construir nuestras propias clases, a continuación, tenemos un ejemplo de cómo se declara una constante y una variable estática:

```
static final int VALOR_MAXIMO = 100;  
  
static int numEmpleados;
```

## 5.2. Métodos estáticos

Los métodos estáticos o de clase, son métodos que se caracterizan por que pueden ser utilizados sin necesidad de crear un objeto de la clase, es decir, que simplemente utilizando la clase para llamarlos.

Si recordáis, cuando hablamos de algunas clases de las que llamamos “wrappers”, como Character, ya utilizamos algunos de estos métodos.

```
Character.isLowerCase(letra);
```

En las clases principales todos los métodos secundarios que creemos deberán ser estáticos, debido a la naturaleza de esta clase. Para declarar un método estático solo tenemos que añadir la palabra reservada “static” detrás del modificador de acceso y delante del tipo de datos que va a devolver.

```
public static void saludar(){  
    System.out.println("Hola a todos");  
}  
  
private static int devuelveMayor(int num1, int num2){  
    if(num1 > num2)  
        return num1;  
    else  
        return num2;  
}
```

## 6. CLASES DE INTERÉS

A continuación tenemos un listado de clases, algunas ya conocidas y otras no, que son de mucha utilidad a la hora de desarrollar aplicaciones. Varias necesitan ser instanciadas para utilizar sus métodos y atributos, mientras que otras no.

- `java.util.Random` (necesita instanciarse)
- `java.util.Scanner` (necesita instanciarse)
- `Byte`
- `Short`
- `Integer`
- `Long`
- `Float`
- `Double`
- `Boolean`
- `Character`
- `String`
- `java.time.LocalDate`
- `java.time.LocalTime`
- `java.time.LocalDateTime`
- `Math`

# **UT-06: DESARROLLO DE CLASES E INSTACIACIÓN DE OBJETOS**

¿Qué vamos a ver?

- Concepto de clase.
- Estructura y miembros de una clase.
- Visibilidad. Modificadores de clase, de atributos y de métodos.
- Creación de atributos. Declaración e inicialización.
- Creación de métodos. Declaración, argumentos y valores de retorno.
- Paso de parámetros. Paso por valor y paso por referencia.
- Constructores.
- Instanciación de objetos. Declaración y creación.
- Características de los objetos.
- Métodos recursivos
- Sobrecarga de métodos.
- Métodos estáticos.
- Librerías y paquetes de clases. Utilización y creación.
- Documentación sobre librerías y paquetes de clases.

## 1. CONCEPTO DE CLASE

En la unidad anterior ya vimos que una clase es la caracterización abstracta de un conjunto de objetos y define los datos que se utilizan para representar un objeto y las operaciones que se pueden realizar sobre esos datos, pero en esta unidad ya vamos a ver cómo crear nuestras propias clases desde 0, las cuales podremos utilizar, y reutilizar, en los programas que creemos.

## 2. ESTRUCTURA Y MIEMBROS DE UNA CLASE

En esta unidad, nos vamos a centrar en la creación de clases secundarias. Recuperaremos el ejemplo de la unidad anterior en la veíamos, de modo general, cómo se estructuraban estas clases.

```
package ut06;

public class Employee {
    //Atributos
    private String name;
    private int employeeNum;
    private final int COMISIONPERCENTAGE = 10;

    //Método constructor
    public Employee(String name, int employeeNum) {
        this.name = name;
        this.employeeNum = employeeNum;
    }

    //Métodos getters
    public String getName() {
        return name;
    }

    public int getEmployeeNum() {
        return employeeNum;
    }

    //Métodos setters
    public void setName(String name) {
        this.name = name;
    }

    public void setEmployeeNum(int employeeNum) {
        this.employeeNum = employeeNum;
    }
}
```

```

    }

//Otros métodos
public double calculateComision(double saleValue) {
    return saleValue * (COMISIONPERCENTAGE/100);
}

```

En el código podemos distinguir:

- Paquete al que pertenecen
- Declaración de la clase
- Atributos
- Método constructor
- Métodos getters
- Métodos setters
- Otros métodos

En los siguientes apartados veremos cómo crear cada uno de estos elementos en nuestras propias clases.

Como vimos en la unidad anterior, tanto las clases como los métodos y los atributos puede tener asociado un modificador de acceso, que será el que determine qué otras clases pueden acceder a ellos. Hasta el momento conocemos tres tipos: private, defecto y public.

Modificador (palabra reservada)	Misma clase	Mismo paquete	Subclase de otro paquete	Resto
<i>private</i>	✓	✗	✗	✗
<i>(defecto)</i>	✓	✓	✗	✗
<i>public</i>	✓	✓	✓	✓

### 3. CREACIÓN DE ATRIBUTOS. DECLARACIÓN E INICIALIZACIÓN

Los atributos son variables y/o constantes que se encuentran dentro la clase, de tal forma que cuando instanciamos un objeto, cada uno de ellos tendrá un valor determinado. Los atributos pueden ser de tipos primitivos o compuestos como arrays, e incluso puede almacenar objetos.

Con respecto a los atributos siempre aplicaremos el principio de ocultación, es decir, que le aplicaremos el modificador de acceso más restrictivo que nos sea posible, y nos apoyaremos en los métodos getters y setters para controlar los accesos que se hagan desde fuera a los datos que contiene el objeto.

Para declarar un atributo de una clase solo tendremos que escribir el modificador de acceso el tipo o clase a la que pertenece y el nombre del atributo.

Como ocurría con las variables y constantes podemos inicializar los atributos en el mismo momento que se crean o más adelante. Una práctica muy habitual es inicializar los atributos en la llamada al constructor de la clase, pero eso lo veremos más adelante.

```
public class Employee {  
    //Atributos  
    private int employeeNum;  
    private FichaEmpleado datos;  
    private final int COMISIONPERCENTAGE = 10;  
}
```

En la construcción de clases podemos hacer uso de un tipo especial de atributos, los atributos estáticos, o de clase. Estos atributos se identifican poniendo la palabra clave “static” entre el modificador de acceso y el tipo de dato de la variable o constante.

Cuando utilizemos una variable estática en la construcción de nuestra clase, significará que todas las instancias, u objetos, de esa clase tendrán el mismo valor a la vez y si este cambia, será cambiado en todas a la vez.

Pueden ser utilizado para crear valores constantes que afecten a todos los objetos y que no se vayan a modificar, o para crear un contador de instancias, entre otras posibilidades.

```
public class Empleado {  
    private static final COMISION = 10;  
    private static int numEmpleados = 0;  
    private int idEmpleado;  
    private String nombreEmpleado;  
  
    public Empleado (String nombreEmpleado){  
        this.numEmpleados++;  
        idEmpleado = numEmpleados;  
        this.nombreEmpleado = nombreEmpleado;  
    }  
  
    ...  
}
```

## 4. CREACIÓN DE MÉTODOS. DECLARACIÓN, ARGUMENTOS Y VALORES DE RETORNO

Los métodos representan las acciones que pueden realizar los objetos o las que se pueden realizar sobre ellos.

La codificación de los métodos consiste en dos partes, la cabecera y el cuerpo.

### 4.1. Cabecera de un método

La cabecera está compuesta por un modificador de acceso, el tipo de dato que devuelve, el nombre del método y, entre paréntesis, los argumentos que debemos pasarle con su tipo de dato correspondiente.

```
public double calculateComision(double saleValue)
```

Los tipos de datos que puede devolver un método son tanto primitivos, como compuestos (arrays) e incluso clases. También, debemos tener en cuenta que en el caso de que no necesitemos que nuestro método devuelva ningún valor, en lugar del tipo de dato vamos a poner la palabra clave “void”.

Con respecto a los argumentos, debemos tener en cuenta que existen dos tipos de paso de parámetros, los denominados **paso de parámetros por valor** y **paso de parámetros por referencia**.

#### 4.1.1. Paso de parámetros por valor

Cuando utilizamos un **paso de parámetro por valor** lo que pasamos a la función es una copia del valor que se introduce en el parámetro al llamar al método.



Esto implica que cualquier cambio que realicemos al valor durante la ejecución del método solo se mantendrá durante la misma, y cuando finalice y volvamos al método main el valor continuará como cuando llamamos al método. Veamos un ejemplo:

```
package ut07;

public class UT07 {

    public static void main(String[] args) {
        int numero = 5;
```

```

        int numero2 = aumentarNumero(numero);

        System.out.println("numero tiene el valor " + numero);
        System.out.println("numero2 tiene el valor " + numero2);
    }

    private static int aumentarNumero(int numero){
        numero++;
        return numero;
    }

}

```

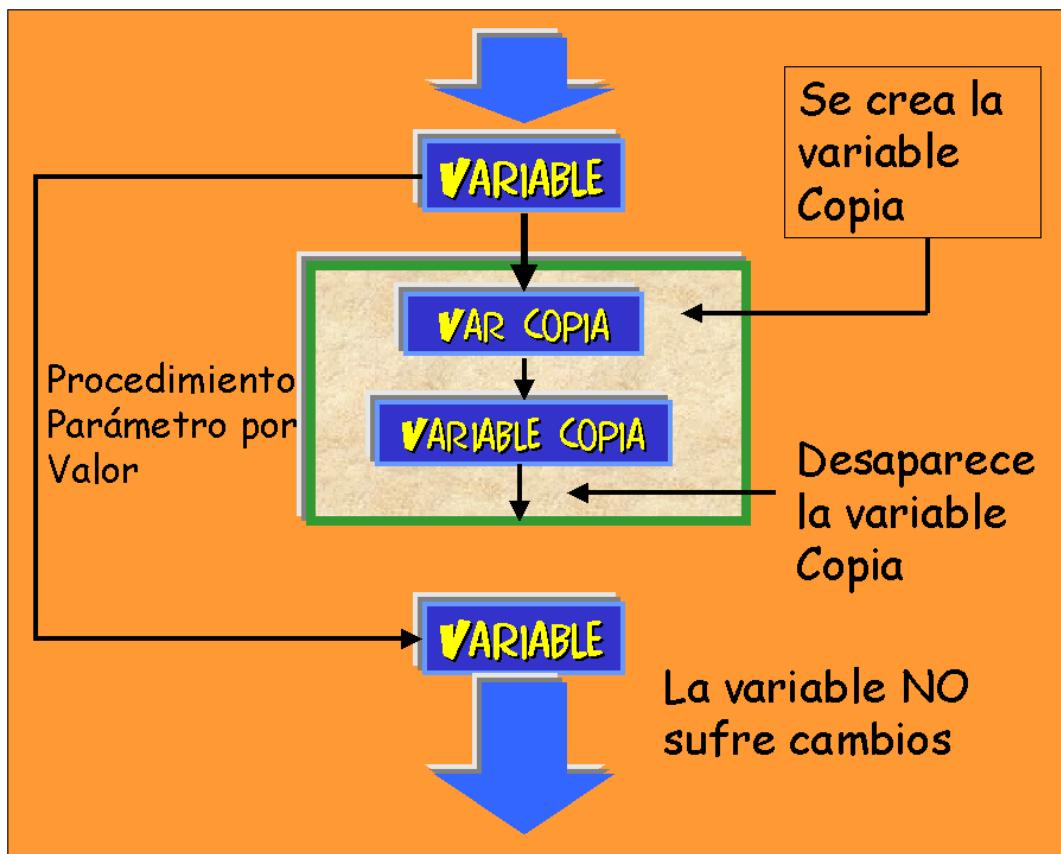
En un principio podríamos pensar que tras ejecutar el método aumentarNúmero, la variable “numero” tendrá el valor 6, pero no es así, puesto que hemos realizado un paso de parámetros por valor.

La salida de la ejecución de este programa es el siguiente:

```

numero tiene el valor 5
numero2 tiene el valor 6

```



El paso de parámetro por valor tiene lugar principalmente cuando los valores que introducimos son de tipo primitivo o un tipo String. Es decir, que cuando creamos un método que opere con este tipo de datos y queramos guardar el resultado, debemos devolverlo como resultado del método.

#### 4.1.2. Paso de parámetros por referencia

Cuando utilizamos un paso de parámetro por referencia lo que pasamos a la función es la dirección de memoria en la que se almacena el valor.



Esto implica que cualquier cambio que se realice sobre el valor durante la ejecución del método va a modificar el valor almacenado en memoria, y por tanto se mantendrá después de finalizar el método. Veamos un ejemplo:

```
package ut07;

import java.util.Arrays;

public class UT07 {

    public static void main(String[] args) {
        String[] texto = new String[4];
        Arrays.fill(texto, "");

        modificarTexto(texto, 2, "Hola");

        for(int i=0; i<texto.length; i++){
            System.out.println("El valor de la cadena " + i + " es: " +
texto[i]);
        }
    }

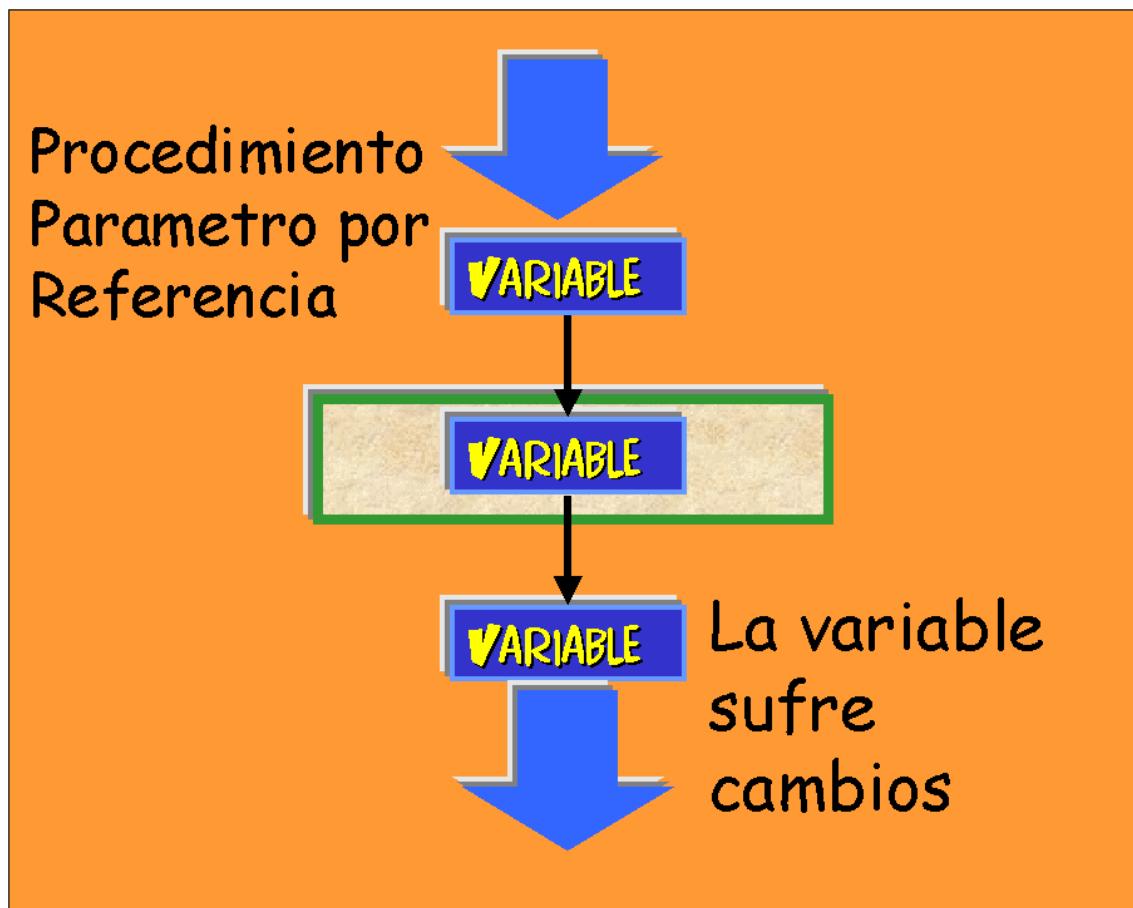
    public static void modificarTexto(String[] texto, int indice, String nuevoValor) {
        texto[indice] = nuevoValor;
    }
}
```

```
        }
    }

    private static void modificarTexto(String[] texto, int pos, String
nuevoValor){
        texto[pos] = nuevoValor;
    }
}
```

El resultado de la ejecución de este programa es el siguiente:

```
El valor de la cadena 0 es:
El valor de la cadena 1 es:
El valor de la cadena 2 es: Hola
El valor de la cadena 3 es:
```



El paso de parámetros por referencia ocurrirá principalmente cuando pasemos por parámetro un array o un objeto de una clase.

## 4.2. Cuerpo de un método

El cuerpo es el código que va a determinar qué hace el método, y se escribe entre llaves {}. Además, si se trata de una función, es decir, un método que devuelve un resultado, debemos hacer uso de la palabra reservada “return” cuando queramos finalizar el método y devolver dicho resultado.

```
{  
    return saleValue * (COMISIONPERCENTAGE/100);  
}
```

El método de ejemplo quedaría de la siguiente manera:

```
public double calculateComision(double saleValue) {  
    return saleValue * (COMISIONPERCENTAGE/100);  
}
```

## 4.3. ÁMBITO

Otro aspecto que debemos tener en cuenta a la hora de utilizar métodos es el ámbito.

Llamamos ámbito a la región del código desde la que se puede acceder a una variable o constante. Distinguimos entre dos tipos de ámbitos:



Ámbito local



Ámbito global

El ámbito local hace referencia a variables o constantes que se declaran dentro del método y no pueden ser accesibles al resto del código.

Mientras que el ámbito global se trata de variables o constantes que se han declarado a nivel global en la clase y que pueden ser accesibles desde cualquier parte de la misma.

```
package ut07;

public class Alumno {
    private int idAlumno; //Ámbito global
    private String nombre; //Ámbito global
    private float[] notas; //Ámbito global

    public Alumno() {
    }

    public int getIdAlumno() {
        return idAlumno;
    }

    public void setIdAlumno(int idAlumno) {
        this.idAlumno = idAlumno;
    }
    ...

    public float calcularMedia(){
        float total = 0; //Ámbito local

        for(int i=0; i<notas.length; i++){
            total += notas[i];
        }

        return total/notas.length;
    }
}
```

Utilizamos la palabra reservada “this” delante de una variable para distinguir una variable global de la clase con respecto a una variable local del método que usa el mismo nombre.

#### 4.4. Constructores



Como dijimos en la unidad anterior, los constructores son unos métodos especiales de las clases, a través de los cuales creamos los objetos de esas clases. Se caracterizan por que siempre tienen el mismo nombre que la clase.

En una clase podemos crear varios constructores diferenciados con los argumentos, o parámetros, que se le pasan, incluso podemos tener constructores a los que no hace falta pasarle ningún parámetro.

Los constructores se suelen utilizar para inicializar los valores de los atributos, pero también pueden realizar otros tipos de operaciones. En el caso de los constructores a los que no se les pasa ningún parámetro, se dejará la inicialización de los atributos a cargo de los métodos getters.

Veamos una clase con distintos tipos de constructores:

```
public class Alumno {  
    private int idAlumno;  
    private String nombre;  
    private float[] notas;  
  
    public Alumno() {}  
  
    public Alumno(int idAlumno, String nombre, float[] notas) {  
        this.idAlumno = idAlumno;  
        this.nombre = nombre;  
        this.notas = notas;  
    }  
  
    public Alumno(int idAlumno, String nombre) {  
        this.idAlumno = idAlumno;  
        this.nombre = nombre;  
    }  
}
```

```
public Alumno(String nombre) {  
    this.nombre = nombre;  
}  
}
```

Otra forma de hacerlo sería la siguiente:

```
public class Alumno {  
    private int idAlumno;  
    private String nombre;  
    private float[] notas;  
  
    public Alumno() {  
        this(0, "", null);  
    }  
  
    public Alumno(int idAlumno, String nombre, float[] notas) {  
        this.idAlumno = idAlumno;  
        this.nombre = nombre;  
        this.notas = notas;  
    }  
  
    public Alumno(int idAlumno, String nombre) {  
        this(idAlumno, nombre, null);  
    }  
  
    public Alumno(String nombre) {  
        this(0, nombre, null);  
    }  
}
```

En esta segunda versión, reutilizamos el segundo de los constructores en el resto de los constructores. Haciendo uso de la palabra clave “this” java entiende que nos referimos a uno de los constructores de esta clase, y al pasarle el número de parámetros adecuados, de los tipos adecuados, identifica cual de ellos es, en este caso el segundo. De esta forma nos aseguramos de que, utilicemos el constructor que utilicemos, siempre se van a inicializar todos los atributos.

#### 4.5. Getters y Setters

Son otro tipo especial de métodos que utilizamos para acceder a los atributos desde fuera de la clase, en el caso de que sean “private”.

Su estructura es como la de cualquier otro método, pero para nombrarlos utilizamos por convención las palabras “get” o “set”, según necesitemos un getter o un setter, y el nombre del atributo.

Los métodos setter siempre tendrán “void” como tipo de dato de salida y necesitará uno o varios tributos, que insertará en el atributo o con los que realizará una serie de operaciones que terminarán insertando un valor en el atributo.

```
public class Producto {  
    private String nombre;  
    private float precioVentaPublico;  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public void setPrecioVentaPublico(float precioSinIva) {  
        this.precioVentaPublico = precioSinIva * 1.21f;  
    }  
}
```

Los métodos getter siempre tendrán como tipo de dato de salida del propio atributo al que hacen referencia. Habitualmente solo devuelven el valor del atributo, pero también pueden realizar cualquier tipo de operación y devolver el resultado.

```
package ut07;  
  
public class Producto {  
    private String nombre;  
    private float precioVentaPublico;  
  
    public String getNombre() {  
        return nombre;  
    }
```

```
    }

    public float getPrecioVentaPublico() {
        return precioVentaPublico;
    }
}
```

## 5. INSTANCIACIÓN DE OBJETOS. DECLARACIÓN Y CREACIÓN

Para declarar un objeto lo haremos como ya vimos en la unidad anterior, en primer lugar, pondremos el nombre de la clase y después el nombre del objeto.

```
Alumno alumno;
```

Y para crearlo tendremos que hacer uso de la palabra clave new y llamar a uno de los constructores de la clase.

```
Alumno alumno1;
Alumno alumno2 = new Alumno();

alumno1 = new Alumno(1,"Scott");
```

### 5.1. Características de un objeto

Llamaremos caracterización de un objeto al conjunto de atributos, métodos y el nombre que le hemos dado.

## 6. USO AVANZADO DE MÉTODOS

### 6.1. Métodos recursivos



Un método recursivo es un método da solución a un problema mediante la búsqueda de un caso base al que llega realizando llamadas recursivas, es decir, llamadas al propio método.

En los métodos recursivos se pueden distinguir dos partes:

- **Caso base:** caso en el que se alcanza la solución y ya no es necesario realizar más llamadas recursivas.
- **Llamadas recursivas:** caso en el que no se ha alcanzado la solución y seguimos buscándola llamando de nuevo al propio método.

Veamos un ejemplo simple:

```
public static int multiplicarNumeros(int num1, int num2){  
    if(num2 == 0)  
        return 0;  
    else  
        return num1 + multiplicarNumeros(num1, num2 -1);  
}
```

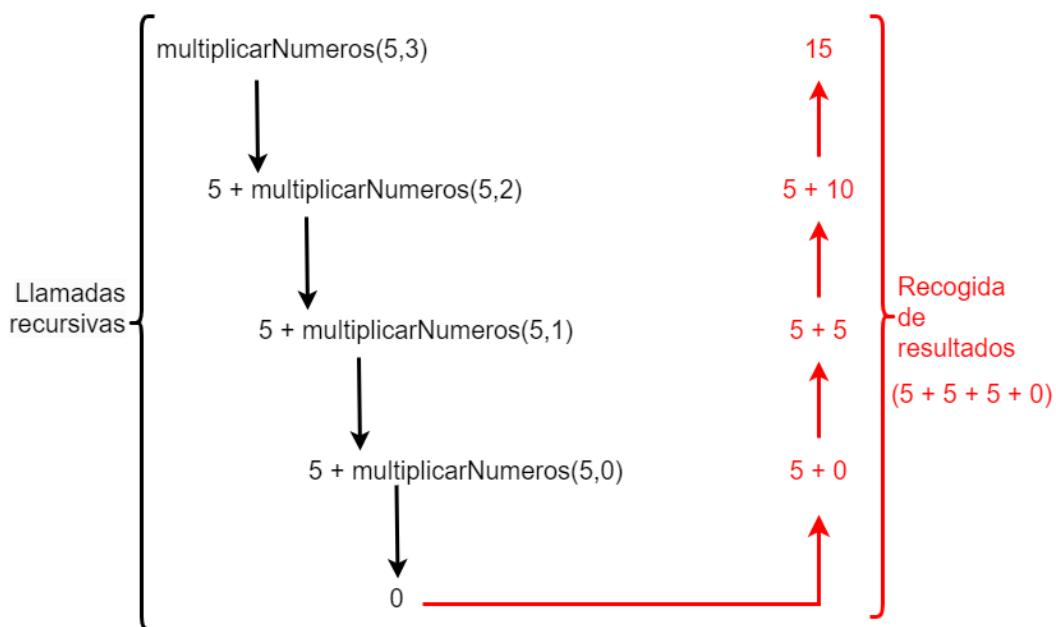
Existen dos tipos de recursividad:

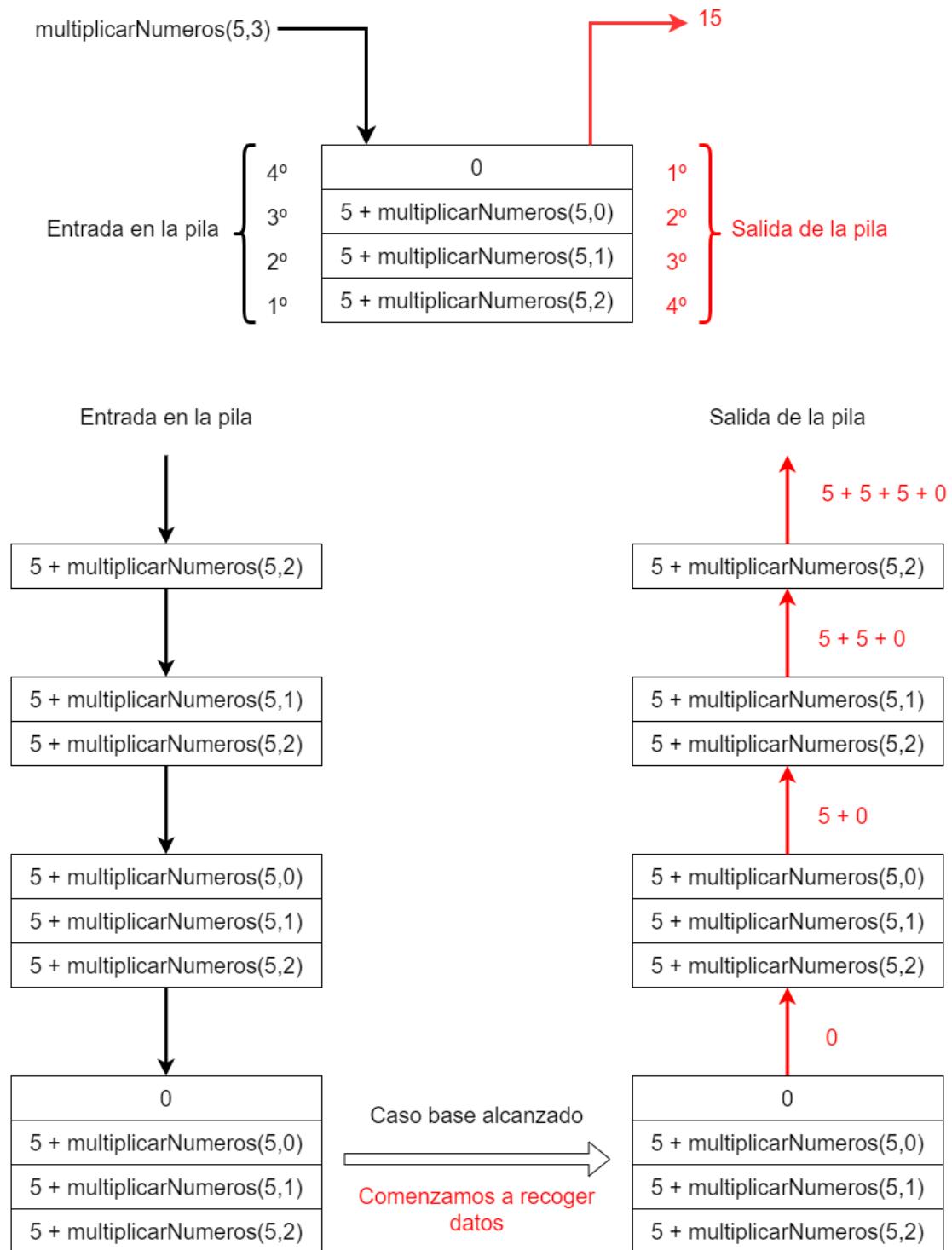
- **Directa:** las llamadas recursivas se hacen al mismo método, como en el ejemplo anterior.
- **Indirecta:** las llamadas se hacen a otro método y este puede llamar a otro, ... y así hasta que en algún momento se vuelve a llamar al método original. Aquí tenemos un ejemplo.

```
public static boolean esPar(int numero){
    if(numero == 0)
        return true;
    else
        return esImpar(numero-1);
}

private static boolean esImpar(int numero){
    if(numero == 0)
        return false;
    else
        return esPar(numero-1);
}
```

Hay que tener en cuenta que los resultados de los métodos recursivos se van almacenando en una pila de datos, de tal forma que se recogerán en orden inverso al que se introdujeron.





Algunas de las ventajas de utilizar métodos recursivos son:

- Resuelve problemas en menos líneas que la solución iterativa.
- Hay problemas, normalmente matemáticos, que se adaptan muy bien a la recursividad, debido a su naturaleza.

Algunas de las desventajas de utilizar métodos recursivos son:

- Si no definimos correctamente el caso base, podemos realizar llamadas recursivas infinitas y desbordar la pila.
- Es menos eficiente que el método iterativo.

## 6.2. Sobrecarga de métodos



Denominamos sobrecarga de métodos cuando en una misma clase creamos métodos con el mismo nombre, pero que reciben distintos parámetros, o argumentos, en número y/o tipo de dato. De esta forma, podremos hacer que un método realice distintas operaciones en función de los parámetros que le lleguen.

En el siguiente ejemplo vemos una clase que tiene 3 definiciones distintas del método calcularPrecioVenta(), en función de si le llega un producto con IVA normal, uno con IVA reducido o uno con IVA superreducido, es decir, con el mismo número de parámetros, pero para distintos tipos de datos.

```
public class CalculoIVA {  
  
    public double calcularPrecioVenta(ProductoNormal producto){  
        return producto.getPrecio() * 1.21;  
    }  
  
    public double calcularPrecioVenta(ProductoReducido producto){  
        return producto.getPrecio() * 1.10;  
    }  
  
    public double calcularPrecioVenta(ProductoSuperReducido producto){  
        return producto.getPrecio() * 1.04;  
    }  
}
```

Los constructores son un claro ejemplo de sobrecarga de métodos con distinto número de parámetros.

```
public class Alumno {  
    private int idAlumno;  
    private String nombre;  
    private float[] notas;  
  
    public Alumno() {  
        this(0,"",null);  
    }  
  
    public Alumno(int idAlumno, String nombre, float[] notas) {  
        this.idAlumno = idAlumno;  
        this.nombre = nombre;  
        this.notas = notas;  
    }  
  
    public Alumno(int idAlumno, String nombre) {  
        this(idAlumno, nombre, null);  
    }  
  
    public Alumno(String nombre) {  
        this(0, nombre, null);  
    }  
}
```

**Without Method Overloading**

```
int add2(int x, int y)
{
    return(x+y);
}
int add3(int x, int y,int z)
{
    return(x+y+z);
}
int add4(int w, int x,int y, int z)
{
    return(w+x+y+z);
}
```

**With Method Overloading**

```
int add(int x, int y)
{
    return(x+y);
}
int add(int x, int y,int z)
{
    return(x+y+z);
}
int add(int w, int x,int y, int z)
{
    return(w+x+y+z);
}
```

**6.3. Métodos estáticos**

Los métodos estáticos, o de clase, son métodos que crearemos en nuestras clases y no será necesario implementar dicha clase para utilizarlos.

Lo habitual es que estos métodos siempre reciban al menos un parámetro y devuelvan un resultado, aunque pueden existir métodos estáticos sin parámetros y/o que no devuelvan un resultado.

Al no necesitar ser instanciados, no podrán hacer uso de los atributos de la clase, a excepción de que sean estáticos y estén inicializados.

```
public class Empleado {
    private static final int COMISION = 10;
    private static int numEmpleados = 0;
    private int idEmpleado;
    private String nombreEmpleado;

    ...

    public static double calcularComision(double precioProducto){
        return precioProducto * COMISION /100;
    }

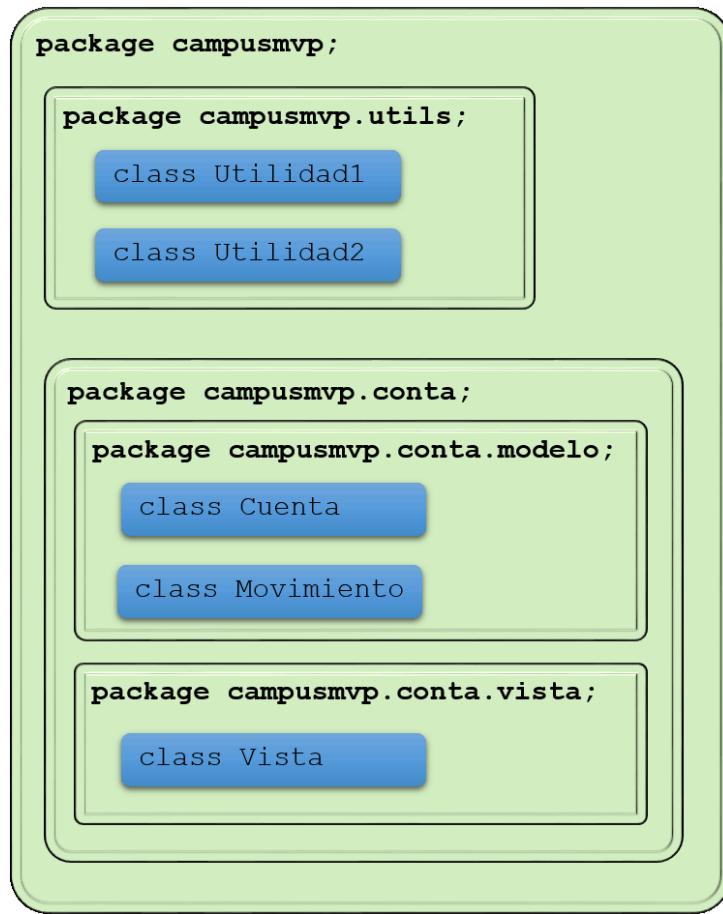
    ...
}
```

## 7. LIBRERÍAS Y PAQUETES DE CLASES. UTILIZACIÓN Y CREACIÓN.

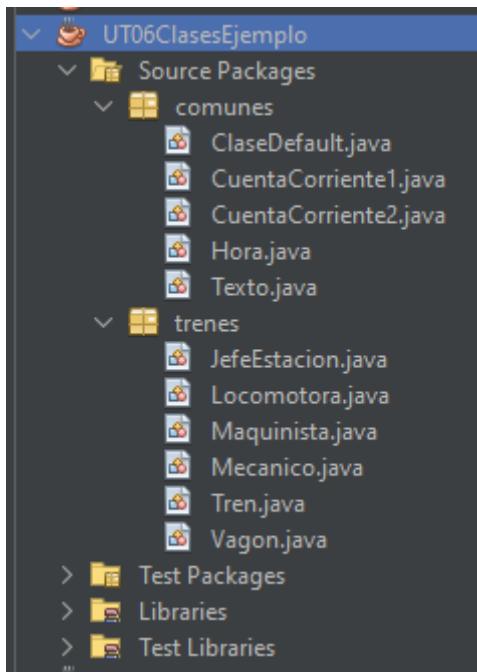


Los paquetes son el mecanismo que usa Java para facilitar la modularidad del código. Un paquete puede contener una o más definiciones de interfaces y clases, distribuyéndose habitualmente como un archivo. Para utilizar los elementos de un paquete es necesario importar este en el módulo de código en curso, usando para ello la sentencia import.

La funcionalidad de una aplicación Java se implementa habitualmente en múltiples clases, entre las que suelen existir distintas relaciones. Las clases se agrupan en unidades de un nivel superior, los paquetes, que actúan como ámbitos de contención de tipos. Cada módulo de código establece, mediante la palabra clave package al inicio, a qué paquete pertenece.



Según vayamos construyendo nuestro software, distribuiremos las distintas clases que creamos en paquetes de clases que están relacionadas entre sí.



Los paquetes Java son como cajas de herramientas, cada una de ellas con una colección distinta de instrumentos útiles para nuestro trabajo, a las que podemos necesitar acceder cada vez que abordamos el desarrollo de un nuevo proyecto. Es en este contexto donde recurriremos a la cláusula import, a fin de importar en el ámbito actual las definiciones de otro paquete y poder usarlas según el procedimiento habitual, creando objetos, accediendo a los servicios de las clases, etc.

Mediante import podemos hacer referencia tanto a una única clase del paquete, como a todas las clases del mismo paquete:

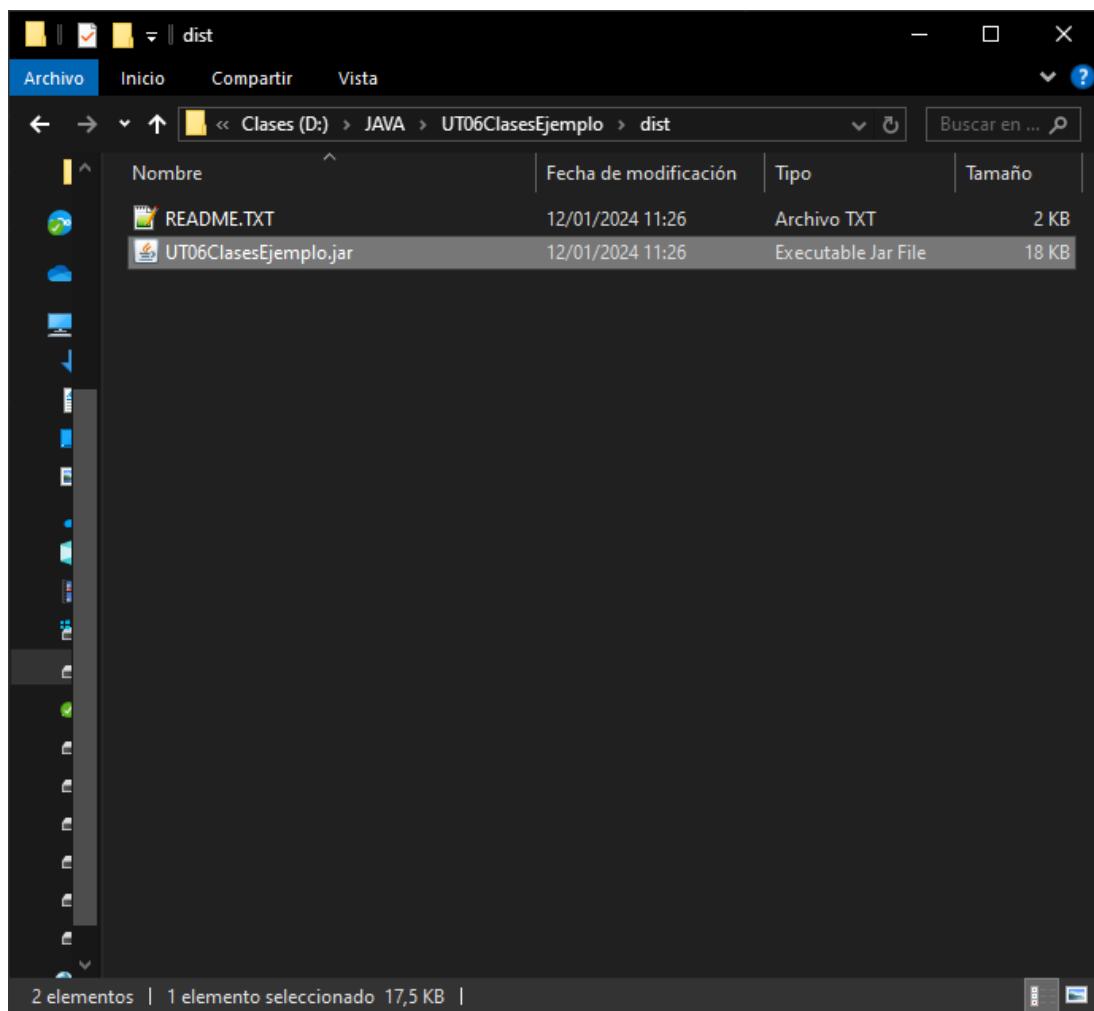
```
import java.lang.Math;
```

```
import java.lang.*;
```

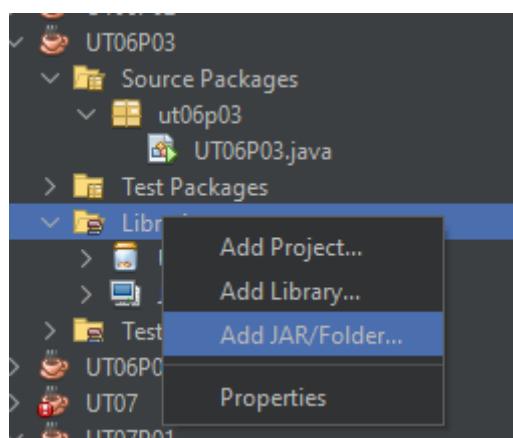
En el caso de nuestro proyecto conste de una clase principal y otras clases y paquetes, cuando necesitemos utilizar una clase que no se encuentre en el mismo paquete desde el que se le llama, necesitaremos utilizar import. Si llamamos a una clase que se encuentra en el mismo paquete no será necesario.

Cuando creamos un proyecto que únicamente consta de paquetes con clases secundarias, es decir, que ninguna de ellas es primaria (contiene un método main), decimos que nuestro proyecto es una librería de clases. El propósito de este tipo de proyectos no es el de generar una aplicación que funciona por si misma, si no la de proporcionar a otros proyectos una serie de herramientas definidas en la librería. Lo habitual en estos casos es que la librería sea compilada, generando un fichero “.jar”, de tal forma que nos llevaremos ese fichero compilado a otro proyecto y lo incluiremos como librería y así podremos hacer uso de toda su funcionalidad.

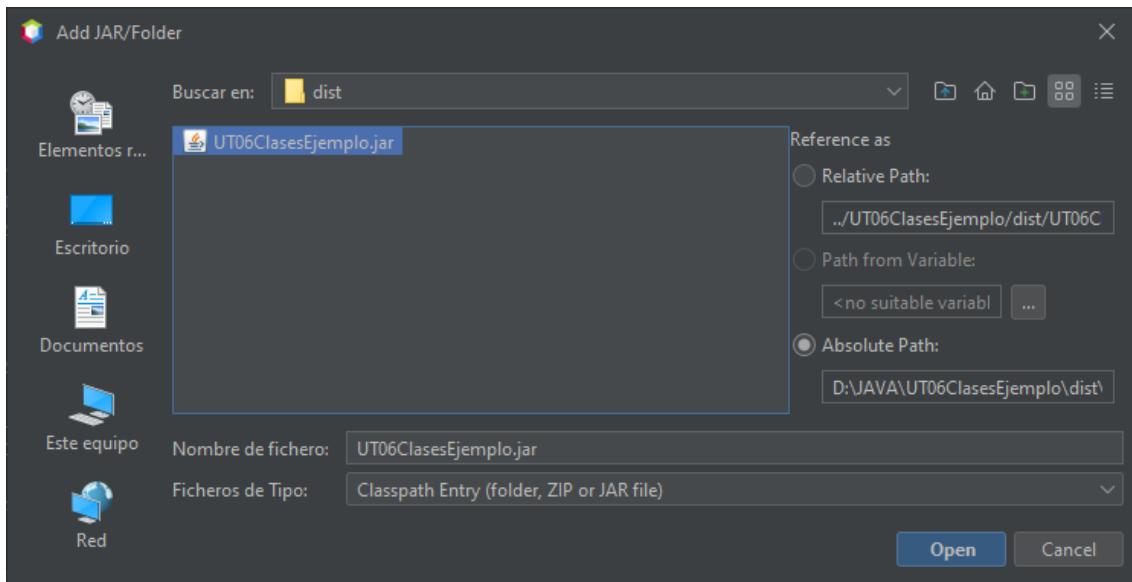
El fichero “.jar” que se genera en carpeta “dist” dentro de la del proyecto:



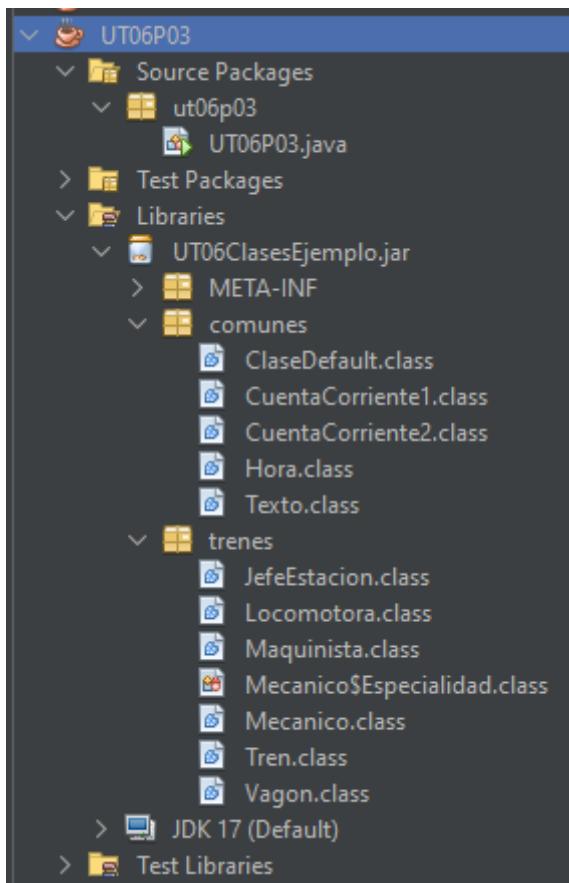
De ahí podemos cogerlo y añadirlo como librería a cualquier otro proyecto, pulsando botón derecho sobre la carpeta “Libraries” en el navegador de proyectos del NetBeans y seleccionamos la opción “Add JAR/folder”.



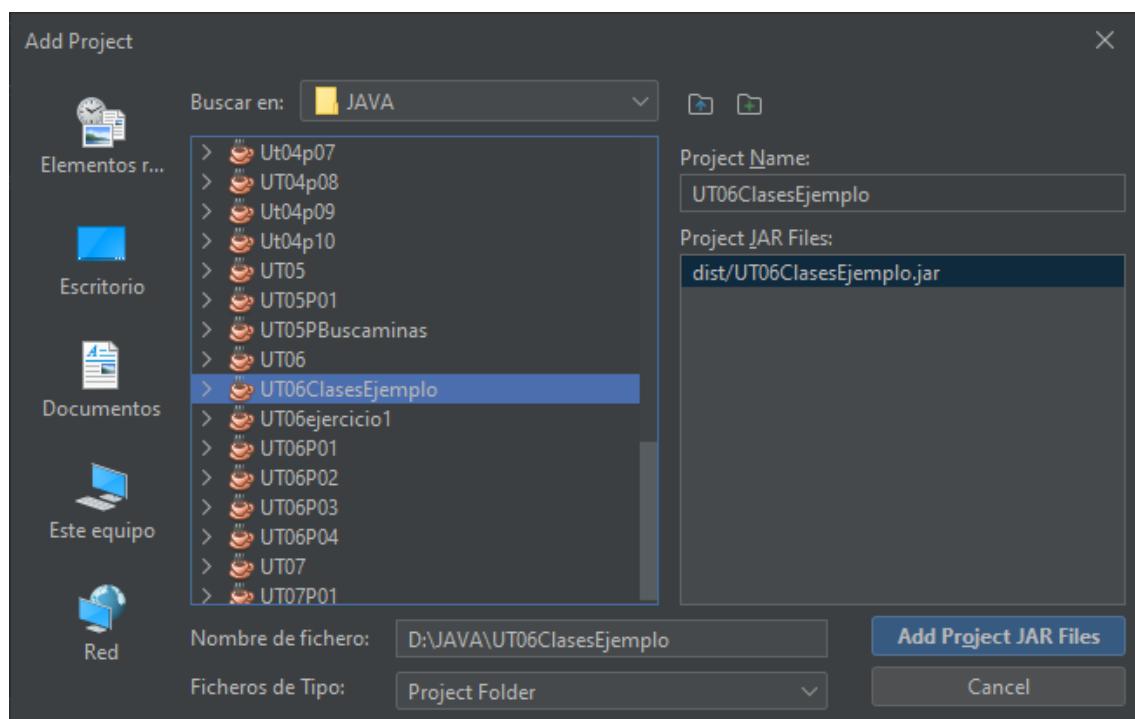
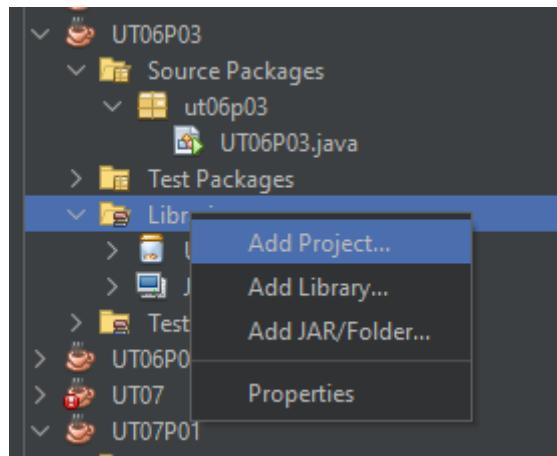
Y buscamos el fichero “.jar” para incluirlo entre nuestras librerías.



Desde ese momento podemos hacer uso de todas las clases que incluye la librería dentro de nuestro proyecto.



Si somos nosotros mismos los que estamos creando tanto la librería clases, como el programa que hará uso de ella, tenemos la opción de incluir el proyecto la librería dentro del proyecto del programa, pulsando botón derecho sobre “Libraries” y seleccionamos “Add Project”.



De esta forma nos añadirá los ficheros “.jar”, de forma automática, desde nuestro proyecto de librería.

Una vez incluida la librería en nuestro proyecto, ya sea como fichero “.jar” o adjuntando el proyecto, solo necesitaremos utilizar import para utilizar las distintas clases que componen la librería.

```
package ut06p03;

import java.util.Scanner;
import trenes.JefeEstacion;
import trenes.Locomotora;
import trenes.Maquinista;
import trenes.Mecanico;
import trenes.Mecanico.Especialidad;
import trenes.Tren;

public class UT06P03 {
    ...
}
```

## 8. DOCUMENTACIÓN SOBRE LIBRERÍAS Y PAQUETES DE CLASES.

A la hora de documentar nuestras librerías y paquetes de clases, haremos uso, tal y como vimos en unidades anteriores, de los comentarios dentro del código.

Existen herramientas como “javadoc” que nos permiten generar documentos de forma automática con las descripciones de los paquetes, clases, métodos, etc ...

Estas herramientas funcionan mediante el uso de una estructura determinada a la hora de comentar el código y símbolos específicos, de tal forma que después se ejecutará un proceso que lea todos los comentarios del código y generará de forma automática un documento con toda la información incluida en los mismos.

En el siguiente link tenéis la página oficial de javadoc:

<https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>

# **UT-07: UTILIZACIÓN AVANZADA DE CLASES**

¿Qué vamos a ver?

- HERENCIA.
  - CONCEPTO Y TIPOS (SIMPLE Y MÚLTIPLE).
  - SUPERCLASES Y SUBCLASES.
  - CONSTRUCTORES Y HERENCIA.
- POLIMORFISMO.
  - CONCEPTO.
  - POLIMORFISMO EN TIEMPO DE COMPILEACIÓN
  - POLIMORFISMO EN TIEMPO DE EJECUCIÓN (LIGADURA DINÁMICA).
- CLASES Y MÉTODOS ABSTRACTOS Y FINALES.
- INTERFACES. CLASES ABSTRACTAS VS. INTERFACES.
- COMPROBACIÓN ESTÁTICA Y DINÁMICA DE TIPOS.
- CONVERSIONES DE TIPOS ENTRE OBJETOS (CASTING).
- CLASES Y TIPOS GENÉRICOS O PARAMETRIZADOS.
- ENUMERADOS

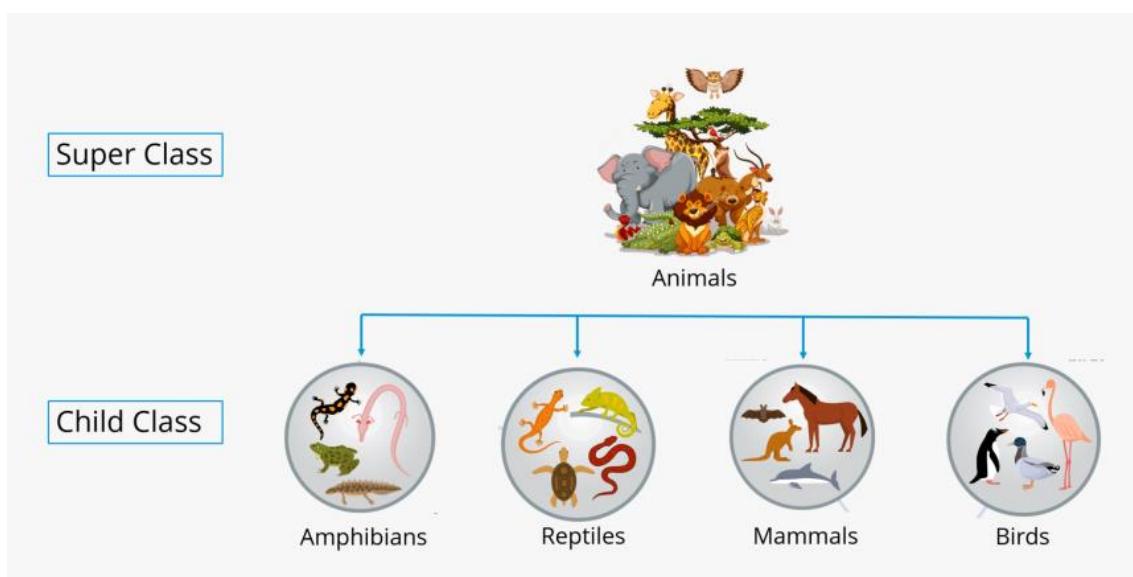
## 1. HERENCIA

### 1.1. Concepto y tipos

La herencia es la propiedad que permite a los objetos ser construidos a partir de otros objetos. La idea fundamental es permitir crear nuevas clases aprovechando las características (atributos y métodos) de otras clases ya creadas evitando así tener que volver a definir esas características (reutilización).

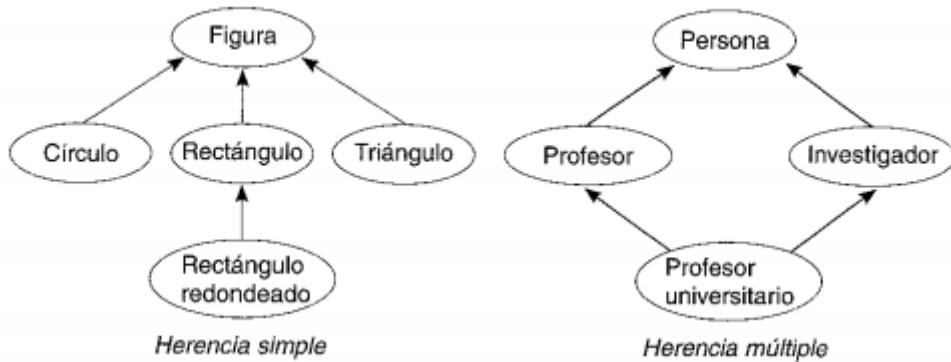
A una clase que hereda de otra se le llama subclase (clase derivada, clase hija) y aquella de la que se hereda es conocida como superclase (clase base, clase padre).

Las clases derivadas heredan el código y datos de su clase base, añadiendo su propio código y datos, incluso puede cambiar aquellos elementos de la clase base que necesita que sean diferentes. Por ejemplo, la clase animal se puede dividir en las subclases anfibios, mamíferos, aves, etc., y la clase vehículo en coches, camiones, etc.



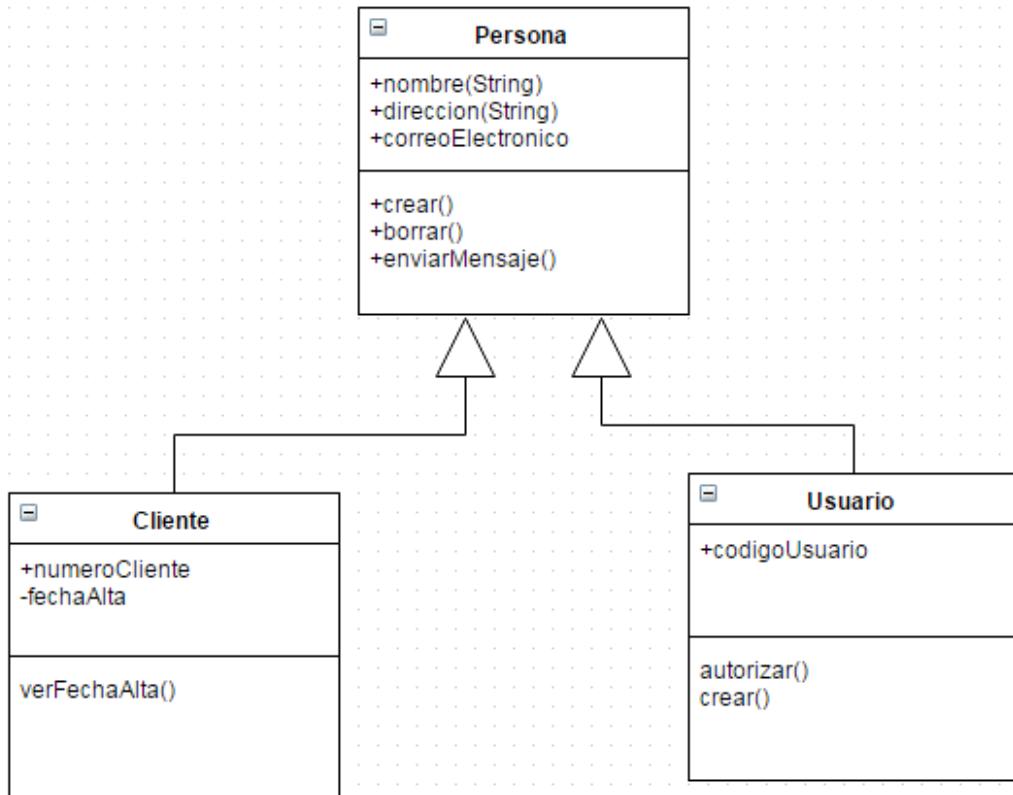
Existen dos tipos de herencia:

- **Herencia simple:** Una subclase puede heredar datos y métodos de una única clase. Java admite solo herencia simple.
- **Herencia múltiple:** Una subclase puede heredar datos y métodos de más de una clase. C++ admite herencia simple y múltiple.



La herencia tiene como ventajas principales la reutilización y compartición de código (evita repetir código ahorrando mucho tiempo), la consistencia de la interfaz (el comportamiento que heredan será el mismo en todos los casos) y la ocultación de información. Pero también tiene inconvenientes como son la velocidad de ejecución (los métodos heredados suelen ser más lentos) y la curva de aprendizaje (al principio el aprendizaje de la programación orientada a objetos suele ser más lento).

## 1.2. Superclases y subclases



Para que una clase (subclase) herede de otra clase (superclase) utilizamos la palabra clave “extends” seguida del nombre de la suplerclase.

```
public class Dog extends Animal{
    ...
}
```

Una subclase que hereda que de una superclase adquiere los atributos y los métodos de la segunda, y además puede añadir unos propios.

Veamos un ejemplo...

```
class Animal {  
  
    // fields of the parent class  
    String name;  
    String species;  
    double weight;  
    double height;  
  
    ...  
  
    // method of the parent class  
    public feed(String food){  
        System.out.println("I eat " + food);  
    }  
}  
  
class Dog extends Animal {  
    String color;  
    String breed;  
  
    ...  
  
    public void display() {  
        System.out.println("My name is " + name);  
        System.out.println("My species is " + species);  
        System.out.println("My weight is " + weight);  
        System.out.println("My height is " + height);  
        System.out.println("My color is " + color);  
        System.out.println("My breed is " + breed);  
    }  
}
```

En este caso la clase Dog hereda de la clase Animal, por tanto, la clase Dog, además de sus atributos color y breed tiene los atributos, name, species, weight y height. Por otro lado, además del método display() cuenta con el método feed().

```
Dog dog1 = new dog();  
...  
dog1.feed("meat");  
System.out.println("The name of my dog is: " + dog1.getName());
```

### 1.3. Constructores y herencia

Al igual que ocurre con el resto de métodos, los constructores también se heredan pero de una forma especial al resto de métodos.

Al llamar al constructor de una subclase vamos a tener que llamar al constructor de la superclase y si necesitamos alguna operación adicional específica de la subclase la realizaremos a continuación. Vamos a verlo con el ejemplo anterior:

```
class Animal {  
  
    String name;  
    String species;  
    double weight;  
    double height;  
  
    public Animal (){}  
  
    public Animal (String name, String species, double weight, double  
height){  
        this.name = name;  
        this.species = species;  
        this.weight = weight;  
        this.height = height;  
    }  
  
    ...  
}  
  
class Dog extends Animal {  
    String color;  
    String breed;  
  
    public Dog(){  
        super();  
    }  
  
    public Dog(String name, String species, double weight, double height,  
String color, String breed){  
        super(name, species, weight, height);  
        this.color = color;  
        this.breed = breed;  
    }  
  
    ...  
}
```

#### 1.4. Modificador de acceso protected

Cuando hablamos de herencia entre clases nos encontramos con un nuevo modificador de acceso, que está precisamente especializado en herencia de clases, este modificador es “protected”.

Este modificador funciona de una forma similar a la visibilidad por defecto, con la diferencia de que los miembros protegidos serán siempre visibles para las clases que hereden, independientemente de si la superclase y la subclase se encuentra en el mismo paquete o son externas, aunque en este último caso, habrá que importar la superclase.

En resumen, un miembro “protected” es visible en las clases del mismo paquete, no es visible para las clases externas, pero siempre es visible, independientemente del paquete al que pertenezca, desde una clase hija.

Modificador (palabra reservada)	Misma clase	Mismo paquete	Subclase de otro paquete	Resto
<i>private</i>	✓	✗	✗	✗
<i>(defecto)</i>	✓	✓	✗	✗
<i>protected</i>	✓	✓	✓	✗
<i>public</i>	✓	✓	✓	✓

```
class Person {
    protected String fname = "John";
    protected String lname = "Doe";
    protected String email = "john@doe.com";
    protected int age = 24;
}
```

```
class Student extends Person {  
    private int graduationYear = 2018;  
    public static void main(String[] args) {  
        Student myObj = new Student();  
        System.out.println("Name: " + myObj.fname + " " + myObj.lname);  
        System.out.println("Email: " + myObj.email);  
        System.out.println("Age: " + myObj.age);  
        System.out.println("Graduation Year: " + myObj.graduationYear);  
    }  
}
```

## 2. POLIMORFISMO

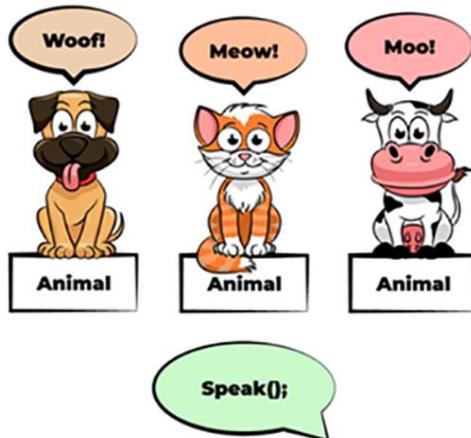
El polimorfismo es la propiedad que permite que un operador o un método actúen de modo diferente en función del objeto sobre el que se aplican.

Cuando un operador existente en el lenguaje tal como +, = o \* se le asigna la posibilidad de operar sobre un nuevo tipo de datos, se dice que está sobrecargado. La sobrecarga es una clase polimorfismo.

En un sentido más general, el polimorfismo supone que un mismo mensaje puede producir acciones totalmente diferentes cuando se reciben por objetos diferentes.

El polimorfismo adquiere su máxima expresión en la herencia de clases, es decir, cuando se obtiene una clase a partir de una clase ya existente.

Por ejemplo, cuando se describe una clase base Mamífero se puede observar que la operación comer es una operación fundamental en su vida, de modo que cada tipo de mamífero (vaca, humano, león) debe poder realizar la operación comer, aunque cada una de las clases derivadas que representan los distintos tipos de mamíferos la realizará de un modo diferente (polimorfismo).



## 2.1. Polimorfismo en tiempo de compilación

El primer tipo de polimorfismo en tiempo de compilación ya lo vimos en la unidad anterior, se trata de la sobrecarga de métodos.

Además de la sobrecarga de métodos se nos está permitido redefinir los tipos de datos de los atributos heredados y de esta forma adaptar mejor dichos atributos a las clases hijas. Aquí tenemos un ejemplo:

```
class Persona {
    String nombre;
    byte edad;
    double estatura;
    void mostrarDatos(){
        System.out.println(nombre);
        System.out.println(edad);
        System.out.println(estatura);
    }
}
```

```
//subclase Empleado que hereda de Persona
class Empleado extends Persona {
    String estatura; //cambiamos de double a String S, M, L, XL
    double salario;
```

```
@Override  
void mostrarDatos(){  
    System.out.println(nombre);  
    System.out.println(edad);  
    System.out.println(estatura);  
    System.out.println(salario);  
}  
}
```

Por otro lado, cuando una subclase sustituye la funcionalidad de un método de su superclase, se le denomina sobreescritura del método.

Para realizar esto en Java vamos a utilizar la anotación “@Override” antes de la definición del método de la subclase.

Veamos un ejemplo:

```
class Animal {  
    void sound() {  
        System.out.println('Animals make sounds');  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    void sound() {  
        System.out.println('Dogs bark');  
    }  
}  
  
// Create an instance of Dog  
Dog dog = new Dog();  
// Call the sound method  
dog.sound();  
  
// Output:  
// 'Dogs bark'
```

## 2.2. Polimorfismo en tiempo de ejecución (ligadura dinámica)

Debemos tener en cuenta que un objeto de una subclase puede ser asignado a una variable de su superclase, por ejemplo, si tenemos una superclase Persona

y una subclase Empleado, podremos crear una variable de la clase Persona y asignarle un objeto que creemos de la clase Empleado:

```
public class Persona{  
    protected String nombre;  
    protected String apellido;  
    protected String DNI;  
  
    void mostrarInformacion(){  
        System.out.println("Mi nombre es " + this.nombre);  
        System.out.println("Mi apellido es " + this.apellido);  
        System.out.println("Mi DNI es " + this.DNI);  
    }  
}  
  
public class Empleado extends Persona{  
    protected int sueldo;  
  
    @Override  
    void mostrarInformacion(){  
        System.out.println("Mi nombre es " + this.nombre);  
        System.out.println("Mi apellido es " + this.apellido);  
        System.out.println("Mi DNI es " + this.DNI);  
        System.out.println("Mi sueldo es " + this.sueldo);  
    }  
}
```

```
Empleado empleado = new Empleado(nombre, apellido, DNI);  
Persona persona = empleado;
```

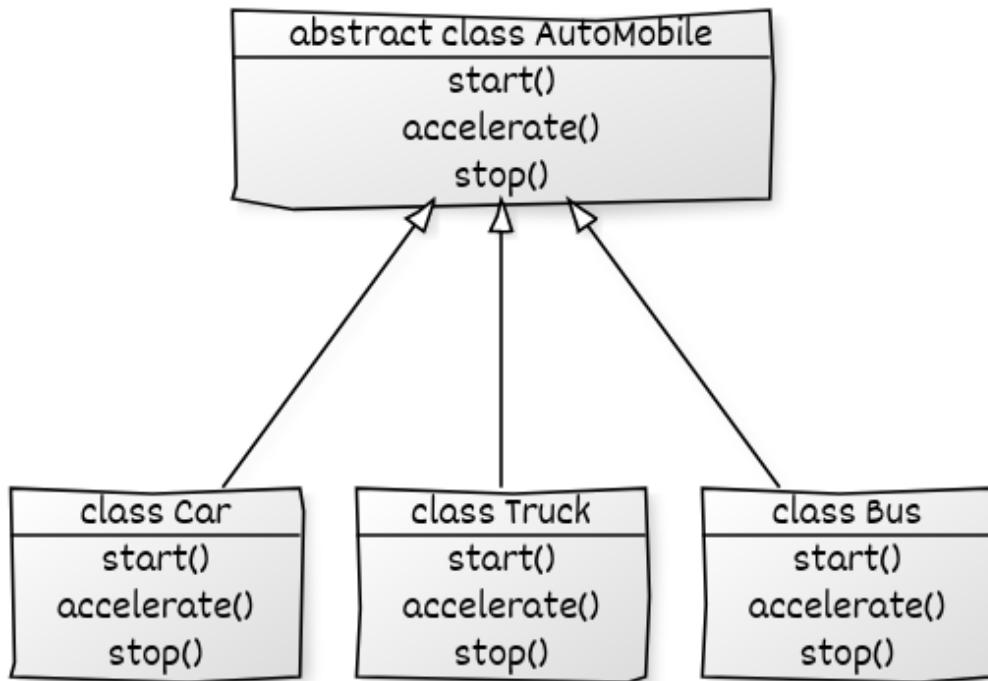
En este caso, si llamamos al método `mostrarInformación()` de empleado, nos devolverá toda la información, incluido el sueldo, pero si llamamos a `mostrarInformacion()` de persona, solo nos mostrará el nombre, apellido y DNI.

```
empleado.mostrarInformacion(); //nombre, apellido, DNI y sueldo  
persona.mostrarInformacion(); //nombre, apellido y DNI
```

A esto se le denomina ligadura dinámica y nos sirve para poder acceder tanto a la versión de la superclase del método, como a la versión de la subclase.

### 3. CLASES Y MÉTODOS ABSTRACTOS Y FINALES

#### 3.1. Clases y métodos abstractos



En la jerarquía de herencia de clases, cuanto más abajo, más específica y particular es la implementación de los métodos. Asimismo, cuanto más arriba, más general.

Hay métodos que no podemos implementar en una clase determinada por falta de datos, pero sí en sus subclases, donde se han añadido los atributos necesarios. La idea es implementarlos “vacíos”, solo con la declaración, en la superclase, y hacen overriding en las subclases, donde ya disponemos de la información necesaria para implementar los detalles.

Un método declarado en una clase, pero cuya implementación se delega a las subclases, se conoce como **método abstracto**. Para declarar un método abstracto se le antepone el modificador “abstract” y se declara un prototipo, sin escribir el cuerpo de la función. Por ejemplo, para declarar un método abstracto que muestra información del objeto escribiremos:

```
abstract void mostrarDatos();
```

Las subclases deberán implementar el método `mostrarDatos()`, cada una con las particularidades específicas de la clase, que no se conocen al nivel de la superclase.

Toda clase que tiene un método abstracto debe ser declarada, a su vez, abstracta, también utilizando la palabra clave “`abstract`”.

Las clases abstractas no son instanciables, es decir, no se pueden crear objetos de esa clase. Las clases abstractas existen para ser heredadas por otras, y no para ser instanciadas.

Una clase abstracta puede tener tanto métodos abstractos como no abstractos, e incluso algunos atributos declarados, pero siempre tendrá al menos un método abstracto.

```
abstract class Persona{
    protected String nombre;
    protected String apellido;
    protected String DNI;

    abstract void mostrarInformacion();

    public void hablar(String mensaje){
        System.out.println(mensaje);
    }
}

public class Empleado extends Persona{
    protected int sueldo;

    @Override
    void mostrarInformacion(){
        System.out.println("Mi nombre es " + this.nombre);
        System.out.println("Mi apellido es " + this.apellido);
        System.out.println("Mi DNI es " + this.DNI);
        System.out.println("Mi sueldo es " + this.sueldo);
    }
}
```

### 3.2. Clases y métodos finales

Existirán ocasiones en las que no deseemos que una de nuestras clases pueda ser heredada. Para poder conseguir esto vamos a colocar el modificador “final” a la clase:

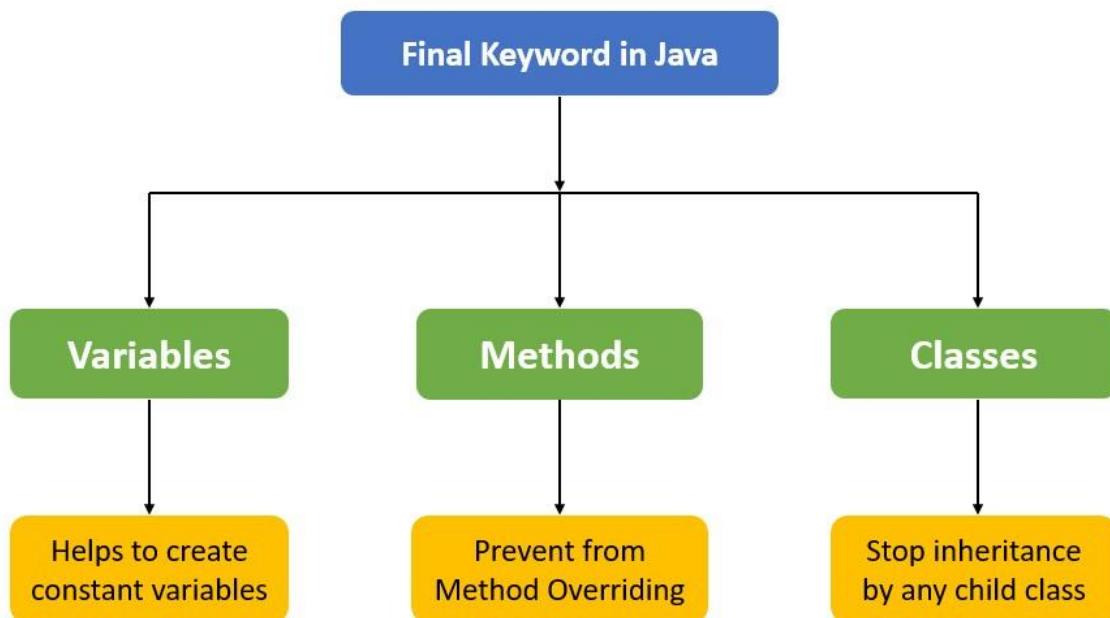
```
public final class Persona{  
    protected String nombre;  
    protected String apellido;  
    protected String DNI;  
  
    void mostrarInformacion(){  
        System.out.println("Mi nombre es " + this.nombre);  
        System.out.println("Mi apellido es " + this.apellido);  
        System.out.println("Mi DNI es " + this.DNI);  
    }  
}
```

De esta forma nadie podrá crear subclases de la clase Persona.

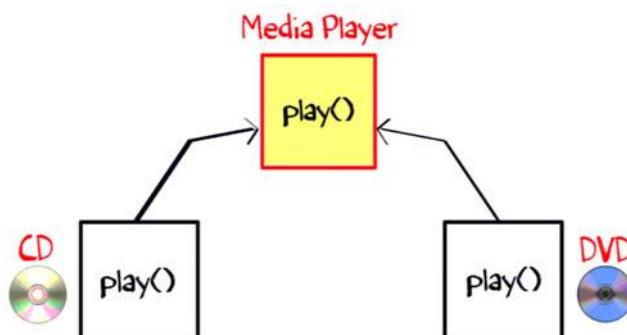
De la misma manera, podemos considerar que alguno de los métodos de nuestra superclase no debe ser sobreescrito por una de sus subclases. Al igual que ocurre con las clases, para evitar esto vamos a añadir el modificador final a la declaración de nuestro método.

```
public class Persona{  
    protected String nombre;  
    protected String apellido;  
    protected String DNI;  
  
    final void mostrarInformacion(){  
        System.out.println("Mi nombre es " + this.nombre);  
        System.out.println("Mi apellido es " + this.apellido);  
        System.out.println("Mi DNI es " + this.DNI);  
    }  
}
```

Así, aunque alguien cree una subclase de Persona no le será posible sobrescribir el método mostrarInformacion().



## 4. INTERFACES



Las interfaces son una especie de plantilla para la construcción de clases. Normalmente una interfaz se compone de un conjunto de declaraciones de cabeceras de métodos (sin implementar, igual que los métodos abstractos) que especifican un protocolo de comportamiento para una o varias clases.

Además, una clase puede implementar una o varias interfaces. En ese caso, la clase debe proporcionar la declaración y definición de todos los métodos de cada una de las interfaces o bien declararse como clase abstracta.

Por otro lado, una interfaz puede emplearse también para declarar constantes que luego puedan ser utilizadas por otras clases.

Es decir, se trata de una clase que no puede ser implementada por si misma, sino que otras clases la heredan y la implementan. De este modo, al emplear las interfaces, es posible establecer un conjunto de reglas que otras clases deberán seguir de forma estricta.

Una interfaz puede contener:

- Constantes estáticas.
- Definición de métodos abstractos.
- Implementación de métodos por defecto.
- Implementación de métodos estáticos.
- Implementación de métodos privados.

#### 4.1. Definición de una interfaz

Para definir una interfaz únicamente tendremos que utilizar la palabra clase “interface” seguido del nombre la interfaz. Por consenso la primera palabra letra del nombre de la interfaz es una “I”.

```
public interface IAnimal {  
}  
}
```

#### 4.2. Constantes estáticas

El único tipo de atributos que se pueden incluir en una interfaz son constantes estáticas, aunque no es necesario identificarlos como “static” y “final”, ya que se toma por defecto.

```
public interface IAnimal {  
    String PLANETA = "Tierra";  
}
```

Estas constantes serán heredadas y podrán ser utilizadas por las instancias de las clases que implementen dicha interface.

También será posible llamar a estos atributos desde la interfaz no implementada.

```
System.out.println("Los animales viven en el planeta" + IAnimal.PLANETA);
```

#### 4.3. Declaración de métodos abstractos

Lo más habitual es que utilicemos las interfaces para declarar métodos abstractos que luego tendrán que implementar las clases que las instancien.

La forma de declarar estos métodos será similar a la forma en que lo hacemos en las clases abstractas, con la diferencia de que en este caso no será necesario utilizar las palabras claves “abstract” ni “public” ya que se asumen por defecto abstractos y públicos.

```
public interface IAnimal {  
    String PLANETA = "Tierra";  
  
    void alimentar(float cantidadComida);  
    void desplazar();  
    float velocidad();  
}
```

#### 4.4. Implementación de métodos por defecto

Los métodos por defecto se declaran anteponiendo la palabra reservada “default” y son “public” sin necesidad de especificarlos.

Estos métodos serán incorporados a las clases que implementen el interfaz y serán accesibles para todos los objetos de dichas clases.

Además, las clases que implementan la interfaz podrán sobreescribir estos métodos.

```
public interface IAnimal {  
    String PLANETA = "Tierra";  
  
    void alimentar(float cantidadComida);  
    void desplazar();  
    float velocidad();  
  
    default void hacerRuido(){  
        System.out.println("Shhhhh");  
    }  
}
```

#### 4.5. Implementación de métodos estáticos

En una interfaz también se pueden implementar métodos estáticos. Si no se especifica un modificador de acceso se tomarán como “public”. Estos métodos pertenecen a la interfaz, y no a las clases que la implementan, por tanto, este método será accesible directamente desde la interfaz.

```
public interface IAnimal {  
    String PLANETA = "Tierra";  
  
    void alimentar(float cantidaComida);  
    void desplazar();  
    float velocidad();  
  
    static int calcularEdad(int anioNacimiento){  
        return LocalDate.now().getYear() - anioNacimiento;  
    }  
}
```

Para llamar a la función lo haríamos así...

```
int edad = IAnimal.calcularEdad(1999);
```

#### 4.6. Implementación de métodos privados

Además de los métodos abstractos, por defecto y estáticos, que son públicos por defecto, en una interfaz se pueden implementar métodos privados anteponiendo el modificador private. Pueden ser tanto estáticos como no estáticos y no son accesibles fuera del código de la interfaz. Estos métodos están implementados y solo pueden ser invocados por el resto de métodos no abstractos de la interfaz. En general, son métodos auxiliares para uso interno de otros métodos de la interfaz.

```
public interface IAnimal {  
    String PLANETA = "Tierra";  
  
    void alimentar(float cantidaComida);  
    void desplazar();  
    float velocidad();  
}
```

```
static int calcularEdad(int anioNacimiento){  
    return getAnioActual() - anioNacimiento;  
}  
  
private static int getAnioActual(){  
    return LocalDate.now().getYear();  
}  
}
```

#### 4.7. Implementación de interfaces

Una vez definida nuestra interfaz, para que una clase la implemente debe utilizar la palabra clave “implements” y además estará obligado a implementar todos los métodos abstractos de la interfaz, a no ser que se trate de una clase abstracta.

```
public class Perro implements IAnimal {  
    private float VELOCIDAD = 30;  
  
    @Override  
    public void alimentar(float cantidadComida) {  
        System.out.println("Como " + cantidadComida + " kilos de carne al  
día");  
    }  
  
    @Override  
    public void desplazar() {  
        System.out.println("Me desplazo a cuatro patas");  
    }  
  
    @Override  
    public float velocidad() {  
        return VELOCIDAD;  
    }  
}
```

Como ya hemos comentado antes una clase puede implementar varias interfaces simultáneamente. En este caso deberíamos implementar los métodos abstractos de todas las interfaces que implementa.

```
public class Perro implements IAnimal, IMamal {  
    ...  
}
```

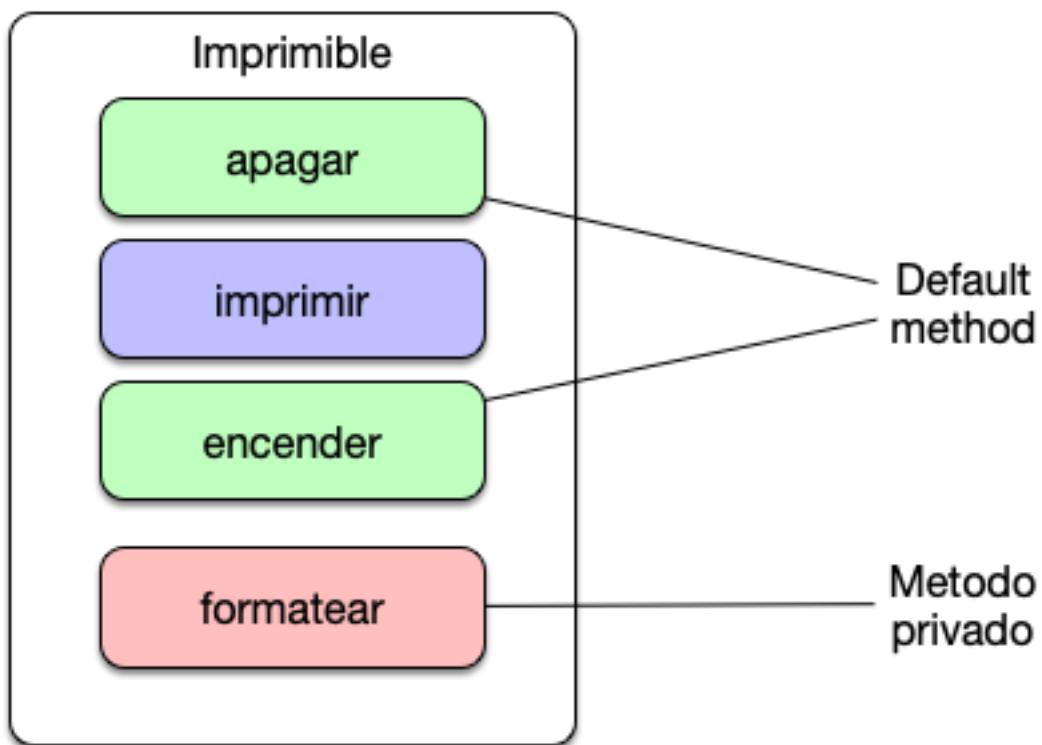
También existe la posibilidad de que una clase herede una clase e implemente una o varias interfaces a la vez.

```
public class Perro extends AnimalTerrestre implements IAnimal, IMamal {  
    ...  
}
```

Y, finalmente, también existe la posibilidad de que una interfaz herede de otra.

```
public interface IMamal extends IAnimal {  
    ...  
}
```

Esto provocaría que la interfaz que estamos definiendo heredaría todos los atributos y todos los métodos, a excepción de los privados de la interfaz madre.



#### 4.8. Clases abstractas vs interfaces

Una interfaz puede parecer similar a una clase abstracta, pero existen una serie de diferencias entre ellas:

- Una interfaz no declara variables de instancia (atributos)
- Una clase puede implementar varias interfaces, pero sólo puede heredar de otra clase
- Una clase abstracta pertenece a una jerarquía de clases, mientras que una interfaz no pertenece a una jerarquía de clases. En consecuencia, clases sin relación de herencia pueden implementar la misma interfaz.

Aunque depende de los casos, habitualmente es más conveniente utilizar interfaces que clases abstractas, debido a que nos estamos tan condicionados por la jerarquía y además nos permite implementación múltiple de interfaces por parte de una clase, mientras que las clases solo puede heredar de una única clase padre.

Abstract Class	Interface
1. <i>abstract</i> keyword	1. <i>interface</i> keyword
2. Subclasses <i>extends</i> abstract class	2. Subclasses <i>implements</i> interfaces
3. Abstract class can have implemented methods and 0 or more abstract methods	3. Java 8 onwards, Interfaces can have default and static methods
4. We can extend only one abstract class	4. We can implement multiple interfaces



#### 4.9. Variables de tipo interfaz

Igual que ocurre con las subclases y las superclases, es posible que una situación determinada nos interese crear una variable de tipo interfaz al que pasaremos como valor un objeto de una clase que implemente dicha interfaz. Esto nos puede servir, entre otras cosas, para ocultar funcionalidad propia de la clase que no deseamos que se utilice desde fuera del objeto.

```
IAnimal animal = new Perro();
```

### 5. COMPARACIÓN ESTÁTICA Y DINÁMICA DE TIPOS



Existen muchas interfaces de gran importancia incluidas en la API de Java. Dos de ellas nos permiten realizar la comparación de valores y objetos, que podremos utilizar para realizar operaciones de búsqueda u ordenación, por ejemplo.

#### 5.1. Interfaz Comparable

Cuando queremos establecer un criterio de comparación natural o por defecto entre los objetos de una clase, haremos que implemente la interfaz Comparable, que consta de un único método abstracto.

```
int compareTo(Object ob);
```

Este método será el encargado de establecer un criterio de ordenación entre los objetos de una clase. Para comparar dos objetos de una determinada clase que implemente Comparable, escribiremos:

```
obj1.compareTo(obj2);
```

que devolverá un número entero.

Si en una ordenación el objeto obj1 debe ir antes que el objeto obj2, el método devolverá un número negativo; si debe ir después, devolverá un número positivo, y si deben ser iguales a efectos de ordenación, devolverá un cero.

Veamos un ejemplo:

```
public class Automovil implements Comparable {  
    private float maxVelocidad;  
    private String marca;  
    private String modelo;  
  
    public Automovil(float maxVelocidad, String marca, String modelo){  
        this.maxVelocidad = maxVelocidad;  
        this.marca = marca;  
        this.modelo = modelo;  
    }  
  
    @Override  
    public int compareTo(Object o) {  
        Automovil otroAutomovil = (Automovil) o;  
        if(maxVelocidad < otroAutomovil.maxVelocidad){  
            return -1;  
        }  
        else if(maxVelocidad > otroAutomovil.maxVelocidad){  
            return 1;  
        }  
        else{  
            return 0;  
        }  
    }  
}
```

```

Automovil coche1 = new Automovil(220, "Audi", "A3");
Automovil coche2 = new Automovil(150, "Ford", "Fiesta");

if(coche1.compareTo(coche2) > 0){
    System.out.println("Coche1 es más rápido");
}
else{
    System.out.println("Coche2 es más rápido");
}

```

Esto podemos utilizarlo para ordenar arrays de objetos, por ejemplo:

```

Automovil[] automoviles = new Automovil[]{
    new Automovil(220, "Audi", "A3"),
    new Automovil(150, "Ford", "Fiesta"),
    new Automovil(300, "Ferrari", "F40")};

Arrays.sort(automoviles);
//Nos ordenará el array así: Ford Fiesta, Audi A3, Ferrari F40

```

## 5.2. Interfaz Comparator

La interfaz Comparable proporciona un criterio de ordenación natural, que es el que usan por defecto los distintos métodos de la API de Java, como sort(). Pero es frecuente que tengamos que ordenar los objetos de la misma clase con distintos criterios en el mismo programa. Para resolver estos casos existe la interfaz Comparator, definida también en la API. Antes de usarla habrá que importarla con la sentencia:

```
import java.util.Comparator;
```

Esta interfaz tiene un único método abstracto:

```
int compare(Object obj1, Object obj2);
```

Recibe como parámetros dos objetos que queremos comparar para determinar cuál va antes y cuál después en un proceso de ordenación. Devuelve un entero, que será negativo si obj1 va antes que obj2, positivo si va después y cero si son iguales. Llamamos comparador a cualquier objeto de una clase que implemente la interfaz Comparator.

Necesitaremos una clase de comparadores distinta por cada criterio de comparación que queramos emplear con objetos de una clase determinada. Pero, a diferencia de la interfaz Comparable, Comparator no se implementa en la clase de los objetos que queremos ordenar, sino en una clase específica, cuyos objetos llamaremos comparadores.

Veamos un ejemplo:

```
public class ComparadorCoches implements Comparator {  
  
    @Override  
    public int compare(Object o1, Object o2) {  
        Automovil automovil1 = (Automovil) o1;  
        Automovil automovil2 = (Automovil) o2;  
  
        if(automovil1.getMaxVelocidad() < automovil2.getMaxVelocidad()){  
            return -1;  
        }  
        else if(automovil1.getMaxVelocidad() >  
automovil2.getMaxVelocidad()){  
            return 1;  
        }  
        else{  
            return 0;  
        }  
    }  
}
```

```
ComparadorCoche comparadorCoche = new ComparadorCoche();  
  
if(comparadorCoches(coche1, coche2)){  
    System.out.println("Coche1 es más rápido");  
}  
else{  
    System.out.println("Coche2 es más rápido");  
}
```

Estas clases también las podremos utilizar para ordenar arrays, pero en esta ocasión debemos poner el objeto del comparador como segundo parámetro.

```
Automovil[] automoviles = new Automovil[]{  
    new Automovil(220, "Audi", "A3"),  
    new Automovil(150, "Ford", "Fiesta"),  
    new Automovil(300, "Ferrari", "F40")};
```

```
Arrays.sort(automóviles, comparadorCoches);
//Nos ordenará el array así: Ford Fiesta, Audi A3, Ferrari F40
```

Comparable	Comparator
An object comparing itself with other object.	Compares two objects.
Class itself must implement Comparable interface.	External Class implements the Comparator Interface.
Can be used to sort according to only one property.	Can be used to sort according to multiple properties.

## 6. CONVERSIONES DE TIPOS ENTRE OBJETOS

Es habitual que, en un array, u otra de las estructuras de almacenamiento en memoria que veremos más adelante, guardemos elementos que corresponden a la misma superclase, pero a subclases distintas. Como por ejemplo en siguiente caso... Imaginemos que Vendedor y JefeSeccion son subclases de Empleado:

```
Empleado[] empleados = new Empleado[3];

empleado[0] = new Vendedor(1, "James", "654789032");
empleado[1] = new Vendedor(2, "Steve", "656889459");
empleado[2] = new JefeSeccion(3, "Mark", "667739723");
```

Como vimos en la sección sobre ligadura dinámica, un objeto de una subclase también pertenece a la superclase de la que hereda, y por ello es posible que en un mismo array puedan convivir objetos de distinta clase, pero misma superclase.

Ahora supongamos que cada una de las supclases tiene un método propio que no comparte con las otras...

```
public clase Vendedor extends Empleado{  
    ...  
  
    public double calcularComision(double importe){  
        ...  
    }  
}  
  
public clase JefeSeccion extends Empleado{  
    ...  
  
    public void generarHorarioTurnos(){  
        ...  
    }  
}
```

En este caso la clase Vendedor cuenta el método propio calcularComision y la clase JefeSeccion con el método propio generarHorarioTurnos.

Si nosotros intentásemos, por ejemplo, acceder al método calcularComision de uno de los Vendedores almacenados en el array, Java no nos lo permitiría, ya que, como vimos en la sección del enlace dinámico mientras que el objeto esté identificado como un objeto de la superclase, no tendrá acceso a las clases propias de las subclases, es decir, no podríamos hacer lo siguiente:

```
empleado[1].calcularComision(100);
```

Sino que para ello necesitaríamos realizar un casting del objeto de la superclase a la subclase. Para ello vamos a utilizar algo que ya hicimos con los valores numéricos... vamos a colocar el tipo de la subclase entre paréntesis delante del objeto...

```
Vendedor vendedor1 = (Vendedor) empleado[1];  
vendedor1.calcularComision(100);
```

Llevando a cabo este casting, o conversión de tipo entre clases, ya nos será posible acceder al método que necesitemos de la subclase.

Existe un operador que nos va a ser de gran utilidad a la hora de identificar si un objeto dentro de un array de la superclase es de una subclase u otra. Este operador es “instanceof”. Veamos un ejemplo práctico para entenderlo mejor.

Cogemos el array del ejemplo anterior, queremos recorrerlo y que cuando nos encontremos con un Vendedor calcule la comisión de una venta de 100€ y cuando se trate de un JefeSeccion, que genere el horario de turnos de su sección.

```
Empleado[] empleados = new Empleado[3];  
  
empleado[0] = new Vendedor(1, "James", "654789032");  
empleado[1] = new Vendedor(2, "Steve", "656889459");  
empleado[2] = new JefeSeccion(3, "Mark", "667739723");
```

A primera vista sería imposible distinguir unos de otros, pero gracias a instanceof va a ser posible de la siguiente manera:

```
Vendedor vendedor;  
JefeSeccion jefeSeccion;  
  
for(Empleado empleado : empleados){  
    if(empleado instanceof Vendedor){  
        vendedor = (Vendedor)empleado;  
        vendedor.calcularComision(100);  
    }  
    if(empleado instanceof JefeSeccion){  
        jefeSeccion = (JefeSeccion)empleado;  
        jefeSeccion.generarHorarioTurnos();  
    }  
}
```

Como veis, instanceof nos devolverá un true o false en función de si la clase del objeto coincide con la que estamos comparando, y va a ser el encargado de determinar si el empleado que hemos seleccionado es un Vendedor o un

JefeSeccion, y una vez identificado podemos realizar el casting y utilizar los métodos o atributos propios de esa subclase.

## 7. CLASES Y TIPOS GENÉRICOS O PARAMETRIZADOS

Los genéricos son clases, o tipos, que tienen un parámetro. Al crear una clase genérica, se especifica no solo la clase, sino también el tipo de dato con el que funcionará.

Un ejemplo muy simple de definición de clase genérica sería el siguiente:

```
public class Box<T> {  
    // T es el "tipo"  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

Se suele utilizar la letra T mayúscula como indicador de que ahí debe incluirse una clase a la hora de instanciar la clase, pero se podría utilizar cualquier otra letra.

Si te fijas, cuando tenemos que utilizar el tipo de dato que corresponde al que hemos pasado como parámetro a la clase, utilizaremos la misma letra para que el compilador identifique que se trata de la misma clase, o tipo de datos.

Vamos a intentar entender para qué sirven las clases genéricas con el siguiente ejemplo...

Cuando nosotros instanciamos un objeto del tipo Box, podremos pasarle cualquier clase como parámetro, por ejemplo:

```
Box<Persona> cajaPersona = new Box<Persona>();
```

De esta forma, podríamos decir, que todas las partes del código donde pone “T” se sustituiría por Persona. Algo así...

```
public class Box<Persona> {
    private Persona persona;

    public void set(Persona persona) { this.persona = persona; }
    public Persona get() { return persona; }
}
```

Pero ¿qué pasaría si en lugar de personas quiero guardar animales en mi clase Box? Pues solo tendríamos que hacer lo siguiente:

```
Box<Animal> cajaAnimal = new Box<Animal>();
```

Y Java identificaría que lo que queremos hacer es esto:

```
public class Box<Animal> {
    private Animal animal;

    public void set(Animal animal) { this.animal = animal; }
    public Animal get() { return animal; }
}
```

Con todo lo que hemos aprendido hasta ahora, si quisiésemos tener una clase Box que almacenara Personas y Animales, estaríamos obligados a crear dos clases distintas, y, por tanto, repetir código. Así:

```
public class BoxPersona {
    private Persona persona;

    public void set(Persona persona) { this.persona = persona; }
    public Persona get() { return persona; }
}

public class BoxAnimal {
    private Animal animal;

    public void set(Animal animal) { this.animal = animal; }
    public Animal get() { return animal; }
}
```

Pero gracias a las clases genéricas podemos utilizar el mismo código para una o para muchas clases distintas, únicamente variando el tipo que le pasamos por parámetro a la clase.

```
public class Box<T> {  
    // T es el "tipo"  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

Igual que con las clases, podemos crear interfaces genéricas:

```
public interface MyInterface<T> {  
    void doSomething(T value);  
}
```

En algunas ocasiones solo nos interesará que se pueden pasar por parámetro una clase y sus subclases, o una subclase y su superclase. Para ello utilizaremos los parámetros genéricos limitados.

En el caso de querer que solo puedan pasarse una clase y sus subclases, utilizaríamos un código similar a este:

```
class nombreClase<T extends claseLmite>{  
    ...  
}
```

Esto obligará a que cuando generemos una instancia de la clase debamos introducir por parámetro la clase claseLmite o alguna de sus subclases.

En el caso de querer que solo puedan pasarse una subclase y su superclase, utilizaríamos un código similar a este:

```
class nombreClase<T super claseLmite>{  
    ...  
}
```

Esto obligará a que, al crear una instancia de nombreClase, debamos introducir como parámetro la subclase claseLmite o su superclase.

También podemos aplicar estas limitaciones a clases que instancien un interfaz, y se haría de la misma forma que con las clases, utilizando la palabra “extends” en lugar de “implements”.

```
class nombreClase<T extends MiInterfaz>{  
    ...  
}
```

## 8. ENUMERADOS

Un enumerado es una clase “especial”, tanto en Java como en otros lenguajes, que limitan la creación de objetos a los especificados explícitamente en la implementación de la clase. La única limitación que tienen los enumerados respecto a una clase normal es que, si tienen constructor, este debe de ser privado para que no se puedan crear nuevos objetos.

En nuestro caso solo los vamos a utilizar para crear agrupaciones de constantes, pero con los enumerados se pueden hacer muchas más cosas. Os recomiendo que, si os interesa, investiguéis sobre el tema.

Para declarar un enumerador necesitaremos utilizar la palabra reservada “enum”, le daremos nombre y, entre llaves introduciremos los literales que deseamos utilizar.

```
public enum Demarcacion {  
    PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO  
}
```

Al tratarse de constantes, se recomienda poner los nombres de los literales en mayúsculas.

Empezando por el primero hasta el último se asigna de forma automática un valor, comenzando por 0, a cada uno de los literales. Es decir, en el ejemplo anterior PORTERO sería el 0, DEFENSA el 1, etc...

Ahora vemos las operaciones básicas que podemos utilizar con este tipo de enumerados.

Utilizar uno de los valores del enumerado:

```
Demarcacion.DELANTERO;
```

Asignar a una variable del tipo enumerado:

```
Demarcacion delantero = Demarcacion.DELANTERO;
```

Extraer el nombre correspondiente de una variable asignada un enumerado:

```
delantero.name(); //Devuelve un String con el nombre de la constante  
(DELANTERO)
```

Conocer la posición que ocupa, o el valor, del enumerado asignado:

```
delantero.ordinal(); // Devuelve un entero con la posición del enum según  
está declarado
```

Comparar dos variables del mismo tipo enum según su ordenación:

```
Demarcacion portero = Demarcacion.PORTERO;  
delantero.compareTo(portero); //Devolverá un valor superior a 0, ya que  
la posición de delantero es mayor que la de portero
```

Obtener un array con todos valores del enum:

```
Demarcacion.values(); //Devuelve un array que contiene todos los enum
```

Veamos el ejemplo completo:

```
public enum Demarcacion {  
    PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO  
}  
  
Demarcacion delantero = Demarcacion.DELANTERO;  
String nombre = delantero.name(); //Devuelve un String con el nombre de  
la constante (DELANTERO)  
int posicion = delantero.ordinal(); // Devuelve un entero con la posición  
del enum según está declarado  
  
Demarcacion portero = Demarcacion.PORTERO;  
if(delantero.compareTo(portero) > 0) { //Devolverá un valor superior a 0,  
ya que la posición de delantero es mayor que la de portero  
    System.out.println("El delantero está en una posición superior al  
portero");  
}  
else{  
    System.out.println("El portero está en una posición superior al  
delantero");  
}  
  
Demarcacion[] demarcaciones = Demarcacion.values(); //Devuelve un array  
que contiene todos los enum
```

Los enums los podemos incluir en el fichero de otra clase, dentro o fuera de la propia clase. Si lo hacemos dentro, normalmente es para que solo esa clase pueda utilizarlo.

```
public enum Demarcacion {  
    PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO  
}  
  
public class Clase{  
    ...  
}  
  
public class Clase{  
    private enum Demarcacion {  
        PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO  
    }  
    ...  
}
```

# **UT-08: COLECCIONES DE DATOS: LISTAS, PILAS Y COLAS.**

**¿Qué vamos a ver?**

- Colecciones de datos.
- Tipos de colecciones: listas, pilas, colas, tablas.
- Operaciones con colecciones.
  - Jerarquías de colecciones.
  - Acceso a elementos.
  - Recorridos. Iteradores.
  - Ordenación de listas
- Clases para la lectura de documentos de intercambio de datos.
- Uso de clases y métodos genéricos.
- Otras colecciones: interfaces map y set.

## 1. COLECCIONES DE DATOS

A menudo necesitamos guardar información, pero no sabemos de antemano el espacio que va a ocupar en memoria. En estos casos, los arrays no son la solución adecuada, ya que su tamaño debe permanecer fijo una vez creadas. Al redimensionarlas, lo que hacemos es crear otras nuevas y copiar todos los datos de la antigua, con la sobrecarga que ello supone para la gestión de memoria. En su lugar, necesitamos estructuras dinámicas de datos, es decir, objetos que alberguen datos que se pueden insertar y eliminar, cambiando el tamaño de la estructura sin alterar los datos restantes, todo ello en tiempo de ejecución.

Una colección es un objeto que sirve para agrupar un conjunto de objetos, llamados elementos, que generalmente guardan una relación entre ellos. Los métodos de las colecciones nos permiten llevar a cabo distintas operaciones con sus elementos, como inserción, eliminación, búsqueda y ordenación.

## 2. TIPOS DE COLECCIONES: LISTAS, PILAS, COLAS Y TABLAS

Hasta el momento hemos visto un único tipo de colecciones, las tablas, que se trata de un contenedor de datos estático.

A partir de ahora vamos a conocer distintos tipos de colecciones dinámicos, que nos permitirán gestionar de una forma mucho más eficiente la memoria y el almacenamiento de los datos. Estos tipos de almacenamiento dinámica se denominan TAD o Tipos Abstractos de Datos.

En los siguientes apartados nos vamos a centrar en cómo se construyen tres tipos de TADs:

- Listas
- Pilas
- Colas

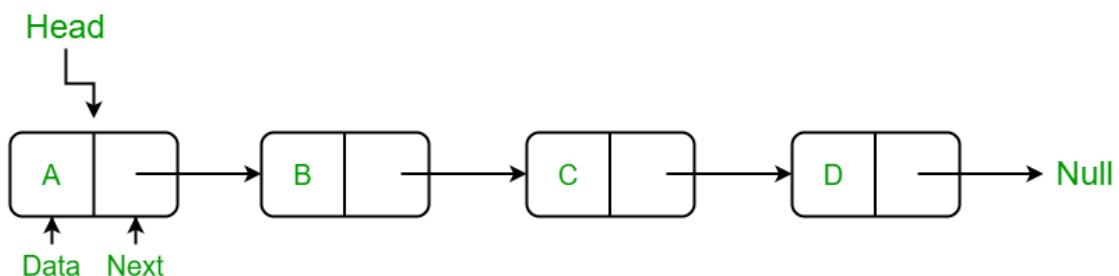
## 2.1. Listas

Responden a la necesidad de manejar sucesiones de datos que pueden estar repetidos y cuyo orden puede ser relevante. De alguna forma sustituyen a las tablas, con la diferencia de que podemos insertar y eliminar datos en ellas sin limitaciones de espacio. A cada elemento que compone una lista lo denominamos nodo, e incluye los datos a almacenar y los punteros necesarios.

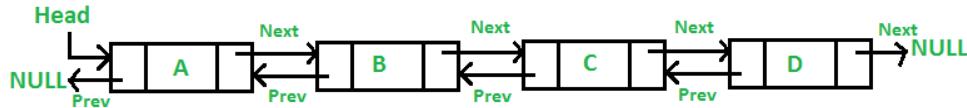


Dentro de las listas nos podemos encontrar con distintos tipos en función de cómo están enlazados sus elementos y cómo se pueden recorrer:

- **Listas enlazadas simples:** cada nodo apunta al siguiente. El último nodo apunta a nulo.



- **Listas doblemente enlazadas:** cada nodo apunta al siguiente y al anterior. En el caso del primer nodo el anterior apunta a nulo y en el caso del último nodo el siguiente apunta a nulo.



- **Listas circulares:** cada nodo apunta al siguiente y al anterior. El primer nodo apunta al último y al segundo. El último nodo apunta al primero y al penúltimo.



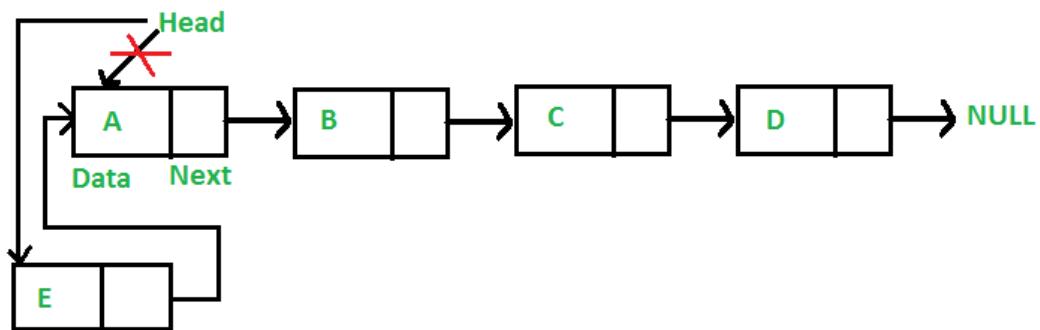
Las listas siempre necesitarán una variable que almacene de forma permanente la dirección del primer nodo, a esto lo llamaremos “cabeza”.

En esta unidad nos vamos a centrar en las listas enlazadas simples.

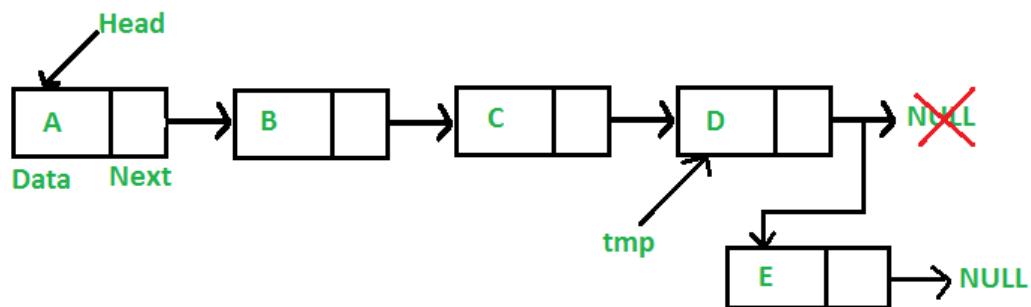
Con las listas (cualquiera de los tres tipos) podremos realizar las siguientes operaciones:

- Creación: establecemos la cabeza de la lista a nulo.
- Recorrido: utilizaremos una variable temporal para apuntar a la posición actual en la que nos encontramos de la lista.
- Inserción

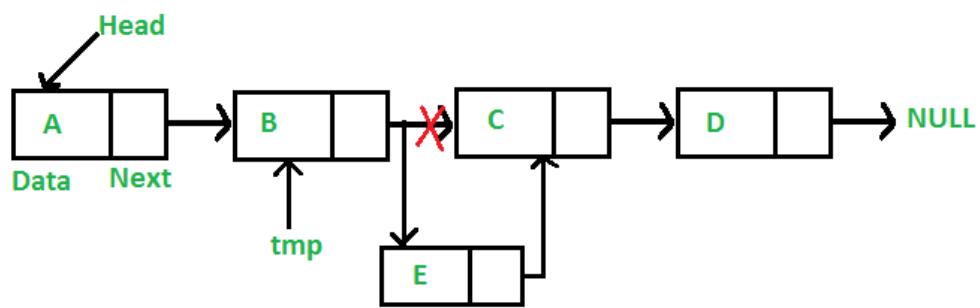
- Inserción en la primera posición (solo listas no ordenadas)



- Inserción en la última posición (solo listas no ordenadas)



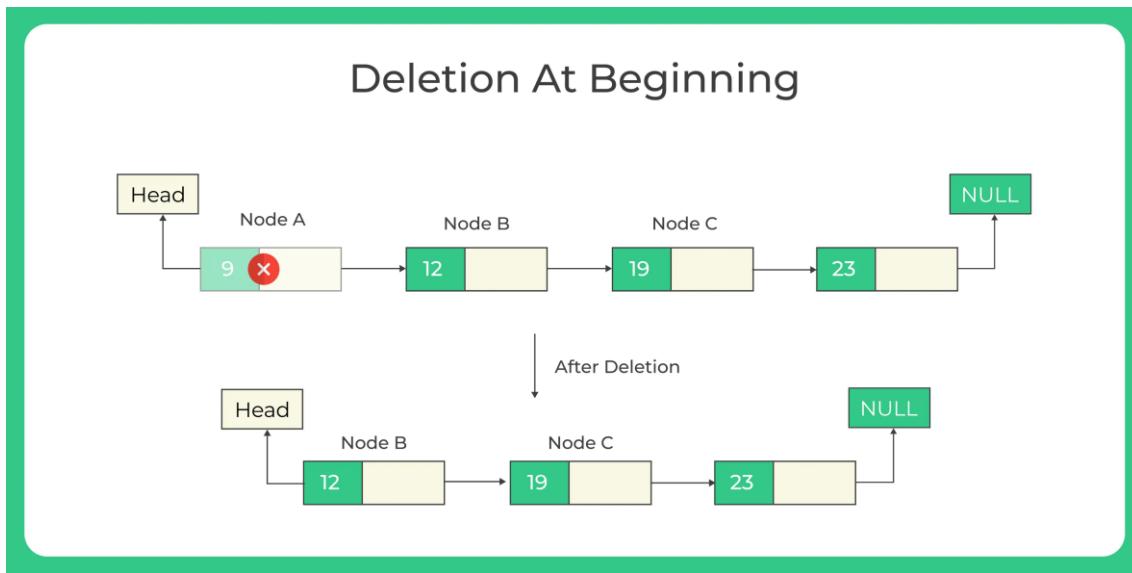
- Inserción en una posición determinada (solo listas no ordenadas)



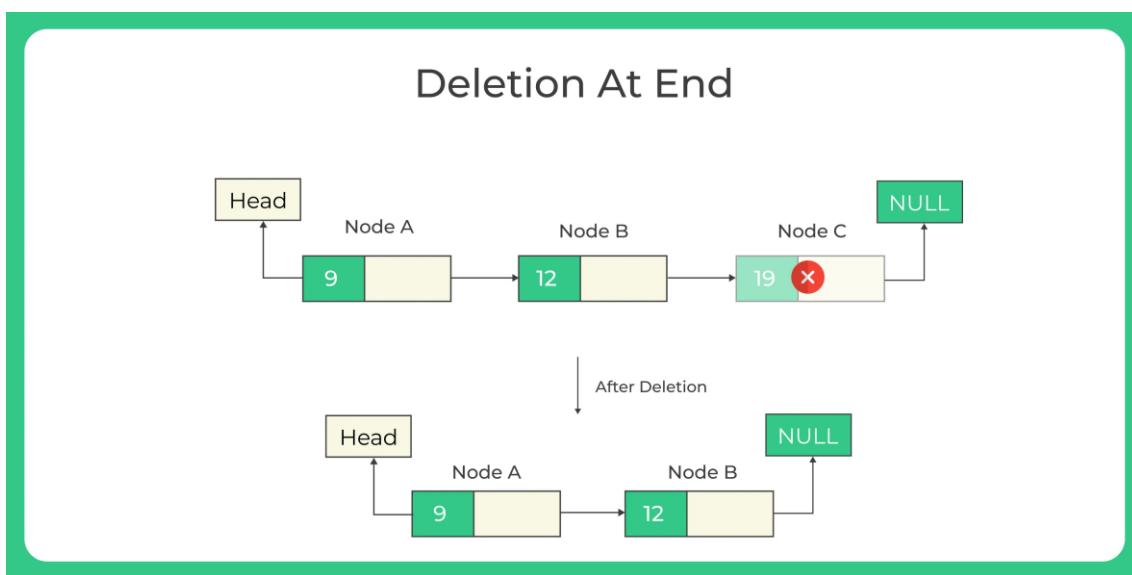
- Inserción ordenada (solo lista ordenadas)

- Eliminación

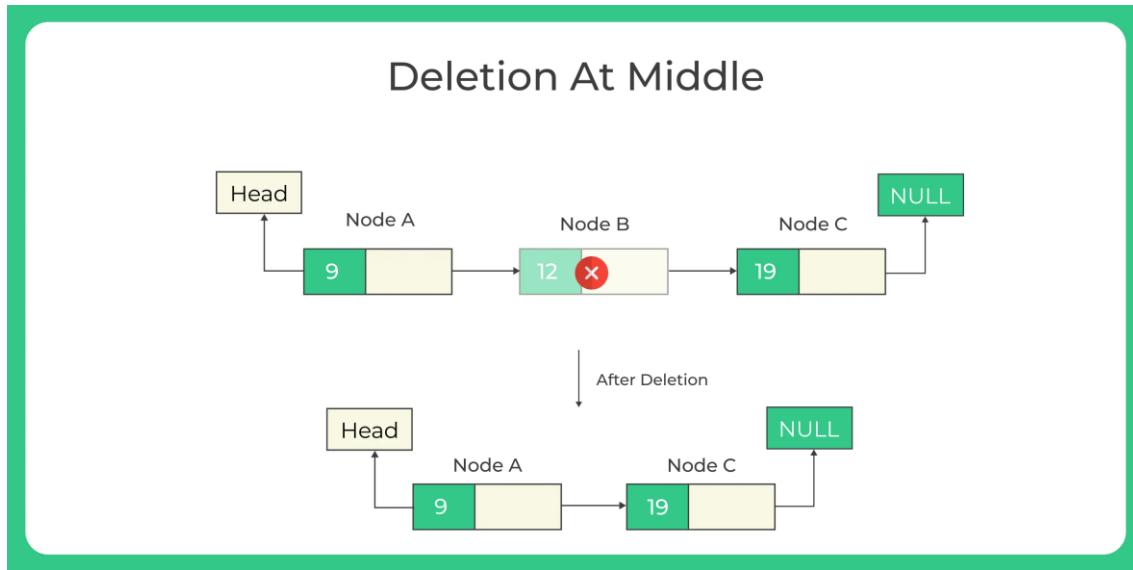
- Eliminación de la primera posición



- Eliminación de la última posición



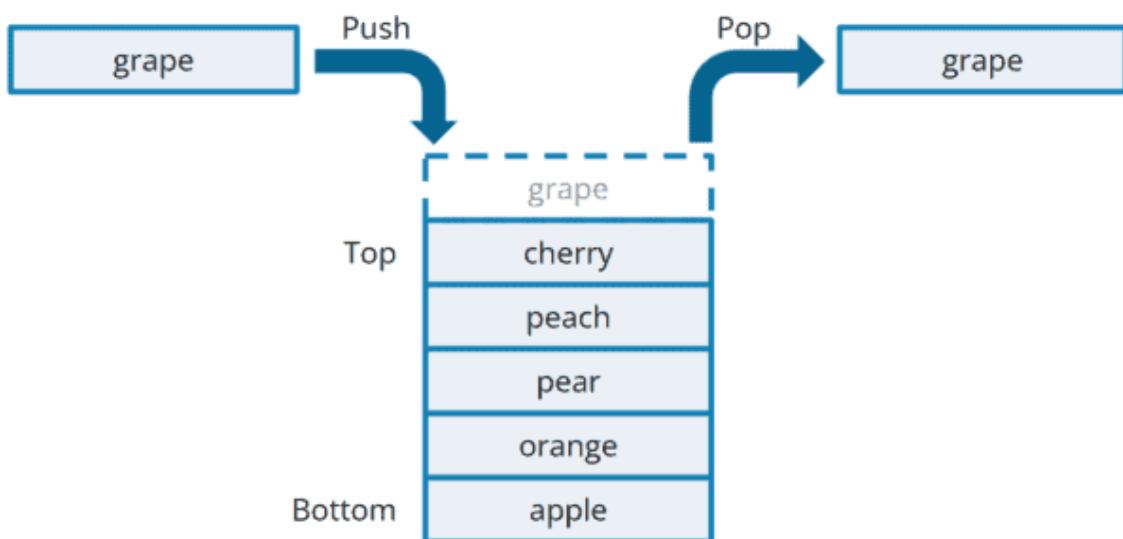
- Eliminación en una posición determinada



- Eliminación de un elemento determinado

## 2.2. Pilas

Las pilas son un tipo de colección similar en funcionamiento a la lista, pero con la particularidad de que sigue el máxima LIFO (Last In First Out), es decir, el último elemento en entrar en la pila será el primero en salir de ella.





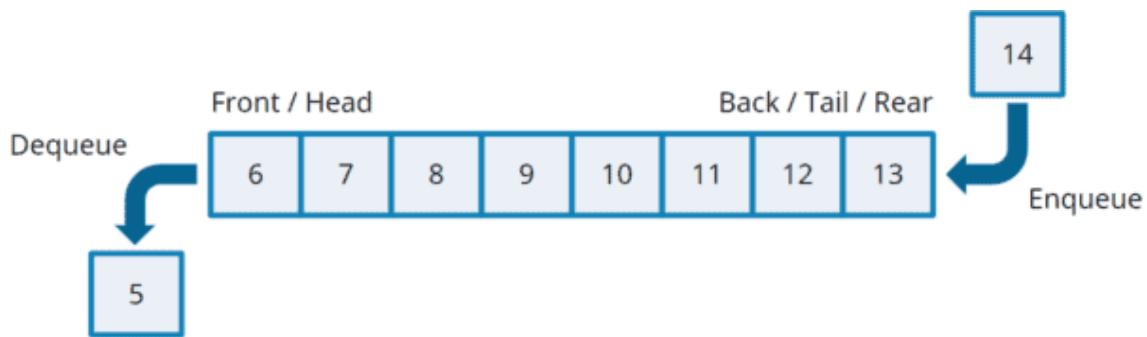
Esto se consigue de la siguiente manera:

- Realizando las inserciones únicamente en la cabecera. A esto se le denomina Apilar o Push.
- No nos será posible acceder a otro dato que no sea el de cabecera.
- Solo se podrá eliminar el nodo de la cabecera. A esto se le denomina Desapilar o Pop.

Por tanto, a la hora de implementar nuestra propia pila de almacenamiento solo tendremos que crear una estructura como la de la lista, pero solo nos centraremos en la inserción y la eliminación en la primera posición.

### 2.3. Colas

Al igual que ocurre con las pilas, las colas tienen un funcionamiento similar a las listas, pero con una determinada restricción... siguen el comportamiento FIFO (First In First Out), es decir, el primer elemento en entrar en la cola será el primero en salir de la cola.



Esto se consigue de la siguiente manera:

- Además de una variable que almacene la posición de la cabecera, necesitaremos una para la cola.
- Realizando las inserciones únicamente en la cola. A esto se le denomina Encolar o Queue.
- No nos será posible acceder a otro dato que no sea el de cabecera.
- Solo se podrá eliminar el nodo de la cabecera. A esto se le denomina Desencolar o Dequeue.

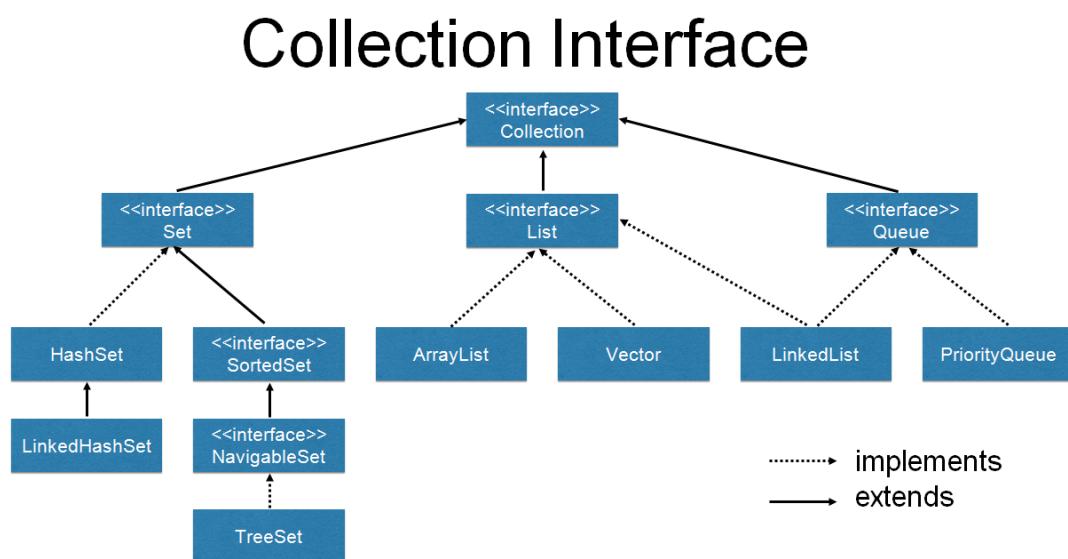
### 3. OPERACIONES CON COLECCIONES

Java proporciona una serie de estructuras dinámicas que comparten un conjunto de métodos declarados en la interfaz Collection.

#### 3.1. Jerarquías de colecciones

Hay colecciones de diferentes tipos, adaptadas a distintos fines más o menos específicos. Por eso no existe una clase colección, sino todo un marco de trabajo (framework), con una estructura jerárquica de interfaces que se implementan en distintas clases, y con un conjunto de algoritmos que permiten la manipulación de los datos almacenados en las colecciones.

Todas ellas son dinámicas, ya que permiten añadir y quitar unidades de información (objetos llamados nodos o elementos) en tiempo de ejecución, sin más límite que la memoria del ordenador. Nosotros llamaremos coleccionees a todas las clases que implementen la interfaz Collection, aunque también usaremos el nombre del tipo especial de colección: lista, conjunto o cola.



En los apartados consecutivos, vamos a centrarnos en las listas (List), más concretamente en los `ArrayList`.

### 3.2. Acceso a elementos

En primer lugar, debemos tener en cuenta que las listas con un tipo de estructura de datos genéricos, es decir, podemos introducir cualquier tipo de dato dentro de ellas, de hecho, la definición de las listas y arraylist es la siguiente:

List<T>

ArrayList<T>

Después, debemos saber que List<T> es una interfaz y, por tanto, no podemos implementar objetos de tipos List<T>, pero sí objetos de clases que implementan la interfaz List<T> como es el caso de ArrayList<T>, y por tanto a la hora de crear una lista podemos hacerlo de dos formas:

```
List<tipoDato> lista = new ArrayList<>();  
  
ArrayList<tipoDato> lista = new ArrayList<>();
```

Una vez que hayamos creado nuestra lista vacía, debemos ir introduciendo los datos. Para ello vamos a utilizar el método add, ya sea para insertarlo al final de la lista con add(dato), o en una posición determinada de la lista con add(posición, dato):

```
List<String> lista = new ArrayList<>();  
  
lista.add("10");  
lista.add("25");  
lista.add("30");  
lista.add(1, "40");  
  
//ahora lista estará compuesto por {"10", "40", "25", "30"}
```

Si queremos acceder a cada uno de los elementos podemos hacerlo por su posición dentro de la lista mediante el método get(posición):

```
lista.get(1); //devolverá "40"  
lista.get(0); // devolverá "10"  
lista.get(2); // devolverá "25"
```

Si necesitamos modificar el dato de que ocupa una determinada posición vamos a utilizar el método `set(posición, nuevoDato)`:

```
lista.set(1, "55");
//ahora lista estará compuesto por {"10", "55", "25", "30"}
```

Para localizar elementos dentro de una lista tenemos varias opciones: `indexOf(dato)` nos devolverá el índice de la primera aparición del dato, `lastIndexOf(dato)` nos devolverá el índice de la última aparición del dato y `contains(dato)` nos indicará si ese dato se encuentra en la lista.

```
lista.indexOf("25"); // {"10", "55", "25", "30"} devolverá 2
lista.lastIndexOf("30");// {"30", "55", "25", "30"} devolverá 3
lista.contains("75"); // {"10", "55", "25", "30"} devolverá false
```

Si queremos eliminar uno de los elementos de la lista podemos hacerlo mediante distintas variantes del método `remove`, por la posición... `remove(posición)`, ... o la primera aparición de un elemento `remove(dato)`:

```
lista.remove(0);
lista.remove("30");
//ahora lista estará compuesto por {"55", "25"}
```

Si queremos eliminar todos los elementos de una lista, para dejarla vacía, utilizaremos el método `clear()`.

```
lista.clear();
//ahora la lista está vacía
```

Para comprobar si una lista está vacía utilizamos el método `isEmpty()`.

```
lista.isEmpty();
//devolverá true si está vacía, false en caso contrario
```

Si deseamos conocer el número de elementos que componen la lista utilizaremos el método `size()`.

```
lista.size();
// {"10", "55", "25", "30"} devolverá 4
```

En la ayuda oficial de Java disponéis de algunos métodos más bastante útiles:

<https://docs.oracle.com/javase/8/docs/api/java/util/List.html>

### 3.3. Recorridos. Iteradores

Podríamos recorrer las listas con cualquiera las distintas instrucciones de bucle que ya vimos en su momento, pero lo más recomendable es utilizar for, ya sea en su variante normal con una variable como contador o con la variante que denominamos foreach. Veamos dos ejemplos que serían equivalentes.

```
TipoDatos objeto;
int max = lista.size();
for(int i = 0; i < max; i++){
    objeto = lista.get(i);
    ...
}
```

La variante del for con contador nos puede servir para recorrer una lista de forma parcial, en lugar de completo, mediante el control del contador.

En el for “clásico” vamos a almacenar el tamaño en una variable, puesto que lista.size() es un método y sería ineficiente consultar en cada vuelta el tamaño si siempre es el mismo. Pero en el caso de que dentro del for vayamos a insertar o eliminar elementos en la lista, esto no nos sería útil.

```
for(TipoDatos objeto : lista){
    objeto ...
}
```

El uso del foreach queda más elegante a la hora de recorrer una lista de forma completa.

Cuando necesitemos aumentar, o reducir, el contenido de la lista será más conveniente usar while y do while.

```
List<Integer> lista = new ArrayList<>();  
while(lista.size() < 10){  
    lista.add(0);  
}
```

### 3.4. Ordenación de listas

El interfaz List cuenta con un método denominado sort, al cual es necesario pasar como parámetro una clase que instancie el interfaz Comparator, que ya conocemos.

```
sort(Comparator <? Super> c)
```

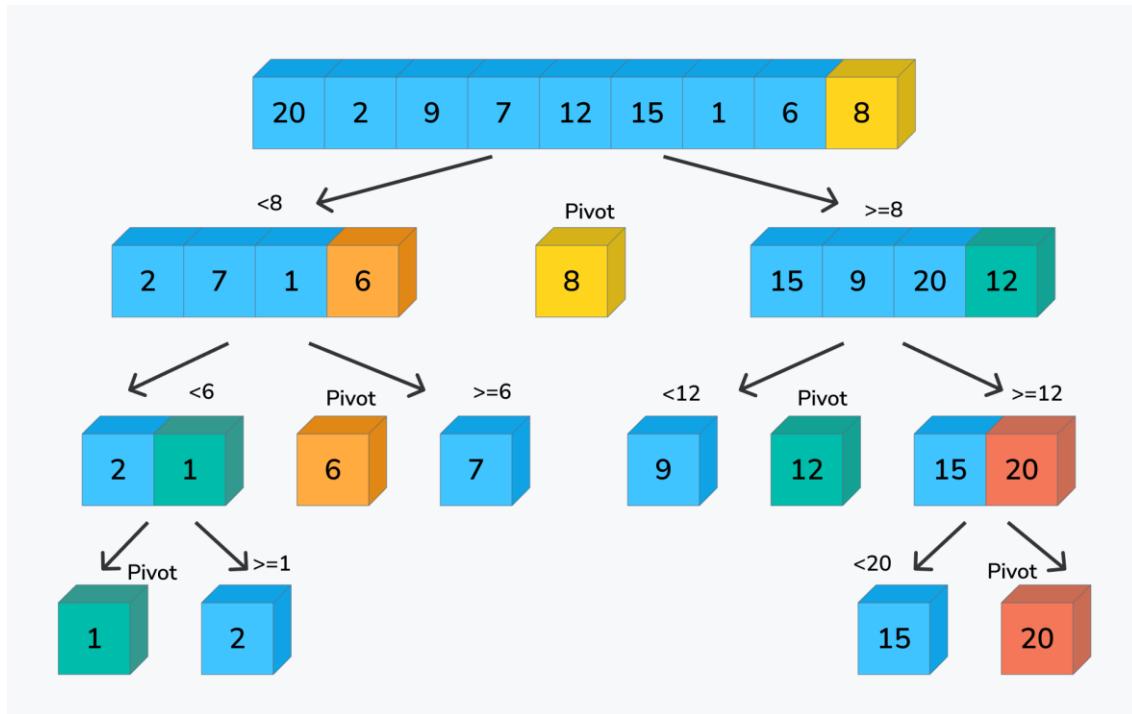
```
lista.sort(new CompartatorLista());  
  
o  
  
Import java.util.Collections;  
  
Collections.sort(lista, new CompartatorLista());
```

De esta forma ordenará los distintos elementos de la lista en función de las comparaciones mediante el Comparator.

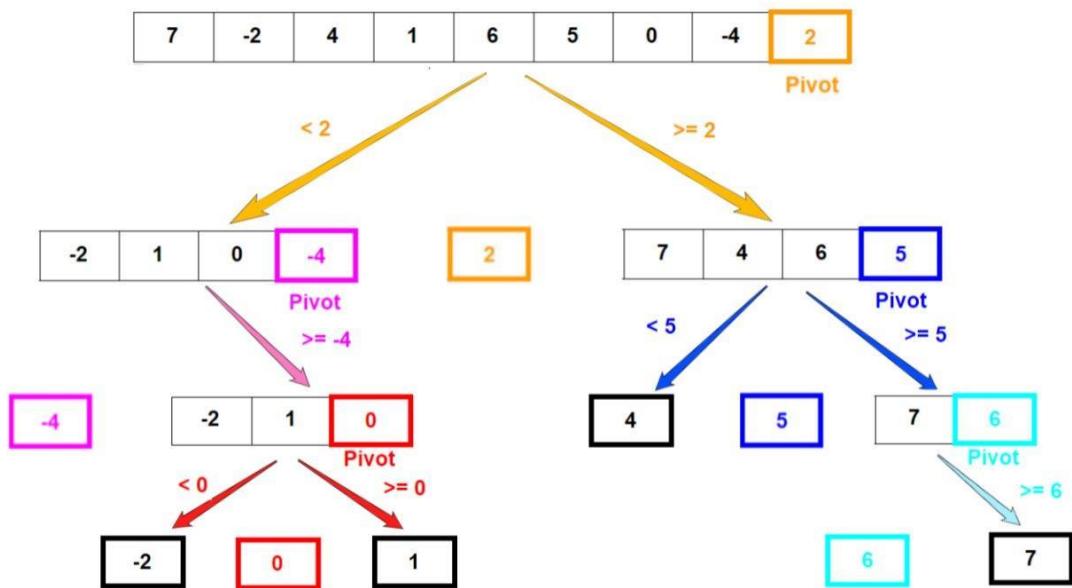
También existen otros modos de ordenación “manuales”, algo más avanzados a los que vimos en su momento con los arrays. Estos métodos de ordenación se basan en el concepto de “Divide y vencerás” y en la recursividad. Se llaman Quick Sort y Merge Sort.

#### *3.4.1. Quick sort*

Ejemplo 1:



Ejemplo 2:

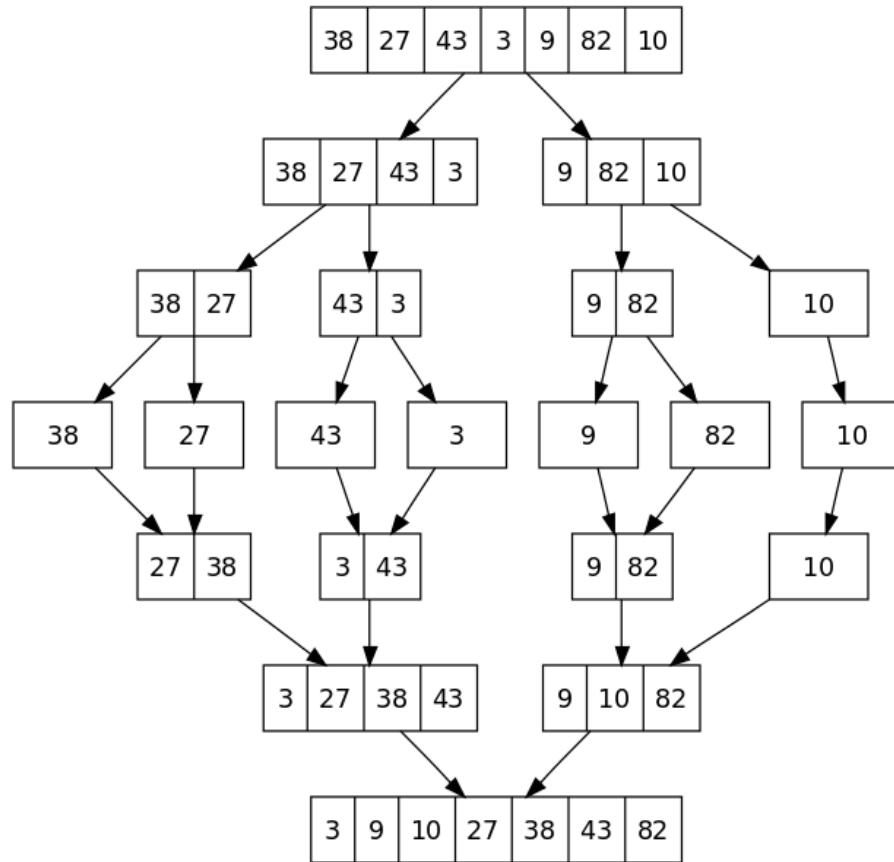


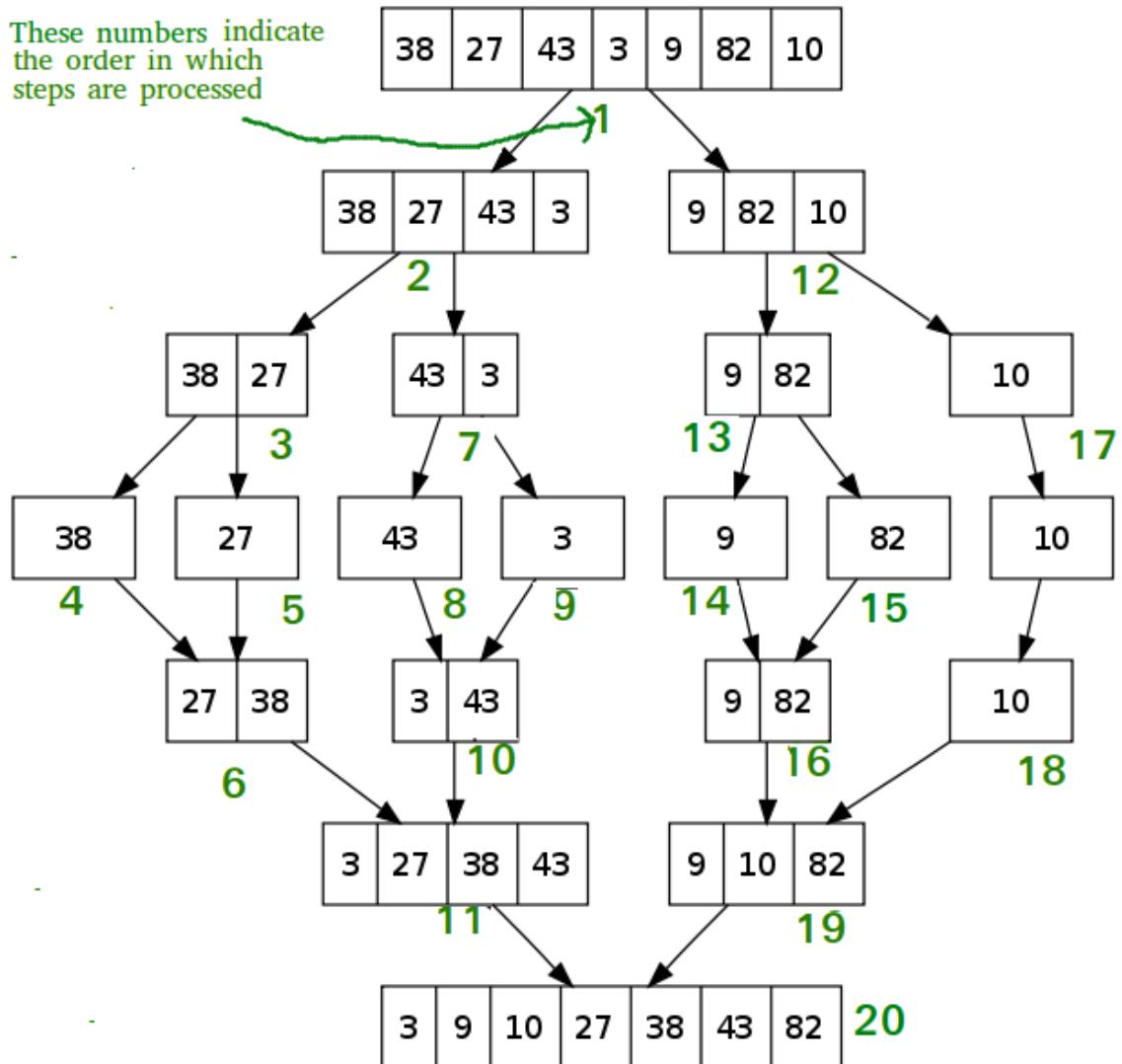
Quick sort se basa en determinar una posición pivote e ir dividiendo la lista en dos sublistas, una con los elementos menores al pivote y otra con los elementos mayores al pivote, quedándonos con este entre ambas sublistas. Después,

realizaremos la misma operación cada una de las sublistas, hasta quedarnos con listas de un solo elemento ordenadas de menor a mayor, y luego las unimos.

En el aula virtual disponéis de un vídeo explicativo de cómo funciona. Vamos a verlo...

### 3.4.2. Merge Sort





Merge sort se basa en ir dividiendo por la mitad la lista en secciones cada vez más pequeñas hasta que nos quedemos con varias listas de un solo elemento, y después la vamos a ir uniendo, por etapas, los distintos elementos en orden, hasta obtener la lista ordenada.

En el aula virtual disponéis de un vídeo explicativo de cómo funciona. Vamos a verlo...

## 4. CLASES PARA LA LECTURA DE DOCUMENTOS DE INTERCAMBIO DE DATOS

Existen una gran variedad de tipos de ficheros que se pueden utilizar para intercambiar información entre organizaciones por medios electrónicos, desde simples txt hasta otros muchos más complejos y encriptados. Algunos de los más conocidos actualmente son Json y XML. En nuestro caso nos vamos a centrar en estos últimos.

XML, siglas de lenguaje de marcado extensible, permite definir y almacenar datos de forma compatible. XML admite el intercambio de información entre sistemas de computación, como sitios web, bases de datos y aplicaciones de terceros. Las reglas predefinidas facilitan la transmisión de datos como archivos XML a través de cualquier red, ya que el destinatario puede usar esas reglas para leer los datos de forma precisa y eficiente.

Un archivo de lenguaje de marcado extensible (XML) es un documento basado en texto que se puede guardar con la extensión .xml.

Cualquier archivo XML incluye los siguientes componentes:

### 4.1. Estructura de los ficheros XML

#### *4.1.1. Declaración XML*

Un documento XML comienza con alguna información sobre el propio XML. Por ejemplo, podría mencionar la versión XML que sigue. Esta apertura se denomina declaración XML. A continuación se muestra un ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
```

#### *4.1.2. Elementos XML*

Todas las demás etiquetas que se crean en un documento XML se denominan elementos XML. Los elementos XML pueden contener las siguientes características:

- Texto

- Atributos
- Otros elementos

Todos los documentos XML comienzan con una etiqueta principal, que se denomina elemento raíz.

Por ejemplo:

```
<ListaInvitación>
<familia>
    <tía>
        <nombre>Cristina</nombre>
        <nombre>Estefanía</nombre>
    </tía>
</familia>
</ListaInvitación>
```

<Listalnvitación> es el elemento raíz; familia y tía son otros nombres de elementos.

#### 4.1.3. Atributos XML

Los elementos XML pueden tener otros descriptores denominados atributos. Puede definir sus propios nombres de atributos y escribir los valores de los atributos entre comillas, como se muestra a continuación.

```
<edad de la persona="22">
```

#### 4.1.4. Contenido XML

Los datos de los archivos XML también se denominan contenido XML. Por ejemplo, en el archivo XML, es posible que veas datos como este.

```
<amigo>
    <nombre>Carlos</nombre>
    <nombre>Esteban</nombre>
</amigo>
```

Los valores de los datos Carlos y Esteban son el contenido.

## 4.2. Escritura de ficheros XML con Java

Para poder crear un fichero XML desde nuestro programa vamos a necesitar importar una gran cantidad de librería. A continuación tenemos el listado:

```
java.io.File;
javax.xml.parsers.DocumentBuilder;
javax.xml.parsers.DocumentBuilderFactory;
javax.xml.parsers.ParserConfigurationException;
javax.xml.transform.Result;
javax.xml.transform.Source;
javax.xml.transform.Transformer;
javax.xml.transform.TransformerException;
javax.xml.transform.TransformerFactory;
javax.xml.transform.dom.DOMSource;
javax.xml.transform.stream.StreamResult;
org.w3c.dom Attr;
org.w3c.dom.DOMImplementation;
org.w3c.dom.Document;
org.w3c.dom.Element;
org.w3c.dom.Text;
```

Gracias a todas estas librerías seremos capaces de crear nuestros propios ficheros XML desde un programa escrito en Java. Veamos el procedimiento:

En primer lugar necesitaremos crear un objeto de tipo DocumentBuilderFactory mediante el método DocumentBuilderFactory.newInstance() y con el objeto resultado vamos a generar un objeto DocumentBuilder usando el método newDocumentBuilder(). Es importante que esto lo introduzcamos dentro de una estructura try...catch, de la siguiente manera.

```
public static void crearFicheroXML(){
    DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
    DocumentBuilder builder;
    try {
        builder = factory.newDocumentBuilder();
        ...
    } catch (ParserConfigurationException | TransformerException ex) {
        System.out.println(ex.getMessage());
```

```
    }  
}
```

Después de esto vamos a necesitar crear un objeto DOMImplementation con el método getDOMImplementation. Gracias a este objeto ya podremos crear el documento XML en si con el método createDocument, al que le pasaremos el nombre de nuestro elemento raíz. Los otros dos parámetros los podemos dejar a null. También le indicaremos la versión de XML que vamos a utilizar, en nuestro caso la 1.0.

```
public static void crearFicheroXML(){  
    DocumentBuilderFactory factory =  
DocumentBuilderFactory.newInstance();  
    DocumentBuilder builder;  
    try {  
        builder = factory.newDocumentBuilder();  
        DOMImplementation implementation =  
builder.getDOMImplementation();  
  
        Document documento = implementation.createDocument(null,  
"elementoRaiz", null);  
        documento.setXmlVersion("1.0");  
        ...  
  
    } catch (ParserConfigurationException | TransformerException ex) {  
        System.out.println(ex.getMessage());  
    }  
}
```

Una vez que tenemos el documento con el nodo raíz, solo tenemos que ir añadiendo los distintos elementos con sus subelementos, atributos y contenido.

Para esto es importante que siempre tengamos en mente la estructura que deseamos construir dentro del documento.

Vamos a realizar un ejemplo sencillo para obtener una estructura como la que se muestra a continuación.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>  
<elementoRaiz>  
    <elementoPadre>
```

```
<elementoHijo1 AtributoHijo1="ValorAtributoHijo1" />
<elementoHijo2>textoElementoHijo2</elementoHijo2>
<elementoHijo3>textoElementoHijo3</elementoHijo3>
</elementoPadre>
</elementoRaiz>
```

Creamos el elemento padre...

```
Element elementoPadre = documento.createElement("elementoPadre");
```

Creamos el elemento hijo 1 y le asignamos el atributo y el valor...

```
Element elemento1 = documento.createElement("elementoHijo1");
elemento1.setAttribute("AtributoHijo1", "ValorAtributoHijo1");
```

Creamos los elementos hijos 2 y 3, sus contenidos y unimos cada uno al que le corresponde...

```
Element elemento2 = documento.createElement("elementoHijo2");
Text textElemento2 = documento.createTextNode("textoElementoHijo2");
elemento2.appendChild(textElemento2);
```

```
Element elemento3 = documento.createElement("elemento3");
Text textElemento3 = documento.createTextNode("textoElementoHijo3");
elemento3.appendChild(textElemento3);
```

Después, añadimos los nodos hijos al nodo padre...

```
elementoPadre.appendChild(elemento1);
elementoPadre.appendChild(elemento2);
elementoPadre.appendChild(elemento3);
```

Añadimos el nodo padre al nodo raíz...

```
documento.getDocumentElement().appendChild(elementoPadre);
```

Finalmente, damos un nombre al nuevo fichero e introducimos el documento XML que hemos generado en dicho fichero.

```
Result result = new StreamResult(new File("elementos.xml"));
Source source = new DOMSource(documento);

Transformer transformer =
TransformerFactory.newInstance().newTransformer();
transformer.transform(source, result);
```

## 4.2. Lectura de ficheros XML con Java

Para leer un fichero XML, y cargarlo en memoria, a través de un programa codificado en Java también necesitaremos hacer uso de varias librerías:

```
java.io.File;
java.io.IOException;
javax.xml.parsers.DocumentBuilder;
javax.xml.parsers.DocumentBuilderFactory;
javax.xml.parsers.ParserConfigurationException;
org.w3c.dom.Document;
org.w3c.dom.Element;
org.w3c.dom.NamedNodeMap;
org.w3c.dom.Node;
org.w3c.dom.NodeList;
org.xml.sax.SAXException;
```

Para leer los documentos XML nos vamos a ayudar de los nombres de las etiquetas de los nodos, y a partir de ahí leeremos sus nodos hijos.

Veamos cómo leeríamos el fichero que hemos creado antes:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<elementoRaiz>
    <elementoPadre>
        <elementoHijo1 AtributoHijo1="ValorAtributoHijo1" />
        <elementoHijo2>textoElementoHijo2</elementoHijo2>
        <elementoHijo3>textoElementoHijo3</elementoHijo3>
    </elementoPadre>
</elementoRaiz>
```

Lo primero que vamos a hacer, al igual que con la escritura es crear los objetos DocumentBuilderFactory y DocumentBuilder.

```
public static void crearFicheroXML(){
```

```

DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
DocumentBuilder builder;
try {
    builder = factory.newDocumentBuilder();
    ...
}

} catch (ParserConfigurationException | SAXException | IOException
ex){
    System.out.println(ex.getMessage());
}
}

```

Después abrimos el fichero xml que vamos a leer...

```
Document documento = builder.parse(new File("elementos.xml"));
```

Buscamos todos los nodos cuya etiqueta sea elementoPadre...

```
NodeList listaPadres = documento.getElementsByTagName("elementoPadre");
```

En nuestro caso solo hay uno, pero al almacenarlo en una lista podemos almacenar un número indeterminado de ellos y luego ir examinando uno a uno.

A continuación, por cada uno de los nodos que nos ha devuelto, vamos a comprobar que es un nodo del tipo Element, y si lo es extraeremos sus hijos.

```

for(int i = 0; i < listaPadres.getLength(); i++){
    Node nodo = listaPadres.item(i);
    if(nodo.getNodeType() == Node.ELEMENT_NODE){
        Element e = (Element) nodo;
        System.out.println("Elemento: " + nodo.getNodeName());
        NodeList hijos = e.getChildNodes();
        ...
    }
}

```

Como sabemos que unos elementos tienen atributos y otros contenido, vamos a preguntar a cada nodo hijo por el número de atributos que tiene. Si tiene más de 0 sabemos que es un elementoHijo1, si no, es de los otros.

```

for (int j = 0; j < hijos.getLength(); j++){
    Node hijo = hijos.item(j);

```

```
if(hijo.getNodeType() == Node.ELEMENT_NODE){  
    NamedNodeMap atributos = hijo.getAttributes();  
    if(atributos.getLength() > 0){  
        Node atributo = atributos.getNamedItem("AtributoHijo1");  
        System.out.println("NodoHijo: " + hijo.getNodeName() + ",  
Atributo: " + atributo.getNodeValue());  
    }  
    else{  
        System.out.println("NodoHijo: " + hijo.getNodeName() + ",  
Valor: " + hijo.getTextContent());  
    }  
}  
}
```

El resultado que nos devolvería por consola al ejecutar este procedimiento de lectura sería el siguiente:

Elemento: elementoPadre

NodoHijo: elementoHijo1, Atributo: ValorAtributoHijo1

NodoHijo: elementoHijo2, Valor: textoElementoHijo2

NodoHijo: elementoHijo3, Valor: textoElementoHijo3

## 5. USO DE CLASES Y MÉTODOS GENÉRICOS

En la unidad ya vimos cómo podemos crear y utilizar nuestras propias clases e interfaces genéricas, con sus métodos correspondientes. Si embargo, dentro de cualquier clase, tanto si está definida con tipos genéricos como si no, podemos implementar métodos con sus propios parámetros genéricos, distintos de los que pueda tener la clase. Dichos métodos se llamarán métodos genéricos.

Veamos un ejemplo...

El siguiente método está preparado para que nos devuelva el número de elementos nulos que hay en un array que se le pasa como parámetro. El tipo T de la tabla es genérico, y se declara en la definición del método, justo antes del tipo devuelto.

```
static <T> int numeroDeNulos(T[] tabla){
    int cont = 0;
    for( T elem : tabla){
        if(elem == null){
            cont++;
        }
    }
    return cont;
}
```

Este método se podría incluir en cualquier clase y depende en nada de los parámetros propios de la misma.

El tipo asociado a un método genérico también puede estar limitado. Por ejemplo, si quisiéramos que nuestro método numeroDeNulos() solo funcionara para arrays de objetos tipos persona o descendientes de la misma pondríamos `<T extends Persona>` en vez de `<T>`.

```
static <T extends Persona> int numeroDeNulos(T[] tabla){
    int cont = 0;
    for( T elem : tabla){
        if(elem == null){
            cont++;
        }
    }
    return cont;
}
```

## 6. OTRAS COLECCIONES: INTERFACES MAP Y SET.

### 6.1. Interfaz Set

La interfaz Set trata los datos como un conjunto matemático, eliminando las repeticiones y sin un orden establecido; aunque, como veremos, una de sus implementaciones permite introducir un orden. Todos sus métodos son herencia de Collection. Lo único que añade es la restricción de no permitir duplicados. Esto significa que si intentamos insertar un elemento que ya existe, no lo hará.

Los métodos más utilizados ya los conocemos, puesto que también son utilizados por la interfaz List. Esto es debido a que son métodos heredados de Collection:

- int size()
- boolean isEmpty()
- boolean contains(Object element)
- boolean add(E element)
- boolean remove(E element)
- boolean addAll(Collection <? Extends E>c)
- void clear()

Para conocer todos los métodos de los que dispone esta interfaz podéis consultarlos en el siguiente link de la ayuda oficial:

<https://docs.oracle.com/javase/8/docs/api/java/util/Set.html>

Para recorrer un conjunto se recomienda el uso de foreach.

Las diferencias más importantes son el orden en que se van insertando los elementos nuevos y que un elemento que ya está en el conjunto no se puede volver a insertar, ya que no son posibles los elementos repetidos. Cuando intentamos insertar un elemento repetido con un método add() o con addAll(), no se producirá ningún error ni se arrojará ninguna excepción; sencillamente, el elemento no se inserta y, como el conjunto no habrá cambiado, el método devuelve false.

Sin embargo, no tendremos los métodos propios de listas, que vienen declarados en la interfaz List. En particular, no es posible el acceso posicional por medio de índices, aunque sí las iteraciones.

Las implementaciones de Set son las clases: HashSet, TreeSet y LinkedHashSet.

- **HashSet**: tiene un buen rendimiento, aunque no garantiza ningún orden en la inserción.
- **TreeSet**: a pesar de tener peor rendimiento, garantiza el orden basado en el valor de los elementos insertados. El criterio de ordenación por defecto es el natural (el proporcionado por el método `compareTo()` de la clase genérica E) o bien se especifica por medio de un comparador pasado como parámetro al constructor.
- **LinkedHashSet**: inserta los elementos al final, con lo cual se garantiza un orden basado en la inserción.

Al contrario de lo que pasa con las listas, las tres implementaciones de Set tienen diferencias en su comportamiento. Es muy común utilizar variables de tipo Set para referenciarlos. Esto permite aprovechar el polimorfismo de las distintas implementaciones y hacer transformaciones de unas en otras

Veamos un ejemplo de TreeSet, en el que la clase cliente tiene un método `compareTo()` basado en el atributo ID:

```
TreeSet<Cliente> conjuntoClientes = new TreeSet<>();  
  
conjuntoClientes.add(new Cliente("111", "Marta", "2004"));  
conjuntoClientes.add(new Cliente("115", "Jorge", "2003"));  
conjuntoClientes.add(new Cliente("112", "Carlos", "2002"));  
System.out.println(conjuntoClientes);
```

Veremos por pantalla:

```
[ID: 111 Nombre: Marta Edad: 20,  
 ID: 112 Nombre: Carlos Edad: 18,  
 ID: 115 Nombre: Jorge Edad: 21  
]
```

Observamos que el orden en el que aparecen no coincide con el orden de inserción, sino que están ordenados por ID creciente.

Si ahora intentamos volver a insertar uno de los elementos anteriores, por ejemplo, el de Marta...

```
boolean insertado = conjuntoClientes.add(new Cliente("111", "Marta",  
"2004));  
System.out.println(insertado);  
System.out.println(conjuntoClientes);
```

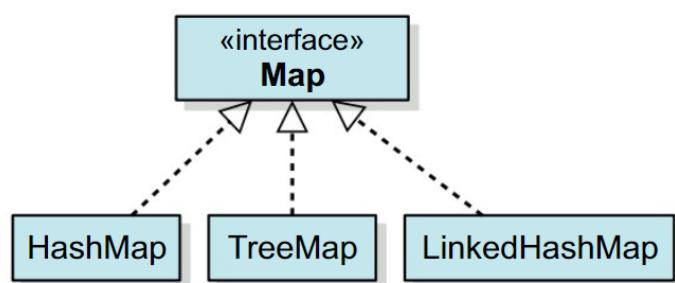
Aparece por pantalla...

```
false  
[ID: 111 Nombre: Marta Edad: 20,  
 ID: 112 Nombre: Carlos Edad: 18,  
 ID: 115 Nombre: Jorge Edad: 21  
]
```

donde vemos que la inserción ha devuelto un false (no ha insertado) y que el conjunto no ha cambiado. Pero si queremos otro conjunto de clientes ordenados por nombre, lo creamos pasando a su constructor, como argumento, un comparador basado en el atributo nombre.

## 6.2. Interfaz Map

Los mapas o diccionarios son estructuras cuyos elementos, a los que llamaremos entradas (objetos del tipo Map.Entry), son pares clave/valor en vez de valores individuales como en las colecciones. Todas ellas implementan la interfaz Map, que no hereda de Collection. Por tanto, los mapas no son colecciones, aunque están íntimamente relacionados con ellas y funcionan dentro del mismo entorno de trabajo. Vamos a conocer tres implementaciones de Map: HashMap, TreeMap y LinkedHashMap, se diferencian entre sí de forma similar a HashSet, TreeSet y LinkedSet.



En un mapa se insertan entradas que constan de una clave, que no se puede repetir, y un valor asociado con ella, que sí puede estar repetido.

Las operaciones fundamentales en un mapa son la inserción, la lectura y la eliminación de entradas.

Como ejemplo del uso de mapas vamos a utilizar la implementación `HashMap`, que no garantiza ningún orden de inserción de las entradas, aunque es muy eficiente en cuanto a la velocidad de acceso a los datos. El constructor que vamos a utilizar es el que se define de la siguiente manera:

```
Map<K, V> m = new HashMap<>();
```

Donde K es el tipo de la clave y V el de los valores. Son tipos genéricos que, necesariamente, serán clases o interfaces y no tipos primitivos. Por tanto, si queremos crear una instancia de un `HashMap` con una clave de tipo `String` y valores de tipo `Double`, lo haremos de la siguiente manera:

```
Map<String, Double> m = new HashMap<>();
```

Para insertar elementos en este mapa vamos a utilizar el método

```
V put(K clave, V valor)
```

En el caso de que no hubiese ningún valor asociado a esa clave anteriormente, se insertará el par y el método devolverá null. En el caso de que ya existiese la clave, se modificará el valor por el nuevo y se devolverá el antiguo.

```
m.put("Ana", 1.65);
m.put("Marta", 1.60);
m.put("Luis", 1.73);
m.put("Pedro", 1.69);
System.out.println(m);
```

Esto devolverá por pantalla:

```
{Marta=1.6, Ana=1.65, Luis=1.73, Pedro=1.69}
```

Si queremos cambiar la estatura de Pedro, insertamos otra vez una entrada con la misma clave y el nuevo valor.

```
m.put("Pedro", 1.71);
System.out.println(m);
```

Y obtenemos...

```
{Marta=1.6, Ana=1.65, Luis=1.73, Pedro=1.71}
```

Para eliminar una entrada del mapa disponemos de

```
V remove (Object k)
```

que elimina la entrada cuya clave es k, si existe. En este caso, devuelve el valor asociado a la clave. En caso contrario, devuelve null.

```
m.remove("Pedro");
System.out.println(m);
```

Devolvería

```
{Marta=1.6, Ana=1.65, Luis=1.73}
```

Para eliminar todas las entradas utilizamos clear().

```
m.clear();
```

Para obtener el valor de una entrada utilizamos, get que devuelve el valor asociado a la clave k o null si no hay ninguna entrada con esa clave.

```
V get (Object k);
```

```
m.get("Ana") // 1.65
```

Finalmente, si queremos saber si una clave se encuentra en el mapa utilizaremos containsKey que nos devolverá true o false en función de si dentro del mapa hay una entrada con la clave k.

```
boolean containsKey(Object k);
```

```
if(containsKey("Ana")){
    System.out.println(m.get("Ana"));
}
// Escribirá por pantalla 1.65
```

Para conocer más operaciones que podemos realizar con los Maps, puedes consultar el siguiente link de la ayuda oficial:

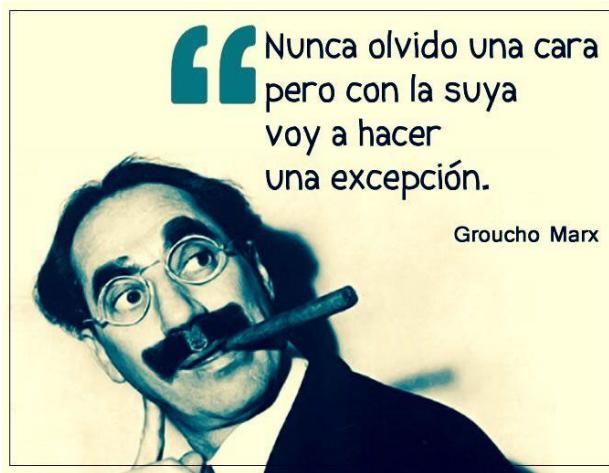
<https://docs.oracle.com/javase/8/docs/api/java/util/Map.html>

# **UT-09: CONTROL Y MANEJO DE EXCEPCIONES**

¿Qué vamos a ver?

- Excepciones. Concepto y Jerarquías de excepciones.
- Manejo de excepciones:
  - Captura de excepciones.
  - Propagar excepciones.
  - Lanzar excepciones.
- Crear excepciones.
- Aserciones

## 1. EXCEPCIONES. CONCEPTO Y JERARQUÍA DE EXCEPCIONES.



Los errores en los programas son prácticamente inevitables, tanto si están originados por códigos deficientes, entrada de datos, parámetros incorrectos o archivos inexistentes...

En la mayoría de los lenguajes de programación, la manipulación de los errores suele ser complicada y confusa. Su código se mezcla con el del resto del programa, haciéndolo poco claro, y suele estar destinado solo a evitar el error y no a controlar la situación una vez que dicho error se ha producido.

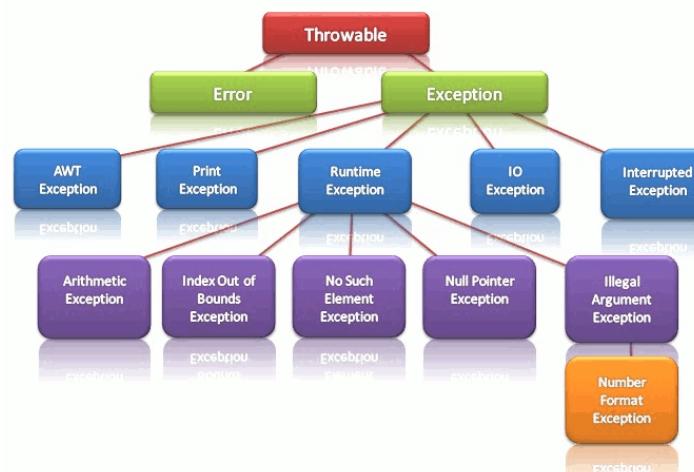
Java es un lenguaje que se enfoca en tratar de evitar, en la medida de lo posible, que el programa se interrumpa a causa de un error. Para ello no basta con evitar que se produzcan los errores, sino que hay que implementar los medios para que un programa se recupere de las condiciones generadas por un error que no se ha podido evitar. Esto depende del tipo de error, y no siempre es posible.

Cuando, en la ejecución de un programa, se produce una situación anormal, es decir, un error, que interrumpe el flujo normal de ejecución, el método que se esté ejecutando genera un objeto de la clase **Throwable** ("arrojable"), que contiene información del error, de su causa y del contexto del programa en el momento

en que se produce, y lo entrega (“lo arroja”) al sistema de tiempo de ejecución (la máquina virtual). Este objeto es susceptible de ser capturado por el programa y analizarlo para dar una respuesta, si procede. En caso contrario, el programa se interrumpe y el sistema de tiempo de ejecución muestra una serie de mensajes que describen el error, como ocurre en otros lenguajes de programación.

Hay errores lo bastante graves para que sea preferible que el programa se interrumpa. Por ejemplo, errores derivados de problemas de hardware. Si intentamos leer de un disco defectuoso, se producirán errores de los que es difícil, si no imposible, recuperarse. Estos y otros errores relacionados directamente con la máquina virtual, y no con el código de nuestro programa, arrojan objetos de la clase `Error`, una subclase de `Throwable`. De ellos no nos vamos a ocupar, ya que poco se puede hacer con estos errores a la hora de programar. Tampoco nos ocuparemos de las excepciones llamadas de tiempo de ejecución (`runtime exceptions`), que proceden de líneas de código equivocadas. La solución a estos errores es corregir el código.

Pero hay otras clases de errores más habituales y menos graves, como entradas de datos de tipos equivocados, aperturas de ficheros con ruta de acceso errónea u operaciones aritméticas no permitidas. A estos errores se les llama **excepciones**, y producen un objeto de la clase `Exception`, otra subclase de `Throwable`. A continuación, explicaremos estas excepciones, ya que son manipulables a través del código. Para ello se usan los bloques `try`, `catch` y `finally`.



Podemos agrupar las excepciones en dos grupos:

- **Excepciones comprobadas**
- **Excepciones no comprobadas**

Entre las **excepciones comprobadas** se encuentran las más comunes, generalmente con un origen fuera del programa (entradas de datos, nombres de ficheros incorrectos, etc). El compilador no exige su tratamiento, por tanto, no tenemos que preocuparnos por saber cuáles son exactamente, aunque con la práctica acabaremos familiarizándonos con las más importantes: **IOException**, **FileNotFoundException**, **NumberFormatException**, ...

Con respecto a las **excepciones no comprobadas**, como **ArithmaticException**, producidas por errores aritméticos, o **ArrayIndexOutOfBoundsException**, que se produce cuando intentamos salirnos de los límites de un array. Dichas excepciones suelen estar asociadas a malas prácticas de programación. Por tanto, más que tratarlas con una estructura **try-catch**, hay que corregir el código para evitar que se produzcan.

## 2. MANEJO DE EXCEPCIONES

### 2.1. Captura de excepciones

Cuando sabemos que en un determinado fragmento de código se puede producir una excepción, lo encerramos dentro de un bloque try. Por ejemplo, si en una división entre las variables a y b sospechamos que divisor b podría hacerse cero en algún momento escribimos:

```
try{
    int c;
    c = a / b;
}
```

Con esta estructura estamos sometiendo a observación al bloque de código cerrado entre llaves. Si salta una excepción en ese bloque, deberá ser capturada por un bloque catch de la siguiente forma:

```
try{
    int c;
    c = a / b;
} catch(ArithmetricException e){
    System.out.println("Error: división por 0");
}
```

Si se produce una excepción y es capturada, se interrumpe la ejecución del código del bloque **try**, saltando a la primera línea del bloque **catch**. Cuando se termina de ejecutar dicho bloque, continúa en la línea inmediatamente posterior a la estructura **try-catch**.

La palabra reservada **catch** va seguida de unos paréntesis donde se encierra un parámetro de la clase de excepción que puede atrapar; en este caso, una excepción aritmética. El parámetro **e** se puede usar como variable local dentro del bloque. Hace referencia al objeto de la excepción y contiene toda la información sobre el error que la ha producido. Entre sus métodos está **getMessage()**, que nos muestra un mensaje descriptivo del error.

Podríamos haber puesto

```
try{
    int c;
    c = a / b;
} catch(ArithmetricException e){
    System.out.println(e.getMessage());
}
```

O incluso

```
try{
    int c;
    c = a / b;
} catch(ArithmetricException e){
```

```
        System.out.println(e);
    }
```

Ya que hace una llamada a `toString()` de la clase de la variable `e`, donde también se describe el error (en inglés).

En el bloque `catch` solo se recogerán las excepciones del tipo declarado entre paréntesis o de una subclase. En el ejemplo anterior podríamos haber escrito lo siguiente:

```
try{
    int c;
    c = a / b;
} catch(Exception e){
    System.out.println(e);
}
```

Ya que **AritmeticException** es una subclase de **Exception**.

Esto nos permitirá recoger otros tipos de excepción en el mismo `catch`, pero, para ese caso, es mejor escribir más de un bloque `catch`. Con un mismo bloque `try` se pueden poner tantos bloques `catch` como deseemos, siempre que vayan seguidos,

```
try{
    //bloque try
} catch (tipoExcepcional1 nombreParametro1){
    //bloque catch
} catch (tipoExcepcional2 nombreParametro2){

}...
```

Cuando se produce una excepción en el bloque `try`, se compara con el tipo del primer bloque `catch`. Si coincide con él o con una subclase, se ejecuta dicho bloque y continúa el programa después del último bloque `catch`. Si no coincide, se compara con el tipo del segundo bloque, y así sucesivamente hasta que se encuentra un bloque cuyo parámetro coincide o sea una superclase de la

excepción producida, de forma que solo se ejecuta un bloque catch, el primero cuyo tipo sea compatible. Si la excepción no coincide con ninguno de los parámetros de los bloques, no es capturada y se interrumpe la ejecución del programa. Aquí hay que tener cuidado de no poner un bloque catch con una excepción que sea superclase de otra que vaya más abajo, pues el bloque de esta última nunca se ejecutará.

Por ejemplo:

```
try{
    int c;
    c = a / b;
} catch (Exception e){
    System.out.println("Estoy en el primer catch");
} catch(ArithmetricException e) {
    System.out.println("Estoy en el segundo catch");
}
```

Si b vale cero, se producirá una excepción de tipo ArithmetricException, pero, al ser un subtipo de Exception, será capturada en el primer catch, cuyo bloque será el que se ejecute, apareciendo el mensaje “Estoy en el primer catch”. La ejecución seguirá después del último bloque, de modo que el segundo bloque catch es inútil, ya que jamás se va a ejecutar. De hecho, en este ejemplo, como todo tipo de excepción es subclase de Exception, cualquier excepción que se produzca en el bloque try será capturada en el primer bloque catch y ningún bloque que pongamos después se va a ejecutar nunca.

Por otra parte, existe la posibilidad de capturar más de un tipo de excepción con un único bloque catch.

```
catch (tipoExpcion1 | tipoExpcion2 | ... e){
    ...
}
```

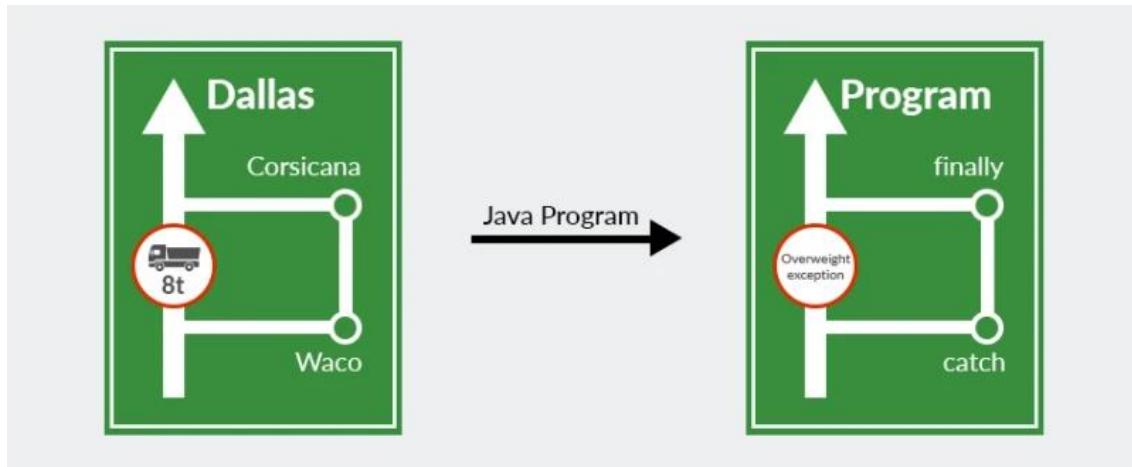
Aquí, la barra vertical equivale al or lógico. Se pueden añadir tantos tipos de excepciones como se quiera, separados por barras verticales. El significado es: “si se arroja una excepción del tipo tipoExpcion1 o del tipo tipoExpcion2 o ..., ejecutar este bloque catch, donde la excepción será referenciada con la variable e”.

Una estructura try-catch supone una bifurcación en el programa. Pero a menudo estamos interesados en que una serie de líneas de código se ejecuten, tanto si se produce una excepción como si no; por ejemplo, para cerrar un archivo en el que estábamos escribiendo.

Para esto se define un bloque opcional **finally** que, cuando está presente, se coloca al final una estructura try-catch (también es posible una estructura **try-finally**, sin bloque catch) y se ejecuta independientemente de si en el bloque try se ha lanzado una excepción o no, y de si la excepción ha sido capturada o no. Se ejecutará antes incluso que cualquier sentencia **return** que aparezca en los bloques **try** o **catch**. Esto nos asegura que, en circunstancias anómalas, se van a ejecutar determinadas tareas, como cerrar ficheros abiertos, antes de que termine la ejecución del programa. En el siguiente trozo de código:

```
try{
    //bloque que trabaja con archivos
} catch (IOException e){
    //bloque si salta excepción
} finally {
    //código para cerrar archivos
}
...
return;
```

el bloque **finally** se ejecuta incluso antes de ejecutarse el **return** del bloque **try**, a pesar de que figura después de dicha sentencia, tanto si se produce una excepción como si no.



## 2.2. Propagación de excepciones

Todo el código en Java forma parte de algún método que, en última instancia, puede ser el método `main()`. Si desde un método `metodo2()` se invoca a un método `metodo1()` y salta una excepción durante la ejecución de este último, podemos capturarla y manipularla, como hemos hecho hasta ahora, con una estructura `try-catch` en el propio código de `metodo1()`. Pero hay un enfoque alternativo. Podemos declarar, en la definición de `metodo1()`, que en su ejecución puede producirse dicha excepción. Para ello usaremos la palabra clave **throws**.

```
tipo metodo1(tipo1 parametro, ...) throws tipoExcepcion{
    ...
}
```

En el cuerpo de `metodo1()` ya no tenemos que insertar los bloques `try-catch` para ese tipo de excepción. En cambio, `metodo2()` será el encargado de gestionarla cuando invoque a `metodo1()`.

```
tipo metodo2(tipo1 parametro, ...){
    ...
    try{
        metodo1();
    } catch (tipoExcepcion e) {
        //tratamiento de la excepción
    }
}
```

Veámoslo con un ejemplo. Supongamos que en método metodo1() se puede dar una división por cero. Por otra parte, un segundo método metodo2() llama a metodo1(). Lo que hemos hecho hasta ahora es:

```
void metodo1(int a, int b){  
    int c;  
    try {  
        c = a/b;  
    } catch(ArithmetricException e) {  
        System.out.println("División por 0");  
    }  
    System.out.println(a + "/" + b " = " + c);  
}  
  
void metodo2(){  
    int x, y;  
    metodo1(x,y);  
}
```

Si la variable y es cero, la excepción producida es capturada y mantenida en el lugar donde se ha intentado la división (en metodo1()) antes de que la ejecución vuelva al método que lo llama (metodo2()).

Pero podríamos hacerlo de otra forma. Primero eliminamos el bloque try-catch de metodo1() y declaramos a este último como susceptible de producir una ArithmetricException. Esto se implementa por medio de la palabra clave throws en su encabezamiento.

```
void metodo1(int a, int b) throws ArithmetricException{  
    int c;  
    c = a/b;  
    System.out.println(a + "/" + b " = " + c);  
}
```

Con esto estamos declarando que, dentro del método, puede producirse una excepción aritmética y que deberá ser manipulada por código externo, en el método que llame a metodo1(), en nuestro caso metodo2(). Además, esta particularidad, que forma parte de la definición de metodo1(), deberá constar en la documentación que la acompaña para que los usuarios del método sepan a

qué atenerse. La implementación de metodo2(), por tanto, deberá hacerse cargo de la excepción.

```
void metodo2(){
    int x, y;
    try{
        metodo1(x,y);
    } catch(ArithmetricException e) {
        System.out.println("División por cero");
    }
}
```

Por supuesto, metodo2() podría propagar la excepción a un tercer método que lo invoque, y así sucesivamente.

### 2.3. Lanzar excepciones

En Java, puedes lanzar una excepción usando la palabra clave "throw" seguida de una instancia de la excepción que deseamos lanzar. Hay varias formas de crear una instancia de una excepción en Java, pero la forma más común es crear una instancia de la excepción con el constructor de la excepción y pasar un mensaje de error como parámetro.

Además, debemos incluir en la declaración del método la palabra clave “throws” seguida del tipo de excepción que vamos a lanzar.

```
public class Operaciones {

    public static double dividir throws ArithmetricException(double numerado, double denominador) {
        if (denominador == 0) {
            throw new ArithmetricException("El denominador no puede ser cero.");
        }
        double resultado = numerado / denominador;
        return resultado;
    }
}
```

En este ejemplo, si el denominador es cero, se lanza una excepción `ArithmaticException` con el mensaje “El denominador no puede ser cero.”. El programa se detendrá y no se imprimirá el resultado.

### 3. CREAR EXCEPCIONES

Hasta ahora, todas las excepciones que hemos visto vienen predefinidas en Java. Pero podemos implementar las nuestras propias con tan solo heredar de alguna que ya exista. Además, es posible lanzarlas cuando nos interese por medio de la palabra reservada **throw**.

Por ejemplo, si estamos introduciendo por teclado el valor entero que representa una edad, no tiene sentido un número negativo. Normalmente, en estos casos nos conformamos con un condicional y un simple mensaje de error. Pero también podemos crear con **new** y lanzar un **throw** una excepción definida por nosotros.

En la definición de una excepción de usuario no necesitamos implementar ningún método; basta con heredar de `Exception`. En todo caso, podemos sustituir el método `toString()` por uno que represente mejor nuestro caso.

```
public class ExpcionEdadNegativa extends Exception{
    public String toString(){
        return "Edad negativa";
    }
}
```

Veamos un ejemplo donde se usa esta excepción:

```
try{
    System.out.print("Introducir edad: ");
    int edad = new Scanner (System.in).nextInt();
    if (edad < 0){
        throw new ExpcionEdadNegativa();
    }
    else {
        System.out.println("edad correcta: " + edad)
    }
} catch (ExpcionEdadNegativa ex){
    e.printStackTrace();
}
```

A la hora de hacer nuestras excepciones, hay que conocer algunos métodos de la clase **Exception**, por si queremos sobreescribirlos:

- **Exception()**: es el constructor por defecto.
- **Exception(String message)**: constructor al que se le pasa un mensaje de error.
- **toString()**: método de la clase **Object** redefinido por **Throwable** para que muestre el nombre de la clase y si tiene mensaje, lo mostrará también.
- **getMessage()**: método heredado de la clase **Throwable** que nos devuelve una cadena que contiene el mensaje con el que fue creada la excepción.
- **printStackTrace()**: método heredado de la clase **Throwable** que directamente imprime a la salida estándar de errores el objeto de la clase desde el que se invoca junto con la traza de llamadas a métodos desde que se ha producido la excepción. Resulta bastante útil para depurar programas y así encontrar el lugar exacto del fallo.

## 4. ASERCIÓNES

Una aserción es una condición lógica insertada en el código Java, de ideas o condiciones que se asumen que son ciertas. El sistema se encarga de comprobarlas y avisar mediante una excepción en caso de que no se cumplan.

Generalmente se suele utilizar para verificar valores de las variables en cierto punto del programa.

Se aconseja su uso durante el desarrollo y las pruebas, y se pueden eliminar de la implantación del sistema. Concretamente en estas situaciones:

- Usarse en la entrada de métodos privados.
- Usarse en la salida de métodos públicos o privados.
- Usarse para verificar cómo se supone que están las variables y estructuras de datos internas.
- Usarse en la sentencia **default** de la estructura **switch** cuando todos los casos correctos están explícitos (es decir, cuando no debería ocurrir).

- Usarse en el último **else** de construcciones **if ... else if ... switch** cuando todos los caos correctos están explícitos (es decir, cuando el último **else** no debería ejecutarse nunca).
- Usarse en bucles largos.

No se recomienda su uso en estas situaciones:

- Usarse para detectar errores en los datos de entrada de los programas.
- Usarse en la entrada de métodos públicos.

Para incluir aserciones a nuestros códigos podemos utilizar dos tipos de sintaxis.

La primera sintaxis sería la siguiente:

```
assert expresion;
```

Por ejemplo:

```
double m = 100.0;
double n = m/2;
assert m == n*n; //condición que asumimos como true
```

La segunda opción de sintaxis es la siguiente:

```
assert expresion1:expresion2;
```

Por ejemplo:

```
double m = 100.0;
double n = m/2;
assert m == n*n : "Java no sabe dividir " + m + " entre 2: " + n;
```

Si te das cuenta, la sintaxis es similar a la del operador ternario, el cuál ya vimos en su momento. La expresión entre “assert” y “:” se evalúa y en caso de que no se cumpla se ejecutará la expresión que va detrás de “:”.

Para entenderlo mejor, veamos una comparación de dos códigos que harán lo mismo:

```
if(!estado_ideal){  
    throw new Error("Fallo en esta zona del programa");  
}
```

Es equivalente a:

```
assert estado_ideal : "Fallo en esta zona del programa";
```

Si queremos que nos salte una excepción en el caso de que el siguiente for se ejecute completamente podríamos hacer los siguiente:

```
for(int i = 0; i < 10; i++){  
    if(array[i] > 1000){  
        return i;  
    }  
}  
assert false;
```

En el caso de que el bucle termine sin haber ejecutado el return, se evaluará la expresión de assert (en este caso false) por lo que lanzará una excepción (AssertionError).

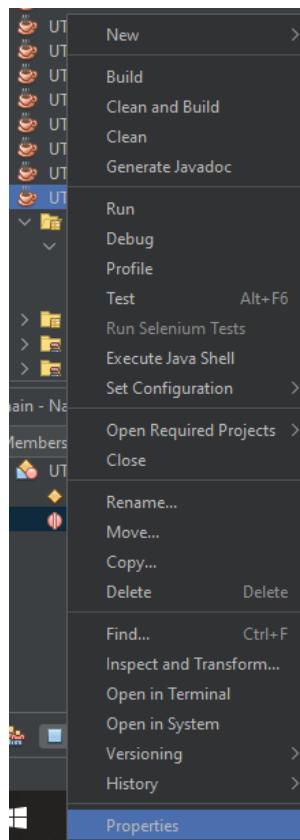
Otro de los casos que nos recomiendan es controlar cuando el default de un switch nunca debería ocurrir, por ejemplo:

```
switch(op){  
    case 1: sumar();  
              break;  
    case 2: restar();  
              break;  
    case 3: multiplicar();  
              break;  
    case 4: dividir();  
              break;  
    default: assert false;  
}
```

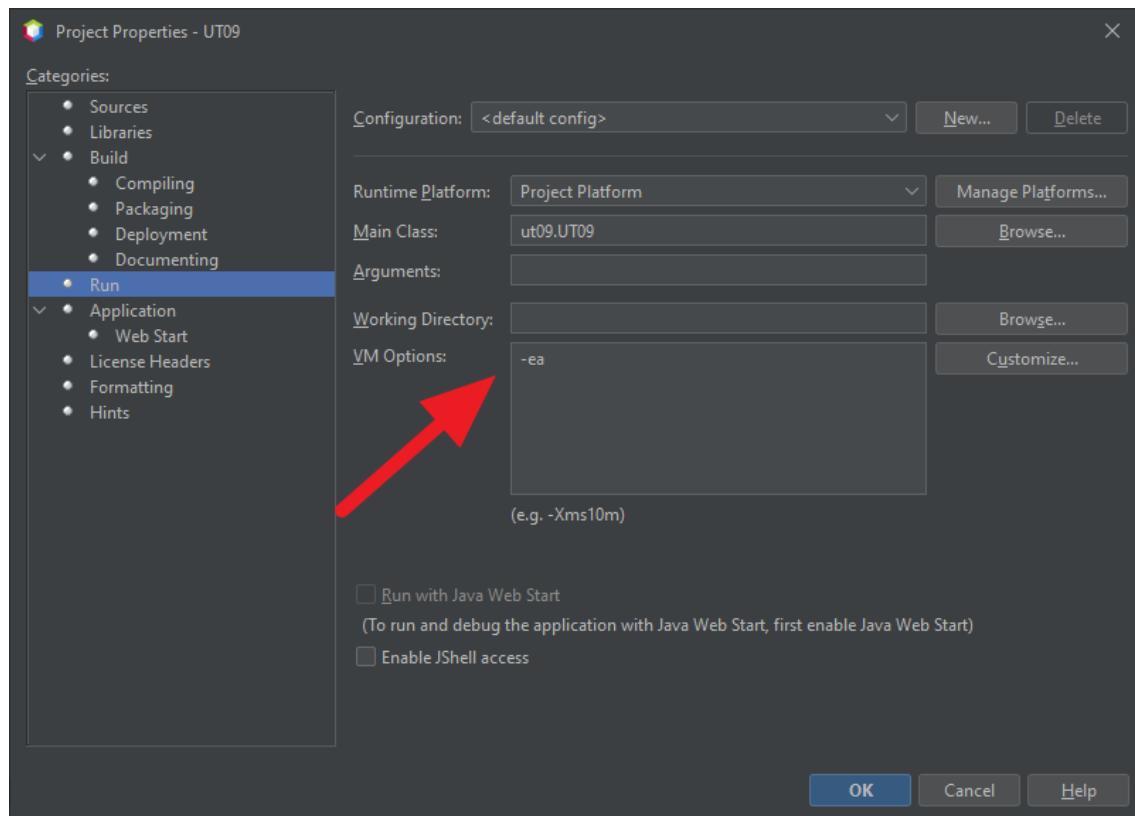
En el caso de no recibir un 1, 2, 3 o 4 nos devolverá una excepción.

#### 4.1. Activar y desactivar las aserciones

Las aserciones se encuentran desactivadas por defecto, por tanto, para poder utilizarlas vamos a necesitar activarlas. Solo tendremos que pulsar botón derecho del ratón sobre el proyecto y seleccionar “properties”...



Después iremos a la sección Run y en el cuadro de texto VM Options escribiremos “-ea”.



Cuando queramos desactivar las aserciones solo tendremos que volver a quitar el “-ea”.

# **UT-10: ESTRUCTURAS EXTERNAS DE DATOS**

¿Qué vamos a ver?

- Ficheros. Concepto de “persistencia” de los datos de una aplicación.
- Clasificación de ficheros:
  - Contenido: ficheros de texto y ficheros binarios.
  - Modo de acceso: ficheros secuenciales y directos.
- Apertura y cierre de ficheros.
- Lectura y escritura de información en ficheros.
- Almacenamiento de objetos en ficheros. Persistencia. Serialización.

## 1. FICHEROS. CONCEPTO DE “PERSISTENCIA” DE LOS DATOS DE UNA APLICACIÓN

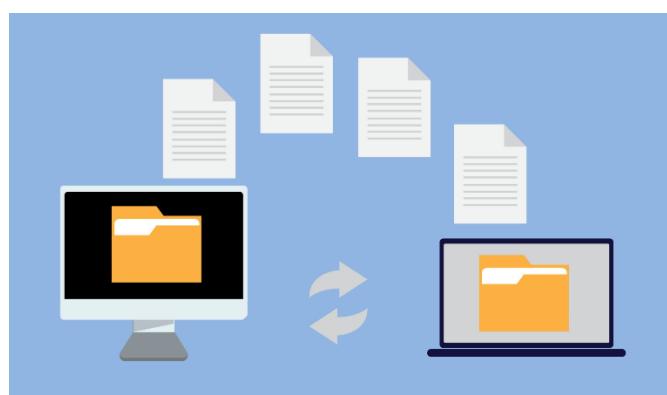


Como ya vimos en la unidad anterior, almacenar la información en memoria, como hacíamos antes de conocer los ficheros XML, tiene el problema de que al finalizar el programa toda esa información se pierde.

Sin embargo, gracias los ficheros podremos almacenar la información que necesitemos, de forma permanente, con la posibilidad de recuperarla más adelante. Al concepto de almacenar la información para luego poder recuperarla de nuevo se le denomina “persistencia de datos”, y podremos aplicarla tanto al contexto de ficheros como al de bases de datos.

Un **fichero** es una colección de datos organizados que se almacenan en un soporte externo de almacenamiento, como un disco duro, disco compacto, etc. Estos soportes no se borran cuando finaliza la ejecución del programa, ni siquiera cuando se apaga el ordenador.

Los ficheros no sólo nos serán útiles a la hora de almacenar la información, sino que también será muy útiles a la hora de transferir información de un dispositivo a otro.



## 2. CLASIFICACIÓN DE FICHEROS

Es posible realizar distintas clasificaciones de ficheros en función de determinadas características. En nuestro caso nos vamos a centrar en las clasificaciones por el tipo de contenido y por el modo de acceso.

### 2.1. Contenido

Principalmente podemos distinguir dos tipos de ficheros en función de su contenido:

#### *2.1.1. Ficheros de texto*

Los datos se guardan en los ficheros en forma de texto, es decir, como secuencias de caracteres. El contenido de este tipo de ficheros puede ser leído por una persona.

#### *2.1.2. Ficheros binarios*

Los datos son guardados en su representación binaria, es decir, en secuencias de bytes. El contenido de este tipo de ficheros no puede ser leído por una persona.

### 2.2. Modo de acceso

Según el modo de acceso a los datos, los ficheros se pueden clasificar en:

#### *2.2.1. Ficheros secuenciales*

Se trata de archivos en los que el contenido se lee o escribe de forma continua. No se accede a un punto concreto del archivo. Para leer cualquier información necesitamos leer todos los datos desde el principio del fichero hasta llegar a la información deseada. En general los archivos de texto se suelen utilizar de forma secuencial.

### 2.2.2. Fichero de acceso directo

Se puede acceder a cualquier dato del archivo conociendo la posición en el mismo. Dicha posición se suele indicar en bytes.

## 3. APERTURA Y CIERRE DE FICHEROS

Para manejar ficheros vamos a hacer uso de la clase `java.io.File`.

### 3.1. Creación de ficheros

Para crear un fichero usaremos el constructor de la clase `File`, proporcionándole el nombre del fichero que queremos crear, y después ejecutaremos el método `createNewFile()`.

```
public File(String pathname)
```

```
public boolean createNewFile() throws IOException
```

Como podemos observar, el método `createNewFile` propaga excepciones `IOException`, por tanto, cuando trabajemos con ficheros en Java, siempre vamos a necesitar encapsularlo dentro de un `try...catch`

Ejemplo:

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

Esto nos creará el fichero “archivo.txt” en la misma ruta que se encuentra el proyecto.

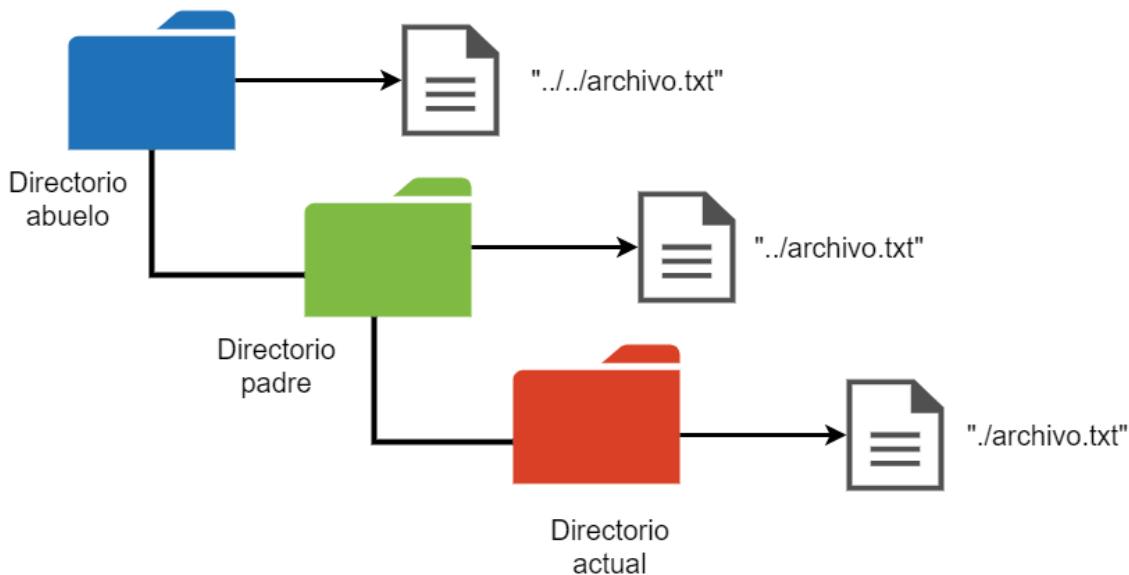
En el caso de queramos crear el fichero en una ruta específica, debemos dar la ruta absoluta o la ruta relativa del mismo. Por supuesto, la ruta de directorios deberá existir, de lo contrario nos dará error.

Ejemplo con ruta relativa:

```
File archivo01 = new File("./ruta01/archivo01.txt");
archivo01.createNewFile();
```

Cuando utilicemos rutas relativas debemos tener en cuenta varias cosas:

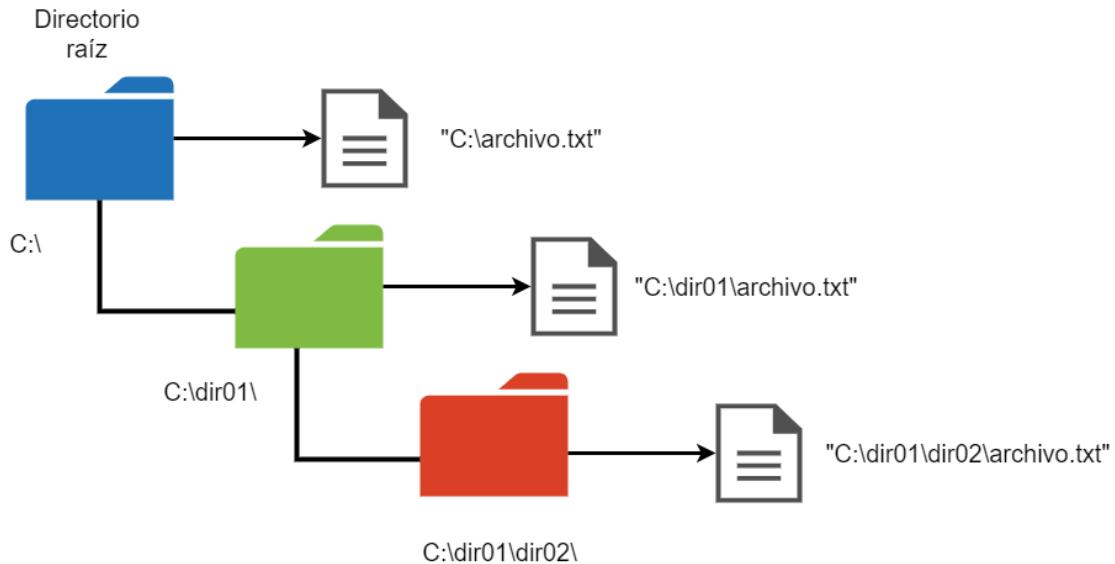
- En las rutas relativas se toma como referencia el directorio actual, es decir, el del proyecto.
- Si utilizamos "./" al comienzo de la ruta estamos indicando que comience desde el directorio actual.
- Si utilizamos "../" al comienzo de la ruta estamos indicando que comience desde el directorio padre del directorio actual. Podremos unir varios "../" para subir niveles de directorios, por ejemplo, con "../../" nos posicionaremos en directorio padre del directorio padre del directorio actual.
- Para separar niveles de directorios utilizamos el carácter "/", por ejemplo, "../../ruta03/archivo.txt"



Ejemplo con ruta absoluta:

```
File archivo02 = new File("c:\\\\ruta02\\\\archivo02.txt");
archivo02.createNewFile();
```

Cuando utilicemos rutas absolutas debemos escribir toda la ruta completa, desde el directorio raíz. Para dividir los niveles de directorios debemos utilizar “\\”.



El código completo quedaría de la siguiente manera:

```
public static void main(String[] args) {
    try {
        File archivo01 = new File("./ruta01/archivo01.txt");
        if(archivo01.createNewFile())
            System.out.println("El fichero se creó");
        else
            System.out.println("El fichero no se creó");
    } catch (Exception e) {
        System.out.println(e.toString());
    }
}
```

### 3.2. Creación de directorios

Para crear directorios utilizaremos los métodos mkdir() y mkdirs().

```
public boolean mkdir()
```

```
public boolean mkdirs()
```

La diferencia entre mkdir y mkdirs consiste en que mkdirs creará también los directorios padre si no existen, mientras que mkdir solo crea el directorio final y si los directorios padres no existen no lo creará.

A la hora de crear los directorios debemos tener en cuenta los mismos factores que con la creación de ficheros, ya sean con ruta relativa o con ruta absoluta.

Veamos un ejemplo:

```
public static void main(String[] args) {
    try {
        File directorio01 = new File("./directorio01");
        if(directorio01.mkdir())
            System.out.println("Se ha creado el directorio01");
        else
            System.out.println("No se ha creado el directorio01");

        File directorio02 = new
File("C:\\\\directorioPadre\\\\directorio02");
        if(directorio02.mkdirs())
            System.out.println("Se ha creado el directorio02");
        else
            System.out.println("No se ha creado el directorio02");

    } catch (Exception e) {
        System.out.println(e.toString());
    }
}
```

### 3.3. Eliminación de ficheros y directorios

Si lo que necesitamos es eliminar ficheros directorios haremos uso del método “delete()”.

```
public boolean delete()
```

Previamente crearemos un objeto de la clase File con un archivo o directorio existente y después ejecutaremos delete().

Cuando eliminemos un directorio, también eliminaremos todo su contenido.

```
public static void main(String[] args) {
    try {

        File archivo02 = new File("c:\\\\ruta02\\\\archivo02.txt");
        if(archivo02.delete())
            System.out.println("archivo02 fue eliminado");
        else
            System.out.println("archivo02 fue eliminado");

        File ruta01 = new File("./ruta01/");
        if(ruta01.delete())
            System.out.println("El directorio rut01 fue eliminado");
        else
            System.out.println("El directorio rut01 fue eliminado");

    } catch (Exception e) {
        System.out.println(e.toString());
    }
}
```

### 3.4. Renombrar o mover ficheros y directorios

Para renombrar o mover ficheros y directorios vamos a utilizar el mismo método:

```
public boolean renameTo(File dest)
```

En el parámetro introduciremos un objeto File con la nueva ruta y/o nombre.

```
public static void main(String[] args) {  
    try {  
  
        File archivo01 = new File("./ruta01/archivo01.txt");  
        File archivo01renamed = new  
File("./ruta01/archivo01renamed.txt");  
        if(archivo01.renameTo(archivo01renamed))  
            System.out.println("El fichero se renombró");  
        else  
            System.out.println("El fichero no se renombró");  
  
        File archivo02 = new File("./ruta01/archivo02.txt");  
        archivo02.createNewFile();  
        File archivo02moved = new File("./archivo02.txt");  
        if(archivo02.renameTo(archivo02moved))  
            System.out.println("El fichero se movió");  
        else  
            System.out.println("El fichero se movió");  
  
        File ruta01 = new File("./ruta01/");  
        File directorio01 = new File("./directorio01/ruta01/");  
        if(ruta01.renameTo(directorio01))  
            System.out.println("El directorio fue movido");  
        else  
            System.out.println("El directorio fue movido");  
    } catch (Exception e) {  
        System.out.println(e.toString());  
    }  
}
```

### 3.5. Leer contenido de directorios

Para conocer el contenido de un directorio tenemos dos opciones.

En primer lugar, podemos obtener los objetos File correspondientes a todos los archivos y directorios que se encuentran en el propio directorio con el método:

```
public File[] listFiles()
```

Este método nos devuelve un array de objetos File.

Si lo que nos interesa son solo los nombres, podemos utilizar:

```
public String[] list()
```

Veámoslo en código:

```
public static void main(String[] args) {
    try {

        File directorio = new File("./nbproject/");
        File[] archivos = directorio.listFiles();
        for(File archivo:archivos){
            System.out.println(archivo.getName());
        }

        System.out.println("");

        String[] archivosString = directorio.list();
        for(String archString: archivosString){
            System.out.println(archString);
        }

    } catch (Exception e) {
        System.out.println(e.toString());
    }
}
```

Link de la ayuda oficial de java de la clase File:

<https://docs.oracle.com/javase/8/docs/api/java/io/File.html>

## 4. LECTURA Y ESCRITURA DE INFORMACIÓN EN FICHEROS



### 4.1. Lectura de información binaria en ficheros

Para leer el contenido de un fichero en formato binario, es decir, byte a byte, utilizaremos la clase `java.io.FileInputStream`, clase hija de `java.io.InputStream` que ya vimos en la unidad 2 con la entrada/salida estándar.

Para crear un objeto de lectura de fichero por bytes tendremos que pasar como parámetro la ruta del fichero que deseamos leer o un objeto `File` que haga referencia a dicho fichero.

```
public FileInputStream(File file) throws FileNotFoundException
```

```
public FileInputStream(String name) throws FileNotFoundException
```

```
FileInputStream lector1 = new FileInputStream("./archivo02.txt");

File archivo03 = new File("./archivo03.txt");
FileInputStream lector2 = new FileInputStream(archivo03);
```

Para leer los datos de este objeto tenemos dos opciones principales, leer carácter a carácter o en bloques de n caracteres.

```
public int read() throws IOException
```

```
public int read(byte[] b) throws IOException
```

Hay que destacar que la primera opción nos devuelve un valor entero, correspondiente al byte que ha leído, siendo -1 en el caso de no leer nada, mientras que a la segunda opción pasamos por parámetro un array de bytes que se llenará, total o parcialmente, con el contenido leído, devolviendo el número de bytes que ha leído. El resultado será -1 si no ha leído nada.

Veamos un ejemplo de cada uno:

```
int palabra;
do{
    palabra = lector2.read();
    if (palabra == -1)
        break;
    else{
        System.out.print(palabra + " ");
    }
}while(true);

Lector2.close();
```

```
byte[] lectura = new byte[100];
int longitud;

do{
    longitud = lector1.read(lectura);
    if (longitud == -1)
        break;
    else{
        for(int i=0; i<longitud; i++){
            System.out.print(lectura[i] + " ");
        }
    }
    System.out.println("");
}while(true);

lector1.close();
```

Algo muy importante, que no debemos olvidar, es cerrar el fichero después de terminar de leer y/o escribir en él, ya que, de lo contrario, puede quedarse bloqueado y no podremos acceder a su contenido.

Para ello vamos a utilizar el método close().

## 4.2. Lectura de información en ficheros de texto

Para este tipo de lectura contamos con dos opciones.

### *4.4.1. FileReader y BufferedReader*

En primer lugar tenemos la clase `java.io.FileReader`, la cual podemos combinar con `java.io.BufferedReader` para leer línea a línea en lugar de carácter a carácter.

Para crear un objeto `FileReader` podemos pasar por parámetro un objeto `File` o el nombre del fichero que queremos leer.

```
public FileReader(File file) throws FileNotFoundException
```

```
public FileReader(String fileName) throws FileNotFoundException
```

Y a la hora de crear el objeto `BufferedReader` pasaremos por parámetro el objeto `FileReader` que hayamos creado previamente.

```
public BufferedReader(Reader in)
```

Ejemplos:

```
File archivo = new File("archivo.txt");
FileReader lector1 = new FileReader(archivo);
BufferedReader lectorBuff1 = new BufferedReader(lector1);
```

```
FileReader lector2 = new FileReader("archivo.txt");
BufferedReader lectorBuff2 = new BufferedReader(lector2);
```

El objeto `FileReader`, de por sí, ya nos permitirá leer el contenido del fichero, pero principalmente carácter a carácter, mediante el método `read()`, que devuelve un entero correspondiente al valor del carácter.

Mientras que si utilizamos el objeto `BufferedReader` podremos realizar también la lectura carácter a carácter, con `read()`, o línea a línea con `readLine()`. Este último método nos devolverá el `String` correspondiente a la línea que ha leído, con un resultado de `null` en el caso de llegar al final del fichero.

Lectura carácter a carácter:

```
int dato;
do{
    dato = lector1.read();
    if(dato == -1)
        break;
    else
        System.out.print(dato + " ");
}while(true);
lector1.close();
System.out.println("");
```

Lectura línea a línea:

```
String linea;
do{
    linea = lectorBuff2.readLine();
    if(linea == null)
        break;
    else
        System.out.println(linea);
}while(true);
lectorBuff2.close();
System.out.println("");
```

#### 4.4.1. Scanner

La segunda opción para realizar la lectura del contenido de un fichero de texto será utilizando la clase Scanner pasándole un objeto File por parámetro, con el cual indicaremos de qué fichero queremos leer.

En este caso solo podremos utilizar los métodos next() y nextLine() para recoger línea a línea los datos del fichero. También haremos uso de los métodos hasNext() y hasNextLine() para realizar una comprobación previa de si queda alguna línea por leer.

```
String datos;

File ficheroDatos = new File("archivo03.txt");
Scanner lectorDatos = new Scanner(ficheroDatos);
while (lectorDatos.hasNextLine()) {
    datos = lectorDatos.nextLine();
    System.out.println(datos);
}

lectorDatos.close();
```

Links a la ayuda de java de todas las clases que hemos visto para la lectura de ficheros:

<https://docs.oracle.com/javase/8/docs/api/java/io/FileInputStream.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html>

#### 4.3. Escritura de información binaria en ficheros

En el caso de la escritura binaria de ficheros haremos uso de la clase `java.io.FileOutputStream`, clase hija de `java.io.OutputStream`.

A la hora de crear el objeto de la clase `FileOutputStream` contamos con varias opciones.

Por un lado, podemos elegir, como ocurría con `FileInputStream`, entre pasar como parámetro un objeto `File` o directamente la ruta del fichero que queremos utilizar.

```
public FileOutputStream(File file) throws FileNotFoundException
```

```
public FileOutputStream(String name) throws FileNotFoundException
```

Con estos dos constructores, en el caso de que el fichero ya existe, se eliminará todo su contenido previamente a realizar la escritura. Es decir, que perderemos todo el contenido que tenía antes de escribir sobre él.

En el caso de que lo que necesitemos sea añadir contenido, tenemos dos constructores, con los que podemos indicar esta opción.

```
public FileOutputStream(File file, boolean append) throws  
FileNotFoundException
```

```
public FileOutputStream(String name, boolean append) throws  
FileNotFoundException
```

Estos dos constructores incluyen un parámetro boolean, denominado append, de tal forma que, si le pasamos true, lo que escribamos se añadirá al final del fichero, pero si pasamos false, se eliminará el contenido actual y se escribirá sobre el fichero vacío.

Para escribir en el fichero, utilizaremos el método write, al cual podremos pasarle como parámetro un array de bytes o un entero correspondiente al valor de un byte.

```
byte[] array = new byte[26];  
  
FileOutputStream escritor1 = new FileOutputStream ("archivo02.txt");  
for(int i = 0; i < 26; i++){  
    escritor1.write(i);  
    array[i] = (byte) i;  
}  
escritor1.close();  
  
FileOutputStream escritor2 = new FileOutputStream ("archivo02.txt",  
true);  
escritor2.write(array);  
escritor2.close();
```

#### 4.4. Escritura de información en ficheros de texto

Al igual que ocurría con la lectura de ficheros de texto, en la escritura tenemos dos opciones.

#### 4.4.1. FileOutputStream, OutputStreamWriter y BufferedWriter

La primera es utilizando java.io.FileOutputStream, combinado con java.io.OutputStreamWriter y java.io.BufferedWriter.

El procedimiento consiste en crear un objeto FileOutputStream, utilizar este objeto para crear un objeto OutputStreamWriter, y después utilizar este último para crear un BufferedWriter.

```
public OutputStreamWriter(OutputStream out)
```

```
public BufferedWriter(Writer out)
```

El método write() será el encargado de la propia escritura en el fichero. A este es método podremos pasarle por parámetro tanto un carácter, como una cadena de caracteres.

```
FileOutputStream fos = new FileOutputStream("archivo01.txt");
OutputStreamWriter osw = new OutputStreamWriter(fos);
BufferedWriter fw = new BufferedWriter(osw);

String[] cadenas = {"Hola,", "¿qué", "tal?"};

for(String cadena:cadenas){
    fw.write(cadena + " ");
}
fw.write("\n");
for(String cadena:cadenas){
    fw.write(cadena + "\n");
}
fw.close();
osw.close();
fos.close();
```

#### 4.4.2. FileWiter y PrintWriter

La segunda opción es utilizar la clase java.io.PrintWriter en combinación con java.io.FileWriter.

FileWriter cuenta con cuatro constructores que nos permiten pasar como parámetro un objeto File, un String con la ruta del fichero, y también nos da la opción de indicar si queremos eliminar el contenido del fichero, si ya existe, o si por el contrario deseamos añadir la nueva información al final del mismo.

```
public FileWriter(File file) throws IOException
```

```
public FileWriter(File file, boolean append) throws IOException
```

```
public FileWriter(String fileName) throws IOException
```

```
public FileWriter(String fileName, boolean append) throws IOException
```

Al constructor de PrintWriter le vamos a pasar el objeto FileWriter que obtengamos.

```
public PrintWriter(Writer out)
```

Como métodos de escritura, PrintWriter cuenta con una gran cantidad, entre los que destacan:

- **print**: que nos permitirá escribir en el fichero datos de tipo primitivo como caracteres, enteros, números en coma flotante, ... y también strings.
- **println**: que funciona igual que “print”, pero este escribirá un salto de línea al final.
- **write**: con él podemos escribir caracteres, arrays de caracteres y Strings.

```
FileWriter archivo = new FileWriter("archivo0.txt", false);
PrintWriter escritor = new PrintWriter(archivo);
String[] cadenas = {"Hola,", "¿qué", "tal?"};

for(String cadena:cadenas){
    escritor.print(cadena + " ");
}
escritor.println("");
for(String cadena:cadenas){
    escritor.println(cadena + " ");
```

```
}
```

```
escritor.close();
```

A continuación, tenéis los links a la ayuda de java de todas las clases que hemos visto para la escritura:

<https://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/OutputStreamWriter.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/BufferedWriter.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html>

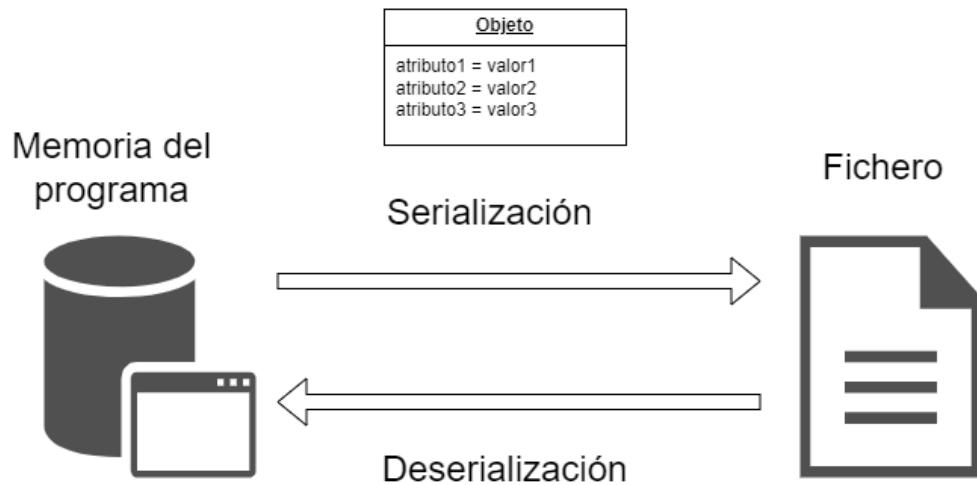
<https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html>

## 5. ALMACENAMIENTO DE OBJETOS EN FICHEROS. PERSISTENCIA. SERIALIZACIÓN.

Además de datos binarios, o texto, java nos permite persistir, es decir, almacenar y recuperar, objetos en ficheros.

Esto consistirá en coger objetos que tengamos almacenados en memoria y los podamos introducir en los ficheros en formato binario, de tal forma que cuando recuperemos esos datos podremos guardarlos de nuevo en objetos del mismo tipo en memoria.

A la operación de almacenar objetos en ficheros le damos el nombre de serialización, mientras que la operación inversa, es decir, recoger los datos del fichero y guardarlo en un objeto le denominamos deserialización.



## 5.1. Serialización

A la hora de almacenar objetos en ficheros vamos a hacer uso de las clases `java.io.FileOutputStream` y `java.io.ObjectOutputStream`. La primera será la que realice la escritura en el fichero, mientras que la segunda se encargará de transformar el objeto a una serie de bytes que se puedan escribir en el fichero.

En primero lugar crearemos un objeto `FileOutputStream`, y luego pasaremos dicho objeto al constructor de `ObjectOutputStream`.

```
protected ObjectOutputStream() throws IOException, SecurityException
```

Después almacenaremos el objeto a través del método `writeObject`, pasando como parámetro el objeto en cuestión.

```
Objeto objeto = new Objeto("Nombre1", 1, "Descripción1");
FileOutputStream fos = new FileOutputStream("objetos.dat");
ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(objeto);
oos.close();
fos.close();
```

Es importante tener en cuenta que solo podremos serializar objetos de clases que implementen la interfaz `Serializable`. En el caso de nuestras propias clases, debemos hacer que nuestra clase implemente dicha interfaz.

```
public class Objeto implements Serializable {  
    private String nombre;  
    private Integer id;  
    private String descripcion;  
  
    ...  
}
```

## 5.2. Deserialización

Para recuperar los objetos de un fichero utilizaremos las clases `java.io.FileInputStream` y `ObjectInputStream`.

En esta ocasión crearemos un objeto `FileInputStream` y luego se lo pasaremos como atributo al constructor de `ObjectInputStream`.

```
public ObjectInputStream(InputStream in) throws IOException
```

Para extraer los objetos haremos uso del método `readObject`, el cual tendremos que utilizar con un casteo explícito para almacenar la información en un objeto de su clase.

```
Objeto objeto2;  
FileInputStream fis = new FileInputStream("objetos.dat");  
ObjectInputStream ois = new ObjectInputStream(fis);  
  
objeto2 = (Objeto) ois.readObject();  
ois.close();  
System.out.println(objeto2.toString());
```

En el caso de necesitar leer todos los objetos de un fichero, debemos comprobar que en el `FileInputStream` aún quedan caracteres por leer, antes de realizar cada lectura.

Veamos un ejemplo:

```
FileInputStream fis = null;
ObjectInputStream ois = null;
Persona persona;
try {
    fis = new FileInputStream(ruta);
    ois = new ObjectInputStream(fis);

    do{
        if(fis.available()==0)
            break;

        persona =(Persona) ois.readObject();
        System.out.println(persona.toString());
    } while (true);
} catch (Exception e) {
    e.printStackTrace();
}
```

Links a la ayuda de java:

<https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/ObjectInputStream.html>