

JAVA CRASH COURSE



@blueslaboratory

UT1 Introducción a la programación:

0. Concepto de programa informático. Instrucciones y datos.
1. Ejecución de programas en ordenadores:
 - 1.1. Datos, algoritmos y programas.
 - 1.2. Hardware vs. Software.
 - 1.3. Estructura funcional de un ordenador: procesador, memoria.
 - 1.4. Tipos de software:
 - 1.4.1. BIOS.
 - 1.4.2. Sistema Operativo.
 - 1.4.3. Aplicaciones.
 - 1.5. Código fuente, código objeto y código ejecutable; máquinas virtuales.
2. Lenguajes de programación:
 - 2.1. Tipos de lenguajes de programación.
 - 2.2. Características de los lenguajes más difundidos.
3. Introducción a la ingeniería del software:
 - 3.1. Proceso software y ciclo de vida del software.
 - 3.2. Fases del desarrollo de una aplicación:
 - 3.2.1. Análisis.
 - 3.2.2. Diseño.
 - 3.2.3. Codificación.
 - 3.2.4. Pruebas.
 - 3.2.5. Documentación.
 - 3.2.6. Explotación.
 - 3.2.7. Mantenimiento.
 - 3.3. Modelos de proceso de desarrollo software:
 - 3.3.1. Cascada.
 - 3.3.2. Evolutivo.
 - 3.3.3. Iterativo.
 - 3.4. Metodologías de desarrollo software.
 - 3.4.1. Características.
 - 3.4.2. Técnicas.
 - 3.4.3. Objetivos.
 - 3.4.4. Tipos de metodologías.
 - 3.5. Herramientas CASE (Computer Aided Software Engineering).
4. Proceso de obtención de código ejecutable a partir del código fuente, herramientas implicadas:
 - 4.1. Editores.
 - 4.2. Compiladores.
 - 4.3. Enlazadores.
5. Errores en el desarrollo de programas.
6. Importancia de la reutilización de código.

0. Concepto de programa informático. Instrucciones y datos.

Según la **RAE**, el **software** es un **conjunto de programas, instrucciones y reglas informáticas** que permiten ejecutar distintas tareas en una computadora.

Se considera que el software es el **equipamiento lógico e intangible** de un ordenador. En otras palabras, el concepto de software abarca a todas las **aplicaciones informáticas**, como los procesadores de textos, las hojas de cálculo y los editores de imágenes.

Un **programa informático** es un conjunto de instrucciones escritas en algún lenguaje de programación. El programa debe ser compilado o interpretado para poder ser ejecutado y así cumplir su objetivo.

Un **programa** está formado por una serie de **INSTRUCCIONES** y de estructuras de **DATOS**. Un programa al ejecutarse en un ordenador, en general, acepta una serie de datos de **ENTRADA** y produce unos resultados de **SALIDA**, ejecutando para ello las instrucciones y manejando las estructuras de datos que componen el programa.

Definición de dato: representación formal de hechos, conceptos o instrucciones adecuada para su comunicación, interpretación y procesamiento por seres humanos o medios automáticos.

Los **tipos de datos** se caracterizan por:

- **Dominio de posibles valores:** qué valores puede tomar.
- **Cómo se representan:** cuál es la representación interna y cómo se representan en el lenguaje de programación.
- **Operadores asociados:** qué operaciones o cálculos se pueden realizar con esos datos.

Algunos tipos de datos en Java:

- Entero (byte, short, int, long)
- Booleano (boolean)
- Carácter (char)
- Real (float, double)

ENTRADA → PROGRAMA → SALIDA

1. Ejecución de programas en ordenadores

1.1. Datos, algoritmos y programas.

PROGRAMA = DATOS + INSTRUCCIONES

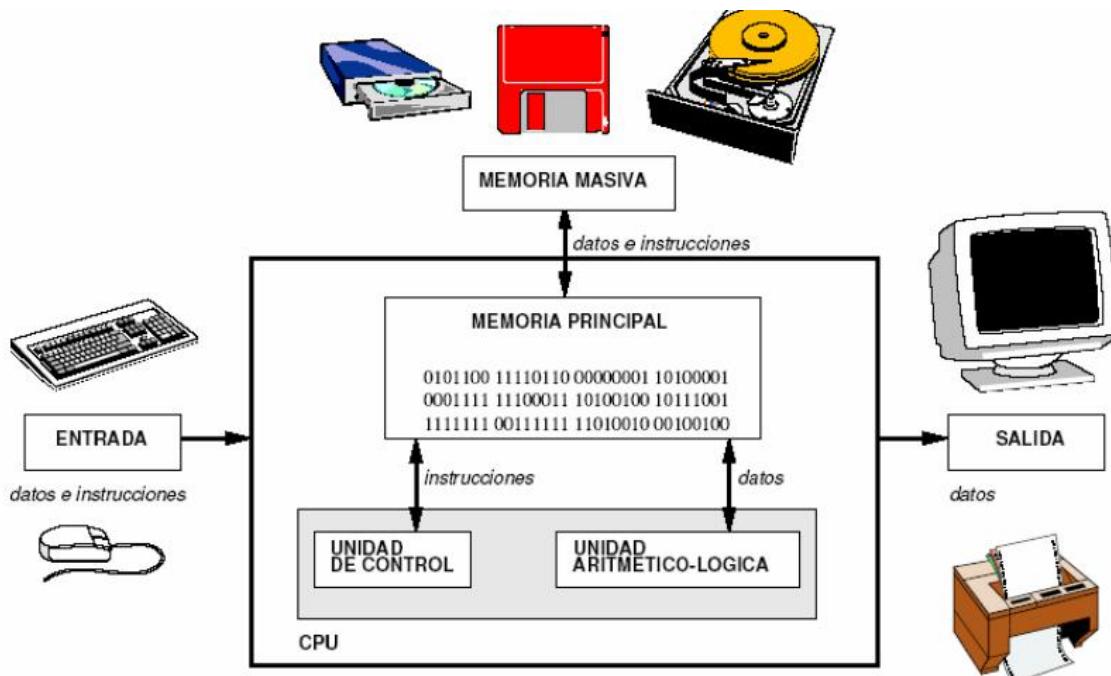
Por otra parte, el término **algoritmo** consiste en una lista ordenada de operaciones que tienen el propósito de buscar la solución a un problema en matemáticas, informática y disciplinas afines (también aplicable en la vida cotidiana).

Un programa es el producto de la traducción de un algoritmo mediante un lenguaje de programación.

Hardware vs. Software.

El *Hardware* son todos los componentes y dispositivos físicos y tangibles que forman una computadora como la CPU o la placa base, mientras que el *Software* es el equipamiento lógico e intangible como los programas y datos que almacena el ordenador.

1.2. Estructura funcional de un ordenador: procesador, memoria.



Los programas se almacenan en una memoria no volátil (por ejemplo un disco), para que luego el usuario del ordenador, directa o indirectamente, solicite su ejecución. En el momento de dicha solicitud, el programa es cargado en la memoria de acceso aleatorio o RAM del equipo, bajo el control del software llamado sistema operativo, el cual puede acceder directamente al procesador. El procesador ejecuta el programa, instrucción por instrucción.

hasta que termina. La memoria se estructura en posiciones o palabras de memoria a las que se accede por su dirección.

A un programa en ejecución se le suele llamar también proceso. Un programa puede terminar su ejecución en forma normal o por causa de un error, dicho error puede ser de software o de hardware.

Ejecución de un programa - IMPORTANTE

Una vez cargado el programa en la memoria se le cede el control del ordenador:

1. Se lee una instrucción del programa.
2. La unidad de control decodifica la instrucción.
3. La unidad de control envía las señales necesarias para ejecutar la instrucción:
 - Se leen los datos de entrada.
 - Se efectúa una operación con ellos en la ALU (UAL, Unidad Aritmético Lógica).
 - Se almacena el resultado.
4. Se determina cuál es la siguiente instrucción que se debe ejecutar.
5. Se vuelve al paso 1.

1.3. Tipos de software.

1.3.1 BIOS/UEFI

La BIOS (siglas en inglés de basic input/output system; en español "sistema básico de entrada y salida") es un software que localiza y reconoce todos los dispositivos necesarios para cargar el sistema operativo en la memoria RAM; es un software muy básico instalado en la placa base que permite que ésta cumpla su cometido. Proporciona la comunicación de bajo nivel, el funcionamiento y configuración del hardware del sistema que, como mínimo, maneja el teclado y proporciona una salida básica (emitiendo pitidos normalizados por el altavoz de la computadora si se producen fallos) durante el arranque. La BIOS **usualmente está escrita en lenguaje ensamblador**.

La BIOS es un programa tipo **firmware**. El firmware no es más que un bloque de instrucciones para propósitos muy concretos, estas instrucciones están grabadas en una memoria **de solo lectura o ROM** (Read Only Memory). Establecen la lógica de más bajo nivel para poder **controlar los circuitos electrónicos de un dispositivo** de cualquier tipo.

UEFI (*Unified Extensible Firmware Interface*) es la versión avanzada y más actual de las BIOS antiguas.

1.3.2 Sistema Operativo

Un Sistema Operativo es el software encargado de ejercer el control y coordinar el uso del hardware entre diferentes programas de aplicación y los diferentes usuarios. Es un administrador de los recursos de hardware del sistema.

En una definición informal, es un programa que realiza una distribución ordenada y controlada de los procesadores, memorias y dispositivos de E/S entre los diversos programas que compiten por ellos.

A pesar de que todos nosotros usamos sistemas operativos casi a diario, es difícil definir qué es un sistema operativo. En parte, esto se debe a que los sistemas operativos realizan dos funciones diferentes:

- Proveer una máquina virtual, es decir, un ambiente en el cual el usuario pueda ejecutar programas de manera conveniente, protegiéndolo de los detalles y complejidades del hardware.
- Administrar eficientemente los recursos del computador.

Ejemplos: Windows, Linux, Mac OS...

1.3.3 Aplicaciones

En Informática, una aplicación es un tipo de programa informático diseñado como herramienta para permitir a un usuario realizar uno o diversos tipos de trabajo.

Suele ser una solución informática para la automatización de ciertas tareas complicadas como pueden ser la contabilidad, la redacción de documentos, o la gestión de un almacén. Algunos ejemplos de programas de aplicación son los procesadores de textos, hojas de cálculo, y bases de datos.

Las aplicaciones desarrolladas “a medida” suelen ofrecer una gran potencia ya que están exclusivamente diseñadas para resolver un problema específico. Otros, llamados “paquetes integrados de software”, ofrecen menos potencia pero a cambio incluyen varias aplicaciones, como un programa procesador de textos, de hoja de cálculo y de base de datos.

Otros ejemplos de programas de aplicación pueden ser: programas de comunicación de datos, multimedia, presentaciones, diseño gráfico, cálculo, finanzas, correo electrónico, compresión de archivos, presupuestos de obras, gestión de empresas, etc.

1.4. Código fuente, código objeto y código ejecutable. Máquinas virtuales. - IMPORTANTE

Código fuente: El código fuente de un programa informático (o software) es un conjunto de líneas de texto que son las instrucciones que debe seguir el ordenador para ejecutar dicho programa.

Código objeto: Se llama código objeto al código que resulta de la compilación del código fuente. Para ello, se usa un programa llamado compilador.

El código objeto consiste en lenguaje máquina o bytecode y se distribuye en uno o varios archivos que corresponden a cada código fuente compilado.

Código ejecutable: Para obtener un programa ejecutable se han de enlazar todos los archivos de código objeto con un programa llamado enlazador (linker).

Máquinas virtuales: Una máquina virtual Java (en inglés Java Virtual Machine, JVM) es una máquina virtual de proceso nativo, es decir, ejecutable en una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (el Java bytecode), el cual es generado por el compilador del lenguaje Java.

El código binario de Java no es un lenguaje de alto nivel, sino un intermedio de bajo nivel (para la JVM). Como todas las piezas del rompecabezas Java, fue desarrollado originalmente por Sun Microsystems. Actualmente el propietario es la empresa Oracle.

La máquina virtual Java es en última instancia la que convierte el código bytecode a código nativo del dispositivo final.

La gran **ventaja** de la máquina virtual java es aportar portabilidad al lenguaje de manera que desde Sun Microsystems se han creado diferentes máquinas virtuales java para diferentes arquitecturas y así un programa (extensión .class) escrito en un Windows puede ser interpretado en un entorno Linux. Tan solo es necesario disponer de dicha máquina virtual para dichos entornos.

En **resumen:**

Una de las características más relevantes del lenguaje de programación Java es el hecho de que utiliza una máquina virtual para la ejecución de los programas, haciendo esta de intermediaria entre la máquina real (HW + SO) y los programas desarrollados por los programadores.

Esto supuso una gran ventaja a la hora de poder distribuir un mismo código ya compilado en diferentes tipos de máquinas físicas con diferentes sistemas operativos. Basta con compilar el código fuente de alto nivel a código intermedio (que es código máquina para la JVM), también llamado bytecode, para que pudiera ser empleado en máquinas virtuales que corren sobre máquinas físicas.

<https://www.adictosaltrabajo.com/tutoriales/byte-code>

2. Lenguajes de programación.

2.1. Tipos de lenguajes de programación.

Los lenguajes de programación se pueden clasificar según varios criterios. Se pueden encontrar hasta 11 criterios diferentes: Nivel de abstracción, propósito, evolución histórica, manera de ejecutarse, manera de abordar la tarea a realizar, paradigma de programación (nos va a interesar), lugar de ejecución, concurrencia, interactividad, realización visual y determinismo.

Hay que tener en cuenta también que en la práctica, la mayoría de lenguajes no pueden ser puramente clasificados en una categoría, pues surgen incorporando ideas de otros lenguajes y de otras filosofías de programación, pero no importa al establecer las clasificaciones, pues el auténtico objetivo de las mismas es mostrar los rangos, las posibilidades y tipos de lenguajes que hay.

1. Nivel de abstracción.

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

- **Lenguajes de bajo nivel:** La programación se realiza teniendo muy en cuenta las características del procesador. Ejemplo: Lenguajes tipo ensamblador.
- **Lenguajes de nivel medio:** Permiten un mayor grado de abstracción pero al mismo tiempo mantienen algunas cualidades de los lenguajes de bajo nivel. Ejemplo: C puede realizar operaciones lógicas y de desplazamiento con bits.
- **Lenguajes de alto nivel:** Más parecidos al lenguaje humano. Manejan conceptos, tipos de datos, etc., de una manera cercana al pensamiento humano ignorando (abstrayéndose) del funcionamiento de la máquina. Ejemplos: Java, Ruby, PHP, Pascal, LISP, ADA.

Hay quien sólo considera lenguajes de bajo nivel y de alto nivel, (en ese caso, C es considerado de alto nivel).

2. Propósito.

Según el propósito, es decir, el tipo de problemas a tratar con ellos:

- **Lenguajes de propósito general:** Aptos para todo tipo de tareas. Ejemplo: C.
- **Lenguajes de propósito específico:** Hechos para un objetivo muy concreto. Ejemplo: Csound (para crear ficheros de audio).
- **Lenguajes de programación de sistemas:** Diseñados para realizar sistemas operativos o drivers. Ejemplo: C.
- **Lenguajes de script:** Para realizar tareas varias de control y auxiliares. Antiguamente eran los llamados lenguajes de procesamiento por lotes (batch) o JCL ("Job Control Languages"). Se subdividen en varias clases

(de shell, de GUI, de programación web, etc.). Ejemplos: bash (shell), JavaScript (programación web).

3. Evolución histórica.

Con el paso del tiempo, se va incrementando el nivel de abstracción, pero en la práctica, los de una generación no terminan de sustituir a los de la anterior:

- **Lenguajes de primera generación (1GL)**: Código máquina.
- **Lenguajes de segunda generación (2GL)**: Lenguajes ensamblador.
- **Lenguajes de tercera generación (3GL)**: La mayoría de los lenguajes modernos, diseñados para facilitar la programación a los humanos. Ejemplos: C, Java.
- **Lenguajes de cuarta generación (4GL)**: se ha dado este nombre a ciertas herramientas que permiten construir aplicaciones sencillas combinando piezas prefabricadas. Hoy se piensa que estas herramientas no son, propiamente hablando, lenguajes. Algunos proponen reservar el nombre de cuarta generación para la programación orientada a objetos.
- **Lenguajes de quinta generación (5GL)**: La intención es que el programador establezca qué problema ha de ser resuelto y las condiciones a reunir, y la máquina lo resuelve. Se usan en inteligencia artificial. Ejemplo: Prolog.

4. Manera de ejecutarse. (“INTERESANTE RECALCARLO”)

Según la manera de ejecutarse:

- Lenguajes compilados: Un programa traductor traduce el código del programa (código fuente) en código máquina (código objeto). Otro programa, el enlazador, unirá los ficheros de código objeto del programa principal con los de las librerías para producir el programa ejecutable. Ejemplo: C, Pascal, Fortran.
- Lenguajes interpretados: Un programa (intérprete), ejecuta las instrucciones del programa de manera directa. Ejemplo: Lisp, Javascript.
- También los hay **mixtos**, como Java, que primero pasan por una fase de compilación en la que el código fuente se transforma en “**bytecode**”, y este “bytecode” puede ser ejecutado luego (interpretado) en ordenadores con distintas arquitecturas (procesadores) que tengan todos instalados la “máquina virtual” **Java**.

5. Manera de abordar la tarea a realizar.

Según la manera de abordar la tarea a realizar, pueden ser:

- Lenguajes imperativos: Indican **cómo** hay que hacer la tarea, es decir, expresan los pasos a realizar. Ejemplo: C.
- Lenguajes declarativos: Indican **qué** hay que hacer. Ejemplos: Lisp, Prolog. Otros ejemplos de lenguajes declarativos, pero que no son lenguajes de

programación propiamente dicha, son HTML (para describir páginas web) o SQL (para consultar bases de datos).

6. Paradigma de programación. (“IMPORTANTE RECALCARLO”)

El paradigma de programación es el estilo de programación empleado. Los principales son:

- Lenguajes de programación procedural: Divide el problema en partes más pequeñas, que serán realizadas por subprogramas (subrutinas, funciones, procedimientos), que se llaman unas a otras para ser ejecutadas. Ejemplos: C, Pascal.
- Lenguajes de programación orientada a objetos: Crean un sistema de clases y objetos siguiendo el ejemplo del mundo real, en el que unos objetos realizan acciones y se comunican con otros objetos. Ejemplos: C++, Java.
- Lenguajes de programación funcional: La tarea se realiza evaluando funciones (como en Matemáticas) de manera recursiva. Ejemplo: Lisp.
- Lenguajes de programación lógica: La tarea a realizar se expresa empleando lógica formal matemática. Ejemplo: Prolog.

7. Lugar de ejecución.

En sistemas distribuidos, según dónde se ejecute:

- Lenguajes de servidor: Se ejecutan en el servidor. Ejemplo: PHP es el más utilizado en servidores web.
- Lenguajes de cliente: Se ejecutan en el cliente. Ejemplo: JavaScript en navegadores web.

8. Conurrencia.

Según admitan o no concurrencia de procesos, esto es, la ejecución simultánea de varios procesos lanzados por el programa:

- Lenguajes concurrentes. Ejemplo: Ada, Java.
- Lenguajes no concurrentes. Ejemplo: C.

9. Interactividad.

Según la interactividad del programa con el usuario u otros programas:

- Lenguajes orientados a sucesos (eventos): El flujo del programa es controlado por la interacción con el usuario o por mensajes de otros programas/sistema operativo, como editores de texto, interfaces gráficas de usuario (GUI) o kernels. Ejemplo: VisualBasic, lenguajes de programación declarativos.
- Lenguajes no orientados a sucesos (eventos): El flujo del programa no depende de sucesos exteriores, sino que se conoce de antemano, siendo los procesos batch el ejemplo más claro (actualizaciones de bases de

datos, colas de impresión de documentos, etc.). Ejemplos: Lenguajes de programación **imperativos**.

10. Realización visual.

Según la realización visual o no del programa:

- Lenguajes de programación visual: El programa se realiza moviendo bloques de construcción de programas (objetos visuales) en un interfaz adecuado para ello. No confundir con entornos de programación visual, como Microsoft Visual Studio y sus lenguajes de programación textuales (como **Visual C#**). Ejemplo: Mindscript, Alice, Scratch.
- Lenguajes de **programación textual**: El código del programa se realiza escribiéndolo. Ejemplos: C, Java, Lisp.

11. Determinismo.

Según se pueda predecir o no el siguiente estado del programa a partir del estado actual:

- Lenguajes **deterministas**. Ejemplos: Todos los anteriores.
- Lenguajes probabilísticos o no deterministas: Sirven para explorar grandes espacios de búsqueda, (como **gramáticas**), y en la investigación teórica de hipercomputación. Ejemplo: mutt (generador de texto aleatorio).

2.2. Características de los lenguajes más difundidos a lo largo del tiempo.

BASIC. Durante mucho tiempo se ha considerado un buen lenguaje para comenzar a aprender, por su sencillez, aunque se podía tender a crear programas poco legibles. A pesar de esta "sencillez" hay versiones muy potentes, incluso para programar en entornos gráficos como Windows.

COBOL. Fue muy utilizado para negocios (para crear software de gestión que tuviese que manipular grandes cantidades de datos). Sigue usándose en el campo de la Banca.

FORTRAN. Concebido para ingeniería, operaciones matemáticas, etc. También va quedando desplazado.

ENSAMBLADOR. Muy cercano al código máquina (es un lenguaje de "bajo nivel"), pero sustituye las secuencias de ceros y unos (bits) por palabras más fáciles de recordar, como MOV, ADD, CALL o JMP.

PASCAL. El lenguaje estructurado por excelencia, y que en algunas versiones tiene una potencia comparable a la del lenguaje C.

C. Uno de los mejor considerados actualmente (junto con C++ y Java, que mencionaremos a continuación), porque no es demasiado difícil de aprender y permite un grado de control del ordenador muy alto, combinando características de lenguajes de alto y bajo nivel. Además, es muy transportable: existe un **estándar**, el **ANSI C**, lo que asegura que se pueden convertir programas en C de un ordenador a otro o de un sistema operativo a otro con bastante menos esfuerzo que en otros lenguajes.

C++. Un lenguaje desarrollado a partir de C, que permite Programación Orientada a Objetos.

Java. Desarrollado a su vez a partir de C++, elimina algunos de sus inconvenientes y ha alcanzado una gran difusión gracias a su uso en Internet. Al igual que C++ es Orientado a Objetos.

Python.

Kotlin.

3. Introducción a la ingeniería del software. - HA DICHO QUE NOS LO SALTAMOS

3.1. Proceso software y ciclo de vida del software

El software de ordenadores es el producto que diseñan y construyen los ingenieros del software. El software es un elemento lógico, no físico. El software tiene unas características considerablemente distintas a las del hardware:

1. El software se desarrolla no se fabrica.
2. La mayoría del software se realiza a medida.

La **Ingeniería del Software** es una disciplina o área de la Informática que ofrece métodos y técnicas para desarrollar y mantener software de calidad.

La Ingeniería del Software aborda todas las fases de ciclo de vida de cualquier tipo de sistema de información.

Sistema de Información. 'Sistema de información' (SI) es un conjunto de elementos orientados al tratamiento y administración de datos e información, organizados y listos para su posterior uso, generados para cubrir una necesidad (objetivo). Dichos elementos formarán parte de alguna de estas categorías:

- Personas.
- Datos.
- Actividades o técnicas de trabajo.
- Recursos materiales en general, típicamente recursos informáticos y de comunicación, aunque no tienen por qué ser de este tipo obligatoriamente.

Existen múltiples definiciones de Ingeniería del Software, casi tantas como autores que han escrito sobre el tema. Vamos a mencionar la de **IEEE** ("Institute of Electrical and Electronics Engineers"):

"Ingeniería del Software es la aplicación de un enfoque sistemático, disciplinado y cuantificable hacia el desarrollo, operación y mantenimiento del software. Es decir, la aplicación de la Ingeniería al software".

El **ciclo de vida** de un sistema abarca toda la vida del sistema, desde su concepción hasta que deja de utilizarse. Se habla también de ciclo de vida del desarrollo software, abarcando las etapas que comienzan en el análisis y finalizan con la entrega del sistema al usuario.

3.2. Fases del desarrollo de una aplicación

3.2.1. Análisis

Para comprender la naturaleza de los programas a construir el analista, (= ingeniero del software) debe comprender el dominio de información del software, así como las funciones requeridas, comportamiento, rendimiento e interconexión.

En esta fase, el analista ha de recoger en colaboración con el cliente, toda la información relativa a los requisitos del *sistema a desarrollar*: su función, rendimiento e interfaces. Hay que analizar costes, recursos y definir tiempos para las siguientes tareas a realizar.

3.2.2. Diseño

El diseño del software es realmente un proceso de muchos pasos que se centra en cuatro atributos distintos de programa:

- Estructura de datos
- Arquitectura de software
- Representaciones de interfaz
- Algoritmo (detalle procedimental)

“El proceso de diseño traduce requisitos en una representación del software donde se pueda evaluar su calidad antes de que comience la codificación”.

3.2.3. Codificación

El diseño se debe traducir en una forma legible por la máquina. Esta tarea se lleva a cabo en el paso de Generación de código (Codificación). Si el diseño se ha realizado de una forma detallada, la generación de código se realiza mecánicamente.

3.2.4. Pruebas

Cuando se ha finalizado la generación del código comienzan las pruebas del programa. El proceso de pruebas se centra en los procesos lógicos internos del software. Se realizan las pruebas para la detección de errores y para asegurar que la entrada definida produce resultados reales de acuerdo con los resultados requeridos.

3.2.5. Documentación

Cada una de las fases por las que pasa un proyecto de Software debe ir acompañada de la correspondiente documentación.

3.2.6 Explotación

La etapa de explotación transcurre una vez que el sistema está a disposición del usuario.

3.2.7. Mantenimiento

El software una vez entregado sufrirá cambios. Los motivos de estos cambios pueden ser:

- Errores que pudieran encontrarse.
- El software debe adaptarse a cambios en su entorno externo (por ejemplo un cambio de sistema operativo).
- El software requiere mejoras funcionales o de rendimiento.

El proceso de soporte y mantenimiento implica que el programa que se va a modificar tendrá que pasar por cada una de las fases precedentes.

3.3. Modelos de proceso de desarrollo software

La estrategia de desarrollo (modelo o proceso) que se elige para realizar una aplicación concreta recibe el nombre de:

Ciclo de vida = modelo de desarrollo

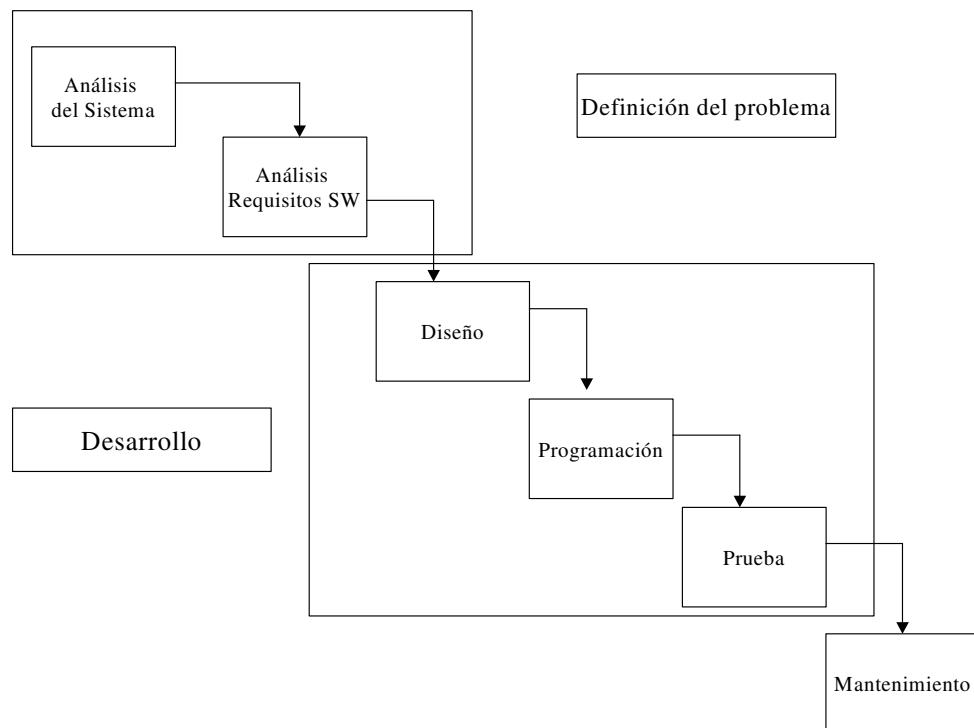
Cada modelo de desarrollo representa un intento de ordenar la actividad de desarrollo del software. Todos los modelos esperan contribuir al control y la coordinación de un proyecto de software real.

Dependiendo de la naturaleza de un proyecto de desarrollo, puede haber razones para adoptar un método en lugar de otro o incluso una combinación de ellos.

3.3.1 Cascada

En este modelo se distinguen una serie de fases, que pueden diferir en algunos autores y libros en nombre y número pero todas tienen en común dos aspectos:

- Implantación ascendente.
- Progresión lineal y secuencial de una fase a otra.



Razones por las que a veces no se debe utilizar el modelo lineal:

- A veces es difícil que el usuario declare explícitamente todos los requisitos al principio del proyecto. El modelo lineal secuencial requiere que esto ocurra y resulta complicado compaginarlo con la incertidumbre existente al comienzo de muchos proyectos.
- Una versión visible del trabajo realizado en el proyecto no estará disponible hasta que todo el trabajo esté realizado, haciendo que si ocurre un error ya sea demasiado tarde.

A pesar de estos problemas el modelo lineal secuencial proporciona una plantilla para el desarrollo del proyecto. Siempre es mejor que realizar un desarrollo al azar.

El modelo lineal es el modelo más antiguo y el que más se ha utilizado en la ingeniería del software.

Cuándo se debe utilizar:

"El modelo lineal se puede utilizar y es aconsejable utilizarlo cuando los requisitos se han entendido y definido correctamente".

3.3.2. Evolutivo

En el desarrollo en cascada no se tiene en cuenta la naturaleza evolutiva del software. Se debe de tener una especificación totalmente detallada de **TODOS** los requerimientos que debe satisfacer el software que desarrollemos para poder iniciar las diferentes etapas de desarrollo.

El desarrollo evolutivo se basa en la idea de desarrollar una implementación inicial e ir refinándola a través de diferentes versiones hasta desarrollar un sistema software que satisfaga todos los requerimientos del cliente.



Un enfoque evolutivo para el desarrollo de software suele ser más efectivo que el desarrollo en cascada ya que desde un principio se le entrega al cliente una versión que satisface los requerimientos principales.

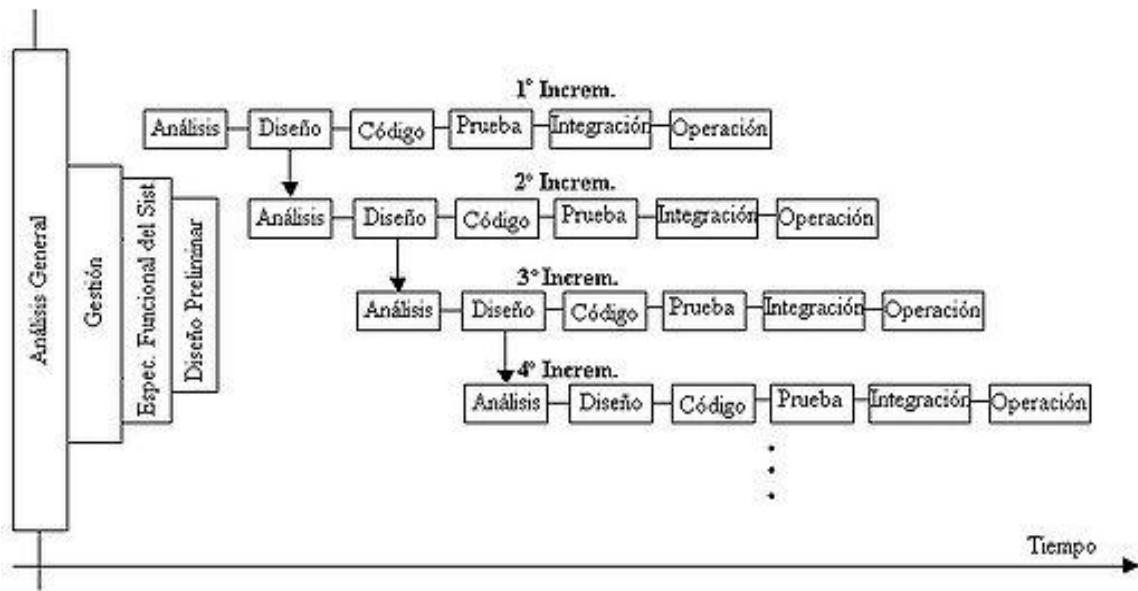
El modelo iterativo incremental y el modelo en espiral son dos modelos de tipo evolutivo.

3.3.3. Iterativo

Como ya hemos dicho el modelo en cascada requiere tener una especificación de requerimientos totalmente detallada para poder iniciar la etapa del diseño. Si a lo largo del proceso de desarrollo se cambian los requisitos hay que rehacer parte del trabajo de diseño e implementación para poder hacer frente a los nuevos cambios.

El desarrollo evolutivo permite que los requerimientos no estén totalmente especificados para comenzar con el desarrollo del software, esto hace que el software desarrollado pueda estar mal estructurado y sea difícil de mantener pero permite adaptarse a los cambios en los requisitos.

Vamos a ver un modelo de proceso basado en el desarrollo evolutivo: el **modelo iterativo incremental** (no lo entiendo).



Siguiendo este modelo, los clientes especifican a grandes rasgos los servicios que tiene que proporcionar el sistema software que se quiere desarrollar. Entonces se definen varios incrementos para desarrollar cada uno un subconjunto de la funcionalidad del sistema software dando prioridad a los requerimientos más importantes.

Cada incremento puede desarrollarse siguiendo un modelo en cascada típico, por lo que al inicio de cada incremento se debe tener una especificación detallada de la funcionalidad que se pretende desarrollar en ese incremento.

Cuando el incremento se completa se entrega y los clientes pueden utilizarlo a la espera del siguiente incremento. Esto significa que los clientes obtienen una parte de la funcionalidad del sistema software que necesitan de manera temprana, y se quedan a la espera de nuevos incrementos que mejoren la funcionalidad del sistema software.

Este proceso de desarrollo incremental tiene varias ventajas en contraste al desarrollo en cascada:

- Los clientes no tienen que esperar hasta la etapa final para sacar provecho del sistema software ya que **el primer incremento satisface los requerimientos principales**.
- Los clientes pueden utilizar cada incremento para analizar nuevos requerimientos para incrementos posteriores.
- Existe un **bajo riesgo de un fallo total del proyecto, ya que los errores encontrados en un incremento pueden arreglarse en el incremento posterior**.

Este enfoque de desarrollo es aconsejable si el software que se pretende desarrollar posee una serie de funcionalidades bien definidas que se pueden desarrollar con total independencia.

3.4. Metodologías de desarrollo software

Metodología de desarrollo software: Conjunto de procedimientos, técnicas, herramientas y un soporte documental que ayuda a los desarrolladores a realizar nuevo software.

3.4.1. Características

- Existencia de reglas predefinidas
- Cobertura total del ciclo de desarrollo
- Verificaciones intermedias
- Planificación y control
- Comunicación efectiva
- Utilización sobre un abanico amplio de proyectos
- Fácil formación
- Uso de herramientas CASE
- Actividades que mejoren el proceso de desarrollo
- Soporte al mantenimiento
- Soporte de la reutilización de software

3.4.2. Técnicas

Técnica: Definición de la forma de ejecutar una tarea.

En las distintas fases de una metodología se pueden usar técnicas como:

- Diagramas de casos de uso
- Diagramas de clase
- Diagramas de Flujo de datos
- Diagramas de estructura

3.4.3. Objetivos

Una metodología puede seguir uno o varios modelos de ciclo de vida, es decir, el ciclo de vida indica qué es lo que hay que obtener a lo largo del desarrollo del proyecto pero no cómo hacerlo.

El objetivo de las metodologías es indicar cómo hay que obtener los distintos productos parciales y finales.

3.4.4. Tipos de metodologías

- **Metodologías estructuradas:**
 - Orientadas a procesos.
 - Orientadas a datos.

- o **Mixtas.**
- **Metodologías orientadas a Objetos.**

3.5. Herramientas CASE (Computer Aided Software Engineering).

Computer Aided Software Engineering

Se puede definir a las Herramientas CASE como un conjunto de programas y ayudas que dan asistencia a los analistas, ingenieros de software y desarrolladores, durante todos los pasos del Ciclo de Vida de desarrollo de un Software: Análisis, Diseño, Implementación e Instalación.

Una herramienta CASE suele incluir:

- Un **diccionario de datos** para almacenar información sobre los datos de la aplicación.
- **Herramientas de diseño** para dar apoyo al análisis de datos.
- **Herramientas** que permitan desarrollar el **modelo de datos** corporativo, así como los esquemas conceptual y lógico.
- **Herramientas** para desarrollar los **prototipos** de las aplicaciones.

El uso de las herramientas CASE puede mejorar la productividad en el desarrollo de una aplicación de bases de datos.

3.6. Ejemplos de Herramientas Case más utilizadas.

ERwin:

ERwin es una herramienta para el diseño de bases de datos, que proporciona productividad en el diseño, generación, y mantenimiento de aplicaciones. Desde un modelo lógico de los requerimientos de información, hasta el modelo físico perfeccionado para las características específicas de la base de datos diseñada, además ERwin permite visualizar la estructura, los elementos importantes, y optimizar el diseño de la base de datos. Genera automáticamente las tablas y miles de líneas de procedimientos almacenados y disparadores para los principales tipos de base de datos.

EasyCASE

EasyCASE Profesional - Es un producto para la generación de esquemas de base de datos e ingeniería inversa - trabaja para proveer una solución comprensible para el diseño, consistencia y documentación del sistema en conjunto.

Esta herramienta permite automatizar las fases de análisis y diseño dentro del desarrollo de una aplicación, para poder crear las aplicaciones eficazmente - desde el procesamiento de transacciones a la aplicación de bases de datos de cliente/servidor, así como sistemas de tiempo real.

Oracle Designer:

Oracle Designer es un conjunto de herramientas para guardar las definiciones que necesita el usuario y automatizar la construcción rápida de aplicaciones cliente/servidor gráficas. Integrado con Oracle Developer, Oracle Designer provee una solución para desarrollar sistemas empresariales de segunda generación.

Todos los datos ingresados por cualquier herramienta de Oracle Designer, en cualquier fase de desarrollo, se guardan en un repositorio central, habilitando el trabajo fácil del equipo y la dirección del proyecto.

En el lado del Servidor, Oracle Designer soporta la definición, generación y captura de diseño de los siguientes tipos de bases de datos, por conexión de Oracle:

4. Proceso de obtención de código ejecutable a partir del código fuente, herramientas implicadas

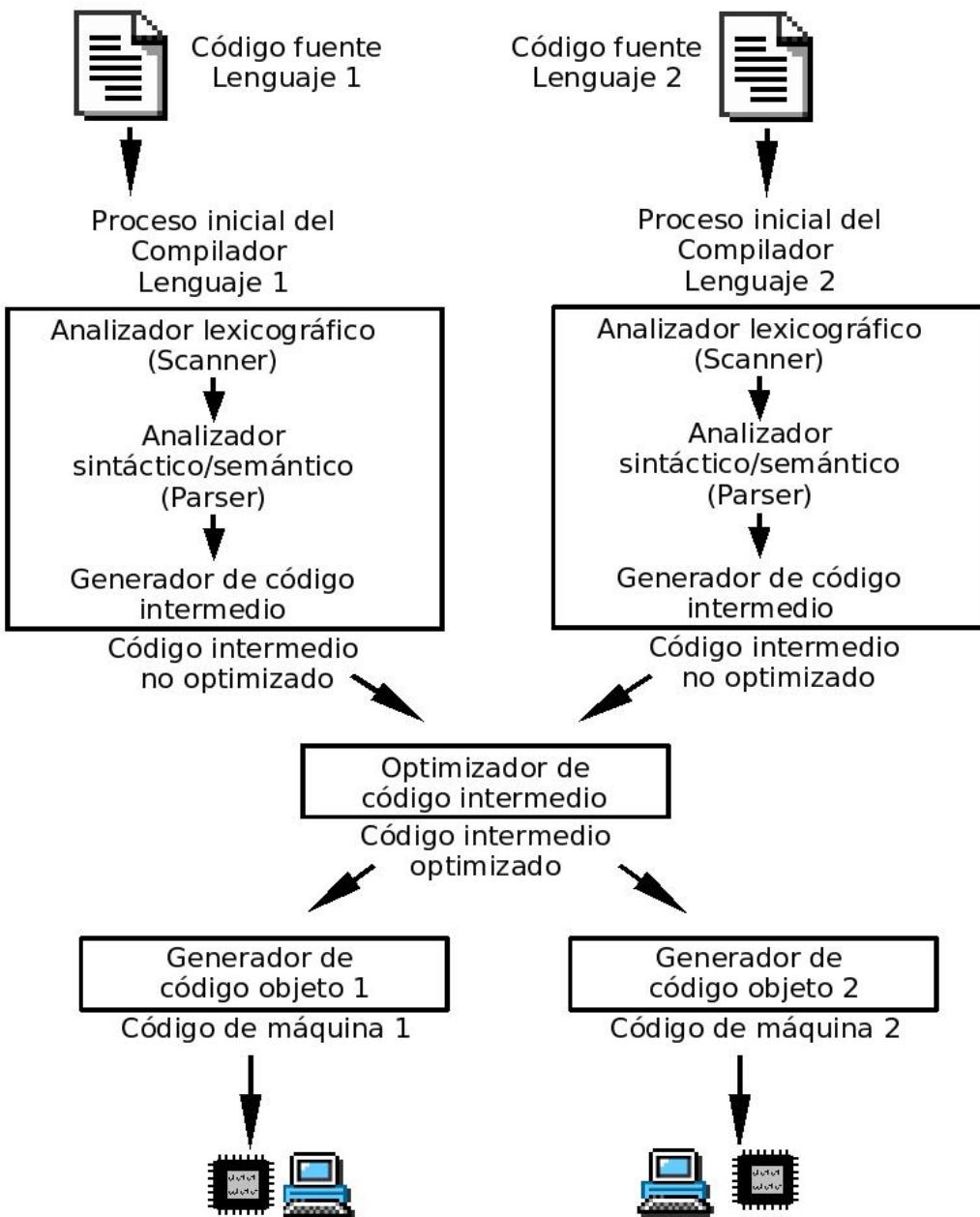
4.1. Editores

Un editor es un programa informático que nos permite crear, modificar (editar) y guardar archivos de texto en la memoria secundaria del ordenador.

Los editores de programas especializados nos ofrecen distintas facilidades para escribir el código: marcar las palabras reservadas en distinto color, terminar las palabras reservadas automáticamente, ayuda online, etc.

4.2. Compiladores

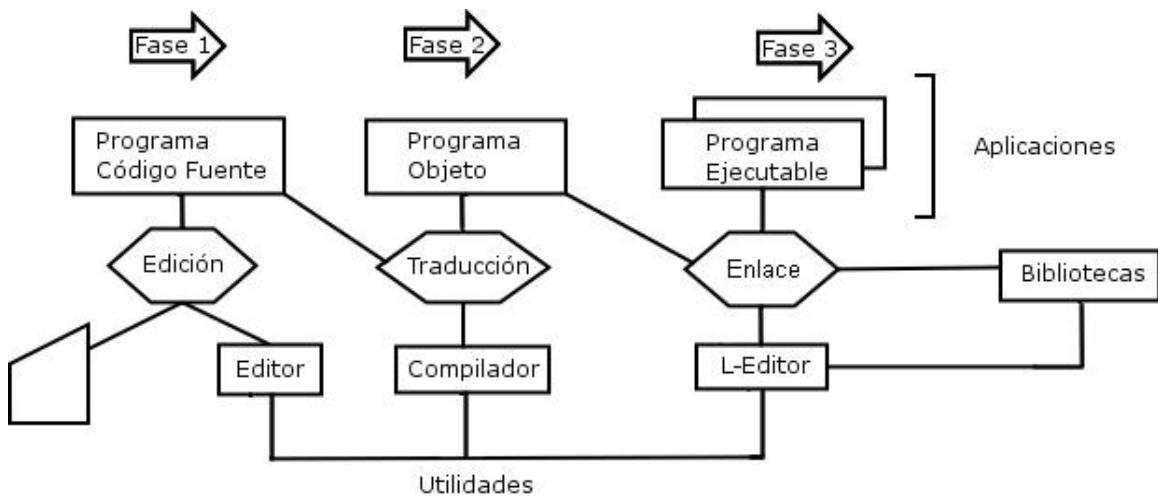
Un **compilador** es un programa informático que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación, generando un programa equivalente que la máquina será capaz de interpretar. Usualmente el segundo lenguaje es lenguaje de máquina, pero también puede ser un lenguaje intermedio (bytecode) o simplemente texto. Este proceso de traducción se conoce como **compilación**.



4.3. Enlazadores

En programación, un **enlazador** (en inglés, *linker*) es un módulo o programa que une los ficheros de código objeto (generados en la compilación), la información de todos los recursos necesarios (librerías/bibliotecas), elimina los recursos que no se necesitan y enlaza el código objeto con sus librerías. Finalmente produce el fichero ejecutable o una librería.

Existen programas que se enlazan dinámicamente, esto significa que este proceso se hace en el momento que se carga el programa.



5. Errores en el desarrollo de programas

En el proceso de desarrollo de un programa, así como durante todo el ciclo de vida de una aplicación, se pueden producir distintos errores. Estos errores pueden ser:

Errores sintácticos. Se detectarán en la fase de compilación del código.

Errores de ejecución. Se detectarán una vez compilado el programa, cuando intentemos ejecutarlo. Un ejemplo de esto es la división por cero. Suponga que tiene la instrucción siguiente:

velocidad = kilómetros / horas

Si la variable horas tiene un valor de 0, se produce un error en tiempo de ejecución en la operación de división. El programa se debe ejecutar para que se pueda detectar este error y si horas contiene un valor válido, no se producirá el error.

Errores lógicos. Los errores lógicos son errores que impiden que un programa haga lo que estaba previsto. El código puede compilarse y ejecutarse sin errores, pero el resultado de una operación puede generar un resultado no esperado.

Por ejemplo, puede tener una variable llamada *nombre* y establecida inicialmente en una cadena vacía. Después en el programa, puede concatenar *nombre* con otra variable denominada *apellido* para mostrar un nombre completo. Si olvida asignar un valor a *nombre*, sólo se mostrará el apellido, no el nombre completo como pretendía.

Los errores lógicos son los más difíciles de detectar y corregir, las herramientas de depuración facilitan el trabajo de encontrar los errores lógicos.

<http://wiki.elhacker.net/bugs-y-exploits/introduccion/errores-de-programacion-comunes>

6. Importancia de la reutilización de código

En programación, reutilización de código es el uso de software existente para desarrollar un nuevo software. La reutilización de código ha sido empleada desde los primeros días de la programación. Los programadores siempre han reusado partes de un código, hojas de cálculo, funciones o procedimientos.

La idea es que parte o todo el código de un programa informático escrito una vez, sea o pueda ser usado en otros programas. La reutilización de código es una técnica común que intenta ahorrar tiempo y energía, reduciendo el trabajo redundante.

Las bibliotecas o librerías de software son un buen ejemplo. Al utilizarlas se está reutilizando código.

El software más fácilmente reutilizable tiene ciertas características: modularidad, bajo acoplamiento, alta cohesión, ocultación de información, etc.

UT1 Ejercicios

1. Secuenciales

Ejercicio 1

Escribir un programa que pregunte al usuario su nombre, y luego le salude.

Ejercicio 2

Calcular el perímetro y área de un rectángulo dada su base y su altura.

Ejercicio 3

Dados los catetos de un triángulo rectángulo, calcular su hipotenusa.

Ejercicio 4

Dados dos números, mostrar la suma, resta, división y multiplicación de ambos.

Ejercicio 5

Escribir un programa que convierta un valor dado en grados Fahrenheit a grados Celsius. Recordad que la fórmula para la conversión es:

$$C = (F-32) * 5 / 9$$

Ejercicio 6

Calcular la media de tres números pedidos por teclado.

Ejercicio 7

Realiza un programa que reciba una cantidad de minutos y muestre por pantalla a cuántas horas y minutos corresponde. Por ejemplo: 1000 minutos son 16 horas y 40 minutos.

Ejercicio 8

Un vendedor recibe un sueldo base más un 10% extra por comisión de sus ventas (se incrementa su sueldo en un 10% de la cantidad vendida), el vendedor desea saber cuánto dinero obtendrá por concepto de comisiones por las tres ventas que realiza en el mes y el total que recibirá en el mes tomando en cuenta su sueldo base y comisiones.

Ejercicio 9

Una tienda ofrece un descuento del 15% sobre el total de la compra y un cliente desea saber cuánto deberá pagar finalmente por su compra.

Ejercicio 10

Un alumno desea saber cuál será su calificación final en la materia de Algoritmos. Dicha calificación se compone de los siguientes porcentajes:

- 55% del promedio de sus tres calificaciones parciales.
- 30% de la calificación del examen final.
- 15% de la calificación de un trabajo final.

Ejercicio 11

Pide al usuario dos números y muestra la “distancia” entre ellos (el valor absoluto de su diferencia, de modo que el resultado sea siempre positivo).

Ejercicio 12

Pide al usuario dos pares de números x_1, y_1 y x_2, y_2 , que representen dos puntos en el plano. Calcula y muestra la distancia entre ellos.

LA DISTANCIA
ENTRE DOS
PUNTOS SE
OBTIENE COMO
CONCLUSIÓN
DEL PROCESO
SIGUIENTE:

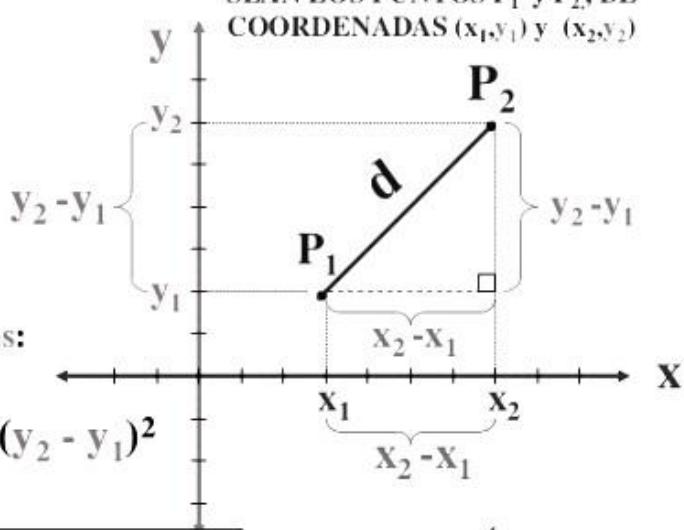
Aquí, Según Pitágoras:

$$d^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

ESTO ES:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

SEAN LOS PUNTOS P_1 y P_2 , DE
COORDENADAS (x_1, y_1) y (x_2, y_2)



ESTA ES LA FÓRMULA
GENERAL PARA DETERMINAR
LA DISTANCIA ENTRE DOS
PUNTOS

Ejercicio 13

Realizar un algoritmo que lea un número y que muestre su raíz cuadrada y su raíz cúbica. PSeInt no tiene ninguna función predefinida que permita calcular la raíz cúbica, ¿Cómo se puede calcular?

Ejercicio 14

Dado un número de dos cifras, diseñe un algoritmo que permita obtener el número invertido. Ejemplo, si se introduce 23 que muestre 32.

Ejercicio 15

Dadas dos variables numéricas A y B, que el usuario debe teclear, se pide realizar un algoritmo que intercambie los valores de ambas variables y muestre cuánto valen al final las dos variables.

Ejercicio 16

Dos vehículos viajan a diferentes velocidades (v_1 y v_2) y están distanciados por una distancia d . El que está detrás viaja a una velocidad mayor. Se pide hacer un algoritmo para ingresar la distancia entre los dos vehículos (km) y sus respectivas velocidades (km/h) y con esto determinar y mostrar en qué tiempo (minutos) alcanzará el vehículo más rápido al otro.

Ejercicio 17

Un ciclista parte de una ciudad A a las HH horas, MM minutos y SS segundos. El tiempo de viaje hasta llegar a otra ciudad B es de T segundos. Escribir un algoritmo que determine la hora de llegada a la ciudad B, también en HH horas, MM minutos y SS segundos.

Ejercicio 18

Pedir el nombre y los dos apellidos de una persona y mostrar las iniciales.

Ejercicio 19

Escribir un algoritmo para calcular la nota final de un estudiante, considerando que: por cada respuesta correcta 5 puntos, por una incorrecta -1 y por respuestas en blanco 0. Imprime el resultado obtenido por el estudiante.

Ejercicio 20

Diseñar un algoritmo que nos diga el dinero que tenemos (en euros y céntimos) después de pedirnos cuantas monedas tenemos (de 2€, 1€, 50 céntimos, 20 céntimos o 10 céntimos).

2. Condicionales (estructuras alternativas)

Ejercicio 1

Algoritmo que pida dos números e indique si el primero es mayor que el segundo o no.

Ejercicio 2

Algoritmo que pida un número y diga si es positivo, negativo o 0.

Ejercicio 3

Escribe un programa que lea un número e indique si es par o impar.

Ejercicio 4

Crea un programa que pida al usuario dos números y muestre su división si el segundo no es cero, o un mensaje de aviso en caso contrario.

Ejercicio 5

Escribe un programa que pida un nombre de usuario y una contraseña y si se ha introducido “pepe” y “asdasd” se indica “Has entrado al sistema”, si no, se da un error.

Ejercicio 6

Programa que lea una cadena por teclado y compruebe si es una letra mayúscula.

Ejercicio 7

Realiza un algoritmo que calcule la potencia, para ello pide por teclado la base y el exponente. Pueden ocurrir tres cosas:

- El exponente sea positivo, sólo tienes que imprimir la potencia.
- El exponente sea 0, el resultado es 1.
- El exponente sea negativo, el resultado es 1/potencia con el exponente positivo.

Ejercicio 8

Algoritmo que pida dos números ‘nota’ y ‘edad’ y un carácter ‘sexo’ y muestre el mensaje ‘ACEPTADA’ si la nota es mayor o igual a cinco, la edad es mayor o igual a dieciocho y el sexo es ‘F’. En caso de que se cumpla lo mismo, pero el sexo sea ‘M’, debe imprimir ‘POSIBLE’. Si no se cumplen dichas condiciones se debe mostrar ‘NO ACEPTADA’.

Ejercicio 9

Algoritmo que pida tres números y los muestre ordenados (de mayor a menor);

Ejercicio 10

Algoritmo que pida los puntos centrales x_1, y_1 , x_2, y_2 y los radios r_1, r_2 de dos circunferencias y las clasifique en uno de estos estados:

- exteriores
- tangentes exteriores
- secantes
- tangentes interiores

- interiores
- concéntricas

Ayuda: <http://mimosa.pntic.mec.es/clobo/geoweb/circun3.htm>

Ejercicio 11

Programa que lea 3 datos de entrada A, B y C. Estos corresponden a las dimensiones de los lados de un triángulo. El programa debe determinar qué tipo de triángulo es, teniendo en cuenta lo siguiente:

- Si se cumple Pitágoras entonces es triángulo rectángulo.
- Si sólo dos lados del triángulo son iguales entonces es isósceles.
- Si los 3 lados son iguales entonces es equilátero.
- Si no se cumple ninguna de las condiciones anteriores, es escaleno.

Ejercicio 12

Escribir un programa que lea un año e indicar si es bisiesto. Nota: un año es bisiesto si es un número divisible por 4, pero no si es divisible por 100, excepto que también sea divisible por 400.

Ejercicio 13

Escribe un programa que pida una fecha (día, mes y año) y diga si es correcta.

Ejercicio 14

La asociación de vinicultores tiene como política fijar un precio inicial al kilo de uva, la cual se clasifica en tipos A y B, y además en tamaños 1 y 2. Cuando se realiza la venta del producto, ésta es de un solo tipo y tamaño, se requiere determinar cuánto recibirá un productor por la uva que entrega en un embarque, considerando lo siguiente: si es de tipo A, se le cargan 20 céntimos al precio inicial cuando es de tamaño 1; y 30 céntimos si es de tamaño 2. Si es de tipo B, se rebajan 30 céntimos cuando es de tamaño 1, y 50 céntimos cuando es de tamaño 2. Realice un algoritmo para determinar la ganancia obtenida.

Ejercicio 15

El director de una escuela está organizando un viaje de estudios, y requiere determinar cuánto debe cobrar a cada alumno y cuánto debe pagar a la compañía de viajes por el servicio. La forma de cobrar es la siguiente: si son 100 alumnos o más, el costo por cada alumno es de 65 euros; de 50 a 99 alumnos, el costo es de 70 euros, de 30 a 49, de 95 euros, y si son menos de 30, el costo de la renta del autobús es de 4000 euros, sin importar el número de alumnos. Realice un algoritmo que permita determinar el pago a la compañía de autobuses y lo que debe pagar cada alumno por el viaje.

Ejercicio 16

La política de cobro de una compañía telefónica es: cuando se realiza una llamada, el cobro es por el tiempo que ésta dura, de tal forma que los primeros cinco minutos cuestan 1 euro, los siguientes tres, 80 céntimos, los siguientes dos minutos, 70 céntimos, y a partir del décimo minuto, 50 céntimos. Además, se carga un impuesto de 3 % cuando es domingo, y si es otro día, en turno de

mañana, 15 %, y en turno de tarde, 10 %. Realice un algoritmo para determinar cuánto debe pagar por cada concepto una persona que realiza una llamada.

Ejercicio 17

Realiza un programa que pida por teclado el resultado (dato entero) obtenido al lanzar un dado de seis caras y muestre por pantalla el número en letras (dato cadena) de la cara opuesta al resultado obtenido.

- Nota 1: En las caras opuestas de un dado de seis caras están los números: 1-6, 2-5 y 3-4.
- Nota 2: Si el número del dado introducido es menor que 1 o mayor que 6, se mostrará el mensaje: "ERROR: número incorrecto.".

Ejemplo:

Introduzca número del dado: 5
En la cara opuesta está el "dos".

Ejercicio 18

Realiza un programa que pida el día de la semana (del 1 al 7) y escriba el día correspondiente. Si introducimos otro número nos da un error.

Ejercicio 19

Escribe un programa que pida un número entero entre uno y doce e imprima el número de días que tiene el mes correspondiente.

Ejercicio 20

Una compañía de transporte internacional tiene servicio en algunos países de América del Norte, América Central, América del Sur, Europa y Asia. El costo por el servicio de transporte se basa en el peso del paquete y la zona a la que va dirigido. Lo anterior se muestra en la tabla:

Zona	Ubicación	Costo/gramo
1	América del Norte	24.00 euros
2	América Central	20.00 euros
3	América del Sur	21.00 euros
4	Europa	10.00 euros
5	Asia	18.00 euros

Parte de su política implica que los paquetes con un peso superior a 5 kg no son transportados, esto por cuestiones de logística y de seguridad. Realice un algoritmo para determinar el cobro por la entrega de un paquete o, en su caso, el rechazo de la entrega.

3. Estructuras repetitivas

Ejercicio 1

Crea una aplicación que pida un número y calcule su factorial (El factorial de un número es el producto de todos los enteros entre 1 y el propio número y se representa por el número seguido de un signo de exclamación. Por ejemplo: $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$),

Ejercicio 2

Crea una aplicación que permita adivinar un número. La aplicación genera un número aleatorio del 1 al 100. A continuación, va pidiendo números y va respondiendo si el número a adivinar es mayor o menor que el introducido, además de informarle de los intentos que le quedan (tiene 10 intentos para acertarlo). El programa termina cuando se acierta el número (además te dice en cuantos intentos lo has acertado), si se llega al límite de intentos te muestra el número que había generado.

Ejercicio 3

Algoritmo que pida números hasta que se introduzca un cero. Debe imprimir la suma y la media de todos los números introducidos hasta ese momento.

Ejercicio 4

Realizar un algoritmo que pida números (se pedirá por teclado la cantidad de números a introducir). El programa debe informar de cuantos números introducidos son mayores que 0, menores que 0 e iguales a 0.

Ejercicio 5

Algoritmo que pida caracteres e imprima 'VOCAL' si son vocales y 'NO VOCAL' en caso contrario, el programa termina cuando se introduce un espacio.

Ejercicio 6

Escribir un programa que imprima todos los números pares entre dos números que se le pidan al usuario.

Ejercicio 7

Realizar un algoritmo que muestre la tabla de multiplicar de un número introducido por teclado.

Ejercicio 8

Escribe un programa que pida el límite inferior y superior de un intervalo. Si el límite inferior es mayor que el superior lo tiene que volver a pedir. A continuación, se van introduciendo números hasta que introduzcamos el 0. Cuando termine el programa dará las siguientes informaciones:

- La suma de los números que están dentro del intervalo (intervalo abierto).
- Cuántos números están fuera del intervalo.
- Informa si hemos introducido algún número igual a los límites del intervalo (solo si ha habido alguno o no).

Ejercicio 9

Escribe un programa que, dados dos números, uno real (base) y un entero positivo (exponente), saque por pantalla el resultado de la potencia. No se puede utilizar el operador de potencia.

Ejercicio 10

Algoritmo que muestre la tabla de multiplicar de los números 1,2,3,4 y 5.

Ejercicio 11

Escribe un programa que diga si un número introducido por teclado es o no primo. Un número primo es aquel que sólo es divisible entre él mismo y la unidad. Nota: Es suficiente probar hasta la raíz cuadrada del número para ver si es divisible por algún otro número.

Ejercicio 12

Realizar un algoritmo para determinar cuánto ahorrará una persona en un año si al final de cada mes deposita cantidades variables de dinero. Además, se quiere saber cuánto lleva ahorrado cada mes.

Ejercicio 13

Una empresa tiene el registro de las horas que trabaja diariamente un empleado durante la semana (seis días) y requiere determinar el total de éstas, así como el sueldo que recibirá por las horas trabajadas.

Ejercicio 14

Una persona se encuentra en el kilómetro 70 de una carretera, otra se encuentra en el km 150, los coches tienen sentido opuesto y tienen la misma velocidad. Realizar un programa para determinar en qué kilómetro de esa carretera se encontrarán.

Ejercicio 15

Una persona adquirió un producto para pagar en 20 meses. El primer mes pagó 10 €, el segundo 20 €, el tercero 40 € y así sucesivamente (cada mes, el doble que el mes anterior). Realizar un algoritmo para determinar cuánto debe pagar mensualmente y el total de lo que pagó después de los 20 meses.

Ejercicio 16

Una empresa les paga a sus empleados en base a las horas trabajadas en la semana y a un precio por hora dado. Realiza un algoritmo para determinar el sueldo semanal de N trabajadores y, además, calcula cuánto pagó la empresa por los N empleados.

Ejercicio 17

Una empresa les paga a sus empleados con base en las horas trabajadas en la semana. Para esto, se registran los días que trabajó y las horas de cada día. Realice un algoritmo para determinar el sueldo semanal de N trabajadores y además calcule cuánto pagó la empresa por los N empleados.

Ejercicio 18

Hacer un programa que muestre un cronómetro, indicando las horas, minutos y segundos.

Ejercicio 19

Realizar un ejemplo de menú, donde podemos escoger las distintas opciones hasta que seleccionamos la opción de “Salir”.

Ejercicio 20

Mostrar en pantalla los N primeros números primos. Se pide por teclado la cantidad de números primos que queremos mostrar.

UT2 Identificación de los elementos de un programa informático en Java

- 1.- Estructura y bloques fundamentales.
- 2.- Variables.
- 3.- Tipos de datos.
- 4.- Literales.
- 5.- Constantes.
- 6.- Operadores y expresiones.
- 7.- Conversiones de tipo.
- 8.- Comentarios.

1. Estructura y bloques fundamentales de un programa Java

El código fuente de un programa en Java debe estar escrito en un **fichero de texto con extensión ".java"**. Para crear o editar el fichero debe utilizarse un editor de texto sin formato como el bloc de notas de Windows, el editor de texto de Linux, o alguno de los programas y entornos de desarrollo específicos para la creación de aplicaciones.

```
public class HolaMundo {  
    public static void main(String args[]) {  
        System.out.println("Hola Mundo!");  
    }  
}
```

La primera línea de código indica el **nombre de la clase Java** que estamos desarrollando. Todos los programas Java están formados por una o más clases. Es importante tener en cuenta que el nombre de la clase debe corresponder exactamente con el nombre del fichero de texto que contiene el código fuente. En el caso de este ejemplo, el código debe almacenarse en un **fichero de texto denominado "HolaMundo.java"**.

```
public class HolaMundo
```

Las **llaves {}** indican el inicio y fin de cada bloque. Siempre deben ir en pareja, es decir, que por cada llave de apertura debe existir siempre una llave de cierre.

La siguiente línea realiza la declaración del método llamado "main". Cada clase Java que forma una aplicación contendrá uno o varios métodos, que son bloques de código que permiten tenerlo más organizado. Si una determinada clase Java contiene un método con el nombre "main", la ejecución de dicha clase comenzará por el código contenido en ese método.

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

La última línea de código del ejemplo, contiene las sentencias que se van a ejecutar. En este caso tan solo hay una sentencia, pero podría haber todas las que se quisiera. Aquí en concreto se utiliza la llamada a *System.out.println()* que permite mostrar en pantalla una serie de caracteres. El texto mostrado es el indicado entre comillas: "Hola Mundo!".

```
System.out.println("Hola Mundo!");
```

2. Variables

Las **variables** identifican datos mediante un nombre simbólico, haciendo referencia a un espacio de memoria principal en los que se sitúan los datos, para que puedan ser utilizados por el procesador, y así poder hacer cualquier tipo de operación con ellos.

Los **nombres utilizados para identificar a las variables** deben cumplir una serie de **condiciones**:

- No pueden empezar por un dígito numérico.
- No pueden utilizarse espacios, y los únicos caracteres especiales válidos son el guión bajo (_) y el símbolo del dólar (\$).
- Son sensibles a las mayúsculas y minúsculas, es decir, las variables "suma" y "Suma" se consideran variables distintas.

No pueden utilizarse como nombres de variables las **palabras reservadas de Java**, que son las siguientes:

```
abstract default goto
assert do if package synchronized
boolean double implements private this
break else import protected throw
byte enum instanceof public throws
case extends int return transient
catch false interface short true
char final long static try
class finally native strictfp void
const float new super volatile
continue for null switch while
```

Es costumbre empezar los nombres de las variables por una letra minúscula.

Cuando el nombre de una variable está formado por más de una palabra, se suele utilizar una letra mayúscula para distinguir el comienzo de las palabras. Por ejemplo: sumaTotal.

Es recomendable utilizar nombres que hagan referencia al contenido que va a almacenar para facilitar la comprensión del código. Es mucho más claro utilizar el nombre "suma" que "s".

Ejemplos de nombres de variables **válidos**: indice, ventas, compras, saldoGeneral, importetotal, contador_lineas, \$valor, num2.

Ejemplos de nombres de variables **no válidos**: 3valores, suma&total, super, edad media.

Antes de poder utilizar una variable, esta debe ser declarada en el programa. La declaración de variables se debe realizar siguiendo el siguiente formato de sentencia:

```
tipoDato nombreVariable;
```

Donde tipoDato es uno de los tipos de datos básicos o el nombre de una clase (byte, short, int, long, float, double, boolean, char, String, etc), y nombreVariable es el nombre que se desea asignar a la variable siguiendo las normas anteriores.

Es posible declarar más de una variable de un mismo tipo en la misma línea separando los nombres con comas:

```
tipoDato nombreVariable1, nombreVariable2, nombreVariable3;
```

Ejemplos de declaraciones de variables:

```
int num1, num2, suma;  
char letraNIF;  
String saludoInicial;  
boolean mayorEdad;
```

Es posible asignar un valor inicial a las variables en el momento de declararlas. Para inicializar variables se debe seguir el siguiente formato de sentencia:

```
tipoDato nombreVariable = valorInicial;
```

Donde valorInicial puede ser un valor literal, otra variable declarada anteriormente o una expresión combinando valores literales y variables con operadores. El valorInicial debe ser del mismo tipo de dato que la variable que se está declarando.

Ejemplos de declaraciones de variables con inicialización:

```
int num1 = 34;  
int doble = num1 * 2;  
String saludo = "Hola";  
char letraA = 'A', letraB = 'B';
```

En el código del programa es posible asignar valores a las variables que previamente han sido declaradas. Al hacerlo, el valor que guardaba la variable anteriormente se perderá. Se debe utilizar el siguiente formato:

```
nombreVariable = valor;
```

Donde valor puede ser de nuevo un valor literal, una variable declarada anteriormente (puede ser la misma variable) o una expresión combinando valores literales y variables con operadores. El valor debe ser del mismo tipo de dato que la variable a la que se está asignando el nuevo valor.

```

package ejemplos;
public class AsignacionVariables {
    public static void main(String[] args) {
        int a=5, b=0, c;

        b = a * 3; // Se cambia el valor de b a 15
        c = a; // Se guarda en c el valor de a que es 5
        a = a + 6; // Se suma 6 al valor que tenía a. Ahora a vale
        11
        b = a - c; // b guarda 11 - 5 , luego vale 6
        System.out.println("La variable a contiene: " + a);
        System.out.println("La variable b contiene: " + b);
        System.out.println("La variable c contiene: " + c);
    }
}

```

Se muestra lo siguiente:

La variable a contiene: 11
 La variable b contiene: 6
 La variable c contiene: 5

Las variables pueden ser utilizadas dentro del bloque de código en el que han sido declaradas, es decir, dentro de las llaves "{}" que marcan el inicio y el fin de un bloque de código. Se denomina **ámbito** de la variable al bloque de código en el que se declara la variable.

```

package ejemplos;
public class AmbitoVariables {
    static int variableGlobal;

    public static void main(String[] args) {
        int variableDelMain = 10;
        /*Aquí se pueden usar variableGlobal y
        variableDelMain. No se puede usar
        variableDeOtroMetodo */
        System.out.println("variableGlobal " + variableGlobal);
        System.out.println("variableDelMain " + variableDelMain);
        otroMetodo();
    }

    static void otroMetodo() {
        int variableDeOtroMetodo=90;
        /* Aquí se pueden usar variableGlobal y
        variableDeOtroMetodo. No se puede usar
        variableDelMain */
        System.out.println("variableGlobal " + variableGlobal);
        //System.out.println("variableDelMain " + variableDelMain);
        System.out.println("variableDeOtroMetodo " +
        variableDeOtroMetodo);
    }
}

```

3. Tipos de datos básicos

El lenguaje de programación Java permite la utilización de los siguientes tipos de datos básicos:

Números enteros: Representan a los números enteros (sin parte decimal) con signo (pueden ser positivos o negativos). Se dispone de varios tipos de datos, ocupando cada uno de ellos un espacio distinto en memoria. Cuanta más capacidad de almacenamiento, más grande es el rango de valores permitidos, aunque ocupará más espacio de memoria principal. Se dispone de los siguientes tipos:

- **byte:** Ocupan 8 bits (1 byte), permitiendo almacenar valores entre -128 y 127.
- ($2^8 = 256$; necesitas 1 bit para el signo y uno de los valores será 0; $2^7 = 128$)
- **short:** Ocupan 16 bits (2 bytes), permitiendo almacenar valores entre -32.768 y 32.767.
- **int:** Ocupan 32 bits (4 bytes), permitiendo almacenar valores entre -2.147.483.648 y 2.147.483.647. Es el tipo de datos por defecto para los valores numéricos enteros. Este tipo de datos es lo suficientemente grande para almacenar los valores numéricos que vayan a usar tus programas. solo se suelen usar los tipos anteriores si se pueden producir problemas con el espacio de memoria.
- **long:** Ocupan 64 bits (8 bytes), permitiendo almacenar valores entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807.

Números reales: Representan a los números reales con parte decimal y signo positivo o negativo. Hay dos tipos de datos numéricos reales que permiten obtener mayor o menor precisión. Utilizan un método para almacenar los datos que puede ocasionar que el valor original varíe levemente del valor almacenado realmente. Cuanta más precisión se utilice, habrá menor variación.

- **float:** Ocupan 32 bits (4 bytes). Se le denomina de simple precisión. Almacenan valores desde -3.40282347E+38 a + 3.40282347E+38 (E+x significa: 10 elevado a x)
- **double:** Ocupan 64 bits (8 bytes). Se le denomina de doble precisión. Es el tipo de datos por defecto para los valores numéricos reales. Almacenan valores desde: - 1.79769313486231570E+308 a +1.79769313486231570E+308

Tipo	Representación / Valor	Tamaño (en bits)	Valor mínimo	Valor máximo	Valor por defecto
boolean	true o false	1	N.A.	N.A.	false
char	Carácter Unicode	16	\u0000	\uFFFF	\u0000
byte	Entero con signo	8	-128	128	0
short	Entero con signo	16	-32768	32767	0
int	Entero con signo	32	-2147483648	2147483647	0
long	Entero con signo	64	-9223372036854775808	9223372036854775807	0
float	Como flotante de precisión simple Norma IEEE 754	32	$\pm 3.40282347E+38$	$\pm 1.40239846E-45$	0.0
double	Como flotante de precisión doble Norma IEEE 754	64	$\pm 1.79769313486231570E+308$	$\pm 4.94065645841246544E-324$	0.0

Valores lógicos: Representan dos únicos posibles valores: verdadero y falso.

- **boolean:** Ocupan 1 bit, pudiendo almacenar los valores true (verdadero) y false (falso).

Caracteres: Representan las letras, dígitos numéricos y símbolos contenidos en la tabla de caracteres Unicode.

- **char:** Ocupan 16 bits (2 bytes). Permite representar un único carácter, encerrado entre comillas simples, por ejemplo la letra 'A'. (<https://unicode-table.com/en/>)
- **String:** Realmente **no es un tipo de dato básico de Java**, pero por su interés se incluye aquí. Permite representar un conjunto de caracteres, encerrado entre comillas dobles, por ejemplo: "Saludos para todos". (String empieza con mayúsculas porque es el nombre de una clase, una clase es el nombre de una plantilla)

4. Valores literales - Leer en casa

Un valor literal es la representación de un valor fijo en el código fuente de un programa.

Los valores correspondientes a los tipos de datos numéricos enteros (byte, short, int y long) se pueden expresar usando el sistema numérico decimal, octal o hexadecimal. Los números en sistema decimal se expresan de la manera habitual, simplemente escribiendo su valor con los dígitos numéricos, por ejemplo, 284. No se pueden emplear separadores de millares, por lo que para indicar el valor 1.000.000 (un millón) se debe escribir como 1000000. Para representar valores negativos se añade el carácter "-" (guion) delante del número, como es habitual es la escritura normal, por ej -376.

Para representar un valor numérico entero en sistema octal, debe ir precedido del carácter 0 (cero), por ejemplo, el valor 284 se representa en octal como 0434. Asimismo, para representar un valor en el sistema hexadecimal, se debe emplear el prefijo 0x, por lo que el valor 284 se representa en hexadecimal como 0x11C.

Por defecto, los valores literales numéricos enteros se almacenan en memoria con el formato del tipo de dato "int". Si se desea almacenar como "long", con el fin de poder obtener resultados con valores muy altos en los cálculos, se debe emplear el sufijo L en mayúscula o minúscula (sería recomendable utilizar la L mayúscula por el parecido de la letra minúscula con el valor 1). Por ejemplo, el valor 284L correspondería al valor entero 284 utilizando 64 bits (tipo long) para almacenarlo en memoria.

Para representar valores literales de los tipos de datos numéricos reales (float y double) se puede emplear el sistema decimal o la notación científica. En el sistema decimal se expresan los números con parte decimal de la forma usual, utilizando el punto como separador de la parte entera y decimal. En este caso tampoco se puede emplear los separadores de millares. Así, por ejemplo, el valor 21.843,83 se debe expresar como 21843.83 en el código fuente.

Los números reales expresados en notación científica deben emplear la letra "E" o "e" para separar la parte correspondiente al exponente. El valor $7,433 \cdot 10^6$ se debe expresar como 7.433e6 en el código fuente. Si el exponente es negativo se escribe el guion detrás de la letra E, por ejemplo, $7,433 \cdot 10^{-6}$ se expresa como 7.433e-6, y si el valor es negativo se indica el guion al principio, por ejemplo, $-7,433 \cdot 10^6$ se expresa como -7.433e6.

Por defecto, los valores literales numéricos reales se almacenan en memoria con el formato del tipo de dato "double". Si se desea almacenar como "float", con el fin de emplear menos espacio de memoria y necesitar menos precisión en los resultados de los cálculos, se debe emplear el **sufijo F en mayúscula o minúscula**. Por ejemplo, el valor 21843.83F correspondería al valor entero 21843.83 utilizando 32 bits (tipo float) para almacenarlo en memoria, en vez de 64 bits.

Los valores literales de los tipos char (carácter) y String (cadena de caracteres), pueden contener cualquier carácter Unicode. Los valores de tipo char se deben expresar encerrados entre comillas simples (en los teclados españoles habituales se encuentra junto a la tecla del cero), por ejemplo, la letra A se debe expresar como 'A'. Por otro lado, las cadenas de caracteres (tipo String) se expresan entre comillas dobles (en la tecla del 2), por ejemplo, el texto Saludos a todos, se debe escribir como "Saludos a todos".

En caso de que se necesite escribir un carácter de la tabla de caracteres Unicode que no se encuentre en el teclado, se puede hacer indicando el código hexadecimal correspondiente a dicho carácter precedido del modificador \u (utilizando la barra invertida situada en la tecla junto al 1). En todo caso se debe encerrar entre comillas simples o dobles según se vaya a tratar como carácter o dentro de una cadena de caracteres. Por ejemplo, para escribir la letra griega beta (β) se puede emplear '\u00DF', o para escribir la palabra España es posible utilizar "Espa\u00F1a".

El lenguaje de programación Java también soporta un pequeño conjunto de caracteres especiales que se pueden utilizar en los valores literales char y String:
\b (retroceso), \t (tabulador), \n (nueva línea), \f (salto de página), \r (retorno de carro), \" (comilla doble), \' (comilla simple), \\ (barra invertida).

Por ejemplo, la cadena de caracteres "La palabra "hola" es un saludo" se tiene que escribir en el código fuente como: "La palabra \"hola\" es un saludo", para que no confunda las comillas dobles de inicio y fin de la cadena de caracteres.

El carácter especial '\n' permite introducir un salto de línea dentro de una cadena de caracteres. Por ejemplo, "Primera línea\nSegunda línea" mostraría ese texto separado en dos líneas.

Los valores literales para el tipo de dato boolean sólo pueden ser true o false, que corresponden a los valores Verdadero y Falso. Hay que observar que se deben indicar sin comillas de ningún tipo, ya que no son cadenas de caracteres. En un programa, es posible mostrar cualquiera de estos valores literales a través de la salida estándar (terminal, ventana de salida, etc) utilizando la siguiente sentencia:

```
System.out.println(valorLiteral);
```

Donde valorLiteral debe ser el valor que se desea mostrar manteniendo las normas comentadas.

Ejemplos:

```
//Mostrar un valor numérico entero
System.out.print("Número entero: ");
System.out.println(284);

//Mostrar un valor numérico real
System.out.print("Número real: ");
System.out.println(21843.83);

//Mostrar un carácter
System.out.print("Carácter: ");
System.out.println('A');

//Mostrar una cadena de caracteres
System.out.print("Cadena de caracteres: ");
System.out.println("Saludos a todos");

//Mostrar un valor lógico
System.out.print("Valor lógico: ");
System.out.println(true);
```

5. Constantes

Son similares a las variables en cuanto que son datos a los que se hace referencia mediante un nombre y a los que se les asigna un valor. Pero a diferencia de las variables, **a las constantes no se les puede modificar el valor asignado.**

El formato de declaración de las constantes es prácticamente igual que el utilizado para las variables. La única diferencia es que se debe indicar el modificador final delante del tipo de dato que almacenará:

```
final tipoDato NOMBRECONSTANTE = valor;
```

Al igual que en la declaración de variables, el valor que se asigna en la declaración puede ser un valor literal, una variable o una expresión.

Ejemplo:

```
final double PI = 3.1415926536;
```

Por convenio, el nombre de las constantes se escribe en mayúsculas.

El beneficio de usar constantes es evitar la repetición de escribir un mismo valor en el programa y facilitar su modificación.

El empleo de las constantes a lo largo del programa es igual que el utilizado con las variables. Se indica su nombre dentro de cualquier expresión o como parámetro para métodos como `println()`.

```
System.out.println("Perímetro = " + 2*PI*radio);
```

6. Operadores

El lenguaje de programación Java incorpora una serie de operadores que permiten realizar cálculos y escribir expresiones que realicen una serie de operaciones sobre los datos.

6.1. Operadores Aritméticos:

+ (suma)

- (resta)

* (multiplicación)

/ (división entera o con decimales según operandos)

% (resto de la división)

Todos estos operadores aritméticos deben utilizarse con dos operandos, situados delante y detrás de los operadores, pudiéndose encadenar las operaciones. Se pueden incluir espacios para aclarar más el código. Los datos usados como operandos deben ser de alguno de los tipos de datos numéricos (byte, short, int, long, float o double).

Ejemplos:

4 + 3

8 - 5 + 2

6 * 2 / 3

8.5 - 3 + 4.3

El resultado de la división tendrá decimales o no según el tipo de operandos que se utilice. Si los dos son enteros, el resultado no tendrá decimales, pero si al menos uno de los operandos es de tipo numérico real (float o double) el resultado será de ese tipo. Ejemplos:

8 / 2 resulta 4

7 / 2 resulta 3

7.0 / 2 resulta 3.5

7 / 2.0 resulta 3.5

7.4 / 2 resulta 3.7

8 / 2.5 resulta 3.2

8.5 / 2.5 resulta 3.4

Es posible modificar el tipo de dato de cualquier operando indicando delante el nuevo tipo de dato entre paréntesis. Así se hace una conversión de tipo (casting):

```
(double) 7 / 2 resulta 3.5  
7 / (float)2 resulta 3.5
```

El operador resto se debe utilizar con tipos de datos numéricos enteros (*a mí me ha funcionado también con double*). El resultado será el resto de la división entre los dos operandos. Ejemplos:

```
7 % 2 resulta 1  
8 % 3 resulta 2
```

El resultado de cualquier operación aritmética será del tipo de dato más grande que se utilice en los operandos. Por ejemplo, si se hace una operación entre dos números enteros (int) el resultado será del mismo tipo, pero si se hace entre un int y un long es resultado es de tipo long.

Ejemplos:

2147483647 * 2 resulta un dato incorrecto, en concreto -2, porque se están multiplicando dos int y el resultado sobrepasa el límite de los enteros.

2147483647L * 2 resulta 4294967294 porque el primer operando es de tipo long (se ha indicado L al final).

Los caracteres pueden ser utilizados para realizar cálculos aritméticos. En caso de que aparezca algún carácter en una expresión aritmética, se toma el valor numérico que le corresponde a cada carácter en la tabla de codificación Unicode.

Ejemplos:

'A' + 1 resulta 66, ya que el carácter 'A' tiene el código 65.

(char) ('A' + 1) resulta 'B' ya que se ha hecho una conversión de tipos del resultado.

6.2. Operadores Relacionales

Los operadores relacionales permiten comparar dos valores numéricos.

```
> (mayor que)  
>= (mayor o igual que)  
< (menor que)  
<= (menor o igual que)  
== (igual que)  
!= (distinto de)
```

Cada uno de estos operadores relacionales debe emplearse con dos valores numéricos a ambos lados, pudiendo ser dos valores literales o resultados de expresiones aritméticas.

Ejemplos:

```
4 > 3 resulta true.  
7 <= 2 resulta false.  
5 + 2 == 4 + 3 resulta true.  
4 * 3 != 12 resulta false.
```

6.3. Operadores Lógicos

Los operadores lógicos permiten unir valores o expresiones lógicas, obteniendo como resultado si es verdadera o falsa la expresión combinada. Son los siguientes:

```
&& (Y lógico – conjunción)  
|| (O lógico – disyunción)  
! (NO lógico)
```

Los operadores **&&** y **||** deben utilizarse con dos valores o expresiones lógicas a ambos lados, mientras que el operador de negación **!** solo se aplica al valor o expresión lógica que tenga a su derecha. El resultado que se obtiene utilizando estos operadores se obtiene de la siguiente tabla de verdad:

El **operador Y (&&)** resulta true solo si ambos operandos son true:

```
true && true resulta true  
true && false resulta false  
false && true resulta false  
false && false resulta false
```

El **operador O (||)** resulta true si al menos uno de los operandos son true:

```
true || true resulta true  
true || false resulta true  
false || true resulta true  
false || false resulta false
```

El **operador NO (!)** resulta true si el operando es false:

```
!true resulta false.  
!false resulta true.
```

Ejemplos:

```
4 > 3 && 5 <= 5 resulta true,  
porque las dos expresiones son true.  
4 > 3 && 5 != 5 resulta false,  
porque al menos una expresión es false.  
4 > 3 || 5 != 5 resulta true,  
porque al menos una expresión es true.  
4 > 3 && !(5 != 5) resulta true,  
porque las dos expresiones son true.
```

6.4. Operadores de Asignación

Permiten asignar valores a variables. El operador de asignación elemental es el igual (=) que asigna un valor o el resultado de una expresión a una variable, siguiendo el siguiente formato:

```
nombreVariable = valorAsignado;
```

El tipo de dato del valor asignado debe ser del mismo tipo que el de la variable a la que se pretende asignar. En caso de que no sean del mismo tipo, hay que utilizar algún tipo de conversión como el casting, utilizando entre paréntesis el nombre del tipo de dato al que se quiere convertir.

```
int numEntero;  
numEntero = 364; //Correcto  
numEntero = 264.38; //Incorrecto  
numEntero = (int)264.83; //Correcto, se asigna el valor 264  
(lo trunca)
```

También se puede utilizar el operador de asignación para guardar en una variable el resultado de una expresión que dé como resultado un valor del mismo tipo que la variable.

```
nombreVariable = Expresión;  
int numEntero;  
numEntero = 364 * 2 + 266;
```

Además del operador igual (=), se pueden emplear otros operadores de asignación, que a la vez que asignan un valor realizan un cálculo:

- +=** (le suma a la variable un valor y guarda el resultado en la misma variable).
- =** (le resta a la variable un valor y guarda el resultado en la misma variable).
- *=** (multiplica la variable por un valor y guarda el resultado en la misma variable).
- /=** (divide la variable por un valor y guarda el resultado en la misma variable).
- %=** (obtiene el resto de dividir la variable por un valor y guarda el resultado en la misma variable).

```
calculo = 5;  
calculo += 6; //Incrementa en 6 el valor de la variable:  
calculo = calculo +6  
System.out.println(calculo); //Muestra 11  
//Duplica el valor de cálculo: calculo = calculo*2  
calculo *= 2;  
System.out.println(calculo); //Muestra 22
```

Por tanto, una sentencia como **a+=3** es lo mismo que **a=a+3**.

6.5. Operadores Incrementales

Son los operadores que nos permiten incrementar las variables en una unidad

```
++ (incrementa en 1 el valor de una variable).  
-- (decrementa en 1 el valor de una variable).
```

Ejemplo:

```
num1 = 3;  
num2 = 7;  
num1++; //Suma 1 a num1  
System.out.println(num1); //Muestra 4  
num2--; //resta 1 a num2  
System.out.println(num2); //Muestra 6
```

Por tanto, una sentencia como `x++` es lo mismo que `x=x+1`, y `x--` es lo mismo que `x=x-1`.

Hay dos formas de utilizar el incremento y el decremento, `x++` ó `++x`. La diferencia estriba en el modo en el que se comporta la asignación.

Ejemplo:

```
int x=5, y=5, z;  
z=x++; /* z vale 5, x vale 6 porque primero se asigna  
el valor de x a z después se incrementa x */  
z=++y; /* z vale 6, y vale 6 porque primero incrementa  
la variable y, después se asigna el valor a z */
```

6.6. Operador Condicional. - (ha dicho que nos cuesta entenderlo)

(*No es lo mismo un operador condicional que una sentencia*)

Existe un operador condicional, que permite asignar a una variable un valor u otro dependiendo de una expresión condicional. El formato es el siguiente:

```
variable = condición ? valor1 : valor2;
```

La condición que se indica debe ser una variable booleana o una expresión condicional que resulte un valor de tipo booleano (true o false).

Ejemplo:

```
mensaje = (num1 >= 0) ? "Positivo" : "Negativo";
```

Si se mostrara en pantalla posteriormente el valor de la variable mensaje, aparecerá "Positivo" si el valor de la variable num1 es mayor o igual que cero, y "Negativo" en caso contrario.

Precedencia de operadores en Java

(Los operadores en una misma fila tienen la misma precedencia.La precedencia disminuye según se baja en la tabla.)

()	[]	.			
- (unario)	--	++	!		
new (tipo)					
*	/	%			
+	-				
>	>=	<	<=		
==	!=				
&&					
 					
? :					
=	+=	-=	*=	/=	%=

Descripción	Operadores
operadores posfijos	op++ op--
operadores unarios	++op --op +op -op ~ !
multiplicación y división	* / %
suma y resta	+ -
desplazamiento	<< >> >>>
operadores relacionales	< > <= =>
equivalencia	== !=
operador AND	&
operador XOR	^
operador OR	
AND booleano	&&
OR booleano	
condicional	?:
operadores de asignación	= += -= *= /= %= &= ^= = <<= >>= >>>=

7. Conversiones de tipo

Java es un lenguaje fuertemente tipificado, lo que significa que es bastante estricto al momento de asignar valores a una variable (*esto ya no suele ser así, pero en versiones previas lo era*). El compilador sólo permite asignar un valor del tipo declarado en la variable; no obstante, en ciertas circunstancias es posible realizar conversiones que permiten almacenar en una variable un tipo diferente al declarado. En Java es posible realizar conversiones en todos los tipos básicos, con excepción de boolean, que es incompatible con el resto de los tipos.

Las conversiones de tipo pueden realizarse de dos maneras: implícitamente y explícitamente.

7.1. Conversión implícita

Las conversiones implícitas se realizan de manera automática, es decir, el valor o expresión que se va a asignar a una variable es convertido automáticamente por el compilador, antes de almacenarlo en la variable.

Para que una conversión pueda realizarse de manera automática (implícitamente), el tipo de la variable destino debe ser de tamaño igual o superior al tipo de origen, si bien esta regla tiene dos excepciones:

- Cuando la variable destino es entera y el origen es decimal (float o double), la conversión **no** podrá ser automática.
- Cuando la variable destino es "char" y el origen es numérico; independientemente del tipo específico, la conversión **no** podrá ser automática.

Ejemplos de conversiones implícitas:

```
// Declaraciones.  
int j = 5, i;  
short s = 10;  
char c = 'ñ';  
float f;  
  
// Conversiones implícitas.  
i = c; // Conversión implícita de char a int.  
f = j; // Conversión implícita de int a float.  
i = s; // Conversión implícita de short a int.
```

Los siguientes ejemplos de conversión implícita provocarán un error:

```
// Declaraciones.  
int i;  
long l = 20;  
float ft = 2.4f;  
char c;  
byte b = 4;
```

```

// Conversiones implícitas.
i = l; // Error, el tipo destino es menor al tipo origen.
c = b; // Cuando la variable destino es "char" y el origen es
numérico; independientemente del tipo específico, la
conversión no podrá ser automática.
i = ft; // Cuando la variable destino es entera y el origen es
decimal (float o double), la conversión no podrá ser
automática.

```

7.2. Conversión explícita

Cuando no se cumplan las condiciones para una conversión implícita, ésta podrá realizarse de manera explícita utilizando la expresión:

```
variable_destino = (tipo_destino) dato_origen;
```

Con esta expresión se obliga al compilador que convierta "dato_origen" a "tipo_destino" para que pueda ser almacenado en "variable_destino". A esta operación se le conoce como "casting" o "conversión".

Ejemplos de conversiones explícitas:

```

// Declaraciones.
char c;
byte b;
int i = 400;
double d = 34.6;
// Conversiones explícitas.
c = (char)d; // Se elimina la parte decimal (trunca), no se redondea.
b = (byte)i; // Se provoca una pérdida de datos, pero la conversión es
posible.

public class Conversiones {
    public static void main (String [] args) {
        int a = 2;
        double b = 3.0;
        float c = (float) (20000*a/b + 5);
        System.out.println("Valor en formato float: " + c);
        System.out.println("Valor en formato double: " +
                           (double) c);
        System.out.println("Valor en formato byte: " +
                           (byte) c);
        System.out.println("Valor en formato short: " +
                           (short) c);
        System.out.println("Valor en formato int: " + (int) c);
        System.out.println("Valor en formato long: " + (long) c);
    }
}

```

8. Comentarios

Cuando se escribe código en general es útil realizar comentarios explicativos. Los comentarios no tienen efecto como instrucciones para el ordenador, simplemente sirven para que cuando una persona lea el código pueda comprender mejor lo que lee. En Java existen dos formas de poner comentarios.

La primera es cuando la línea de comentario solo ocupa una línea de código. En este caso se pone dos barras inclinadas (//) antes del texto.

```
// Comentario de una línea
```

En el caso de comentarios de más de una línea se pone /* para empezar y */ para finalizar.

El código nos quedará de la siguiente forma:

```
/* Comentario  
de varias  
líneas */
```

UT2 Literales en Java

1. LITERAL JAVA

Un literal Java es un valor de tipo entero, real, lógico, carácter, cadena de caracteres o un valor nulo (null) que puede aparecer dentro de un programa. Por ejemplo: 150, 12.4, "ANA", null, 't'.

1.1 LITERAL JAVA DE TIPO ENTERO

Puede expresarse en decimal (base 10), octal (base 8) y hexadecimal (base 16). El signo + al principio es opcional y el signo - será obligatorio si el número es negativo.

El tipo de un literal entero es **int** a no ser que su valor absoluto sea mayor que el de un int o se especifique el sufijo L o L en cuyo caso será de tipo **long**.

Literal Java de tipo entero en Base decimal

Está formado por 1 o más dígitos del 0 al 9.
El primer dígito debe ser distinto de cero.

Por ejemplo:

1234	literal java entero de tipo int
1234L	literal java entero de tipo long
123400000000	literal java entero de tipo long

Literal Java de tipo entero en Base octal

Está formado por 1 o más dígitos del 0 al 7.
El primer dígito debe ser cero.

Por ejemplo:

01234
025

Literal Java de tipo entero en Base hexadecimal

Está formado por 1 o más dígitos del 0 al 9 y letras de la A a la F (mayúsculas o minúsculas).
Debe comenzar por 0x ó 0X.

Por ejemplo:

0x1A2
0x430
0xf4

1.2 LITERAL JAVA DE TIPO REAL

Son números en base 10, que deben llevar un parte entera, un punto decimal y una parte fraccionaria. Si la parte entera es cero, puede omitirse. El signo + al principio es opcional y el signo - será obligatorio si el número es negativo.

Por ejemplo:

12.2303
-3.44
+10.33
0.456
.96

También pueden representarse utilizando la notación científica. En este caso se utiliza una E (mayúscula o minúscula) seguida del exponente (positivo o negativo). El exponente está formado por dígitos del 0 al 9.

Por ejemplo, el número en notación científica 14×10^{-3} se escribe: 14E-3

Más ejemplos de literal Java de tipo real:

2.15E2 -> 2.15×10^2
.0007E4 -> 0.0007×10^4
-50.445e-10 -> -50.445×10^{-10}

El tipo de estos literales es siempre **double**, a no ser que se añada el sufijo F ó f para indicar que es float.

Por ejemplo:

2.15 literal real de tipo double
2.15F literal real de tipo float

También se pueden utilizar los sufijos d ó D para indicar que el literal es de tipo double:

12.002d literal real de tipo double

1.3 LITERAL JAVA DE TIPO CARÁCTER

Contiene un solo carácter encerrado entre comillas simples.

Es de tipo **char**.

Las secuencias de escape se consideran literales de tipo carácter.

Por ejemplo:

'a'
'4'
\n
\u0061'

1.4 LITERAL JAVA DE CADENAS DE CARACTERES

Está formado por 0 ó más caracteres encerrados entre comillas dobles.
Pueden incluir secuencias de escape.

Por ejemplo:

“Esto es una cadena de caracteres”

“Pulsa ‘C’ para continuar”

“” -> cadena vacía

“T” -> cadena de un solo carácter, es diferente a ‘t’ que es un carácter

Las cadenas de caracteres en Java son objetos de tipo **String**.

Nota: Sobre atajos de teclado en Eclipse:

Ctrl+May+L nos visualiza en Eclipse todos los atajos de teclado

UT2 Ampliación Operadores

1. Operadores lógicos

Operador	Descripción	Uso	Retorna cierto si
&&	and	op1 && op2	op1 y op2 son true. Solo evalúa op2 si op1 es true
	or	op1 op2	op1 o op2 es true. Solo evalúa op2 si op1 es false
!	not	!op	op es false

2. Operadores lógicos a nivel de bits

Operador	Descripción	Uso
&	and	op1 & op2
	or	op1 op2
^	or exclusivo	op1 ^ op2
~	complemento	~op1

El operador **AND bit por bit (&)** realiza un and lógico entre cada par de bits paralelos que forman los operandos. Si ambos bits valen 1 el bit resultante vale 1, de lo contrario vale 0. Por ejemplo:

```
int bits_orig = 6;           // binario 00000000 00000000 00000000 00000110
int bits_bandera = 5;         // binario 00000000 00000000 00000000 00000101
int respuesta = (bits_orig & bits_bandera);           // 000000 00000100
```

El operador **OR bit por bit (|)** realiza un or lógico entre cada par de bits paralelos que forman los operandos. Si alguno de los bits vale 1 el bit resultante vale 1, de lo contrario vale 0. Por ejemplo:

```
int bits_orig = 6;           // binario 00000000 00000000 00000000 00000110
int bits_bandera = 5;         // binario 00000000 00000000 00000000 00000101
int respuesta = (bits_orig | bits_bandera);           // 000000 00000111
```

El operador **XOR bit por bit (^)** realiza un or exclusivo entre cada par de bits paralelos que forman los operandos. Si ambos bits tienen valores opuestos el bit resultante vale 1, de lo contrario vale 0. Por ejemplo:

```
int bits_orig = 6;           // binario 00000000 00000000 00000000 00000110
int bits_bandera = 5;         // binario 00000000 00000000 00000000 00000101
int respuesta = (bits_orig ^ bits_bandera);           // 000000 00000011
```

El operador **NOT bit por bit (~)** invierte los bits del operando. Por ejemplo:

```
byte valor = 6;             // binario 00000110
byte respuesta = (~ valor); // binario 11111001
```

3. Operadores de desplazamiento de bits

Existen muchas operaciones que requieren desplazar los valores de los bits que componen un número a la izquierda o a la derecha. Los operadores de desplazamiento de bits suelen utilizarse para llevar a cabo operaciones muy rápidas de multiplicación y de división de enteros. Un desplazamiento a la izquierda equivale a una multiplicación por 2 y un desplazamiento a la derecha a una división por 2.

Un desplazamiento no es una rotación. A medida que se desplazan los bits hacia un extremo se van rellenando con ceros por el otro extremo. Los bits que salen se pierden.

Operador	Uso	Operación
<<	op1 << op2	Desplaza a la izquierda los bits del primer operando op1 tantas veces como indica el segundo operando op2(por la derecha siempre entra un cero).
>>	op1 >> op2	Desplaza a la derecha los bits del primer operando op1 tantas veces como indica el segundo operando op2(por la izquierda entra siempre el bit más significativo anterior).
>>>	op1 >>> op2	Desplaza a la derecha los bits del primer operando op1 tantas veces como indica el segundo operando op2(sin signo- por la izquierda entra siempre un cero)

A diferencia de C y C++, Java siempre conserva el bit de signo (el bit izquierdo) después de hacer un desplazamiento. Este tipo de desplazamiento se conoce como desplazamiento aritmético o desplazamiento de extensión de signo. Las siguientes instrucciones desplazan un valor, primero dos bits a la izquierda y luego dos bits a la derecha, con los operadores de desplazamiento de bits de Java:

```
int original = -3;
//                                binario 10000000 00000000 00000000 00000011
int izquierda = original << 2;
// izquierda vale -12          binario 10000000 00000000 00000000 00001100
int derecha = izquierda >> 2;
// derecha vale -3            binario 10000000 00000000 00000000 00000011
int x = 7;
// x vale 7                  binario 00000000 00000000 00000000 00000111
x = x << 1;
// x vale 14                 binario 00000000 00000000 00000000 00001110
```

El **operador >>>** realiza un desplazamiento a la derecha similar al operador **>>**. La diferencia es que el operador **>>>** no conserva el bit de signo, sino que coloca el bit de signo en cero, a diferencia del operador **>>**, que lo pone en uno cuando es negativo, de forma que permanezca negativo.

Normalmente se utiliza el operador **>>>** cuando el operador no representa una cantidad con signo, sino que es una máscara de bits de algún tipo en el que no importa el bit de signo. El siguiente ejemplo muestra el uso del operador **>>>** para desplazar el valor 256 dos bits a la derecha, dando como resultado 64:

```
int mascara = 256;
// mascara 256                binario 00000000 00000000 00000001 00000000
int respuesta = mascara >>> 2;
// respuesta 64               binario 00000000 00000000 00000000 01000000
```

Nota:

Java convierte a tipo int los tipo short y byte antes de realizar el desplazamiento de bits. Esto significa que el tipo byte tendrá 32 bits en lugar de 8. Después de ejecutar el desplazamiento trunca el dato para convertirlo al tipo original.

UT2 Java Scanner para la lectura de datos

Java Scanner para lectura de datos

La clase Scanner está disponible a partir de Java 5 y facilita la lectura de datos en los programas Java. Primero veremos varios ejemplos de lectura de datos en Java con Scanner y después explicaremos en detalle cómo funciona.

Para utilizar Scanner en un programa tendremos que hacer lo siguiente:

1. Escribir el import

La clase Scanner se encuentra en el paquete java.util por lo tanto se debe incluir al inicio del programa la instrucción:

```
import java.util.Scanner;  
import java.util.*;
```

2. Crear un objeto Scanner

Tenemos que crear un objeto de la clase Scanner asociado al dispositivo de entrada. Si el dispositivo de entrada es el teclado escribiremos:

```
Scanner sc = new Scanner(System.in);  
//(System.in representa la entrada de datos por defecto en mi  
programa)
```

Se ha creado el objeto sc asociado al teclado representado por System.in, una vez hecho esto podemos leer datos por teclado.

Ejemplos de lectura:

Para leer podemos usar el método nextXxx() donde Xxx indica el tipo, por ejemplo nextInt() para leer un entero, nextDouble() para leer un double, etc.

Ejemplo de lectura por teclado de un número entero:

```
int n;  
System.out.print("Introduzca un número entero: ");  
n = sc.nextInt();
```

Ejemplo de lectura de un número de tipo double:

```
double x;  
System.out.print("Introduzca número de tipo double: ");  
x = sc.nextDouble();
```

Ejemplo de lectura de una cadena de caracteres (una línea hasta intro):

```
String s;  
System.out.print("Introduzca texto: ");  
s = sc.nextLine();
```

Ejemplo de programa Java con lectura de datos con Scanner:

El programa pide que se introduzca el nombre de la persona y lo muestra por pantalla. A continuación lee por teclado el radio de una circunferencia de tipo double y muestra su longitud. Además lee un entero y muestra su cuadrado.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in); //crear un objeto Scanner
        String nombre;
        double radio;
        int n;

        System.out.print("Introduzca su nombre: ");
        nombre = sc.nextLine(); //leer un String/linea
        System.out.println("Hola " + nombre + "!!!");
        System.out.print("Introduzca el radio de la circunferencia: ");
        radio = sc.nextDouble(); //leer un double
        System.out.println("Longitud de la circunferencia: " +
        2*Math.PI*radio);
        System.out.print("Introduzca un número entero: ");
        n = sc.nextInt(); //leer un entero
        System.out.println("El cuadrado es: " + Math.pow(n,2));
    }
}
```

Funcionamiento de la clase Java Scanner.

De forma resumida podemos decir que cuando se introducen caracteres por teclado, el objeto Scanner toma toda la cadena introducida y la divide en elementos llamados tokens (los guarda en un buffer interno).

El carácter predeterminado que sirve de separador de tokens (*en el .nextLine()*) es el espacio en blanco. Por ejemplo, si introducimos:

Esto es un ejemplo, lectura de datos.

Scanner divide la cadena en los siguientes tokens:

Esto
es
un
ejemplo,
lectura
de
datos.

Si introducimos la cadena:

12 20.001 Lucas w

Los tokens que se crean son:

12
20.001
Lucas
w

A continuación, utilizando los métodos que proporciona la clase Scanner se puede acceder a esos tokens y trabajar con ellos en el programa.

Ya hemos visto el método nextXxx(). Además la clase Scanner proporciona otros métodos, algunos de los más usados son:

METODO	DESCRIPCIÓN
nextXxx()	Devuelve el siguiente token como un tipo básico. Xxx es el tipo. Por ejemplo, nextInt() para leer un entero, nextDouble para leer un double, etc.
next()	Devuelve el siguiente token como un String.
nextLine()	Devuelve la línea entera como un String. Elimina el final \n del buffer
hasNext()	Devuelve un boolean. Indica si existe o no un siguiente token para leer.
hasNextXxx()	Devuelve un boolean. Indica si existe o no un siguiente token del tipo especificado en Xxx, por ejemplo hasNextDouble()
useDelimiter(String)	Establece un nuevo delimitador de token.

Cómo limpiar el buffer de entrada en Java

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.

Es decir, cuando en un programa introducimos un dato y pulsamos el intro como final de entrada, el carácter intro también pasa al buffer de entrada.

Buffer de entrada si se introduce un 5: 5\n

En esta situación, la instrucción:

```
n = sc.nextInt();
```

Asigna a n el valor 5 pero el intro permanece en el buffer.

Buffer de entrada después de leer el entero: \n

Si ahora se pide que se introduzca por teclado una cadena de caracteres:

```
System.out.print("Introduzca su nombre: ");
nombre = sc.nextLine(); //leer un String
```

El método nextLine() extrae del buffer de entrada todos los caracteres hasta llegar a un intro y elimina el intro del buffer.

En este caso asigna una cadena vacía \n a la variable nombre y limpia el intro. Esto provoca que el programa no funcione correctamente, ya que no se detiene para que se introduzca el nombre.

Solución:

Se debe **limpiar el buffer** de entrada si se van a leer datos de tipo carácter a continuación de la lectura de datos numéricos.

La forma más sencilla de limpiar el buffer de entrada en Java es ejecutar la instrucción:

```
sc.nextLine();
```

Lo podemos comprobar si cambiamos el orden de lectura del ejemplo y leemos el nombre al final:

```
import java.util.Scanner;
public class Ejemplo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String nombre;
        double radio;
        int n;

        System.out.print("Introduzca el radio de la circunferencia: ");
        radio = sc.nextDouble();
        System.out.println("Longitud de la circunferencia: " +
        2*Math.PI*radio);
        System.out.print("Introduzca un número entero: ");
        n = sc.nextInt();
        System.out.println("El cuadrado es: " + Math.pow(n, 2));
        System.out.print("Introduzca su nombre: ");
        nombre = sc.nextLine(); //leemos el String después del double
        System.out.println("Hola " + nombre + "!!!");
    }
}
```

Si lo ejecutamos obtendremos:

```
Introduzca el radio de la circunferencia: 34
Longitud de la circunferencia: 213.62830044410595
Introduzca un número entero: 3
El cuadrado es: 9.0
Introduzca su nombre: Hola !!!
```

Comprobamos que no se detiene para pedir el nombre.

Una solución es escribir la instrucción

```
sc.nextLine();
```

Después de la lectura del int y antes de leer el String:

```
n = sc.nextInt();
System.out.println("El cuadrado es: " + Math.pow(n, 2));
sc.nextLine();
System.out.print("Introduzca su nombre: ");
nombre = sc.nextLine();
System.out.println("Hola " + nombre + "!!!");
```

Ahora la ejecución es correcta:

```
Introduzca el radio de la circunferencia: 23
Longitud de la circunferencia: 144.51326206513048
Introduzca un número entero: 5
El cuadrado es: 25.0
Introduzca su nombre: Lucas
Hola Lucas!!!
```

Hay una solución más elegante: leer siempre líneas con sc.nextLine(); y realizar la conversión al tipo correspondiente después:

```
import java.util.Scanner;
public class Scanner4 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String nombre, radioS, nS;
        double radio;
        int n;

        System.out.print("Introduzca el radio de la circunferencia: ");

        radioS = sc.nextLine();
        radio = Double.parseDouble(radioS);

        //https://docs.oracle.com/en/java/javase/11/docs/api/java.base/
        //java/lang/Double.html#parseDouble(java.lang.String)

        System.out.println("Longitud de la circunferencia: " +
        2*Math.PI*radio);
        System.out.print("Introduzca un numero entero: ");

        nS = sc.nextLine();
        n = Integer.parseInt(nS);

        System.out.println("El cuadrado es: " + Math.pow(n, 2));

        System.out.print("Introduzca su nombre: ");
        nombre = sc.nextLine(); //leemos el String despues del double
        System.out.println("Hola " + nombre + "!!!!");

        sc.close();
    }
}
```

UT2 Ejercicios de expresiones

Ejemplos:

Expresión algebraica	Expresión aritmética algorítmica
$x^2 + y^4$	$x^{**2} + y^{**4}$
$\frac{x^2}{a^3 + b^3}$	$x^{**2} / (a^{**3} + b^{**3})$
$u + \frac{x^2}{y}$	$u + x^{**2} / y$
$\frac{a + b}{a + \frac{c^2}{d + e}}$	$(a + b) / (a + c^{**2} / (d + e))$

Deducir el valor de las expresiones siguientes: Siendo:

A = 5; B = 25; C = 10

- | | |
|----------------|-----|
| 1. A + B / C | 7,5 |
| 2. (A + B) / C | 3 |
| 3. A + B % C | 10 |

Si el valor de A es 4, el valor de B es 5 y el valor de C es 1, evaluar las siguientes expresiones:

- | | |
|--|-------------|
| 1. B * A - B * B / 4 * C | |
| 20-25/4*1 | 13,75 |
| 2. (A * B) / 3 * 3 | |
| 20/3*3 | 20 |
| 3. (((B + C) / 2 * A + 10) * 3 * B) - 6 | |
| ((6/2*4+10)*3*5)-6 | 22*15-6=324 |

Realizar las conversiones de expresiones matemáticas a expresiones algorítmicas indicando el orden de ejecución de cada una de ellas

$\frac{m+n}{p-q}$	$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$	$\frac{m+n/p}{a-r/5}$
-------------------	--------------------------------------	-----------------------

(m+n)/(p-q)
 $(-b + (\text{Math.pow}(\text{Math.pow}(b, 2) - 4*a*c, 1/2))) / (2*a)$
 $(-b - (\text{Math.pow}(\text{Math.pow}(b, 2) - 4*a*c, 1/2))) / (2*a)$
 $((m+n)/p) / ((a-r)/5)$

Evaluar las expresiones lógicas aplicando la jerarquía de operadores.

1. $((A * B) < (B + C)) \&\& (A == C)$ $(12 < 6) \&\& (3 == 2)$ FALSE	A=3, B=4 y C=2
2. $((A + B) > C) ((B / D > B))$ $(7 > 3) (5/5 > 5)$ T (1>5) T	A=2, B=5, C=3 y D=5
3. $(A/B) * C + (A / B)$ $2*3 + (2) 8$	A = 4, B = 2, C = 3
4. $\pi * X * X > Y 2 * \pi * X <= Z$ 3.141592*1*1 > 4 2*3.141592*1 <= 10 F T T	X=1, Y=4, Z=10, $\pi = 3.141592$ E=2.718281
5. $X > 3 \&\& Y == 4 X + Y <= Z$ F && T 5 <= 10 F T T	
6. $X > 3 \&\& (Y == 4 X + Y <= Z)$ F && (F 5 <= 10) F && (F) F	
7. $!(Y/2 == 2 * X) \&\& !(Y < \pi * E * Z)$!(4/2 == 2) && !(4 < 3.141592 - 27.18281) !(T) && !(F) F && T F	
8. $A == B \% C$ 5 == 4%3 5 == 1 F	A=5, B=4, C=3, X=0.05, Y=2.3
9. $6/C < C \% 6$ 6/3 < C \% 6 2 < 3 T	
10. $C + B - 1 != A B >= -B * A \&\& A * A <= 10$ 3 + 4 - 1 != 5 4 >= -4 * 5 && 5 * 5 <= 10 6 != 5 4 >= -20 && 25 <= 10 T T && F T F T	
11. $B \% A / C$ 4 % 5 / 3 4/3	
12. $!(X * A > Y / B)$!(0.05 * 5 > 2.3/4) !(0.25 > 0.575) !(F) T	

Convertir en expresiones numéricas los siguientes enunciados.

1. Elabore una expresión que sólo permita valores entre 1 y 10.

$$(x >= 1 \&\& x <= 10)$$

2. Elabore una expresión que permita valores entre 1 y 3, y entre 5 y 7 exclusivamente.

$$(x >= 1 \&\& x <= 3) || (x >= 5 \&\& x <= 7)$$

3. Elabore una expresión que permita edades entre 18 y 25 años.

$$(edad >= 18 \&\& edad <= 25)$$

Comprobar qué resultado se obtiene en estas 2 expresiones. ¿Son el mismo resultado? Justificar la respuesta

1. $7 == 4 + 3 || 6 < 2 \&\& 5 >= 8$

T || F && F

T || F T

2. $(7 == 4 + 3 || 6 < 2) \&\& 5 >= 8$

(T || F) && F

T && F F

UT3 Estructuras de Control

1. Estructuras de selección.

1.1 Estructura If.

1.2 Switch.

1.3 Operador Condicional.

2. Estructuras de repetición.

2.1 Bucle For.

2.2 Bucle While.

2.3 Bucle Do-While

1.- Estructuras de selección.

Las estructuras de selección son aquellas que se utilizan cuando se quiere ejecutar una instrucción o bloque de instrucciones si se cumple una determinada condición, de ahí que también se conozcan con el nombre de condicionales.

1.1 Estructura If.

Permite ejecutar una instrucción (o secuencia de instrucciones) si se da una condición. La sentencia **if** es la sentencia básica de selección y existen tres variantes: simple, doble y selección múltiple, pero las 3 se basan en la misma idea.

a) **Selección Simple.** Solo tiene la parte positiva de la selección. Su sintaxis es:

```
if (condición){  
    sentencias;  
}
```

Donde **condición** es una expresión booleana y **sentencias** representa una sentencia o bloque de sentencias, las cuales se ejecutarán si el valor de la condición es True (cierta).

Si es una **sentencia única**, se pueden quitar las llaves

Ej:

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

b) **Selección Doble.** Se añade una parte else a la sentencia if la cual se ejecutará si la condición es falsa. Su sintaxis es:

```
if (condición){  
    sentencias1;  
}  
else{  
    sentencias2;  
}
```

Ej:

```
if (a>b)
    System.out.println ("El número mayor es " + a);
else
    System.out.println("El número mayor es " + b+ " o son
iguales");
```

c) **Selección Múltiple.** Es habitual cuando hay más de una condición. Su sintaxis es:

```
if (condición1){
    sentencias1;
}
else if (condición2){
    sentencias2;
}
else if (condición3){
    sentencias3;
}
else {
    sentencias;
}
```

Ej:

```
if (mes == 12 || mes == 1 || mes == 2)
    System.out.println ( "Inviero");
else if (mes == 3 || mes == 4 || mes == 5)
    System.out.println( "Primavera");
else if (mes == 6 || mes == 7 || mes == 8)
    System.out.println ("Verano");
else
    System.out.println ( "Otoño");
```

1.2 Switch.

Es una estructura de selección múltiple más cómoda de leer y utilizar que el else-if. Selecciona un bloque de sentencias dependiendo del valor de una expresión.

Su sintaxis es:

```
switch (expresion) {  
    case valor1:sentencias;  
    break;  
    case valor2:sentencias;  
    break;  
    case valor3:sentencias;  
    break;  
    ...  
    default: sentencias;  
    break;  
}
```

Donde **expresión** tiene que tomar un valor entero o un carácter. A partir de la versión 7 de Java también se permite que la expresión sea un String (cadena), **break** indica que ha acabado la ejecución de ese caso y seguiría ejecutando la sentencia siguiente al switch y **default** es opcional y se ejecutará cuando la expresión tome un valor que no esté recogido en los distintos casos especificados.

Ej:

```
switch (mes) {  
    case 4:  
    case 6:  
    case 9:  
    case 11: dias = 30;  
              break;  
    case 2:  dias = 28;  
              break;  
    default: dias = 31;  
              break;  
}
```

Es posible juntar distintos casos, dejándolos en blanco y especificando las instrucciones en el último de los casos de grupo.

En el ejemplo, si la variable mes toma los valores 4, 6, 9 u 11 se hace en los 4 casos lo mismo, se le asigna a la variable día el valor 30.

1.3 Operador Condicional.

El operador condicional es un operador ternario, es decir, consta de tres operandos y su función es asignar un valor entre dos posibles a una variable si se cumple o no una condición.

Su sintaxis es la siguiente:

```
tipo variable = (condicion) ? v_cond_true: v_cond_false;
```

Realmente es un if...else simple que podemos utilizar si solo queremos asignar un valor a una variable si se cumple o no una condición.

Ejemplo:

```
public class EjemploOperadorCondicional
{
    public static void main(String[] args)
    {
        int edad = 18;

        //Utilizando el operador condicional
        String resultado = (edad >= 18) ? "Mayor de edad." : "Menor de edad.";
        System.out.println(resultado);

        //Utilizando if...else
        if(edad >= 18)
        {
            System.out.println("Mayor de edad.");
        }
        else
        {
            System.out.println("Menor de edad.");
        }
    }
}
```

2.- Estructuras de repetición.

Estas estructuras se utilizan para repetir un bloque de sentencias un número de veces. Son también llamadas sentencias de iteración o bucles.

Los tipos de bucles que hay son: **for**, cuando se sabe el número de veces que se va realizar y **while o do-while**, cuando no se conoce de antemano el número de veces que se repetirá el bloque de sentencias.

2.1 Bucle For (PARA)

La sintaxis de la sentencia es:

```
for(inicialización;condición;incremento/decremento){  
    sentencias;  
}
```

Dónde:

- **Inicialización** se realiza solo una vez, **antes de la primera iteración**.
- **Condición** se comprueba **cada vez, antes de entrar** al bucle y si es cierta entra, si no lo es se termina la ejecución de la sentencia y pasa a ejecutarse la siguiente sentencia al for.
- **Incremento/decremento** se realiza siempre **después de terminar de ejecutar las sentencias del cuerpo de la iteración** y antes de volver a comprobar la condición de nuevo.

Ej:

```
int i;  
for (i=0; i<5; i++)  
    System.out.println (i);
```

El **funcionamiento** de esta instrucción seria el siguiente:

1º. Se inicializa la i.

En el ejemplo i toma el valor 0.

2º. Se comprueba si cumple la condición, si se cumple entra en el bucle pero si no la cumple no entraría.

En el ejemplo sí la cumple pues $0 < 5$.

3º. Se ejecuta el bloque de sentencias que pertenezcan al for.

En el ejemplo escribirá en pantalla un 0

4º. Vuelve al principio del bucle, incrementa la i.

En el ejemplo se incrementa en 1, con lo cual pasa a tomar valor 1.

5º. Se repite el proceso desde el paso 2, hasta que deje de cumplirse la condición.

En el ejemplo se repetirá hasta que i tome el valor de 5 pues no se cumple que $5 < 5$. Los siguientes dos ejemplos son un bucle infinito y un bucle que nunca se llega a ejecutar.

Bucle infinito: `for (i=1; i<5; i--)` pues nunca se dejará de cumplir que $i < 5$.

Bucle que no ejecuta nunca sus sentencias: `for (i=5; i<3; i++)` pues desde el principio no se cumple la condición pues 5 no es menor que 3.

2.2 Bucle While (MIENTRAS)

La sintaxis de la sentencia es:

```
while (condición){
    sentencias;
}
```

Donde **condición** es una expresión booleana que se evalúa al principio del bucle y antes de cada iteración de las sentencias.

Si la **condición es verdadera**, se ejecuta el bloque de sentencias, y se vuelve al principio del bucle.

Si la **condición es falsa**, no se ejecuta el bloque de sentencias, y se continúa con las siguientes sentencias del programa.

Si la condición es falsa desde un principio, entonces el bucle nunca se ejecuta.

Si la condición nunca llega a ser falsa, se tiene un bucle infinito.

Ej:

```
int i=0;
while (i<5) {
    System.out.println (i);
    i++;
}
```

El resultado de ejecutar este bloque de instrucciones es el mismo que para el for, escribirá en pantalla del 0 al 4 en diferentes líneas.

Los dos siguientes ejemplos son también un bucle infinito y un bucle que no se ejecutará nunca.

Bucle infinito:

```
int i=0;
while (i<5){
    System.out.println(i);
}
```

Sera infinito pues al no cambiar el valor de la i nunca, la condición siempre se está cumpliendo.

Bucle que nunca se ejecutará:

```
int i=6;
while (i<5){
    System.out.println(i);
    i++;
}
```

No se ejecuta nunca el bucle pues desde el principio no se cumple la condición que permita entrar en el while, pues 6 no es menor que 5.

2.3 Bucle Do-While (REPETIR HASTA QUE)

La **sintaxis** de la sentencia es:

```
do{
    sentencias;
}while (condición);
```

Es muy parecida a while. El bloque de sentencias se repite mientras se cumpla la condición pero en este caso, la condición se comprueba después de ejecutar el bloque de sentencias por lo que el bloque **se ejecuta siempre al menos una vez**.

Este tipo de bucle será muy útil cuando se quiere obligar a que una determinada variable solo pueda tomar unos ciertos valores, o cuando se quiere comprobar una contraseña antes de seguir ejecutando el bucle, etc...

Ej:

```
int i;
do {
    System.out.println ("Introducir un numero distinto de 0");
    i = teclado.nextInt();
} while(i == 0);
```

En este ejemplo se pide introducir un número que no sea 0, con lo cual no se saldrá del bucle mientras se introduzca un 0. (Probarlo en clase).

UT3 Java printf para dar formato a los datos de salida

Vamos a ver cómo utilizar printf para dar formato a los datos que se imprimen por pantalla en Java. Este problema se nos plantea por ejemplo cuando queremos mostrar un número de tipo float o double con un número determinado de decimales y no con los que por defecto muestra Java.

A partir de la versión Java 5 se incorporan los métodos format y printf que permiten aplicar un formato a la salida de datos por pantalla. Ambos realizan la misma función, tienen exactamente el mismo formato y emulan la impresión con formato printf() de C.

Veamos primero varios ejemplos de printf en Java y después explicaremos en detalle la sintaxis de printf. Si queremos mostrar el número 12.3698 de tipo double con dos decimales:

```
System.out.printf("% .2f %n", 12.3698);
```

El primer % indica que en esa posición se va a escribir un valor. El valor a escribir se encuentra a continuación de las comillas, .2 indica el número de decimales. La f indica que el número es de tipo **float** o **double**. En la tabla que aparece más adelante podéis ver todos los caracteres de conversión para todos los tipos de datos. %n indica un salto de línea, equivale a \n. Con printf podemos usar ambos para hacer un salto de línea.

La salida por pantalla es: 12,37

Comprobamos que printf realiza un redondeo para mostrar los decimales indicados.

Lo más común será que tengamos el valor en una variable, en ese caso si queremos escribir el valor de n con tres decimales:

```
double n = 1.25036;  
System.out.printf("% .3f %n", n);  
Salida:  
1,250
```

Para mostrar el signo + en un número positivo:

```
double n = 1.25036;  
System.out.printf("%+.3f %n", n);  
Salida:  
+1.250
```

Si el número a mostrar es un **entero** se utiliza el carácter d:

```
int x = 10;  
System.out.printf("%d %n", x);  
Salida:  
10
```

Para mostrarlo con signo:

```
int x = 10;  
System.out.printf("%+d %n", x);  
Salida:  
+10
```

Para mostrar varias variables pondremos **tantos % como valores vamos a mostrar**. Las variables se escriben a continuación de las comillas separadas por comas:

```
double n = 1.25036;  
int x = 10;  
System.out.printf("n = %.2f x = %d %n", n, x);
```

Salida:

```
n = 1,25 x = 10
```

Cuando hay varias variables podemos indicar de cuál de ellas es el valor a mostrar escribiendo 1\$, 2\$, 3\$, ... indicando que el valor a mostrar es el de la primera variable que aparece a continuación de las comillas, de la segunda, etc.

La instrucción anterior la podemos escribir así:

```
System.out.printf("n = %1$.2f x = %2$d %n", n, x);
```

Este número es opcional, si no aparece se entenderá que el primer valor proviene de la primera variable, el segundo de la segunda, etc.

Si queremos mostrar el número 123.4567 y su cuadrado ambos con dos decimales debemos escribir:

```
double n = 123.4567;  
System.out.printf("El cuadrado de %.2f es %.2f\n", n, n*n);
```

Salida:

```
El cuadrado de 123,46 es 15241,56
```

printf permite mostrar valores con un ancho de campo determinado. Por ejemplo, si queremos mostrar el contenido de n en un ancho de campo de 10 caracteres escribimos:

```
double n = 1.25036;  
System.out.printf("%+10.2f %n", n);
```

Salida:

```
bbbbbb+1.25
```

Donde cada *b* indica un espacio en blanco.

El 10 indica el tamaño en caracteres que ocupará el número en pantalla. Se cuentan además de las cifras del número el punto decimal y el signo si lo lleva. En este caso el número ocupa un espacio de 5 caracteres (3 cifras, un punto y el signo) por lo tanto se añaden 5 espacios en blanco al principio para completar el tamaño de 10.

Si queremos que en lugar de espacios en blancos nos muestre el número completando el ancho con ceros escribimos:

```
System.out.printf("%+010.2f %n", n);
```

Salida:

+000001.25

Más ejemplos de printf:

Mostrar el número 1.22 en un ancho de campo de 10 caracteres y con dos decimales.

```
double precio = 1.22;  
System.out.printf("%10.2f", precio);
```

Salida:

bbbbbb1.22

(el carácter b indica un espacio en blanco)

El número ocupa un espacio total de 10 caracteres incluyendo el punto y los dos decimales.

Mostrar la cadena "Total:" con un ancho de 10 caracteres y alineada a la izquierda:

```
System.out.printf("%-10s", "Total:");
```

Salida:

Total:bbbb

El carácter s indica que se va a mostrar una cadena de caracteres.

El signo - indica alineación a la izquierda.

Mostrar la cadena "Total:" con un ancho de 10 caracteres y alineada a la derecha:

```
System.out.printf("%10s", "Total:");
```

Salida:

bbbbTotal:

Al final puedes ver un ejemplo completo con distintos usos de printf.

Veamos ahora detenidamente la sintaxis de printf:

La sintaxis general de printf es:

```
printf (String de formato, Object ... datos);
```

El *String de formato* es una cadena de caracteres que contiene:

- **Texto fijo** que será mostrado tal cual.
 - **Especificadores de formato** que determinan la forma en que se van a mostrar los datos.

Los *datos* representan la información que se va a mostrar y sobre la que se aplica el formato anterior. El número de datos que se pueden mostrar es variable.



Explicación de cada una de las partes que aparecen en la instrucción printf:

Especificadores de formato:

La sintaxis para los especificadores de formato de printf es:

%[posición_dato\$] [indicador_de_formato] [ancho] [.precisión] carácter_de conversión

Los elementos entre corchetes son opcionales.

posición_dato: indica la posición del dato sobre el que se va aplicar el formato. El primero por la izquierda ocupa la posición 1.

indicador_de_formato: es el conjunto de caracteres que determina el formato de salida. Los indicadores de formato de printf en Java son:

INDICADORES DE FORMATO			
Indicador	Significado	Indicador	Significado
-	Alineación a la izquierda	+	Mostrar signo + en números positivos
(Los números negativos se muestran entre paréntesis	0	Rellenar con ceros
,	Muestra el separador decimal		

ancho: Indica el tamaño mínimo, medido en número de caracteres, que debe ocupar el dato en pantalla.

.precisión: Indica el número de decimales que serán representados. Solo aplicable a datos de tipo float o double.

carácter_de_conversión: Carácter que indica cómo tiene que ser formateado el dato. Los más utilizados se muestran en la tabla.

CARACTERES DE CONVERSIÓN			
Carácter	Tipo	Carácter	Tipo
d	Número entero en base decimal	X, x	Número entero en base hexadecimal
f	Número real con punto fijo	s	String
E, e	Número real notación científica	S	String en mayúsculas
g	Número real. Se representará con notación científica si el número es muy grande o muy pequeño	C, c	Carácter Unicode. C: en mayúsculas

Ejemplo completo con distintos usos de printf en Java:

```
public class Printf2 {
    public static void main(String[] args) {
        double q = 1.0/3.0;

        System.out.printf("1.0/3.0 = %5.3f %n", q);
        System.out.printf("1.0/3.0 = %7.5f %n", q);

        q = 1.0/2.0;
        System.out.printf("1.0/2.0 = %09.3f %n", q);

        q = 1000.0/3.0;
        System.out.printf("1000/3.0 = %7.1e %n", q);

        q = 3.0/4567.0;
        System.out.printf("3.0/4567.0 = %7.3e %n", q);

        q = -1.0/0.0;
        System.out.printf("-1.0/0.0 = %7.2e %n", q);

        q = 0.0/0.0;
        System.out.printf("0.0/0.0 = %5.2e %n", q);

        System.out.printf("pi = %5.3f; e = %10.4f %n", Math.PI,
        Math.E);

        double r = 1.1;
        System.out.printf("%1$.5f *%1$.5f %n", Math.PI, r);
        System.out.printf("%1$6.6f *%1$6.6f %n", Math.PI, r);

        System.out.printf("C = 2 * %1$5.5f * %2$4.1f, +"A = %2$4.1f
        * %2$4.1f * %1$5.5f %n", Math.PI, r);
    }
}
```

Salida:

```
1.0/3.0 = 0,333
1.0/3.0 = 0,33333
1.0/2.0 = 00000,500
1000/3.0 = 3,3e+02
3.0/4567.0 = 6,569e-04
-1.0/0.0 = -Infinity
0.0/0.0 =  NaN
pi = 3,142; e =      2,7183
3,14159 *3,14159
3,141593 *3,141593
C = 2 * 3,14159 * 1,1, A = 1,1 * 1,1 * 3,14159
```

1. Formatter (clase) java.util.Formatter

Esta clase permite generar String a partir de datos usando una especificación de formato.

La funcionalidad se presta a diferentes clases para un uso cómodo:

```
String s = String.format(formato, valores ...);  
System.out.printf(formato, valores ...);  
System.out.format(formato, valores ...);
```

Es muy similar a printf.

```
Formatter fmtr = new Formatter();  
  
String saludo =  
String.format("Hola amigos, bienvenidos a %s !", "Madrid");  
System.out.println(saludo);  
  
saludo =  
String.format("Hola %2$s, bienvenidos %1$s !", "Madrid", "chicos");  
System.out.println(saludo);  
  
int numero = 425;  
fmtr.format("%08d", numero);  
System.out.println("El numero formateado " + fmtr);  
  
Object result = fmtr.format("%1$4d - el año %2$4.2f", 1951, Math.PI);  
System.out.println(result);  
  
// Otra forma de hacerlo  
System.out.format("%1$04d - el año %2$4.2f%n", 1951, Math.PI);  
  
// Parecido al printf que ya vimos:  
System.out.printf("%1$04d - el año %2$4.2f%n", 1951, Math.PI);  
System.out.printf("PI es mas o menos %1$4.2f%n", Math.PI);  
System.out.printf("Tu nombre es %1$s y el mio es %2$s%n", "Juan",  
"Pepi");  
  
fmtr.close();
```

1.1. Como usar el Formateador decimal en Java

La clase **DecimalFormat**: Es una **clase de Java** que nos permite mostrar los números con un formato deseado, puede ser limitando los dígitos decimales, si usaremos punto o coma, etc, incluso podemos tomar los valores desde un **JTextField** y reconstruir el número, ejemplo:

```
//import requerido para el Formateador  
  
import java.text.DecimalFormat;  
  
DecimalFormat formateador = new DecimalFormat("####.#####");  
  
// Imprime esto con cuatro decimales, es decir: 7,1234  
  
System.out.println (formateador.format (7.12342383));
```

Si utilizamos ceros en lugar de los #, los dígitos no existentes se rellenarán con ceros, ejemplo:

```
DecimalFormat formateador = new DecimalFormat("0000.0000");  
  
// Imprime con 4 cifras enteras y 4 decimales: 0001,8200  
  
System.out.println (formateador.format (1.82));
```

También podemos utilizar el signo de porcentaje (%) en la máscara y así el número se multiplicará automáticamente por 100 al momento de Imprimir.

```
DecimalFormat formateador = new DecimalFormat("###.##%");  
  
// Imprime: 68,44%  
  
System.out.println (formateador.format(0.6844));
```

1. DecimalFormat (clase)

La clase **DecimalFormat** usa por defecto el formato para el lenguaje que tengamos instalado en el ordenador. Es decir, si nuestro sistema operativo está en español, se usará la coma para los decimales y el punto para los separadores de miles. Si estamos en inglés, se usará el punto decimal. Una opción para cambiar esto, es utilizar la clase **DecimalFormatSymbols**, que vendrá rellena con lo del idioma por defecto, y cambiar en ella el símbolo que nos interese. Por ejemplo, si estamos en español y queremos usar el punto decimal en vez de la coma, podemos:

```
import java.text.DecimalFormat;

import java.text.DecimalFormatSymbols;

DecimalFormatSymbols simbolos = new DecimalFormatSymbols();

simbolos.setDecimalSeparator('.');

DecimalFormat formateador = new DecimalFormat("###.###", simbolos);

// Imprime: 3.4324

System.out.println(formateador.format (3.43242383));
```

Reconstruyendo un Número:

Supongamos que leemos algún número desde consola o un JTextField y deseamos reconstruirlo con DecimalFormat, se hace de la siguiente forma:

```
JTextField textField = new JTextField();  
DecimalFormat formateador = new DecimalFormat("####.#####");  
  
String texto = textField.getText();  
  
try  
{  
    // parse() lanza una ParseException en caso de fallo que hay que capturar.  
  
    Number numero = formateador.parse(texto);  
    double valor = numero.doubleValue();  
  
    // Optimizando las Lineas anteriores:  
  
    // double valor = formateador.parse(texto).doubleValue();  
}  
  
catch (ParseException e)  
{  
    // Error. El usuario ha escrito algo que no se puede convertir a número.  
}
```

UT3 Otra forma de leer del usuario

1. Alertas y diálogos en Java

La clase `JOptionPane` es una clase que hereda de `JComponent`. Esta clase nos permitirá crear alertas o cuadros de diálogo simples para poder solicitar o mostrar información al usuario.

Los métodos que veremos son:

1. `JOptionPane.showMessageDialog(...);`
2. `JOptionPane.showConfirmDialog(...);`
3. `JOptionPane.showInputDialog(...);`
4. `JOptionPane.showOptionDialog(...);`

Cada uno de estos métodos presenta una particularidad distinta pero todos ellos nos muestran una ventana pop up que nos permitirá captar información del usuario.

a) showMessageDialog

JOptionPane.showMessageDialog (Component componentePadre, Object mensaje, String titulo, int tipoDeMensaje)

Nos sirve para mostrar información, por ejemplo alguna alerta que queremos hacerle al usuario. Veamos cuales son los principales argumentos del método:

componentePadre: es el Frame desde el cual lo llamamos. Si queremos lo podemos poner null de momento. Si hubiera un padre, el dialogo se mostraría sobre él.

mensaje: es lo que queremos que diga el cuadro de dialogo.

titulo: el título de la ventana.

tipoDeMensaje: son constantes que le dirán a java qué tipo de mensaje queremos mostrar. De acuerdo a esto serán los iconos que se mostrarán en el cuadro de dialogo.

Las opciones son:

- **ERROR_MESSAGE**
- **INFORMATION_MESSAGE**
- **WARNING_MESSAGE**
- **QUESTION_MESSAGE**
- **PLAIN_MESSAGE**

Si quisiéramos mostrar un ícono personalizado, podemos agregarlo al final como un argumento más.

Ejemplo:

```
JOptionPane.showMessageDialog(null, "Acceso Denegado",  
"Error", JOptionPane.ERROR_MESSAGE);
```

b) showConfirmDialog

Este método sirve para pedirle al usuario una confirmación. Por ejemplo, una confirmación de salida del sistema

```
Int respuesta = JOptionPane. showConfirmDialog(Component  
componentePadre, Object mensaje, String titulo, int tipoDeOpcion);
```

Lo anterior es la versión corta de los argumentos del método. La versión larga incluye el tipo de mensaje y el icono, por si queremos personalizarlo.

Los argumentos son idénticos a los del método anterior. Excepto por el tipo de opción, que es otra constante y los valores pueden ser:

- **DEFAULT_OPTION**
- **YES_NO_OPTION**
- **YES_NO_CANCEL_OPTION**
- **OK_CANCEL_OPTION**

Como vemos, el método devuelve un entero que nos permitirá captar cual es la opción elegida por el usuario. Los valores serán **0** para **Sí**, **1** para **No**, **2** para **Cancelar** y **-1** para el **cierre de la ventana**. Así podremos preguntar cuál es el valor devuelto y realizar la acción que deseamos.

Ejemplo:

```
int i = JOptionPane.showConfirmDialog(this,  
"¿Realmente Desea Salir de Hola Swing?",  
"Confirmar Salida", JOptionPane.YES_NO_OPTION);  
  
if(i==0){  
    System.exit(0);  
}
```

c) showInputDialog

Este método nos muestra una ventana donde podremos insertar un String. Por ejemplo, cuando queremos que el usuario inserte su nombre. **La versión corta del método es:**

String respuesta = JOptionPane.showInputDialog(Object mensaje)

Este método devolverá un String para poder utilizarlo después. La versión larga de los argumentos del método es similar a los anteriores.

Ejemplo:

```
String nombre = JOptionPane.showInputDialog("Inserte su nombre");
```

También podemos crear un cuadro de dialogo que contenga un combo con las opciones predeterminadas que le queremos dar al usuario.

Ejemplo:

```
Object[] valoresPosibles = {"Pedro", "Juan", "Carlos" };

Object jefe = JOptionPane.showInputDialog(null, "Seleccione cuales son sus Jefes Inmediatos", "Seleccionar Jefe",
JOptionPane.QUESTION_MESSAGE, null, valoresPosibles,
valoresPosibles[0]);
```

El array de valores posibles nos muestra en un combo cuáles serán los jefes que podemos mostrar. El último argumento del método nos muestra cuál será la opción seleccionada por defecto

d) showOptionDialog

Con este método podemos crear cuadros de diálogos con botones personalizados. Está bien para personalizar los cuadros de dialogo.

El método tiene la forma:

```
int res = JOptionPane.showOptionDialog(Component componentePadre,  
Object mensaje, String titulo, int tipoDeOpcion, int tipoMensaje,  
Icon icono, Object[] botones, Object botonDefault)
```

Aquí lo único que varía con el resto, es el array de botones que vamos a tener, debemos destacar que no hace falta que creamos botones, solo tenemos que poner cuál será el texto que saldrá en él.

```
Object[] botones = {"No, nada", "Un poquito", "Me  
estalla"};  
  
int i=JOptionPane.showOptionDialog(null, "¿Te duele la  
cabeza?", "ventanita",  
JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE,  
null, botones, botones[0]);  
  
System.out.println(i);
```

Después podemos tomar la respuesta como lo hacíamos con el confirmDialog.

Se puede consultar el API y <https://serprogramador.es/programando-mensajes-de-dialogo-en-java-parte-1/>

2. Convertir String a número

Instrucciones de Java: ParseInt, ParseDouble y ParseFloat:

La función parseFloat() analiza una cadena y devuelve un número de punto flotante.

```
Float.parseFloat(String str)
```

La función parseInt() analiza una cadena y devuelve un entero.

```
Integer.parseInt(String str)
```

La función parseDouble() analiza una cadena y devuelve un número decimal doble.

```
Double.parseDouble(String str)
```

UT3 Sus Métodos en Java, funciones y procedimientos. Cómo hacerlos y usarlos

Los métodos en Java, las funciones y los procedimientos, especialmente en Java, son una herramienta indispensable para programar. Java nos permite crear o hacer nuestros propios métodos y usarlos sencillamente como también nos facilita hacer uso de los métodos de otras librerías (funciones matemáticas, aritméticas, de archivos, de fechas, etc. Cualquiera que sea el caso, las funciones permiten automatizar tareas que requiramos con frecuencia y que además se puedan generalizar por medio de parámetros o argumentos.

Aprender a crear métodos en Java y usarlos correctamente es de gran importancia, separar nuestro código en módulos y según las tareas que requerimos. En java una función debe contener la implementación de una utilidad de nuestra aplicación, esto nos pide que por cada utilidad básica (abrir, cerrar, cargar, mover, etc.) sería adecuado tener al menos una función asociada a ésta, pues sería muy complejo usar o crear un método que haga todo de una sola vez, por esto es muy buena idea separar cada tarea en una función o método (según corresponda).

En Java es mucho más común hablar de métodos que de funciones y procedimientos y esto se debe a que en realidad un método, una función y un procedimiento NO son lo mismo, veamos la diferencia:

1. ¿Funciones, métodos o procedimientos?

Es muy común entre programadores que se hable indistintamente de estos tres términos, sin embargo poseen diferencias fundamentales.

a) Funciones:

Las funciones son un conjunto de líneas de código (instrucciones), encapsulados en un bloque, usualmente reciben parámetros, cuyos valores utilizan para efectuar operaciones y adicionalmente devuelven/retornan un valor. En otras palabras, una función puede recibir parámetros o argumentos (algunas no reciben nada), hace uso de dichos valores recibidos como sea necesario y devuelve un valor usando la instrucción **return**. **Si no devuelve algo, entonces no es una función (se suele llamar procedimiento en este caso).**

b) Métodos:

Los métodos y las funciones (y/o procedimientos) en Java pueden realizar las mismas tareas, es decir, son funcionalmente idénticos, pero su diferencia radica en la manera en que hacemos uso de uno u otro (el contexto). Un método también puede recibir valores, efectuar operaciones con estos y retornar valores, sin embargo un método está asociado a un objeto, **SIEMPRE, básicamente un método es una función que pertenece a un objeto o clase, mientras que una función existe por sí sola, sin necesidad de un objeto para ser usada.**

Nota: En Java se debe hablar de métodos y no de funciones, pues en Java estamos siempre obligados a crear un objeto para usar el método. Para que sea una función/procedimiento, esta debe ser *static*, para que no requiera de un objeto para ser llamada.

c) Procedimientos:

Los procedimientos son básicamente un conjunto de instrucciones que se ejecutan sin retornar ningún valor, pueden recibir o no argumentos. En el contexto de Java un procedimiento es básicamente un método cuyo tipo de retorno es *void*, que no nos obliga a utilizar una sentencia return.

2. Crear un método en Java

La sintaxis para declarar una función es muy simple, veamos:

```
Ejemplo: public static void main (String[] args)  
[acceso] [modificador] tipoDevuelto nombreFuncion([tipo nombreArgumento, [tipo nombreArgumento]...])  
//Argumento es lo mismo que variable  
{  
    /*  
     * Bloque de instrucciones  
    */  
  
    return valor;  
    //return: devuelve solo el valor de la variable, no la variable  
}
```

El primer componente corresponde al **modificador de acceso**, que puede ser public, private o protected, este último es opcional, si no ponemos nada, se asume el modificador de acceso por defecto.

El segundo componente es el **modificador** que puede ser final o static (o ambas), también es opcional. Recordemos que un método o función siempre retorna algo, por lo tanto es obligatorio declararle un **tipo** (el tercer componente de la sintaxis anterior), puede ser entero (int), booleano (boolean), o cualquiera que consideremos, incluso tipos complejos. Si no es función, pondremos void.

Luego debemos darle un **nombre** a dicha función/método/procedimiento, para poder identificarla y llamarla (invocarla) durante la ejecución. Después, en el interior del paréntesis, podemos poner los **argumentos** o parámetros.

Cuando hemos acabado con la definición de la "firma" o prototipo del método, definimos su funcionamiento entre llaves. Todo lo que esté dentro de las llaves, es parte del cuerpo del método y este se ejecuta hasta llegar a una instrucción *return* o cuando se alcanza la última sentencia a ejecutar.

3. Return y void

En algunas ocasiones, no es necesario que el método estático tenga que devolver un valor al finalizar su ejecución. En este caso, el tipo de dato que debe indicar en la cabecera de declaración del método es el tipo void y la sentencia return no viene seguida de ninguna expresión. Este return solitario se puede omitir.

Sintaxis:

```
return;
```

//Quiere decir vuelvo al sitio donde me han llamado para seguir con la siguiente instrucción, no continua a partir de aquí

Ejemplo:

Seguidamente se muestra un ejemplo de declaración y uso de un método que devuelve el cubo de un valor numérico real con una sentencia return:

```
/**  
 * Demostracion del metodo cubo  
 */  
public class PruebaCubo {  
    public static void main (String [] args){  
        System.out.println("El cubo de 7.5 es: " + cubo(7.5));  
        // llamada dentro de la sentencia System.out.println  
    }  
  
    public static double cubo (double x) {  
        // devuelve un double  
        // declaracion  
        return x*x*x;  
    }  
}
```

Si no hay sentencia return dentro de un método, su ejecución continúa hasta que se alcanza el final del método y entonces se devuelve la secuencia de ejecución al lugar dónde se invocó al método.

Un método cuyo tipo de retorno no es void necesita siempre devolver algo. Si el código de un método contiene varias sentencias if debe asegurarse de que cada una de las posibles opciones devuelva un valor. En caso contrario, se generaría un error de compilación.

Por ejemplo:

```
/***
 * Demostracion de la funcion esPositivo
 */
public class PruebaPositivo {
    public static void main (String [] args) {
        for (int i=5; i>=-5; i--)
            System.out.println(i + " es positivo: " + esPositivo(i));
    }

    public static boolean esPositivo(int x) {
        if (x<0) return false;
        if (x>0) return true;
    }
}
```

Ejemplo de intento de compilación del código anterior:

```
$>javac PruebaPositivo.java
pruebaPositivo.java:14: missing return statement
}^
```

Lo que pasa es que falta el valor 0, que no es ni positivo ni negativo, por eso te da error.

UT3 Generación de números aleatorios

1. Números aleatorios en Java

Para generar números aleatorios en Java tenemos dos opciones. Por un lado, podemos usar `Math.random()`, por otro la clase `java.util.Random`. La primera es de uso más sencillo y rápido. La segunda nos da más opciones.

- `Math.random()`
- Clase `java.util.Random`

2. Math.random()

La llamada a `Math.random()` devuelve un número aleatorio entre 0.0 y 1.0, excluido este último valor, es decir, puede devolver 0.346442, 0.2344234, 0.98345,....

En muchas de nuestras aplicaciones no nos servirá este rango de valores. Por ejemplo, si queremos simular una tirada de dado, queremos números entre 1 y 6 sin decimales. Debemos echar unas cuentas para obtener lo deseado.

En primer lugar, miramos cuántos valores queremos. En nuestro caso del dado son 6 valores, del 1 al 6 ambos incluido. Debemos entonces multiplicar `Math.random()` por 6.

```
Math.random()*6      // Esto da valores de 0.0 a 6.0, excluido el 6.0  
Math.random()*limite+1;
```

Donde límite será el número más alto del rango. Los números genéricos que generaremos irán entre el 1 y el límite.

```
Math.random()*6 + 1  
// Esto da valores entre 1.0 y 7.0 excluido el 7.0
```

Finalmente, para conseguir un entero, quitamos los decimales usando la clase `Math.floor()`

```
int valorDado = Math.floor(Math.random()*6+1);  
// A mi me sale que Math.floor te lo pasa a un double, igual es  
Math.round
```

O también haciendo un casting a un entero:

```
int numeroAleatorio = (int) (Math.random()*6+1);
```

En general, para conseguir un número entero entre [N, M] con N < M y ambos incluidos, debemos usar esta fórmula:

```
int valorEntero = Math.floor(Math.random() * (M-N+1) +N);  
// Valor entre M y N, ambos incluidos.
```

3. Clase java.util.Random

La clase `java.util.Random` debemos instanciarla, a diferencia del método `Math.random()`. A cambio, tendremos bastantes más posibilidades.

Podemos usar un constructor sin parámetros o bien pasarle una semilla. Si instanciamos varias veces la clase con la misma semilla, tendremos siempre la misma secuencia de números aleatorios.

```
Random r1 = new Random();
Random r2 = new Random(4234);
Random r3 = new Random(4234); // r2 y r3 darán la misma secuencia.
```

Lo más fácil es usar el constructor sin parámetros, que normalmente dará secuencias distintas en cada instancia. De todas formas, una manera de obtener una semilla que sea distinta cada vez que ejecutemos nuestro programa puede ser obtener el tiempo actual en milisegundos con `System.currentTimeMillis()`, que dará números distintos salvo que hagamos la instancia justo en el mismo instante de tiempo.

Con esta clase, una vez instanciada, nuestro problema del dado sería bastante más sencillo, usando el método `nextInt(int n)`, que devuelve un valor entre 0 y n, excluido n

```
Random r = new Random();
int valorDado = r.nextInt(6)+1; // Entre 0 y 5, más 1. Es decir, 1-6
```

También tenemos funciones que nos dan un valor aleatorio siguiendo una curva de Gauss o que nos rellenan un array de bytes de forma aleatoria. Y por supuesto, el `nextDouble()` que devuelve un valor aleatorio entre 0.0 y 1.0, excluido este último.

La clase `Random` proporciona un generador de números aleatorios que es más flexible que la función estática `random` de la clase `Math`.

Para crear una secuencia de números aleatorios tenemos que seguir los siguientes pasos:

1. Proporcionar a nuestro programa información acerca de la clase `Random`. Al principio del programa escribiremos la siguiente sentencia.

```
import java.util.Random;
```

2. Crear un objeto de la clase `Random`
3. Llamar a una de las funciones miembro que generan un número aleatorio
4. Usar el número aleatorio.

3.1 Constructores

La clase dispone de dos constructores, el primero crea un generador de números aleatorios cuya semilla es inicializada en base al instante de tiempo actual.

```
Random rnd = new Random();
```

El segundo, inicializa la semilla con un número del tipo **long**.

```
Random rnd = new Random(3816L);
```

El sufijo L no es necesario, ya que aunque 3816 es un número **int** por defecto, es promocionado automáticamente a **long**.

Aunque no podemos predecir que números se generarán con una semilla particular, podemos sin embargo, duplicar una serie de números aleatorios usando la misma semilla. Es decir, cada vez que creamos un objeto de la clase *Random* con la misma semilla obtendremos la misma secuencia de números aleatorios. Esto no es útil en el caso de loterías, pero puede ser útil en el caso de juegos, exámenes basados en una secuencia de preguntas aleatorias, las mismas para cada uno de los estudiantes, simulaciones que se repitan de la misma forma una y otra vez, etc.

3.2 Funciones miembro

Podemos cambiar la semilla de los números aleatorios en cualquier momento, llamando a la función miembro *setSeed*.

```
rnd.setSeed(3816);
```

Función miembro	Descripción	Rango
rnd.nextInt()	Número aleatorio entero de tipo int	2^{-32} y 2^{32}
rnd.nextLong()	Número aleatorio entero de tipo long	2^{-64} y 2^{64}
rnd.nextFloat()	Número aleatorio real de tipo float	$[0,1[$
rnd.nextDouble()	Número aleatorio real de tipo double	$[0,1[$

Podemos generar números aleatorios de cuatro formas diferentes:

```
rnd.nextInt();
```

Genera un número aleatorio entero de tipo **int**

```
rnd.nextLong();
```

Genera un número aleatorio entero de tipo **long**

```
rnd.nextFloat();
```

Genera un número aleatorio de tipo **float** entre 0.0 y 1.0, aunque siempre menor que 1.0

```
rnd.nextDouble();
```

Genera un número aleatorio de tipo **double** entre 0.0 y 1.0, aunque siempre menor que 1.0

Casi siempre usaremos esta última versión. Por ejemplo, para generar una secuencia de 10 números aleatorios entre 0.0 y 1.0 escribimos

```
for (int i = 0; i < 10; i++) {  
    System.out.println(rnd.nextDouble());  
}
```

Para crear una secuencia de 10 números aleatorios enteros comprendidos entre 0 y 9 ambos incluidos escribimos

```
int x;  
for (int i = 0; i < 10; i++) {  
    x = (int) (rnd.nextDouble() * 10.0);  
    System.out.println(x);  
}
```

(int) transforma un número decimal **double** en entero **int** eliminando la parte decimal.

4. Comprobación de la uniformidad de los números aleatorios

Podemos comprobar la uniformidad de los números aleatorios generando una secuencia muy grande de números aleatorios enteros comprendidos entre 0 y 9 ambos inclusive. Contamos cuantos ceros aparecen en la secuencia, cuantos unos, ... cuantos nueves, y guardamos estos datos en los elementos de un array.

Primero creamos un array *ndigitos* de 10 de elementos que son enteros.

```
int[] ndigitos = new int[10];
```

Inicializamos los elementos del array a cero.

```
for (int i = 0; i < 10; i++) {  
    ndigitos[i] = 0;  
}
```

Creamos una secuencia de 100000 números aleatorios enteros comprendidos entre 0 y 9 ambos inclusive (véase el apartado anterior)

```
for (long i=0; i < 100000L; i++) {  
    n = (int)(rnd.nextDouble() * 10.0);  
    ndigitos[n]++;  
}
```

Si *n* sale cero suma una unidad al contador de ceros, *ndigitos[0]*. Si *n* sale uno, suma una unidad al contador de unos, *ndigitos[1]*, y así sucesivamente.

Finalmente, se imprime el resultado, los números que guardan cada uno de los elementos del array *ndigitos*

```
for (int i = 0; i < 10; i++) {  
    System.out.println(i+": " + ndigitos[i]);  
}
```

Observaremos en la consola que cada número 0, 1, 2...9 aparece aproximadamente 10000 veces.

5. Secuencias de números aleatorios

En la siguiente porción de código, se imprime dos secuencias de cinco números aleatorios uniformemente distribuidos entre [0, 1), separando los números de cada una de las secuencias por un carácter tabulador.

```
System.out.println("Primera secuencia");
for (int i = 0; i < 5; i++) {
    System.out.print("\t"+rnd.nextDouble());
}
System.out.println("");

System.out.println("Segunda secuencia");
for (int i = 0; i < 5; i++) {
    System.out.print("\t"+rnd.nextDouble());
}
System.out.println("");
```

Comprobaremos que los números que aparecen en las dos secuencias son distintos.

En la siguiente porción de código, se imprime dos secuencias iguales de números aleatorios uniformemente distribuidos entre [0, 1). Se establece la semilla de los números aleatorios con la función miembro *setSeed*.

```
rnd.setSeed(3816);
System.out.println("Primera secuencia");
for (int i = 0; i < 5; i++) {
    System.out.print("\t"+rnd.nextDouble());
}
System.out.println("");

rnd.setSeed(3816);
System.out.println("Segunda secuencia");
for (int i = 0; i < 5; i++) {
    System.out.print("\t"+rnd.nextDouble());
}
System.out.println();
```

```
package azar;

import java.util.Random;

public class AzarApp {
    public static void main (String[] args) {
        int[] ndigitos = new int[10];
        int n;

        Random rnd = new Random();

        // Inicializar el array
        for (int i = 0; i < 10; i++) {
            ndigitos[i] = 0;
        }

        // verificar que los números aleatorios están uniformemente distribuidos
        for (long i=0; i < 100000L; i++) {
        // genera un número aleatorio entre 0 y 9
            n = (int)(rnd.nextDouble() * 10.0);
        //Cuenta las veces que aparece un número
            ndigitos[n]++;
        }

        // imprime los resultados
        for (int i = 0; i < 10; i++) {
            System.out.println(i+": " + ndigitos[i]);
        }

        //Dos secuencias de 5 número (distinta semilla)
        System.out.println("Primera secuencia");
        for (int i = 0; i < 5; i++) {
            System.out.print("\t"+rnd.nextDouble());
        }
        System.out.println("");

        System.out.println("Segunda secuencia");
        for (int i = 0; i < 5; i++) {
            System.out.print("\t"+rnd.nextDouble());
        }
        System.out.println("");

        //Dos secuencias de 5 número (misma semilla)
        rnd.setSeed(3816L);
        System.out.println("Primera secuencia");
        for (int i = 0; i < 5; i++) {
            System.out.print("\t"+rnd.nextDouble());
        }
        System.out.println("");

        rnd.setSeed(3816);
        System.out.println("Segunda secuencia");
        for (int i = 0; i < 5; i++) {
            System.out.print("\t"+rnd.nextDouble());
        }
        System.out.println("");
    }
}
```

UT3 Ejercicios

Funciones 1

- 1.- Crea una subrutina en java a la que se le pase un número N que se le pedirá al usuario y muestre por pantalla la frase “Módulo ejecutándose” N veces.
- 2.-Muestra por pantalla los resultados de las 4 operaciones básicas entre 2 números que se le pedirán al usuario, realizando una subrutina para cada una de dichas operaciones. El resultado de la operación se imprime dentro de la subrutina. Y fuera
- 3.- Crea una función en java que reciba dos números reales y devuelva su suma.
- 4.- Crea una función en java que reciba un número entero y devuelva su factorial.
- 5.- Escribe una aplicación en java que le pida al usuario que escriba el nombre de una operación en java entre las siguientes: suma, resta, multiplicación y división. En caso de introducir una distinta se le comunicará el error y se pedirá de nuevo la operación.

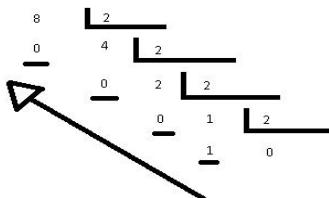
Con una instrucción switch, diseña un menú que en función de cuál sea la operación elegida ejecutará una de las cuatro funciones que escribirás para llevar a cabo la operación (sumar(float, float), multiplicar(float, float), restar(float, float), dividir(float, float)). Cada función de las anteriores recibirá los dos operadores y devolverá al programa principal el resultado, desde donde se imprimirá. En el menú habrá una opción de salir para poder terminar el programa.
- 6.- Crea una aplicación que nos calcule el área de un círculo, cuadrado o un triángulo. Pediremos de qué figura queremos calcular su área (nos darán una de las tres) y según lo introducido pedirá los valores necesarios para calcular el área. Crea un método por cada figura para calcular cada área que devolverá un número real como valor del área. Muestra el resultado por pantalla. Añade un menú para mejorar el ejercicio. En el menú habrá una opción de salir para poder terminar el programa.

Datos para el cálculo del área de cada figura:

- **Círculo:** $(radio^2)*PI$
- **Triángulo:** $(base * altura) / 2$
- **Cuadrado:** lado * lado

7.- Crea una aplicación que nos convierta un número en base decimal a binario. Esto lo realizará un método al que le pasaremos el número como parámetro, devolverá un String con el número convertido a binario. Para convertir un número decimal a binario, debemos dividir entre 2 el número y el resultado de esa división se divide entre 2 de nuevo hasta que no se pueda dividir más (la división da 0), el resto que obtengamos de cada división formara el número binario, de abajo a arriba.

Ejemplo: si introducimos un **8** nos deberá devolver **1000**



8.- Crea una aplicación que nos cuente el número de cifras de un número **entero positivo** (hay que controlarlo) pedido por teclado. Crea un método que realice esta acción, pasando el número por parámetro, devolverá el número de cifras.

9.- Crea un aplicación que nos convierta una cantidad de euros introducida por teclado a otra moneda, estas pueden ser dolares, yenes o libras. El método tendrá como parámetros, la cantidad de euros y la moneda a pasar que será una cadena. Haz dos versiones del programa:

El método no devolverá ningún valor, mostrará un mensaje indicando el cambio (void).

El método devolverá un valor que indica el cambio y este se mostrará en el programa principal.

El cambio de divisas es:

- 0.86 libras es un 1 €
- 1.28611 \$ es un 1 €
- 129.852 yenes es un 1 €

Funciones 2

Crea una biblioteca de funciones matemáticas que contenga las siguientes funciones. Recuerda que puedes usar unas dentro de otras si es necesario. Los números serán todos long. Todas las funciones devuelven al código llamante lo que se pide.

1.- voltear: Le da la vuelta a un número.

2.- esCapicua: Devuelve verdadero si el número que se pasa como parámetro es capicúa y falso en caso contrario.

3.- digitos: Cuenta el número de dígitos de un número.

4.- digitoN: Devuelve el dígito que está en la posición n de un número. Se empieza contando por el 0 y de izquierda a derecha.

5.- posicionDeDigito: Da la posición de la primera ocurrencia de un dígito dentro de un número. Si no se encuentra, devuelve -1.

6.- quitaPorDetras: Le quita a un número n dígitos por detrás (por la derecha).

7.- quitaPorDelante: Le quita a un número n dígitos por delante (por la izquierda).

8.- pegaPorDetras: Añade un número a otro número por detrás.

9.- pegaPorDelante: Añade un número a otro número por delante.

10.- trozoDeNumero: Toma como parámetros las posiciones inicial y final dentro de un número y devuelve el trozo correspondiente.

11.- juntaNumeros: Pega dos números para formar uno. Se usa para apoyar la solución de 8 y 9.

Ejercicios Cadenas (Strings)

- 1.-** Realiza un programa que lea 11 cadenas de caracteres, para mostrar al final la que contenga el mayor número de vocales, ya sean mayúsculas o minúsculas. Se consideran además de las 5 vocales, todas las vocales acentuadas y la ü.
- 2.-** Realiza un programa que lea un número entero y creando una función llamada esCapicua devuelva si es capicúa o no lo es (true y false respectivamente). Para ello, conviértelo en una cadena de caracteres y utiliza métodos de cadenas/caracteres.
- 3.-** Crea una función en Java que reciba una cadena de caracteres y devuelva esta cadena invertida.
- 4.-** Crea una función en Java que reciba una palabra y la devuelva invertida con efecto espejo, esto es, se concatena a la palabra original su inversa, compartiendo la última letra, que hará de espejo, por lo que la palabra obtenida se lee igual hacia adelante que hacia atrás. Por ejemplo, al introducir "teclado" devolverá "tecladodalcet" y al introducir "goma" devolverá "gomamog".
- 5.-** Crea una función en Java que reciba dos cadenas, la primera cadena será en la que se buscará y la segunda será la cadena buscada. La función devolverá cuántas veces aparece la segunda cadena en la primera.
- 6.-** Crea una función en Java que reciba dos cadenas de caracteres y que devuelva la primera cadena, pero transformando en mayúsculas la parte que coincide con la segunda cadena introducida. Por ejemplo, si se introducen las cadenas "Este es mi amigo Juan" y "amigo", devolverá "Este es mi AMIGO Juan".

Básicos Hoja 1

- 1.** Hacer un programa que sume dos números leídos por teclado y escriba el resultado con el mayor detalle posible.
- 2.** Haz un programa que convierta un número de galones en litros y escriba el resultado.
En un galón hay 3.7854 litros.
- 3.** Haz un programa que solicite una medida en pies y realice la conversión a pulgadas, yardas, cm y metros. Ten en cuenta que un pie tiene 12 pulgadas, una pulgada equivale a 2.54 cm y 1 yarda son 0.9144 metros.
- 4.** Un año es bisiesto si es múltiplo de 4, exceptuando los múltiplos de 100, que solo son bisiestos cuando son múltiplos además de 400.

Por ejemplo, el año 1900 no fue bisiesto, pero el año 2000 sí.
Haz un programa que dado un año A nos diga si es o no bisiesto.
- 5.** Hacer un programa que dado un día D, un mes M y un año A, calcule cual es el día siguiente. Se debe tener en cuenta que en los años bisiestos Febrero tiene 29 días y en los no bisiestos 28.
- 6.** Hacer un programa para resolver una ecuación de segundo grado, sin tener en cuenta las soluciones complejas.
- 7.** Modificar el ejercicio 1 para sumar 10 números leídos por teclado.
- 8.** Modificar el ejercicio 7 para que permita sumar N números. El valor de N se debe leer previamente por teclado.
- 9.** Modifica el programa 2 para que imprima una tabla de conversión de 1 hasta 100 galones, cada 10 galones imprimirá una línea de salida en blanco.
- 10.** Hacer un programa que permita escribir los primeros 100 números pares.
- 11.** La sucesión de Fibonacci se define de la siguiente forma:
a₁=1
a₂=1
...
a_n=a_{n-1}+a_{n-2} para n>2,

Es decir, los dos primeros son 1 y el resto cada uno es la suma de los dos anteriores, los primeros son: 1, 1, 2, 3, 5, 8, 13, 21,...
Haz un programa que calcule e imprima N términos de la sucesión, N se pedirá por teclado.

12. Haz un programa que calcule el total de una factura, partiendo de una lista de parejas importe, IVA pertenecientes a la misma factura. La lista finaliza cuando el importe sea 0. El IVA puede ser el 4%, el 10% o el 21%, en cualquier otro caso se rechazan importe e IVA y se deben introducir de nuevo. Finalmente, hay que realizar un descuento en función de la suma de los importes. Dicho descuento es del 0% si es menor que 1000, del 5% si es menor que 10000 (y mayor o igual que 1000) y del 10% si es mayor o igual que 10000. El descuento se debe aplicar al importe final.

El programa debe imprimir el **importe total** sin descuento y el **importe final** al aplicar el descuento que corresponda al importe total.

13. Hacer un programa que lea N números (se piden al usuario), calcule la suma de los pares y el producto de los impares y escriba ambos resultados.

14. Hacer un diagrama de flujo para calcular el máximo común divisor de dos números enteros positivos N y M siguiendo el algoritmo de Euclides, que es el siguiente:

1. Se divide N por M, sea R el resto.
2. Si R=0, el máximo común divisor es M y se acaba.
3. Si no, se asigna a N el valor de M y a M el valor de R y volvemos al paso 1.

15. Hacer un programa para calcular el factorial de N ($N!=1\cdot2\cdot3\cdots\cdot N$).

16. Hacer un programa para sumar los N primeros términos de una progresión geométrica de primer término A y razón R (dados por teclado). Se debe realizar la suma sin emplear la fórmula que existe para ello. Muestra también los términos de la serie.

Ejemplo de ejecución:

Introducir número de términos

6

Introducir el primer término

5

Introducir la razón

3

Salida:

5 15 45 135 405 1215

La suma de los términos de la serie es 1820

17. Se dice que un número N es perfecto si la suma de sus divisores, excluido el propio número es N. Por ejemplo, 28 es perfecto, pues sus divisores son: 1, 2, 4, 7 y 14 y su suma es $1+2+4+7+14=28$.

Haz un programa que dado un número N nos diga si es o no perfecto. Cambia el programa para que siga pidiendo números mientras el número introducido sea distinto de cero, que será la señal para parar el programa.

18. Hacer un programa que sea capaz de calcular el impuesto sobre la renta y lo escriba por pantalla hasta que se introduzca un salario igual a cero. El impuesto de la renta es el 15% del salario anual de cada persona, al que previamente se debe realizar una deducción en función del número de hijos, que es del 0% si tiene 0, del 5% si tiene 1 o 2 y del 15% si tiene más de 2. Tanto el salario como el número de hijos se pedirán por teclado.

19. Hacer un programa que lea por teclado un numero N e imprima un triángulo rectángulo, de N filas. Ej: N=5, se pintará lo siguiente:

```
*  
* *  
* * *  
* * * *  
* * * * *
```

20. Hacer un programa en el que se pida por teclado un número mayor que 2 (el programa controlará que cumpla esto), y lo imprima de todas las formas posibles como producto de dos factores (no se tiene en cuenta la multiplicación por 1).

Ej: Con el número 36, tendría que visualizarse: 18*2, 12*3, 9*4, 6*6, 3*12, 4*9, 2*18

Básicos Hoja 2

1. Escribir un programa que imprima cada uno de los términos de la serie 2, 5, 7, 10, 12, 15, 17,..., 1800. Además calcule e imprima la suma de los términos.

2. Leer un capital C y averiguar e imprimir en cuantos meses se duplica, si lo colocamos a un interés compuesto del 5% mensual.

La fórmula a aplicar es:

$$C_f = C_i (1 + i)^n$$

Donde:

C_f = Capital final

C_i = Capital inicial

i = Tasa de interés

n = Período del ahorro

3. En 1980 la ciudad A tenía 3.5 millones de habitantes y una tasa de crecimiento del 7% anual; y la ciudad B tenía 5 millones de habitantes y una tasa de crecimiento del 5% anual.

Si el crecimiento poblacional se mantiene constante en las dos ciudades, hacer un programa que calcule e imprima en qué año la población de la ciudad A es mayor que la de la ciudad B.

4. Haz un programa que lea un capital y calcule e imprima en cuantos meses se triplica si se coloca a un interés del 6% mensual.

5. Haz un programa que lea un número entero N y calcule el resultado de la siguiente serie: $1+1/2+1/3+1/4+1/5+\dots+1/N$.

6. Haz un programa que lea un número entero N y calcule el resultado de la siguiente serie: $1-1/2+1/3-1/4+1/5-\dots+1/N$.

7. Haz un programa que calcule la suma de los números pares comprendidos entre 10 y 50.

8. Haz un programa para imprimir una tabla de tres columnas y N filas con los cuadrados y los cubos de los N primeros números. Pide N al usuario.

```
Introducir numero
5
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
```

9. Haz un programa que pida 2 números por teclado y calcule su producto mediante sumas sucesivas. Imprimir su resultado.

10. Hacer un programa que pida 2 números por teclado y calcule su división mediante restas sucesivas. Imprimir su resultado. Divide siempre el más grande entre el más pequeño.

Ejemplo: 1324 entre 312.

1324 - 312 = 1012 contamos una vez y seguimos porque $1012 \geq 312$
1012 - 312 = 700 contamos 2 veces y continuamos porque $700 \geq 312$
700 - 312 = 388 contamos 3 veces y continuamos porque $388 \geq 312$
388 - 312 = 76 contamos 4 veces y paramos porque $76 < 312$

Luego la división es 4 y el resto 76.

11. Haz un programa que lea un número entero y genere y escriba el número resultante de invertir sus cifras.

12. Haz un programa que indique si un número entero N es primo o no.
//Ejercicio 11 Estructuras repetitivas Pseint

13. Haz un programa que imprima el triángulo de Floyd hasta un valor dado. El triángulo contiene los números naturales correlativos, uno en la primera línea, dos en la segunda, etc.; es decir, en la fila n-ésima aparecen n valores.

Ejemplo:

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

14. Escribe un programa para calcular A elevado a B, siendo A un número real cualquiera y B un valor entero positivo o nulo (sin emplear métodos ya hechos, claro). Si el exponente no es positivo o nulo se sigue pidiendo hasta que lo sea.
//Ejercicio 9 Estructuras repetitivas Pseint

Básicos Hoja 3

1. Haz un programa en Java para jugar contra el ordenador a adivinar un número, generado aleatoriamente, entre 1 y 200. El usuario debe introducir un número por teclado y el programa le dirá mediante los símbolos '<' o '>', si el número introducido es menor o mayor que el generado por el ordenador. Finalmente, se mostrará un mensaje informando de cuantos intentos se han necesitado para adivinar el número y si no se adivina se mostrará un mensaje diciendo que ha perdido.
El número máximo de intentos se pedirá por teclado.

Explicación: Para generar el número aleatorio poner:

```
int numAleatorio = (int) (Math.random()*200+1);
```

2. Haz un programa en Java que muestre si dos números son o no amigos. Los números se pedirán por teclado.

Explicación: Se dice que dos números son amigos si la suma de los divisores del primero (excluido el propio número) es el segundo número y viceversa.

Ej: Los números 220 y 284 son amigos

$$220 = 1+2+4+5+10+11+20+22+44+55+110 = 284$$

$$284 = 1+2+4+71+142 = 220$$

3. Haz un programa que muestre un contador con 3 dígitos. Mostrará los números del 0-0-0 al 9-9-9, con la particularidad que cada vez que aparezca un 3 lo sustituya por una E.

4. Para obtener el número del tarot de una persona, hay que sumar los números de su fecha de nacimiento y reducirlo a un solo dígito.

Ej: 1 de Julio de 1990 → 1+7+1990 = 1998 → 1+9+9+8 = 27 → 2+7=9 El número del tarot es el 9.

Haz un programa que lea la fecha de nacimiento por teclado y escriba el número del tarot. La fecha estará formada por 3 números enteros, el día, el mes y el año (4 dígitos).

Básicos Hoja 4 - Nivel Examen

1. Haz un programa en Java que pida números hasta que se teclee uno negativo, y mostrar cuántos números se han introducido.
2. Crear un programa para calcular el salario semanal de los empleados a los que se les paga 15 euros por hora si estas no superan las 35 horas. Cada hora por encima de 35 se considerará extra y se paga a 22 €. El programa pide las horas del trabajador y devuelve el salario que se le debe pagar.
Además, el programa debe preguntar si deseamos calcular otro salario, si es así el programa se vuelve a repetir.
3. Haz un programa que pida un número n, y diga cuántos y cuáles son los números primos que hay entre 1 y n.
4. Haz un programa que vaya pidiendo números, hasta introducir uno negativo, y diga cuál es el mayor número introducido y cuantas veces se repite.

5. Haz un programa que pida un número, entre 0 y 10, y escriba un triángulo invertido con dichos números.

Ej: Si n=10, quedaría

0,1,2,3,4,5,6,7,8,9

1,2,3,4,5,6,7,8,9

2,3,4,5,6,7,8,9

3,4,5,6,7,8,9

4,5,6,7,8,9

5,6,7,8,9

6,7,8,9

7,8,9

8,9

9

6. Un alumno desea saber que nota necesita en el tercer examen para aprobar una evaluación. El promedio de la evaluación se calcula con la siguiente formula.

$$NC = (E1 + E2 + E3) / 3$$

$$NF = NC * 0.7 + NL * 0.3$$

Donde NC es el promedio de los exámenes, NL el promedio de laboratorio y NF la nota final.

Escribe un programa que pregunte al usuario las notas de los dos primeros exámenes y la nota promedio de laboratorio, y muestre la nota que necesita el alumno en el tercer examen para aprobar la evaluación con nota final 6.0.

Ejemplo:

Ingresar nota examen 1: 4.5

Ingresar nota examen 2: 5.5

Ingresar nota laboratorio: 6.5

Necesita nota 7.4 en el examen 3

7. La secuencia de Collatz de un número entero se construye de la siguiente forma:

- si el número es par, se divide por dos;
- si es impar, se multiplica tres y se le suma uno;
- la sucesión termina al llegar a uno.

La conjetura de Collatz afirma que, al partir desde cualquier número, la secuencia siempre llegará a 1. A pesar de ser una afirmación a simple vista muy simple, no se ha podido demostrar si es cierta o no.

Usando computadores, se ha verificado que la sucesión efectivamente llega a 1 partiendo desde cualquier número natural menor que 258.

a) Hacer un programa que muestre la secuencia de Collatz de un número entero, que se pedirá por teclado.

Ej:

n: **18**

18 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

n: **19**

19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

n: **20**

20 10 5 16 8 4 2 1

b) Hacer un programa que grafique los largos de las secuencias de Collatz (número de elementos que hay que generar hasta que se llega al 1) de los números enteros positivos menores que el ingresado por el usuario:

n: **20**

```
1 *
2 **
3 ******
4 ***
5 ****
6 ******
7 *********
8 ****
9 *********
10 *****
11 *********
12 ******
13 ******
14 *********
15 *********
16 ****
17 *********
18 *********
19 *********
20 *****
```

UT4 Programación Orientada a Objetos

1. Programación Orientada a Objeto (POO)

La programación Orientada a Objetos es una metodología que basa la estructura de los programas en torno a los objetos.

Los lenguajes de POO ofrecen medios y herramientas para describir los objetos manipulados por un programa. Más que describir cada objeto individualmente, estos lenguajes proveen una construcción (Clase) que describe a un conjunto de objetos que poseen las mismas propiedades.

2. Clase

Una clase es un tipo definido por el usuario (plantilla) que describe los atributos y los métodos de los objetos que se crearán a partir de ella.

El estado de un objeto viene determinado por sus **atributos**, y su comportamiento está definido por sus **métodos**.

Dentro de las clases también se encuentran los **constructores** que permiten inicializar un objeto.

Los atributos y los métodos se denominan en general **miembros de la clase**.

La definición de una clase consta de dos partes: el nombre de la clase precedido por la palabra reservada **class** y el cuerpo de la clase entre llaves. La sintaxis queda:

```
class nombre-clase {  
    cuerpo de la clase  
}
```

Dentro del cuerpo de la clase se puede encontrar los atributos y métodos.

Ej:

```
class Circunferencia {  
    private double x, y, radio;  
    public Circunferencia(){ }  
  
    public Circunferencia(double cx, double cy, double r){  
        x=cx;  
        y=cy;  
        radio=r;  
    }  
    public void ponRadio(double r){  
        radio=r;  
    }  
    public double longitud(){  
        //No hay que pasarle el radio, coge el radio de la  
        circunf  
        return 2*Math.PI*radio;  
    }  
}
```

En el ejemplo, se define la **clase Circunferencia**, que puede ser usada dentro de un programa de la misma manera que cualquier otro tipo. Un objeto de esta clase tiene tres atributos (coordenadas del centro y valor del radio), dos constructores y dos métodos.

Los constructores se distinguen fácilmente porque tienen el mismo nombre que la clase.

Los atributos se declaran de la misma manera que cualquier variable. En una clase, cada atributo debe tener un nombre único.

Siguiendo las recomendaciones de la Programación Orientada a Objetos, **cada clase se debe implementar en un fichero .java, de esta manera es más sencillo modificar la clase.**

El fichero que contiene la clase debe llevar el nombre de la clase pública con la extensión .java. **A veces tendremos varias clases en un .java, pero sólo una puede ser pública**

3. Métodos

Los métodos forman lo que se denomina interfaz de los objetos, definen las operaciones que se pueden realizar con los atributos. Desde el punto de vista de la Programación Orientada a Objetos, el conjunto de métodos se corresponde con el conjunto de mensajes a los que los objetos de una clase pueden responder.

Los métodos implementan la funcionalidad asociada al objeto. Los métodos son el equivalente a las funciones en Programación Estructurada. Se diferencian de ellos en que es posible **acceder** a las **variables de la clase de forma implícita** (atributos).

Cuando se desea realizar una acción sobre un objeto, se dice que se le manda un mensaje invocando a un método que realizará la acción.

Los métodos permiten al programador modularizar sus programas.

Todas las variables declaradas en las definiciones de métodos son variables locales, solo se conocen en el método que las define. Casi todos los métodos tienen una lista de parámetros que permiten comunicar información.

La sintaxis para definir un método es:

```
<modificador-acceso> <modificador > tipoR nombre-método(<parámetros>) {  
    <cuerpodelmétodo>  
}
```

<modificador-acceso> indica cómo es el acceso a dicho método: **public**, **private**, **protected** y sin modificador.

<modificador > indica si es **estática** o no.

<tipoR> es el tipo de dato que retorna el método. Es obligatorio indicar un tipo. Para los métodos que no devuelven nada se utiliza la palabra reservada **void**.

Nombre-método es cómo el programador quiere llamar a su método y <parámetros> son los datos que se van a enviar al método para trabajar con ellos, no son obligatorios.

Para devolver el valor se utiliza el operador **return**. Una vez que se ejecute **return**, el control se devuelve al código que lo llamó y la ejecución del método termina. Los métodos que no lleven **return**, tendrán tipo **void**.

Ej:

```
int suma(int x, int y) {  
    return x+y;  
}
```

En este ejemplo el método recibe 2 parámetros de tipo int, realiza su suma y devuelve esta suma, que es un valor de tipo int.

```
void imprimir(){  
    System.out.println("Este método no devuelve nada y tampoco  
    recibe parámetros");  
}
```

En este ejemplo el método no devuelve ningún valor, ni recibe parámetros, aun así es necesario poner los paréntesis vacíos.

3.1 Métodos Estáticos o de Clase - hay otro word

Van precedidos del modificador **static**.

Para invocar a un **método** estático no se necesita crear un objeto de la clase en la que se define. **Si se invoca desde la clase** en la que se encuentra definido, basta con escribir su nombre.

Si se le invoca desde una clase distinta, debe anteponerse a su nombre, el de la clase en que se encuentra seguido del operador punto (.). La sintaxis es:

```
< NombreClase> .metodoEstatico( );
```

Suelen emplearse para realizar operaciones comunes a todos los objetos de la clase. No afectan a los estados de los objetos.

Por ejemplo, si se necesita un método para contabilizar el número de objetos creados de una clase, se define estático ya que su función, aumentar el valor de una variable entera, se realiza independientemente del objeto empleado para invocarle. Esa variable entera, sería única para todos los objetos.

Otra razón por la que tendríamos que usar métodos estáticos es si se utilizan fuera del contexto de cualquier instancia

Los métodos estáticos sólo pueden llamar a otros métodos static directamente, y no se pueden referir a this o super de ninguna manera.

No es conveniente usar muchos métodos estáticos, pues si bien se aumenta la rapidez de ejecución, se pierde flexibilidad, no se hace un uso efectivo de la memoria y no se trabaja según los principios de la POO.

Las variables también se pueden declarar como static, y es equivalente a declararlas como variables globales, que son accesibles desde cualquier fragmento de código.

Se puede declarar un bloque static que se ejecuta una sola vez si se necesitan realizar cálculos para inicializar las variables static.

Ej:

```
class Estatica {  
    static int a=3,b;  
    static{  
        System.out.println("Bloque static inicializado");  
        b=a*4;  
    } // Si no es static no compilará  
  
    static void metodo(int x){  
        System.out.println("x= "+x);  
        System.out.println("a= "+a);  
        System.out.println("b= "+b);  
    }  
  
    public static void main(String[] args) {  
        metodo(42);  
    }  
}
```

En el ejemplo la clase que tiene dos variables static, un bloque de inicialización static y un método static. La salida del programa es:

```
Bloque static inicializado:  
x = 42  
a = 3  
b = 12
```

3.2 Modificadores de acceso

Se profundizará sobre los modificadores de acceso cuando veamos Herencia y Paquetes en Java. De momento, pondremos que un atributo/método es privado (**private**) si solo lo utilizamos dentro de su clase, si lo queremos usar fuera lo pondremos público (**public**).

Modificadores de acceso				
	La misma clase	Otra clase del mismo paquete	Subclase de otro paquete	Otra clase de otro paquete
public	X	X	X	X
protected	X	X	X	
default	X	X		
private	X			

4. Objetos

Es una entidad (tangible o intangible) que posee características y acciones que realiza por sí solo o interactuando con otros objetos.

Un **objeto** es una entidad caracterizada por sus atributos propios y cuyo comportamiento está determinado por las acciones o funciones que pueden modificarlo (métodos), así como también las acciones que requiere de otros objetos.

Un objeto constituye una unidad que **oculta** tanto datos como la descripción de su manipulación. Puede ser definido como una **encapsulación** y una **abstracción**: una encapsulación de atributos y servicios, y una abstracción del mundo real.

Para el contexto del Enfoque Orientado a Objetos (EOO) un objeto es una entidad que encapsula datos (atributos) y acciones o funciones que los manejan (métodos). También para el EOO un objeto se define como una instancia o particularización de una clase.

Los objetos de interés durante el desarrollo de software no solo son tomados de la vida real (objetos visibles o tangibles), también pueden ser abstractos. En general, son entidades que juegan un rol bien definido en el dominio del problema. Un libro, una persona, un coche, un polígono son algunos **ejemplos** de objeto.

Cada objeto puede ser considerado como un proveedor de servicios utilizados por otros objetos que son sus clientes. Cada objeto puede ser a la vez proveedor y cliente. De ahí que un programa pueda ser visto como un conjunto de relaciones entre proveedores clientes. Los servicios ofrecidos por los objetos son de dos tipos:

- 1.- Los datos, que llamamos **atributos**.
- 2.- Las acciones o funciones, que llamamos **métodos**.

Un objeto consta de una estructura interna (los atributos) y de una interfaz que permite acceder y manipular dicha estructura (los métodos). Para **construir** un objeto de una clase cualquiera hay que llamar a un método de iniciación, el **constructor**. Para ello, se utiliza el operador **new**. La sintaxis es la siguiente:

```
Nombre-clase nombre-objeto = new Nombre-clase (<valores>);
```

Donde **Nombre-clase** es el nombre de la clase de la cual se quiere crear el objeto, **nombre-objeto** es el nombre que el programador quiere dar a ese objeto y <valores> son los valores con los que se inicializa el objeto. Dichos valores son opcionales y aparecen en el constructor.

Ej: Circunferencia circ = new Circunferencia();

Se crea el objeto circ, de la clase Circunferencia, con los valores predeterminados.

Cuando se crea un objeto, Java hace lo siguiente:

- **Asignar memoria** al objeto por medio del operador **new**.
- **Llamar al constructor** de la clase para inicializar los atributos de ese objeto con los valores iniciales o con los valores predeterminados por el sistema: los atributos numéricos a cero, los alfanuméricos a nulos y las referencias a objetos a null.

Si no hay suficiente memoria para ubicar el objeto, el operador new lanza una excepción OutOfMemoryError.

Para acceder desde un método de una clase a un miembro de un objeto de otra clase diferente se utiliza la sintaxis: **objeto.miembro**. Cuando el miembro accedido es un método se entiende que el objeto ha recibido un mensaje, el especificado por el nombre del método y responde ejecutando ese método. En este caso, después del nombre del método siempre se pone paréntesis, aunque no haya parámetros.

Características Generales:

- Un objeto posee **estados**. El estado de un objeto está determinado por los **valores** que poseen sus **atributos** en un momento dado.
- Un objeto tiene un **conjunto de métodos**. El comportamiento general de los objetos dentro de un sistema se describe o representa mediante sus operaciones o métodos. Los métodos se utilizarán para obtener o cambiar el estado de los objetos, así como para proporcionar un medio de **comunicación** entre objetos.
- Un objeto tiene un **conjunto de atributos**. Los atributos de un objeto contienen valores que determinan el estado del objeto durante su **tiempo de vida**. Se implementan con variables, constantes y estructuras de datos (similares a los campos de un registro).

4.1 Asignación

Una vez el objeto está creado, se tiene una **referencia a ese objeto en la variable** donde se asigna. Si se realiza la asignación de un objeto a otro los dos harán referencia al mismo objeto.

Ej:

```
Circunferencia c1 = new Circunferencia(0, 0, 15);
Circunferencia c2 = c1;
c1.ponRadio(25);
System.out.println(c2.radio); // Es el mismo objeto
```

En el ejemplo, la variable c1 apunta a un objeto de la clase Circunferencia. Debido a la asignación, la variable c2 apunta al mismo objeto. Después se modifica el valor del radio del objeto apuntado por c1. Y por último, se visualiza c2 que será el valor 25.

4.2 Igualdad

Creamos dos objetos iguales (con los mismos valores de sus variables miembro), al preguntar si las variables que apuntan a esos objetos son iguales nos devolverá false, pues aunque tengan el mismo valor no son el mismo objeto.

Ej:

```
Circunferencia c1 = new Circunferencia(0,0,15);  
Circunferencia c2 = new Circunferencia(0,0,15);  
if (c1==c2) /* Esto es falso*/
```

4.3 Constructor

Un Constructor es un método especial en Java empleado para inicializar valores en instancias de objetos. A través de este tipo de métodos es posible generar distintos tipos de instancias para la clase en cuestión.

Los métodos constructores tienen las siguientes **características**:

- Se llaman igual que la clase.
- No devuelven nada, ni siquiera void.
- Puede haber varios constructores, que deberán distinguirse por el tipo de valores que reciban.
- De entre los que existan, solo uno se ejecutará al crear el objeto.
- El código de un constructor, generalmente, suele ser inicializaciones de variables y objetos, para conseguir que el objeto sea creado con dichos valores iniciales.
- Si no se define ningún constructor el compilador crea uno **por defecto** sin parámetros que, al ejecutarse, inicializa el valor de cada atributo de la nueva instancia a 0, false o null, dependiendo de si el atributo es numérico, alfanumérico o una referencia a otro objeto respectivamente, pero dicho constructor desaparece en el mismo momento en que se defina otro constructor, por lo que **si se quiere tener el de por defecto habrá que definirlo.**

La **declaración de constructores** sigue la siguiente sintaxis:

```
<modificador deVisibilidad> NombredelaClase ( <argumentos> ) {  
    < declaraciones>  
}
```

Donde **<modificador deVisibilidad>** es el tipo de modificador de acceso del constructor, **<Nombredelaclase>** es el nombre del constructor y debe coincidir con el de la clase y **<argumentos>** son las variables que recibe el constructor y contienen los valores con los que se inicializaran los atributos.

Ej:

```
class Circunferencia{  
    private double x, y, radio;  
    public Circunferencia(){ }  
    public Circunferencia(double cx, double cy, double r){  
        x=cx; y=cy; radio=r;  
    }  
}
```

En este ejemplo existen 2 constructores, el primero que es el de por defecto y el segundo que inicializa los atributos x, y y radio con los valores cx, cy y r, respectivamente.

4.4 Referencia this

Java incluye un valor de referencia especial llamado **this**, que se utiliza dentro de cualquier método para referirse al objeto actual. **El valor this se refiere al objeto sobre el que ha sido llamado el método actual.**

Se puede utilizar this siempre que se requiera una referencia a un objeto del tipo de una clase actual. Si hay dos objetos que utilicen el mismo código, seleccionados a través de otras instancias, cada uno tiene su propio valor único de this.

Normalmente, dentro del cuerpo de un método de un objeto se puede referir directamente a las variables miembros del objeto. Sin embargo, algunas veces no se querrá tener ambigüedad sobre el nombre de la variable miembro y uno de los argumentos del método que tengan el mismo nombre y usaremos la referencia this.

Ej:

```
class Circunferencia {  
    private double x, y, radio;  
    public Circunferencia(double x, double y, double radio) {  
        this.x=x;  
        this.y=y;  
        this.radio=radio;  
    }  
}
```

En el ejemplo, el constructor de la clase inicializa las variables con los argumentos pasados al constructor. Se debe utilizar `this` en este constructor para evitar la ambigüedad entre los argumentos y los atributos miembro.

También se puede utilizar `this` para llamar a uno de los métodos del objeto actual. Esto solo es necesario si existe alguna ambigüedad con el nombre del método y se utiliza para intentar hacer el código más claro.

UT4 Objetos y clases: Introducción

1. El concepto de clase

Si pensamos en las **clases de objetos** que vemos, tenemos coches, bares, semáforos, personas...

Si simplificamos esta definición de *clases de objetos* por simplemente *clases*:

¿Qué **clases** vemos? Coches, bares, semáforos, personas...

Y ya estaremos usando terminología de la **programación orientada a objetos**:

Tenemos la **clase Coche**, la clase *Bar*, la clase *Semáforo*, la clase *Persona*...

No nos importa si esa persona es *Marta*, está en el bar “*Casa Manolo*” (*Bar Casa Manolo sería un objeto de clase Bar*), si tiene un *Seat Panda* o si el semáforo está en *rojo*. Estamos pensando en términos **clasificatorios**.

Vamos a pensar ahora en cada una de estas clases:

¿Cómo son los coches? ¿Qué **características** pueden diferenciar uno de otro? Por ejemplo, podríamos indicar la *marca*, el *modelo*, el *color*, la *cilindrada*, etc.

A las características las vamos a denominar **atributos**, término muy utilizado en la **programación orientada a objetos**.

Atributo: variable dentro del objeto

¿Qué **atributos** tiene la **clase Bar**? Todo bar tiene un **nombre**, una **ubicación**, un **estado** (si está abierto o cerrado), una **lista de precios**,...

Como atributos de la clase *Semáforo* podríamos indicar su *ubicación* y *estado* (rojo, verde o ámbar).

La clase *Persona* podría definir como atributos el *nombre*, *sexo*, *edad*, *estado civil*, *altura*, etc.

A continuación vamos a pensar en el **comportamiento** de estas clases. Nos preguntaremos qué cosas hacen, qué tipo de acciones pueden realizar...

Un coche puede arrancar, detenerse, girar a la izquierda, acelerar, frenar, encender sus luces.

Método: funciones que cambian el estado, a veces cambian atributos

Un semáforo puede cambiar de estado.

Un bar puede abrir, cerrar, servirte una cerveza, cobrarla, modificar la lista de precios.

Una persona puede hablar, dormir, conducir un coche, tomarse una cerveza en un bar.

En terminología de la **programación orientada a objetos**, a estas funciones que determinan el *comportamiento* de una clase se las conoce como **métodos**.

Sabemos lo que es una **clase** y que está compuesta de **atributos** y **métodos**. Nuestra labor ahora consistirá en **modelar** en Java estas clases.

Modelar no es otra cosa sino crear una **abstracción** que represente de algún modo una determinada realidad.

Para crear en Java la clase *Coche* procederíamos del siguiente modo:

```
class Coche  
{  
}
```

Entre las llaves introduciremos los *atributos* y *métodos* de que consta la clase. Usamos para ello la palabra reservada **class**.

El nombre de la clase *Coche* lo escribimos con la primera letra en mayúsculas, pues es de común acuerdo entre los programadores en Java que los nombres de clases empiecen así.

Introduzcamos algunos atributos:

```
class Coche  
{  
    String marca;  
    String modelo;  
    String color;  
    int numeroDePuertas;  
    int cuentaKilometros;  
    int velocidad;  
    boolean arrancado;  
}
```

Declarar atributos es algo similar a declarar una variable normal. El nombre del atributo se precede por su tipo. Así, en el ejemplo, tenemos tres atributos de tipo *String* que contendrán cadenas de caracteres; otros tres de tipo *int* para almacenar valores enteros; finalmente, el atributo *arrancado*, que utilizaremos para indicar si el coche está en marcha o no, es de tipo *boolean*, admitiendo como posibles valores *true* o *false*.

Los tipos *int* y *boolean* forman parte de los tipos básicos de Java, conocidos como tipos **primitivos**. Los presentaremos formalmente a su debido momento; por ahora es suficiente con que conozcamos su existencia y cómo los utilizamos.

El tipo *String*, que escribimos con la primera letra en mayúsculas (lo que debería darte una pista), no es más que otra clase, como lo son las clases *Coche* y *Persona*. Es una clase muy importante en Java que *encapsula* un buen conjunto de métodos para trabajar con cadenas de caracteres.

El concepto importante que debes entender es que los atributos no necesariamente son siempre de tipos básicos, sino que también pueden ser de cualquier clase, incluso de una propia que nosotros mismos hayamos creado.

Por ejemplo, podríamos definir un nuevo atributo de la clase *Coche*, llamado *conductor*, en el que figure la persona (de tipo *Persona*) que lo conduce:

```
Persona conductor;
```

Fíjate también en la convención utilizada para nombres de variables compuestos de varias palabras. Se escriben todas juntas, pero iniciando cada palabra en mayúsculas, a excepción de la primera. Esto forma parte también del estilo de escritura Java.

Ten en cuenta que Java distingue mayúsculas de minúsculas. No es lo mismo *numeroDePuertas* que *numerodepuertas*.

Definamos ahora los métodos de la clase *Coche*:

```
class Coche
{
    String marca;
    String modelo;
    String color;
    int numeroDePuertas;
    int cuentaKilometros;
    int velocidad;
    boolean arrancado;

    void arrancar()
    {
    }

    void parar()
    {
    }

    void acelerar()
    {
    }

    void frenar()
    {
    }

    void pitar()
    {
    }

    int consultarCuentaKilometros()
    {
    }
}
```

El nombre de método viene seguido por un par de paréntesis, pues en ocasiones los métodos podrán recibir *argumentos* que luego se utilizarán en el cuerpo del método. Aunque el método no requiera argumentos los paréntesis son absolutamente necesarios.

Por otro lado, el nombre del método va precedido por el *tipo* del valor que devuelve. Hay que indicarlo incluso si el método no devuelve explícitamente ningún valor. Ese caso se indica con el tipo **void**.

Entre el par de llaves { } introduciremos el cuerpo del método, las instrucciones que indican su operatividad.

Escribamos algo de código básico en cada uno de ellos:

```
class Coche
{
    String marca;
    String modelo;
    String color;
    int numeroDePuertas;
    int cuentaKilometros;
    int velocidad;
    boolean arrancado;

    void arrancar()
    {
        arrancado = true;
    }

    void parar()
    {
        arrancado = false;
    }

    void acelerar()
    {
        velocidad = velocidad + 1;
    }

    void frenar()
    {
        velocidad = velocidad - 1;
    }

    void pitar()
    {
        System.out.println("Piiiiiiiiiiiiiiii");
    }

    int consultarCuentaKilometros()
    {
        return cuentaKilometros;
    }
}
```

Los métodos *arrancar()* y *parar()* establecen el atributo booleano *arrancado* a *true* y *false*, respectivamente. Los métodos *acelerar()* y *frenar()* incrementan y decrementan en una unidad, respectivamente, el atributo *velocidad*. El método *pitar()* imprime en consola una cadena de caracteres. El método *consultarCuentaKilometros()* devuelve a **quien lo invoca** (fíjate en el *return*) lo que contiene el atributo *cuentaKilometros*. Observa que es el único que devuelve explícitamente un valor (de tipo entero); los restantes, aunque algunos modifican los atributos de la misma clase, no devuelven ningún valor a quien los llama.

Podríamos modelar la clase *Persona* del siguiente modo:

```
class Persona
{
    char sexo;
    String nombre;
    int edad;
    Coche coche; // El coche que conduce esa
    persona

    void saludar()
    {
        System.out.println("Hola, me llamo " +
    nombre);
    }

    void dormir()
    {
        System.out.println("Zzzzzzzzz");
    }

    int obtenerEdad()
    {
        return edad;
    }
}
```

Fíjate que uno de los atributos es precisamente de la clase *Coche* que acabamos de definir:

Coche coche

2. El concepto de objeto

Una vez hemos entendido el concepto de **Clase** en el paradigma de la *Programación Orientada a Objetos*, es momento de matizar qué es un **Objeto** en su sentido más práctico.

Una clase es una plantilla a partir de la cual vamos a crear objetos.

Una **Clase**, como hemos visto, no es más que una especificación que define las *características* y el *comportamiento* de un determinado tipo de objetos. Piensa en ella como si se tratara de una plantilla, molde o esquema a partir del cual podremos construir **objetos concretos**.

Consideremos la clase *Coche* que definimos en el artículo anterior:

Con esta plantilla, vamos a crear coches **concretos** de la marca, modelo y color que nos apetezca. Cada uno que creemos será un **objeto**, o **instancia**, de la clase *Coche*.

Fíjate, en esta terminología, en la equivalencia de los términos **objeto** e **instancia** para referirnos, ahora sí, a **entidades concretas (objetos)** de una determinada clase.

Fabriquemos, entonces, nuestro primer coche...

Cada objeto, como todas las variables en Java, ha de ser **declarado** antes de ser utilizado:

```
Coche coche1
```

Con esta instrucción declaramos la variable *coche1* de tipo *Coche*. En cuanto creemos, con el comando que escribiremos a continuación, el objeto concreto, *coche1* contendrá una **referencia** a ese objeto, es decir, almacenará la dirección de memoria en la que realmente se halla el objeto propiamente dicho.

Esto es muy importante: *coche1* no contendrá el objeto en sí, sino una dirección de memoria que apunta a él.

Materialicemos nuestro primer coche del siguiente modo:

```
coche1 = new Coche()
```

La palabra reservada **new** se emplea para **crear nuevos objetos, instancias** de una determinada clase que indicamos a continuación, seguida de un par de paréntesis.

Veremos esto a su debido momento, pero por ahora tenemos bastante con retener que se está invocando a un método especial que tienen todas las clases, que sirve para **construir** el objeto en cuestión facilitándole sus valores iniciales. A este método se le conoce como **constructor** de la clase.

Podríamos haber realizado la declaración y la creación en una sola instrucción:

```
Coche coche1 = new  
Coche()
```

Ahora, vamos a decir de qué marca y modelo es:

```
coche1.marca = "Seat"  
coche1.modelo = "Panda"
```

(entramos dentro del objeto *coche1* y cambiamos los atributos)

Consulta el cuadro de arriba con la definición de la clase y observa que tanto *marca* como *modelo* son dos atributos de tipo *String*, por lo que referencian *cadenas de caracteres* que escribimos entre comillas.

Observa con cuidado la **notación punto**. Separamos el nombre de la variable que referencia al objeto del atributo empleando un **punto como separador**.

Pintemos de azul nuestro vehículo:

```
coche1.color = "Azul"
```

Además tiene tres puertas, el cuenta kilómetros indica 250.000 y su velocímetro refleja 0 Km/h. Valores correspondientes a los atributos *numeroDePuertas*, *cuentaKilometros* y *velocidad*, respectivamente, todos de tipo entero.

```
coche1.numeroDePuertas = 3  
coche1.cuentaKilometros = 250000  
coche1.velocidad = 0
```

Su motor está detenido, hecho que representamos a través de la variable booleana *arrancado*:

```
coche1.arrancado = false
```

Ahora vamos a experimentar con los *métodos*. Arranquemos el *Panda*:

```
coche1.arrancar()
```

De nuevo, empleamos también la notación punto para separar la variable del método.

Si observas el código verás que este método se limita a hacer que la variable booleana *arrancado* valga ahora *true*. Podrías decir que hubiéramos logrado el mismo resultado actuando sobre el atributo directamente, en lugar de invocar al método:

```
coche1.arrancado = true
```

En efecto, es así; pero, como comprenderás más adelante, no suele ser buena idea dejar que los programas campen a sus anchas y modifiquen arbitrariamente los atributos de un objeto, siendo preferible que sean los métodos los que se ocupen de esa labor. Imagina que el programa intenta hacer que el cuenta kilómetros marque una cantidad negativa, o que el número de puertas sea igual a cincuenta. Un método correctamente diseñado podría gestionar que los valores estuvieran dentro del rango adecuado y asegurarse de que se cumplen las condiciones que permitirían la modificación del atributo.

La *Programación Orientada a Objetos* implementa un mecanismo, denominado **encapsulación**, que nos permite **ocultar** determinadas facetas de nuestro objeto y dejar solo accesibles aquellas partes que nos interesen. Pero vayamos por orden, todo a su momento...

Vamos a acelerar nuestro coche:

```
coche1.acelerar()
```

Lo que provocará, si consultas el código, que el velocímetro incremente en una unidad su valor.

```
coche1.pitar()
```

Lo que ocasionará un pitido.

Puedes crear todos los objetos de la clase *Coche* que deseas:

```
Coche coche2 = new Coche()
```

Cada uno con su propia colección de atributos:

```
coche2.marca = "Ford"  
coche2.modelo = "Fiesta"  
coche2.color = "Negro"
```

Incluso podrías crear otros *Seat Pandas*, por supuesto. Aunque, en un instante dado, compartieran los mismos valores en sus atributos, se trataría de objetos distintos, **ubicados en direcciones de memoria diferentes** y cada uno podría seguir su propia trayectoria vital.

Extraído de:

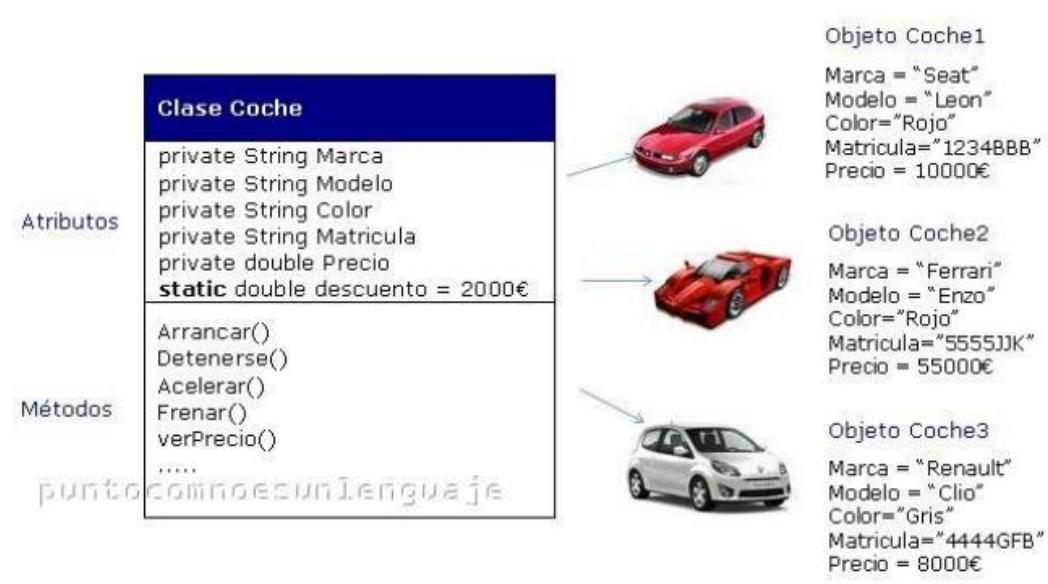
<http://elclubdelautodidacta.es/wp/indice-java/>

UT4 Java static. Atributos y métodos estáticos o de clase

Los atributos y métodos estáticos también se llaman **atributos de clase** y **métodos de clase**. Se declaran como **static**.

Supongamos una clase **Coche** sencilla que se utiliza en un programa de compra-venta de coches usados y de la que se crean 3 objetos de tipo **Coche**.

La clase contiene 6 atributos: marca, modelo, color, matrícula, precio y descuento. Supongamos que el descuento es una cantidad que se aplica a todos los coches sobre el precio de venta. Como este dato es el mismo para todos los coches y es un valor que se puede modificar en cualquier momento no debe formar parte de cada coche sino que es un dato que deben compartir todos. Esto se consigue declarándolo como **static**.



1. Atributos static:

Son **propios** únicamente **de la clase** y **no de los objetos que pueden crearse de la misma**, por lo tanto, sus valores son compartidos por todos los objetos de la clase. Van **precedidos del modificador static**.

Un atributo static:

- No es específico de cada objeto. Solo hay una copia del mismo y su valor es compartido por todos los objetos de la clase.
- Podemos considerarlo como una variable global a la que tienen acceso todos los objetos de la clase.
- Existe y puede utilizarse aunque no existan objetos de la clase.

Para invocar a una variable estática **no se necesita crear un objeto de la clase en la que se define**.

- Si se invoca **desde la clase** en la que se encuentra definido, basta con escribir su nombre.

- Si se le invoca **desde una clase distinta**, debe anteponerse a su nombre, el de la clase en la que se encuentra definido seguido del operador punto (.) <NombreClase>.variableEstatica.

Ejemplo1:

```

public class SerHumano{
    String nombre;
    String colorOjos;
    int edad;

    /*
     * Declaración e inicialización de una variable de instancia estática
     * Tiene sentido declararla estática pues todos los objetos
     * de la clase, teniendo en cuenta que esta modela a un ser humano,
     * habitan en el mismo planeta
    */
    static String planeta="Tierra";

    void mostrarCaracteristicas () {
        System.out.println(nombre+" tiene "+edad+" años");
        System.out.println("Sus ojos son "+colorOjos);
        System.out.println("Su planeta es "+planeta);
    }
    void esMayorEdad() {
        if(edad>=18) {
            System.out.println(nombre+" es mayor de edad");
            System.out.println("Tiene "+edad+" años");
        }
        else{
            System.out.println(nombre+" es menor de edad");
            System.out.println("Tiene "+edad+" años ");
        }
    }
    public static void main(String args[]){
        SerHumano sh1=new SerHumano ();
        sh1.nombre="Jesus";
        sh1.colorOjos="azules";
        sh1.edad=28;
        sh1.mostrarCaracteristicas();
        sh1.esMayorEdad();

        System.out.println("-----");

        SerHumano sh2=new SerHumano ();
        sh2.nombre="Rebeca";
        sh2.colorOjos="verdes";
        sh2.edad=27;
        sh2.mostrarCaracteristicas();
        sh2.esMayorEdad();

        System.out.println("-----");
        System.out.println("FIN DEL PROGRAMA");
    }
}

```

2. Métodos static:

Van **precedidos del modificador static.**

Para invocar a un método estático **no se necesita crear un objeto de la clase en la que se define:**

- Tiene acceso solo a los **atributos estáticos de la clase.**
- No es necesario instanciar un objeto para poder utilizarlo.
Si se le invoca desde una clase distinta, debe anteponerse a su nombre, el de la clase en la que se encuentra seguido del operador punto (.)
`<NombreClase>.metodoEstatico`

Ejemplo:

Vamos a escribir una clase Persona que contendrá un atributo contadorPersonas que indique cuántos objetos de la clase se han creado. contadorPersonas debe ser un atributo de clase ya que no es un valor que se deba guardar en cada objeto persona que se crea, por lo tanto se debe declarar static:

```
public static int contadorPersonas;
```

Si lo declaramos como privado:

```
private static int contadorPersonas;
```

Desde fuera de la clase Persona solo podremos acceder al atributo a través de métodos static:

```
public static void incrementarContador() {  
    contadorPersonas++;  
}  
public static int getContadorPersonas() {  
    return contadorPersonas;  
}
```

En este caso un ejemplo de uso puede ser:

```
System.out.println(Persona.getContadorPersonas());
```

Cada vez que se crea una persona se incrementará su valor.

Si no es private, desde otra clase podemos hacerlo así:

```
Persona.contadorPersonas++;
```

Si es private, desde otra clase debemos incrementarlo así:

```
Persona.incrementarContador();
```

```
//Clase Persona
public class Persona {

    private String nombre;
    private int edad;
    private static int contadorPersonas;

    public Persona() {
    }

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public void setNombre(String nom) {
        nombre = nom;
    }

    public String getNombre() {
        return nombre;
    }

    public void setEdad(int ed) {
        edad = ed;
    }

    public int getEdad() {
        return edad;
    }

    public static int getContadorPersonas() {
        return contadorPersonas;
    }

    public static void incrementarContador() {
        contadorPersonas++;
    }
}
```

```
//Clase Principal
public class Estatico1 {
    public static void main(String[] args) {
        Persona p1 = new Persona("Tomás Navarra", 22);
        Persona.incrementarContador();
        Persona p2 = new Persona("Jonás Estacio", 23);
        Persona.incrementarContador();
        System.out.println("Se han creado: " +
    Persona.getContadorPersonas() + " personas");
    }
}
```

En lugar de utilizar el método incrementarContador() cada vez que se crea un objeto, podemos hacer el incremento de la variable estática directamente en el constructor.

El código de la clase Persona y de la clase principal quedaría ahora así:

```
//Clase Persona
public class Persona {

    private String nombre;
    private int edad;
    private static int contadorPersonas;

    public Persona() {
        contadorPersonas++;
    }

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
        contadorPersonas++;
    }

    public void setNombre(String nom) {
        nombre = nom;
    }

    public String getNombre() {
        return nombre;
    }

    public void setEdad(int ed) {
        edad = ed;
    }

    public int getEdad() {
        return edad;
    }

    public static int getContadorPersonas() {
        return contadorPersonas;
    }
}

//Clase Principal
public class Estatico1 {

    public static void main(String[] args) {
        Persona p1 = new Persona("Tomás Navarra", 22);
        Persona p2 = new Persona("Jonás Estacio", 23);
        System.out.println("Se han creado: " +
        Persona.getContadorPersonas() + " personas");
    }
}
```

No conviene usar muchos métodos estáticos, pues si bien se aumenta la rapidez de ejecución, se pierde flexibilidad, no se hace un uso efectivo de la memoria y no se trabaja según los principios de la Programación Orientada a Objetos.

NOTA: muchas clases de la API disponen de métodos estáticos. Por ejemplo, la **clase Math** del paquete java.lang cuenta con multitud de ellos. Estos métodos se emplean para realizar operaciones matemáticas. La **clase Thread**, del mismo paquete, cuenta con varios: uno que se emplea para retardar la ejecución de código es “**void sleep(long retardo)**”. Consultar la API. Lo importante de estos métodos es que para su utilización no es necesario instanciar un objeto de las clases en las que se encuentran ya que son estáticos.

```
public class MetodosEstaticosAPI{  
    public static void main(String args[]){  
        int num=100;  
        System.out.println("La raiz cuadrada de "+num+" es "+Math.sqrt(num));  
  
        //Bloque try ... catch. Se estudiará más adelante. A nivel de  
        //ejecución no afecta.  
  
        //Se introduce un retardo en la ejecución del código de 3 sg  
        try{  
            Thread.sleep(3000); //espera 3000ms  
        }catch(InterruptedException e){}  
  
        System.out.println("La potencia de 2 elevado a 8 es "+Math.pow(2, 8));  
    }  
}
```

NOTA: this no se puede utilizar en un método estático

UT4 Ejercicios básicos

Ejercicio 1.-

Define una clase Persona considerando los siguientes atributos de clase: nombre (String), apellidos (String), edad (int), casado (boolean), numeroDocumentoldentidad (String). Define un constructor y los métodos para poder establecer y obtener los valores de los atributos.

Ejercicio 2.-

Considera que estás desarrollando un programa Java donde necesitas trabajar con objetos de tipo DiscoMusical. Define una clase DiscoMusical considerando los siguientes atributos de clase: titulo (String), autor (String), añoEdicion (int), formato (String), digital (boolean). Define un constructor y los métodos para poder establecer y obtener los valores de los atributos.

Ejercicio 3.-

Define una clase Profesor considerando los siguientes **atributos de clase**: nombre (String), apellidos (String), edad (int), casado (boolean), especialista (boolean). Define **un constructor** que reciba los parámetros necesarios para la inicialización y otro constructor que no reciba parámetros. Crea los métodos para poder establecer y obtener los valores de los atributos.

Ejercicio 4.-

Define una clase Bombero considerando los siguientes **atributos de clase**: nombre (String), apellidos (String), edad (int), casado (boolean), especialista (boolean). Define **un constructor** que reciba los parámetros necesarios para la inicialización y los métodos para poder establecer y obtener los valores de los atributos.

Ejercicio 5.-

Se pide definir una clase Medico (que representa a un médico de un hospital) con los siguientes atributos de clase: nombre (String), apellidos (String), edad (int), casado (boolean), numeroDocumentoldentidad (String), especialidad (String). Definir **un constructor que permita asignar** valores de defecto a los atributos y los métodos para poder establecer y obtener los valores de los atributos. En cada método, incluye una instrucción para que se muestre por consola un mensaje informando del cambio. Por ejemplo si cambia la especialidad del médico, debe aparecer un mensaje que diga: "Ha cambiado la especialidad del médico de nombre La nueva especialidad es: ...".

Ejercicio 6.-

Dada la clase Medico del ejercicio anterior, añade **un método** de nombre "calculoParaMultiploEdad" que no recibe parámetros y que permita determinar cuántos años faltan para que la edad del médico sea múltiplo de 5 y mostrar un mensaje informativo por pantalla. Por ejemplo si el médico tiene 22 años deberá en primer lugar obtener el resto de la división de 22 entre 5, que es 2. Ahora obtendrá los años que faltan para que el médico tenga una edad múltiplo de 5, que serán $5 - 2 = 3$ años. A continuación deberá mostrar **un mensaje por consola** del tipo: "El médico de nombre ... con especialidad ... tendrá una edad múltiplo de 5 dentro de ... años".

Ejercicio 7.-

Diseñar una clase Rueda que permita representar una rueda de un vehículo. Sus **atributos de clase serán**: tipo (String), grosor (double), diametro (double), marca (String). Define un constructor asignando unos valores de defecto a los atributos y los

métodos para poder establecer y obtener los valores de los atributos. Crea un método denominado **comprobarDimensiones** donde a través de condicionales if realices las siguientes comprobaciones:

- a) Si el diámetro es superior a 1.4 debe mostrarse por consola el mensaje "La rueda es para un vehículo grande". Si es **menor o igual** a 1.4 pero mayor que 0.8 debe mostrarse por consola el mensaje "La rueda es para un vehículo mediano". Si no se cumplen ninguna de las condiciones anteriores debe mostrarse por pantalla el mensaje "La rueda es para un vehículo pequeño".
- b) Si el diámetro es superior a 1.4 con un grosor inferior a 0.4, ó si el diámetro es menor o igual a 1.4 pero mayor que 0.8, con un grosor inferior a 0.25, deberá mostrarse por consola el mensaje "El grosor para esta rueda es inferior al recomendado".

Ejercicio 8.-

Diseñar una clase Motor que representa el motor de una bomba para mover fluidos. Define la clase Motor considerando los siguientes atributos de clase: tipoBomba (int), tipoFluido (String), combustible (String). Define un constructor asignando unos valores de defecto a los atributos y los métodos para poder establecer y obtener los valores de los atributos.

Añade un método **tipo procedimiento** denominado **dimeTipoMotor()** donde a través de un condicional switch hagas lo siguiente:

- a) Si el tipo de bomba es 0, mostrar un mensaje por consola indicando "No hay establecido un valor definido para el tipo de motor".
- b) Si el tipo de bomba es 1, mostrar un mensaje por consola indicando "El motor es un motor de agua".
- c) Si el tipo de bomba es 2, mostrar un mensaje por consola indicando "El motor es un motor de gasolina".
- d) Si el tipo de bomba es 3, mostrar un mensaje por consola indicando "El motor es un motor de hormigón".
- e) Si el tipo de bomba es 4, mostrar un mensaje por consola indicando "El motor es un motor de pasta alimenticia".
- f) Si no se cumple ninguno de los valores anteriores mostrar el mensaje "No se puede clasificar el motor".

Ejercicio 9.-

Dada la clase del ejercicio anterior Motor, diseña un método tipo función que devuelva un booleano (true o false) denominado **dimeSiMotorEsParaAgua()** donde se cree una variable local booleana motorEsParaAgua de forma que si el tipo de motor tiene valor 1 tomará valor true y si no lo es tomará valor false. El método debe devolver la variable local booleana motorEsParaAgua.

UT4 Ejercicios de clases

1. Ejercicios de clases 1

1.- Indica la salida al siguiente programa.

```
class Ejercicio{  
    public static void main(String[] args) {  
        Clase1 obj1=new Clase1();  
        obj1.imprimir(24.3,5);  
    }  
}  
  
class Clase1{  
//objeto  
    private double valor=9.8;  
    private int x=7;  
  
    public void imprimir(double valor, int x){  
        System.out.print(valor + " " + this.x);  
        //valor se refiere al de Ejercicio  
        //this.x se refiere al de la Clase1  
    }  
}
```

Salida Solucion:

24.3 7

2.- Indicar la salida al siguiente programa.

```
class Ejercicio {  
    public static void main(String[] args) {  
        Clase1 obj1=new Clase1(5,4);  
        System.out.print(obj1.modificar(4)+" ");  
        Clase1 obj2=new Clase1(5,4);  
        System.out.print(obj2.modificar(5)+" ");  
        obj2=obj1;  
        System.out.print(obj2.modificar(5)+" ");  
    }  
}  
  
class Clase1{  
    int p1,p2;  
    public Clase1(int i, int j){  
        p1=i;  
        p2=j;  
    }  
  
    public int modificar(int i){  
        p1=p1+i;  
        p2=p2+i;  
        System.out.print(p2+" ");  
        return p1;  
    }  
}
```

//SERIA UTIL PARA DEPURAR

Salida solucion:

1er print

P1 = 5+4

P2 =4+4

Print 8

Print 9

2do print

Print 9

Print 10

3er print

8+5=13

9+5=14

3.- Crea una clase que represente a un círculo. Debe tener tres atributos, las coordenadas x e y de su centro y su radio.

Tendrá tres constructores, sin parámetros (el centro será 0,0 y el radio 1), pasándole sólo el radio (el centro será 0,0) y pasándole el radio y las coordenadas X e Y.

También tendrá tres métodos uno que calcule el área, otro que calcule la longitud y otro que escriba los resultados.

Después, haz una clase principal en la que se creen 3 objetos círculos y se prueben los métodos.

4.-. Crea una clase Punto que modele un punto en un espacio bidimensional. Tendrá dos atributos, x e y, que guardan las coordenadas. Habrá un constructor sin parámetros que crea un punto en (0, 0) y otro al que se le pueden pasar las coordenadas del punto. También habrá métodos para obtener las coordenadas y para imprimir el punto con el formato (x,y).

5.- Realiza un programa en Java con la creación de una clase llamada Coche con los atributos color, marca, matricula, número de puertas. Crea un constructor que inicialice el objeto con estos valores y otro que funcione como un constructor por defecto. Escribe métodos para devolver cada uno de los valores de los atributos (dameMatricula, dameColor,...) y un método que simule la operación de pintar el coche cambiando su color. Crea un método main que implemente la solución

6.- Realiza un programa en Java que permita crear cuentas bancarias pidiendo la cantidad inicial al usuario, así como realizar operaciones de ingresar y sacar dinero de esas cuentas. Si la cantidad de dinero a sacar es superior a la que hay en la cuenta se mostrará un mensaje advirtiendo que no se puede realizar la operación. El programa irá contando el número de cuentas creadas y lo mostrará al final. El programa dispondrá de un método que imprima la cantidad de dinero que queda en la cuenta. Crea un método main que implemente la solución.

2. Ejercicios de clases 2

1.- Crea una clase llamada Libro que guarde la información de cada uno de los libros de una biblioteca. La clase debe guardar el título del libro, autor, número de ejemplares del libro y número de ejemplares prestados. La clase contendrá los siguientes métodos:

Constructor por defecto.

Constructor con parámetros.

Métodos Setters/getters

Método préstamo que incremente el atributo correspondiente cada vez que se realice un préstamo del libro. No se podrán prestar libros de los que no queden ejemplares disponibles para prestar. Devuelve true si se ha podido realizar la operación y false en caso contrario.

Método devolución que decremente el atributo correspondiente cuando se produzca la devolución de un libro. No se podrán devolver libros que no se hayan prestado. Devuelve true si se ha podido realizar la operación y false en caso contrario.

Método `toString` para mostrar los datos de los libros. Este método se heredada de `Object` y lo debemos modificar (`override`) para adaptarlo a la clase `Libro`.

Escribe un programa para probar el funcionamiento de la clase `Libro`.

2.- Creamos una clase llamada empleado con los atributos nombre, apellido, edad, salario de forma que no se puedan acceder de ninguna clase externa. Crea un constructor vacío y otro que reciba cuatro parámetros con los valores de sus cuatro atributos. Crea un método para cambiar cada uno de los atributos y otro para devolver cada uno de los atributos, todos deben poder ser usados desde cualquier clase externa.

Crea un método que permita que a un empleado se le sume un plus a su salario siempre y cuando su edad sea de 40 años o más.

Para probar la clase `Empleado` que has desarrollado crea una clase que cree varios empleados (pide el número al usuario) y los referencie en un array que también crearás. Sube a los empleados que tengan derecho 50 euros a su sueldo, indicando en el programa a qué empleados se les ha podido realizar la subida y a cuáles no (imprime los datos de cada empleado, a continuación si se les ha subido o no el sueldo y el nuevo sueldo que tendrán).

Si para hacer lo que te piden necesitas a algún método más de los que se te indican, añádelo.

3. Ejercicios de clases 3

1.1.- Escribe el código en Java de una clase para representar un empleado de una empresa. Del empleado vamos a querer tener los datos de su nombre, apellido, edad y salario.

Realiza dos métodos constructores, uno con los valores de los cuatro atributos y otro sin atributos.

Crea también los métodos que devuelvan los valores de los atributos y otros que permitan modificarlos, así como un método que si el empleado tiene más de 40 años se le aumente el sueldo una cantidad que se pasa por parámetro. A este último método le llamaremos **comprobarPlus** y devuelve true si se aumenta el sueldo y falso si no se aumenta.

Se debe crear también un método **toString()** para imprimir un objeto Empleado.

Utiliza la clase **JOptionPane** para recibir los valores del usuario para ir creando los empleados. Para ello, crearás un método denominado **leerEmpleado()** en la misma clase donde tengamos el método **main()**, este método pide los datos de un empleado y devuelve un objeto de la clase Empleado.

Crea tres empleados para probar los métodos descritos.

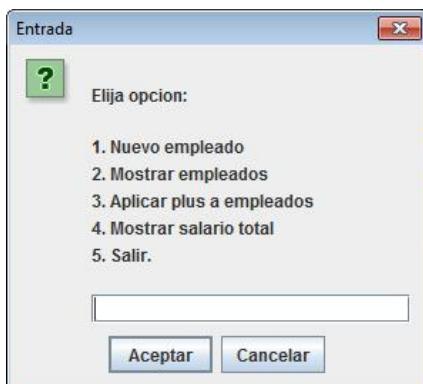
1.2.- Haz el ejercicio creando un array de empleados, donde se puedan acumular hasta 100 empleados (usa menos para probar el programa). Crea los empleados guardándolos en el array, para posteriormente comprobar si hay que aumentar el plus a los empleados.

Muestra los datos de cada empleado del array.

Muestra la suma del salario de todos los empleados antes y después de realizar el aumento que corresponda.

Haz el ejercicio con **JOptionPane**.

1.3.- Haz el ejercicio ahora con un menú como el siguiente:



Para ir creando los empleados usarás la **1ª opción** y un **atributo estático** que va contando los empleados según se van creando en el/los constructor/es.

En la **2^a opción** mostrarás todos los empleados creados hasta el momento en que se selecciona esa opción

En la **3^a opción**, se recorre el array y se sube la cantidad indicada como plus a los mayores de 40 años. Fíjate en que si se elige varias veces esta opción, habrá empleados a los que se les aplique la subida más de una vez. Busca una solución a este problema.

En la **4^a opción**, se muestra la suma de todos los empleados creados hasta el momento en que se selecciona la opción.

En esta versión del programa, deberás crear métodos estáticos en el fichero fuente del main, que reciban el array de empleados y lleven a cabo lo que se pide en cada opción. Es decir, **en el menú aparecerán solo las llamadas a los métodos**.

Si el usuario selecciona alguna opción y no hay empleados, el programa deberá notificarlo.

2.- Implementa una clase en Java que permita realizar promedios. La clase debe tener dos métodos, uno para ingresar un nuevo número, llamado *agregarNúmero(int numero)* y otro para obtener el promedio hasta el momento, llamado *obtenerPromedio()*. Determina qué atributos son necesarios para implementarla.

Implementa luego una clase de prueba que permita introducir algunos valores y que muestre el promedio.

3.- Implementa una clase en Java que represente una fracción de números enteros. Implementa asimismo los siguientes métodos en la clase Fracción:

- Suma de dos fracciones, que será una nueva fracción. Resta de dos fracciones, similar a la suma.
- División de dos fracciones.
- Multiplicación de dos fracciones.
- Calculo de la inversa de una fracción (cambiar numerador por denominador y viceversa).

Todos los métodos se realizan sobre un objeto Fracción pasándole por parámetro la segunda fracción cuando sea necesario.

Intenta simplificar los resultados usando un nuevo método.

4.- Realiza un programa en Java que, mediante un menú con varias opciones haga las siguientes tareas (hazlo con Scanner y con la clase JOptionPane):

Opción 1: Pide los datos de un alumno (nombre, apellido, curso, nota) y da de alta al alumno en nuestro programa.

Opción 2: Muestra los datos de los alumnos cuya nota media es mayor o igual a 5 y el número de ellos que hay que cumplan esa media.

Opción 3: Muestra los datos de los alumnos cuya nota media es menor a 5 y el número de ellos que hay que cumplan esa media.

Utiliza funciones (métodos estáticos) para llevar a cabo las tareas pedidas.

5.- Crea una clase en Java llamada Cubo con dos atributos:

```
int capacidad; // capacidad máxima en litros  
int contenido; // contenido actual en litros
```

La clase tendrá un constructor que recibe la capacidad inicial del cubo. Una vez establecida la capacidad, ya no será posible modificarla.

Habrá un método llamado **llena()**, que llenará el cubo hasta su capacidad máxima y otro **vacia()** que pone el contenido a 0 litros.

Haz un método llamado **vuelcaEn()** que vuelca el contenido de un cubo sobre otro. Antes de echar el agua se comprueba cuánto le cabe al cubo destino (será el que se manda por parámetro (void vuelcaEn(Cubo destino))).

Se pide también un método llamado **pinta()** que pinta un cubo en la pantalla. Se muestran los bordes del cubo con el carácter # y el agua que contiene con el carácter ~.

Cada litro se representa con una línea.

Para probar el ejercicio, ve haciendo las sentencias correspondientes a las siguientes salidas:

Cubo pequeño:

```
#  #  
#  #  
#####
```

Cubo grande:

```
#  #  
#  #  
#  #  
#  #  
#  #  
#  #  
#  #  
#####
```

Lleno el Cubo pequeño:

```
#~~~~~#
#~~~~~#
#####
```

El Cubo grande sigue vacío:

```
#    #
#    #
#    #
#    #
#    #
#    #
#    #
#####
```

Ahora vuelco lo que tiene el Cubo pequeño en el Cubo grande.

Cubo pequeño:

```
#    #
#    #
#    #
#    #
#####
```

Cubo grande:

```
#    #
#    #
#    #
#    #
#    #
#~~~~~#
#~~~~~#
#####
```

Ahora vuelco lo que tiene el Cubo grande en el Cubo pequeño.

Cubo pequeño:

```
#~~~~~#
#~~~~~#
#####
```

Cubo grande:

```
#    #
#    #
#    #
#    #
#    #
#    #
#    #
#####
```

4. Ejercicios de clases 4

1.- Desarrolla una clase Cafetera con atributos:

capacidadMaxima (la cantidad máxima de café que puede contener la cafetera)

cantidadActual (la cantidad actual de café que hay en la cafetera)

Implementa, al menos, los siguientes métodos:

- **Constructor** predeterminado: establece la capacidad máxima en 1000 (c.c.) y la actual en cero (cafetera vacía).
- **Constructor** con la capacidad máxima de la cafetera; inicializa la cantidad actual de café igual a la capacidad máxima.
- **Constructor** con la capacidad máxima y la cantidad actual. Si la cantidad actual es mayor que la capacidad máxima de la cafetera, la ajustará al máximo.
- **Accedentes y mutadores (getter y setter).**
- **llenarCafetera()**: hace que la cantidad actual sea igual a la capacidad máxima.
- **servirTaza(int)**: simula la acción de servir una taza con la capacidad indicada por parámetro. Si la cantidad actual de café en la cafetera “no llega” para llenar la taza, se sirve lo que quede.
- **vaciarCafetera()**: pone la cantidad de café actual en cero.
- **agregarCafetera(int)**: añade a la cafetera la cantidad de café indicada en el parámetro. Si esa cantidad excede el máximo, ajusta al máximo.

Escribe el código en el programa principal para probar lo siguiente:

Capacidad máxima = 1000 Cantidad actual = 0

Agregamos 20 c.c. de cafe...

Capacidad máxima = 1000 Cantidad actual = 20

Llenamos la cafetera...

Capacidad máxima = 1000 Cantidad actual = 1000

Servimos una taza de 500 c.c....

Todavia quedan 500 c.c.

Capacidad máxima = 1000 Cantidad actual = 500

Servimos una taza de 600 c.c....

Se sirve todo lo que quedaba (500 c.c.)

Capacidad máxima = 1000 Cantidad actual = 0

Servimos una taza de 200 c.c....

Lo siento, pero no queda nada de cafe.

Capacidad máxima = 1000 Cantidad actual = 0

Llenamos la cafetera con 700 c.c....

Capacidad máxima = 1000 Cantidad actual = 700

Llenamos la cafetera con 400 c.c....

Capacidad máxima = 1000 Cantidad actual = 1000

2.- Implementa una clase en Java que permita representar cuadrados. Cada objeto Cuadrado vendrá representado por sus cuatro vértices, que serán los atributos de la clase. Además de los métodos para modificar y devolver los valores de los vértices, se piden los siguientes métodos:

Cada cuadrado tendrá un método (dibujar) para representarse por medio de asteriscos. Por ejemplo, el cuadrado con vértices $(0, 0)$, $(0, 10)$, $(10, 10)$ y $(10, 0)$ se representaría:

The image shows a decorative border made of asterisk characters (*). This border is composed of two concentric rectangular patterns, creating a frame around the central content area. The border is approximately 10 pixels wide and is positioned at the top and bottom of the page.

También queremos un **método (dibRellenando)** que represente el cuadrado relleno de asteriscos.

A 10x10 grid of black asterisks (*). The grid is composed of 100 individual asterisks arranged in 10 rows and 10 columns.

Otro **método** (**dibujarVertices**) para escribir los 4 vértices del cuadrado de la forma siguiente para el ejemplo anterior:

$$(\theta, -1\theta) \quad (\bar{1}\theta, \bar{1}\theta)$$

$$(-10, -10)$$

$$(\theta, \theta)$$

$$(-10, 0)$$

Un **método (esCuadrado)** que devuelva true si se trata de los vértices de un cuadrado y false si no lo es.

La *distancia entre dos puntos* P1: (x1, y1) y P2: (x2, y2) viene dada por la fórmula:

$$D = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Cada cuadrado tendrá un **método (lado)** que devuelve el valor del lado del cuadrado creado.

Escribe un **método (compara)** en la clase Cuadrado para poder comparar el área de dos cuadrados. Este método recibe un cuadrado y lo compara con el cuadrado que invoca el método, actuando igual que el método compare de la clase String.

Implementa luego una clase de prueba que permita crear objetos de la clase Cuadrado y probar todos sus métodos.

Otro ejemplo:

```
* * * * *
*      *
*      *
*      *
* * * * *
```

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

(1, 5) (6, 5)

(1, 0) (6, 0)

Cuadrado definido por los vértices: (1,0) (1,5) (6,5) (6,0)

Observa que para dibujar los cuadrados vamos a ignorar la distancia al eje de las x y de las y, los pintamos siempre desde el punto P1(x1, y1), teniendo únicamente en cuenta la longitud del lado.

3.- Vamos a escribir un programa para representar el consumo de energía de una instalación eléctrica. Para ello, se hará una clase que representa los aparatos conectados en la instalación.

Cada aparato tiene un consumo eléctrico determinado. Al encender un aparato eléctrico, el consumo de energía se incrementa en la potencia de dicho aparato. Al apagarlo, se disminuye el consumo en dicha potencia.

Inicialmente, los aparatos están todos apagados.

Además, se desea consultar el consumo total de la instalación.

Haz un programa que declare dos aparatos eléctricos, una bombilla de 150 watos y una plancha de 2000 watos. El programa deberá imprimir el consumo nada más crear los objetos. Después, se enciende la bombilla y la plancha, y el programa imprime el consumo. Luego se apaga la bombilla, y se vuelve a imprimir el consumo.

Ejemplo de salida:

Inicialmente el consumo eléctrico es 0.0

Encendemos la bombilla Potencia 150.0

Encendemos la plancha Potencia 2000.0

El consumo eléctrico es 2150.0

Apagamos la plancha Potencia 2000.0

El consumo eléctrico es 150.0

5. Enunciado clase 1 de diciembre - Fraccion

Implementa una clase en Java que represente una fracción de números enteros. Implementa asimismo los siguientes métodos en la clase Fracción:

- Suma de dos fracciones, que será una nueva fracción. Resta de dos fracciones, similar a la suma.
- División de dos fracciones.
- Multiplicación de dos fracciones.
- Calculo de la inversa de una fracción (cambiar numerador por denominador y viceversa).

Todos los métodos se realizan sobre un objeto Fracción pasándole por parámetro la segunda fracción cuando sea necesario.

Intenta simplificar los resultados usando un nuevo método.

http://www.objetos.unam.mx/matematicas/leccionesMatematicas/01/1_005/index.html

UT5 Arrays, Matrices y ArrayLists

1. Arrays

1.1 ¿Qué es un array?

Un array es una colección finita de datos del mismo tipo, que se almacenan en posiciones consecutivas de memoria y reciben un nombre común.

Ej: Se quieren guardar las notas de los 20 alumnos de una clase. Gráficamente el array se puede representar de la siguiente forma:

Array notas:

8.50	6.35	5.75	8.50	...	3.75	6.00	7.40
notas[0]	notas[1]	notas[2]	notas[3]	...	notas[17]	notas[18]	notas[19]

Para acceder a cada elemento del array se utiliza el nombre del array y un índice que indica la posición que ocupa el elemento dentro del array. El índice se escribe entre corchetes.

El primer elemento del array ocupa la posición 0, el segundo la posición 1, etc. En un array de N elementos el último ocupará la posición N-1.

En el ejemplo, notas[0] contiene la nota del primer alumno y notas[19] contiene la del último.

El atributo **length** de un array contiene el tamaño del array independientemente de que tenga el valor por defecto u otro valor.
(te devuelve el numero de elementos del array)

Los índices deben ser enteros no negativos.

1.2 Crear arrays unidimensionales

Para crear un array se deben realizar dos operaciones:

1. Declaración
2. Instanciación

1. Declaración de un array: En la declaración se crea la referencia al array. La referencia es el nombre del array en el programa. Se debe indicar el nombre del array y el tipo de datos que contendrá.

De forma general un array unidimensional se puede declarar de cualquiera de estas dos formas:

```
tipo [] nombreArray; o tipo nombreArray[];
```

tipo: indica el tipo de datos que contendrá. Un array puede contener elementos de tipo básico o referencias a objetos.

nombreArray: es la referencia al array.

Ej:

```
int [] ventas; //array de tipo int llamado ventas  
double [] temperaturas; //array de tipo double llamado temperaturas  
String [] nombres; //array de tipo String llamado nombres
```

2. Instanciar un array: Mediante la instanciación se reserva un bloque de memoria para almacenar todos los elementos del array. La dirección, donde comienza el bloque de memoria, donde se almacenará el array se asigna al nombre. De forma general:

```
nombreArray = new tipo[tamaño];
```

nombreArray: es el nombre creado en la declaración.

tipo: indica el tipo de datos que contiene.

tamaño: es el número de elementos del array. Debe ser una expresión entera positiva. El tamaño no se puede modificar durante la ejecución del programa.

new: operador para crear objetos. Mediante new se asigna la memoria necesaria para ubicar el objeto. Java implementa los arrays como objetos.

Ej: ventas = new int[5]; //se reserva memoria para 5 enteros

Lo normal es que la declaración y la instanciación se hagan en una sola instrucción:

```
tipo [] nombreArray = new tipo[tamaño];
```

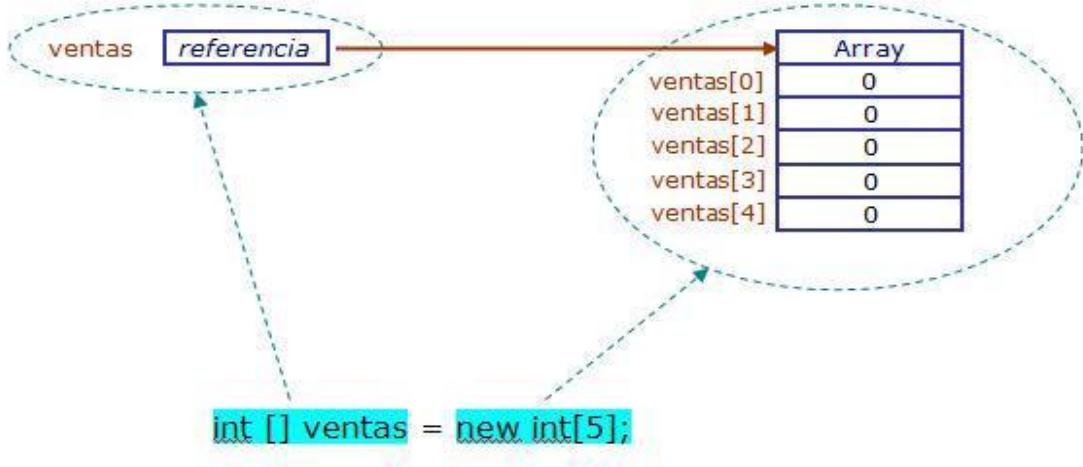
Ej: int [] ventas = new int[5];

El tamaño del array también se puede indicar durante la ejecución del programa, es decir, en tiempo de ejecución se puede pedir por teclado el tamaño del array y crearlo.

Ej:

```
.....  
Scanner teclado = new Scanner(System.in);  
System.out.print("Número de elementos del array: ");  
int numeroElementos = teclado.nextInt();  
int [] ventas = new int[numeroElementos];
```

Si no hay memoria suficiente para crear el array, new lanza una excepción `java.lang.OutOfMemoryError`.



Esquema importante: el int ventas va a tener una referencia a una dir de memoria; una vez que haga el new te reserva en memoria espacio para 5 elementos.

Diferencia entre la referencia y el contenido del array

Debe quedar clara la diferencia entre la referencia (manejador del array o nombre del array) y el contenido del array.

El nombre del array contiene la dirección de memoria del contenido del array.

(Por eso son contiguos, java para pasar al 2º elemento suma el tamaño del 1º.)

1.3 Inicializar arrays unidimensionales

Un array es un objeto, por lo tanto, cuando se crea, a sus elementos se les asigna automáticamente un valor inicial. Los valores iniciales por defecto para un array en java son:

0 para arrays numéricos
 '\u0000' (carácter nulo) para arrays de caracteres
 false para arrays booleanos
 null para arrays de String y de referencias a objetos.

También se pueden dar otros valores iniciales al array cuando se crea. Los valores iniciales se escriben entre llaves separados por comas y deben aparecer en el orden en que serán asignados a los elementos del array. El número de valores determina el tamaño del array.

Ej:

```
double [] notas = {6.7, 7.5, 5.3, 8.75, 3.6, 6.5};
boolean [] resultados = {true, false, true, false};
String [] dias = {"Lunes", "Martes", "Miércoles", "Jueves", "Viernes",
  "Sábado", "Domingo"};
```

1.4 Acceder a los elementos de un array

Para acceder a cada elemento del array se utiliza el nombre del array y el índice que indica la posición que ocupa el elemento dentro del array. El índice se escribe entre corchetes. Se puede utilizar como índice un valor entero, una variable de tipo entero o una expresión de tipo entero.

Un elemento de un array se puede utilizar igual que cualquier otra variable. Se puede hacer con ellos las mismas operaciones que se pueden hacer con el resto de variables (incremento, decremento, operaciones aritméticas, comparaciones, etc).

```
int m = 5;  
int [] a = new int[5];
```

0	0	0	0	0
a[0]	a[1]	a[2]	a[3]	a[4]

```
a[1] = 2;
```

0	2	0	0	0
a[0]	a[1]	a[2]	a[3]	a[4]

```
a[2] = a[1];
```

0	2	2	0	0
a[0]	a[1]	a[2]	a[3]	a[4]

```
a[0] = a[1] + a[2] + 2;
```

6	2	2	0	0
a[0]	a[1]	a[2]	a[3]	a[4]

```
a[0]++;
```

7	2	2	0	0
a[0]	a[1]	a[2]	a[3]	a[4]

```
int m = 5;  
a[3] = m + 10;
```

7	2	2	15	0
a[0]	a[1]	a[2]	a[3]	a[4]

Si se intenta acceder a un elemento que está fuera de los límites del array (índice negativo o con un índice mayor que el último elemento del array) el compilador no avisa del error. El error se producirá durante la ejecución. En ese caso se lanza una excepción `java.lang.ArrayIndexOutOfBoundsException`.

Se puede saber el número de elementos del array mediante el atributo **length**. Se puede utilizar `length` para comprobar el rango del array y evitar errores de acceso. (`length - 1`)

Ej: Para asignar un valor a un elemento del array que se leen por teclado:

```
Scanner teclado = new Scanner(System.in);  
int i, valor;  
int [] a = new int[10];  
System.out.print("Posición: ");  
i = teclado.nextInt();  
System.out.print("Valor: ");  
valor = teclado.nextInt();  
if (i>=0 && i < a.length)  
a[i] = valor;
```

1.5 Recorrer un array unidimensional

Para recorrer un array se utiliza una instrucción iterativa (normalmente una instrucción for, aunque también puede hacerse con while o do while) usando una variable entera como índice que tomará valores desde el primer elemento al último o desde el último al primero.

Ej:

```
double[] notas = {2.3, 8.5, 3.2, 9.5, 4, 5.5, 7.0};  
for (int i = 0; i < 7; i++)  
    System.out.print(notas[i] + " ");
```

Ej: Programa que lee por teclado la nota de los alumnos de una clase y calcula la nota media del grupo. También muestra los alumnos con notas superiores a la media. El número de alumnos se lee por teclado.

```
import java.util.*;  
public class Ejemplo {  
    public static void main(String[] args) {  
        Scanner teclado= new Scanner(System.in);  
        int numAlum, i;  
        double suma = 0, media;  
  
        do {  
            System.out.print("Número de alumnos de la clase: ");  
            numAlum = teclado.nextInt();  
        } while (numAlum <= 0);  
  
        double[] notas = new double[numAlum];  
  
        for (i = 0; i < notas.length; i++) {  
            System.out.print("Alumno " + (i + 1) + " Nota final: ");  
            notas[i] = teclado.nextDouble();  
        }  
        teclado.close();  
  
        for (i = 0; i < notas.length; i++)  
            suma = suma + notas[i];  
  
        media = suma / notas.length;  
        System.out.printf("Nota media del curso: %2f %n", media);  
        System.out.println("Listado de notas superiores a la media: ");  
        for (i = 0; i < notas.length; i++)  
            if (notas[i] > media)  
                System.out.println("Alumno numero " + (i + 1)+ " Nota  
final: " + notas[i]);  
    }  
}
```

1.6 Arrays de caracteres en Java

Un array de caracteres en Java se crea de forma similar a un array de cualquier otro tipo de datos.

Ej: Array de 8 caracteres llamado cadena. Por defecto se inicializa con el carácter nulo.

```
char [] cadena = new char[8];
```

\u0000	\u0000	\u0000	\u0000	\u0000	\u0000	\u0000	\u0000
cadena[0]	cadena [1]	cadena [2]	cadena [3]	cadena [4]	cadena [5]	cadena [6]	cadena [7]

Ej: Array de 5 caracteres llamado vocales.

Se asignan valores iniciales: a, e, i, o, u

```
char [] vocales = {'a', 'e', 'i', 'o', 'u'};
```

a	e	i	o	u
vocales[0]	vocales [1]	vocales [2]	vocales [3]	vocales [4]

A diferencia de los demás arrays, se puede mostrar el contenido completo de un array de caracteres mediante una sola instrucción print o printf.

Ej:

```
System.out.println(cadena); //Muestra 8 caracteres nulos (en blanco)  
System.out.println(vocales); //Muestra aeiou
```

El atributo length de un array de caracteres contiene el tamaño del array independientemente de que sean caracteres nulos u otros caracteres.

Ej: char [] cadena = new char[8];

\u0000	\u0000	\u0000	\u0000	\u0000	\u0000	\u0000	\u0000
cadena[0]	cadena [1]	cadena [2]	cadena [3]	cadena [4]	cadena [5]	cadena [6]	cadena [7]

```
System.out.println(cadena.length); // Muestra: 8  
cadena[0] ='m';  
cadena[1] ='n';
```

m	n	\u0000	\u0000	\u0000	\u0000	\u0000	\u0000
cadena[0]	cadena [1]	cadena [2]	cadena [3]	cadena [4]	cadena [5]	cadena [6]	cadena [7]

```
System.out.print(cadena);  
System.out.print(cadena);  
System.out.println(" .");
```

Muestra: mnbbbbbbmnbbbbbb. //b representa los espacios en blanco

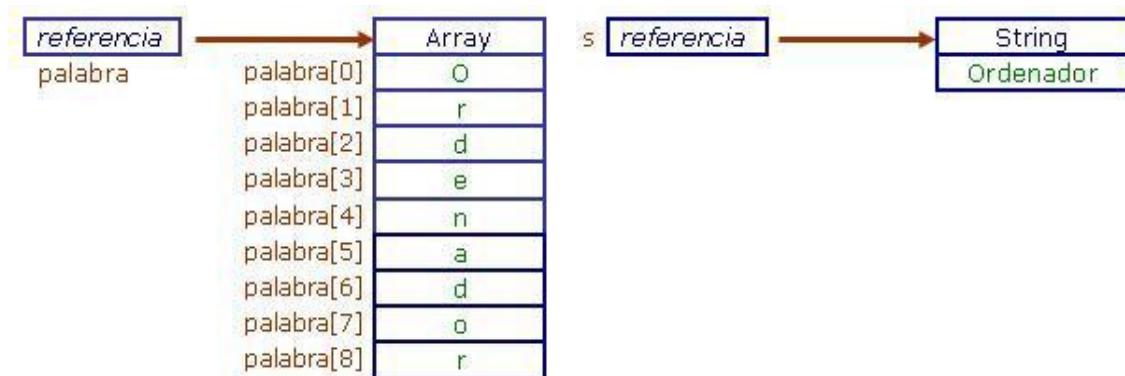
Se puede asignar un String a un array de caracteres mediante el método `toCharArray()` de la clase String.

Ej: `String s = "Ordenador";`



`char [] palabra = s.toCharArray();`

Se crea un nuevo array de caracteres con el contenido del String s y se asigna la dirección de memoria a palabra.

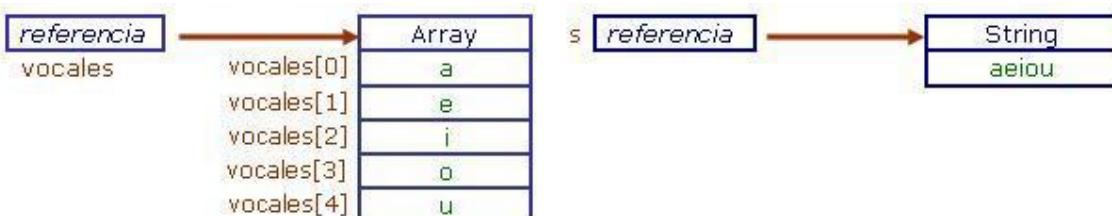


Se puede crear un String a partir de un array de caracteres.

Ej: `char [] vocales = {'a', 'e', 'i', 'o', 'u'};`
`String s = new String(vocales);`



Se crea un nuevo String con el contenido del array vocales y se asigna la dirección de memoria a s.



1.7 Recorrer un array de caracteres unidimensional

Se puede recorrer de forma completa utilizando una instrucción iterativa, normalmente un for.

Ej:

```
char [] s = new char[10];
s[0]='a';
s[1]='b';
s[2]='c';

//Muestra todos los caracteres del array,
//incluidos los nulos.
for(int i = 0; i<s.length; i++)
    System.out.print(s[i]+ " ");
```

Si los caracteres no nulos se encuentran al principio del array se puede recorrer utilizando un while, mientras que no encontremos un carácter nulo.

Ej:

```
char [] s = new char[10];
s[0]='a';
s[1]='b';
s[2]='c';
int i = 0;

// Muestra los caracteres del array hasta que
// encuentra el primer nulo.

while(s[i] != '\0'){
    System.out.print(s[i]); i++;
} // Ojo que si está lleno del todo da error
```

2. Arrays Bidimensionales (Matrices)

Un array puede tener más de una dimensión. El caso más general son los arrays bidimensionales, también llamados matrices o tablas. La dimensión de un array la determina el número de índices necesarios para acceder a sus elementos.

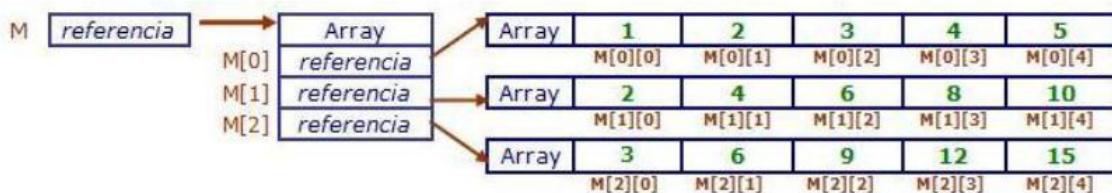
Una matriz necesita dos índices para acceder a sus elementos. Gráficamente se puede representar una matriz como una tabla de n filas y m columnas cuyos elementos son todos del mismo tipo.

La siguiente figura representa un array M de 3 filas y 5 columnas:

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Pero en realidad una matriz en Java es un array de arrays.

Gráficamente, podemos representar la disposición real en memoria del array anterior así:



La longitud del array M (`M.length`) es 3. Se corresponde con el número de filas del array bidimensional.

La longitud de cada fila del array (`M[i].length`) es 5. Se corresponde con el número de columnas de cada fila del array bidimensional.

Para acceder a cada elemento de la matriz se utilizan **dos índices**. El primero indica la fila y el segundo la columna.

2.1 Crear matrices en java

Se crean de forma similar a los arrays unidimensionales, añadiendo un índice.

Ej:

Matriz de datos de tipo int llamado ventas de 4 filas y 6 columnas.

```
int [][] ventas = new int [4] [6];
```

Matriz de datos double llamado temperaturas de 3 filas y 4 columnas.

```
double [][] temperaturas = new double [3] [4];
```

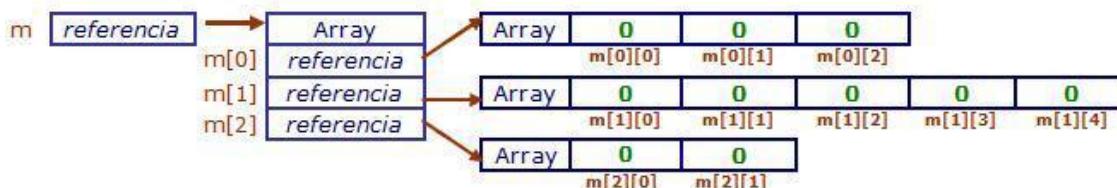
En Java se pueden crear arrays irregulares en los que el número de elementos de cada fila es variable. Solo es obligatorio indicar el número de filas.

Ej: int [][] m = new int[3][]; //crea una matriz m de 3 filas.

A cada fila se le puede asignar un número distinto de columnas:

```
m[0] = new int[3];  
m[1] = new int[5];  
m[2] = new int[2];
```

Gráficamente podemos representar la disposición real en memoria del array anterior así:



2.2 Inicializar matrices

Una matriz es un objeto, por tanto, cuando se crea, a sus elementos se les da automáticamente un valor inicial, al igual que a los array unidimensionales.

- 0 para arrays numéricos
- '\u0000' (carácter nulo) para arrays de caracteres
- false para arrays booleanos
- null para arrays de String y de referencias a objetos.

También es posible dar otros valores iniciales a la matriz cuando se crea. Los valores iniciales se escriben entre llaves separados por comas. Los valores que se le asignen a cada fila aparecerán a su vez entre llaves separados por comas. El número de valores determina el tamaño de la matriz.

Ej: int [][] numeros = {{6,7,5}, {3, 8, 4}, {1,0,2}, {9,5,2}};

Se crea la matriz numeros de tipo int, de 4 filas y 3 columnas, y se le da esos valores iniciales.

2.3 Recorrer matrices

Para recorrer una matriz se anidan dos bucles for. En general, para recorrer un array multidimensional se anidan tantas instrucciones for como dimensiones tenga el array.

Se usa siempre length para obtener el número de columnas que tiene cada fila

```
for (i = 0; i < a.length; i++) { //número de filas  
    for (j = 0; j < a[i].length; j++) //número de columnas de cada fila  
        System.out.print(a[i][j] + " ");  
    System.out.println();  
}
```

Ej: Programa que lee por teclado números enteros y los guarda en una matriz de 5 filas y 4 columnas. A continuación muestra los valores leídos, el mayor y el menor y las posiciones que ocupan.

3. ArrayList

La clase ArrayList permite el almacenamiento de datos en memoria de forma similar a los arrays convencionales, pero con la gran ventaja de que el número de elementos que puede almacenar es dinámico. La cantidad de elementos que puede almacenar un array está limitada a la dimensión que se declara a la hora de crearlo o inicializarlo. Los ArrayList, en cambio, pueden almacenar un número variable de elementos sin estar limitados por un número prefijado.

3.1 Declaración de un objeto ArrayList

La declaración genérica de un ArrayList se hace con un formato similar al siguiente:

```
ArrayList nombreDeLista;
```

De esta manera no se indica el tipo de datos que va a contener, cosa que suele ser recomendable, para que así se empleen las operaciones y métodos adecuados para el tipo de datos manejado.

Para especificar el tipo de datos que va a contener la lista se debe indicar entre los caracteres '<' y '>' la clase de los objetos que se almacenarán:

```
ArrayList<nombreClase> nombreDeLista;
```

En caso de almacenar datos de un tipo básico como char, int, double, etc, se debe especificar el nombre de la clase asociada: Character, Integer, Double, etc.

Ej: `ArrayList<String> listaPaises;`
 `ArrayList<Integer> edades;`

3.2 Creación de un ArrayList

Para crear un ArrayList se puede seguir el siguiente formato:

```
nombreDeLista = new ArrayList();
```

Se puede declarar la lista a la vez que se crea:

```
ArrayList<nombreClase> nombreDeLista = new ArrayList<nombreClase>();
```

Ej: `ArrayList<String> listaPaises = new ArrayList<String>();`

La clase ArrayList forma parte del paquete `java.util`, por lo que hay que incluir en la parte inicial del código la importación de ese paquete.

3.3 Añadir elementos al final de la lista

El método **add** posibilita añadir elementos. Los elementos que se van añadiendo, se colocan después del último elemento que hubiera en el ArrayList. El primer elemento que se añada se colocará en la posición 0.

Ej:

```
ArrayList<String> listaPaises = new ArrayList<String>();  
listaPaises.add("España"); //Ocupa la posición 0  
listaPaises.add("Francia"); //Ocupa la posición 1  
listaPaises.add("Portugal"); //Ocupa la posición 2
```

Ej: Se pueden crear ArrayList para guardar datos numéricos de igual manera

```
ArrayList<Integer> edades = new ArrayList<Integer>();  
edades.add(22);  
edades.add(31);  
edades.add(18);
```

3.4 Añadir elementos en cualquier posición de la lista

También es posible insertar un elemento en una determinada posición desplazando el elemento que se encontraba en esa posición, y todos los siguientes, una posición más. Para ello, se emplea también el método **add** indicando como primer parámetro el número de la posición donde se desea colocar el nuevo elemento.

Ej:

```
ArrayList<String> listaPaises = new ArrayList<String>();  
listaPaises.add("España");  
listaPaises.add("Francia");  
listaPaises.add("Portugal"); //El orden es: España, Francia, Portugal  
listaPaises.add(1, "Italia"); //Ahora es: Italia, España, Francia, Portugal
```

Si se intenta insertar en una posición que no existe, se produce una excepción (IndexOutOfBoundsException)

3.5 Suprimir elementos de la lista

Si se quiere eliminar un determinado elemento de la lista se puede emplear el o método **remove** al que se le puede indicar por parámetro un valor int con la posición a suprimir bien, se puede especificar directamente el elemento a eliminar si es encontrado en la lista.

Ej:

```
ArrayList<String> listaPaises = new ArrayList<String>();  
listaPaises.add("España");  
listaPaises.add("Francia");  
listaPaises.add("Portugal"); //El orden es: España, Francia, Portugal  
listaPaises.add(1, "Italia");  
//Ahora es: España, Italia, Francia, Portugal  
listaPaises.remove(2);  
//Eliminada Francia, queda: España, Italia, Portugal  
listaPaises.remove("Portugal");  
//Eliminada Portugal, queda: España, Italia
```

3.6 Consulta de un determinado elemento de la lista

El método **get** permite obtener el elemento almacenado en una determinada posición que es indicada con un parámetro de tipo **int**. Con el elemento obtenido se podrá realizar cualquiera de las operaciones posibles según el tipo de dato del elemento (asignar el elemento a una variable, incluirlo en una expresión, mostrarlo por pantalla, etc).

Ej: `System.out.println(listaPaises.get(3)); // Mostraría: Portugal`

3.7 Modificar un elemento contenido en la lista

Es posible modificar un elemento que previamente ha sido almacenando en la lista utilizando el **método set**. Como primer parámetro se indica, con un valor **int**, la posición que ocupa el elemento a modificar, y en el segundo parámetro se especifica el nuevo elemento que ocupará dicha posición sustituyendo al elemento anterior.

Ej: En la lista de países se desea modificar el que ocupa la posición 1 (segundo en la lista) por "Alemania".

```
listaPaises.set(1, "Alemania");
```

3.8 Buscar un elemento

La clase **ArrayList** facilita mucho las búsquedas de elementos gracias al método **indexOf** que retorna, con un valor **int**, la posición que ocupa el elemento que se indique por parámetro. Si el elemento se encontrara en más de una posición, este método retorna la posición del primero que se encuentre. El método **lastIndexOf** obtiene la posición del último encontrado.

En caso de que no se encuentre en la lista el elemento buscado, se obtiene el valor **-1**.

Ej: Comprobar si Francia está en la lista, y mostrar su posición.

```
String paisBuscado = "Francia";  
int pos = listaPaises.indexOf(paisBuscado);  
if(pos!=-1)  
    System.out.println(paisBuscado + " encontrado en la  
                      posición: "+pos);  
else  
    System.out.println(paisBuscado + " no se ha encontrado");
```

3.9 Recorrer el contenido de la lista

Es posible obtener cada uno de los elementos de la lista utilizando un bucle con tantas iteraciones como elementos contenga, de forma similar a la usada con los arrays convencionales. Para obtener el número de elementos de forma automática se puede emplear el método **size()** que devuelve un valor int con el número de elementos que contiene la lista.

Ej: `for (int i=0; i<listaPaises.size(); i++)
System.out.println(listaPaises.get(i));`

También se puede emplear otro formato del bucle for (**bucle foreach**) en el que se va asignando cada elemento de la lista a una variable declarada del mismo tipo que los elementos del ArrayList.

Ej: `for (String pais:listaPaises)
System.out.println(pais);`

Si el ArrayList contiene objetos de tipos distintos o se desconoce el tipo se pondría.

`for (Object o: nombreArrayList)`

También es posible hacerlo utilizando un objeto **Iterator**. La ventaja de usar un Iterator es que no se necesita indicar el tipo de objetos que contiene el ArrayList. Iterator tiene como métodos:

hasNext: devuelve true si hay más elementos en el array.

next: devuelve el siguiente objeto contenido en el array

remove: borra el elemento que se está recorriendo dentro del iterador

Ej: `ArrayList<Integer> nros = new ArrayList<Integer>();
//se insertan elementos
Iterator<Integer> it = nros.iterator();
//se crea el iterador it para el ArrayList nros
while(it.hasNext()) //mientras queden elementos
 System.out.println(it.next()); //se obtienen y se muestran`

3.10 Otros métodos

void clear(): Borra todo el contenido de la lista.

Object clone(): Retorna una copia de la lista.

boolean contains(Object elemento): Retorna true si se encuentra el elemento indicado en la lista, y false en caso contrario.

boolean isEmpty(): Retorna true si la lista está vacía.

3.11 Ejemplos de uso de ArrayList

Ej:

```
ArrayList<String> nombres = new ArrayList<String>();
nombres.add("Ana");
nombres.add("Luisa");
nombres.add("Felipe");
System.out.println(nombres); // [Ana, Luisa, Felipe]
nombres.add(1, "Pablo");
System.out.println(nombres); // [Ana, Pablo, Luisa, Felipe]
nombres.remove(0);
System.out.println(nombres); // [Pablo, Luisa, Felipe]
nombres.set(0,"Alfonso");
System.out.println(nombres); // [Alfonso, Luisa, Felipe]
String s = nombres.get(1);
String ultimo = nombres.get(nombres.size() - 1);
System.out.println(s + " " + ultimo); // Luisa Felipe
```

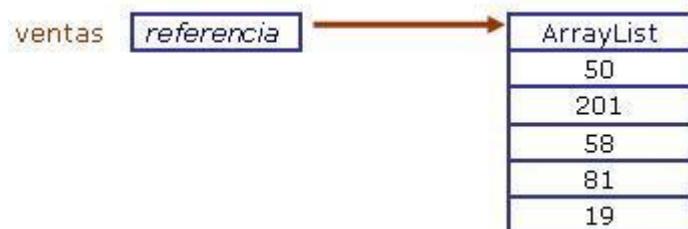
Ej: Escribe un programa que lea números enteros y los guarde en un ArrayList hasta que se lea un 0 y muestra los números leídos, su suma y su media.

```
import java.util.*;
public class ArrayList2 {
    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        ArrayList<Integer> numeros = new ArrayList<Integer>();
        int n;
        do {
            System.out.println("Introduce números enteros. 0 para acabar: ");
            System.out.println("Número: ");
            n = teclado.nextInt();
            if (n != 0)
                numeros.add(n);
        }while (n != 0);
        System.out.println("Ha introducido: " + numeros.size() +
" números:");
        System.out.println(numeros); //Muestra el arrayList completo
        Iterator<Integer> it = numeros.iterator();
        while(it.hasNext())
            System.out.println(it.next());
        double suma = 0;
        for(Integer i: numeros)
            suma = suma + i;
        System.out.println("Suma: " + suma);
        System.out.println("Media: " + suma/ numeros.size());
    }
}
```

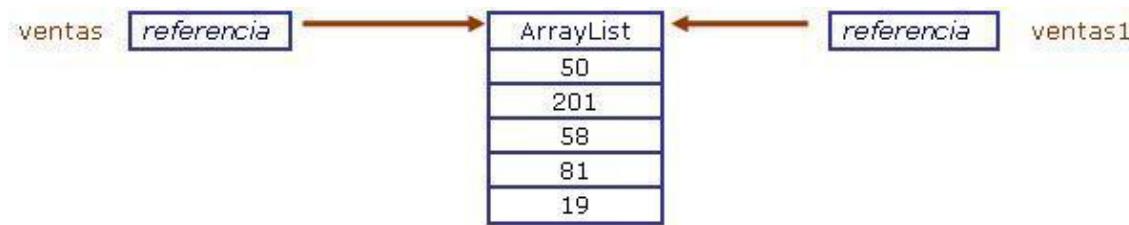
3.12 Copiar un ArrayList

El nombre de un ArrayList contiene la referencia al ArrayList, es decir, la dirección de memoria donde se encuentra el ArrayList, igual que sucede con los arrays estáticos.

Ej: Se tiene un ArrayList de enteros llamado ventas.



La instrucción `ArrayList<Integer> ventas1 = ventas;` no copia el array ventas en el nuevo array ventas1 sino que crea una referencia.

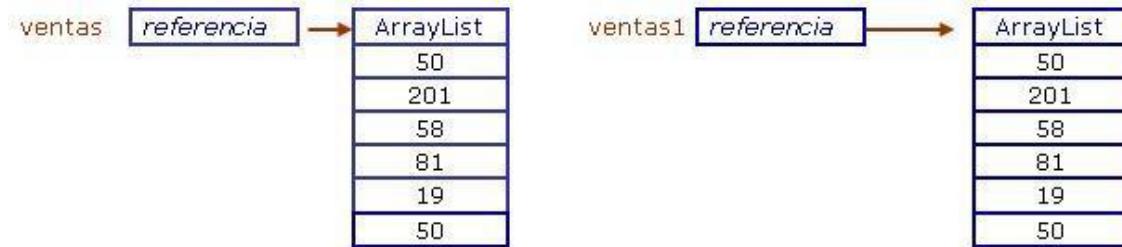


De esta forma se tiene dos formas de acceder al mismo ArrayList: mediante la referencia ventas y mediante la referencia ventas1.

Para hacer una copia se puede hacer de forma manual elemento a elemento o se puede pasar la referencia del ArrayList original al constructor del nuevo.

Shallow & Deep copy ver.

```
ArrayList<Integer> ventas1 = new ArrayList<Integer>(ventas);
```



3.13 ArrayList como parámetro de un método

Un ArrayList puede ser usado como parámetro de un método. Además un método también puede devolver un ArrayList mediante la sentencia return.

Ej: Método que recibe un ArrayList de String y lo modifica invirtiendo su contenido.

```
import java.util.*;  
  
public class ArrayList4 {  
    public static void main(String[] args) {  
        ArrayList<String> nombres = new ArrayList<String>();  
        nombres.add("Ana");  
        nombres.add("Luisa");  
        nombres.add("Felipe");  
        nombres.add("Pablo");  
        System.out.println(nombres);  
        nombres = invertir(nombres);  
        System.out.println(nombres);  
    }  
  
    public static ArrayList<String> invertir(ArrayList<String> nombres) {  
        ArrayList<String> resultado = new ArrayList<String>();  
        for (int i = nombres.size() - 1; i >= 0; i--)  
            resultado.add(nombres.get(i));  
        return resultado;  
    }  
}
```

Como ejercicio, hacer un nuevo método invertir, que modifique el ArrayList que se recibe en lugar de devolver una copia:

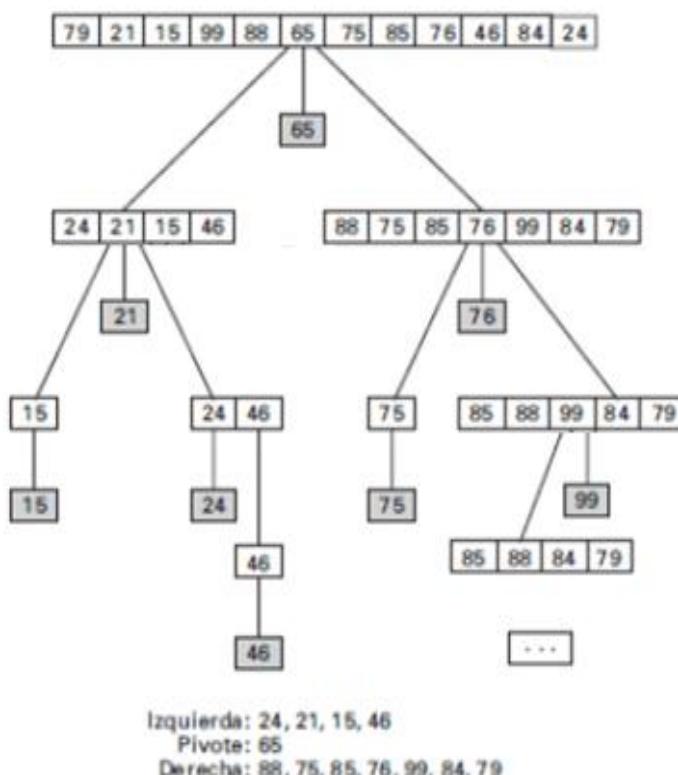
```
public static void invertir(ArrayList<String> nombres)
```

UT5 Algoritmos de Ordenación de Arrays

Intercambio: Consiste en comparar el primer valor con el resto de las posiciones posteriores, cambiando el valor de las posiciones en caso de que el segundo sea menor que el primero comparado. Después la segunda posición con el resto de posiciones posteriores y así sucesivamente hasta que el array se ordena. Se realizan muchas pasadas sobre el array, por lo que es lento y costoso.

Burbuja: Consiste en comparar el primero con el segundo, si el segundo es menor que el primero se intercambian los valores. Después el segundo con el tercero y así sucesivamente, cuando no haya ningún intercambio, el array estará ordenado. Lo peor de este método de ordenación, es que tiene una complejidad de **O(n^2)** haciendo que cuantos más valores a ordenar tengamos, mayor tiempo tardará en ordenar.

Quicksort: Consiste en ordenar un array mediante un pivote, que tomaremos como un punto intermedio en el array. Es como si se ordenaran pequeños trozos del array, haciendo que a la izquierda estén los menores a ese pivote y en la derecha los mayores a este. Después se vuelve a calcular el pivote de trozos de listas. Usa recursividad. Le pasamos el array, su posición inicial y su posición final como parámetro. Tiene una complejidad de **O($n \log_2 n$)**, haciendo que mejore el rendimiento aun teniendo muchos valores que ordenar.



Sort: Consiste en usar el método **sort** de **java.util.Arrays**.

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html>

Para ejecutarlo escribimos:

```
Arrays.sort(array a ordenar);
```

Simplemente insertamos como parámetro el array que queremos ordenar.

Por ejemplo, dado el siguiente array de Strings:

```
String [] nombres = {"juan", "pedro", "ana", "maria", "felipe", "luis",  
"eduardo"};
```

Con el método sort, se ordena ascendente con la instrucción:

```
Arrays.sort(nombres);
```

Para ordenar un array de forma descendente (de mayor a menor) hay que indicarlo utilizando el método **reverseOrder()** de la clase **Collections**.

Para utilizar reverseOrder es necesario incluir el import:

```
import java.util.Collections;
```

Por ejemplo, para ordenar el array nombres de forma descendente escribimos la instrucción Arrays.sort de la siguiente forma:

```
Arrays.sort(nombres, Collections.reverseOrder());
```

Con Arrays.sort podemos ordenar arrays de cualquier tipo de datos. Por ejemplo, para ordenar un array de enteros:

```
int [] numeros = {3, 5, 1, 2, 1, 7, 0, -1};  
Arrays.sort(numeros);  
// Mostrarlo ordenado  
for (int n : numeros) {  
    System.out.println(n);  
}
```

Collections.reverseOrder() solo funciona para arrays de objetos. Por este motivo si queremos ordenar de forma descendente arrays de tipos de datos simples debemos utilizar la **clase envolvente** equivalente al tipo de dato básico.

Por ejemplo, para ordenar un array de enteros de forma descendente hay que declararlo de tipo Integer en lugar de int.

```
Integer [] numeros = {3, 5, 1, 2, 1, 7, 0, -1};  
Arrays.sort(numeros, Collections.reverseOrder());  
for (int n : numeros) {  
    System.out.println(n);  
}
```

Algoritmos De Búsqueda Binaria En Arrays

Es un algoritmo de búsqueda que encuentra la posición de un valor en un array ordenado.

Compara el valor con el elemento que tengamos en medio del array, si no son iguales, la mitad en la cual el valor no puede estar es eliminada y la búsqueda continúa en la mitad restante hasta que el valor se encuentre.

Tienes un ejemplo en la implementación del algoritmo QuickSort de los ejemplos.

UT5 Ejercicios Arrays

Entrega 1

1.- Escribe el valor final de las variables:

a)

```
double[] A= new double[3];
double x;
for (int i=0; i<A.length; i++)
    A[i] = i*3;
x = A[0]+A[1]+A[2];
```

El valor de x es ____ 0+3+6

El valor de A[0] es ____ 0

El valor de A[1] es ____ 3

El valor de A[2] es ____ 6

b)

```
int[] B = new int[5];
```

```
B[4]=1;
B[B[4]]=2;
B[B[B[4]]]=0;
B[B[B[B[4]]]]=3;
B[B[B[B[B[4]]]]]=4;
```

El valor de B[0] es ____ 3

El valor de B[1] es ____ 2

El valor de B[2] es ____ 0

El valor de B[3] es ____ 4

El valor de B[4] es ____ 1

se va sustituyendo en las matrices

B→ 0 0 0 0 0
 0 1 2 3 4

c)

```
int[] C= new int[4];
int y;
for (int i=C.length-1; i>=0; i--)
    C[i] = i/2;
y = C[0]+C[1]+C[2]+C[3];
```

El valor de y es ____ 2

El valor de C[0] es ____ (0)/2

El valor de C[1] es ____ (1)/2

El valor de C[2] es ____ (2)/2

El valor de C[3] es ____ (3)/2

2.- Escribe el código en Java de los siguientes procedimientos:

- a) Un procedimiento que recibe un array de enteros y le asigna a cada componente los valores $0, 3, 6, \dots, 3*(n-1)$ donde n es el indice del array.
- b) Un procedimiento que recibe un array de enteros y le asigna a cada componente los valores $n-1, n-2, \dots, 2, 1, 0$ donde n es la longitud del array.
- c) Un procedimiento que recibe un array de valores booleanos y le asigna de manera intercalada los valores true y false.

3.- Escribe el código en Java de los siguientes ejercicios:

tienen que ser static

- a) El método public int cuentaCeros (int arr[]) que recibe un array de enteros y devuelve el número de ceros que se encuentran dentro del array.
- b) El método public int sumaPares (int arr[]) que recibe un array de enteros y devuelve el resultado de sumar sólo los números pares que hay dentro del array.
- c) El método public int cuentaRepeticiones (int arr[], int x) que recibe un array de enteros y un valor entero x. El método devuelve el número de veces que se repite el valor de x en el array.
- d) El método public void sustituye (int arr[], int viejo, int nuevo) que recibe un array de enteros y dos valores enteros viejo y nuevo. El método debe reemplazar todos los valores viejo del array por el valor de nuevo.
- e) El método public void intercambia (int arr[], int x, int y), que recibe un array de enteros y dos valores enteros que corresponden a dos componentes del array e intercambie los valores de las componentes del array. El ejercicio se hará teniendo en cuenta que x e y son el contenido de dos posiciones del array y después se hará teniendo en cuenta que x e y son las posiciones del array.
- f) El método public void invierte (int arr[]), que recibe un array de enteros e invierte la secuencia de valores del array.
- g) El procedimiento public void rotaDerecha (int arr[]), que recibe un array de enteros y mueve a cada elemento una posición adelante, colocando el último valor del array en la primera componente del array resultante.
- h) El método public boolean iguales (int a1[], int a2[]), que recibe dos arrays de enteros y devuelve true si los dos arrays contienen la misma secuencia de valores y false de otra manera.

4.- Programa Java que lea por teclado 10 números enteros y los guarde en un array. A continuación, calcula y muestra por separado la media de los valores positivos y la de los valores negativos.

5.- Programa Java que lea 10 números enteros por teclado y los guarde en un array.

Calcula y muestra la media de los números que estén en las posiciones pares del array. Considerar la primera posición del array (posición 0) como par.

6.- Haz un programa en Java para leer la altura de N personas y calcular la altura media. Calcular cuántas personas tienen una altura superior a la media y cuántas tienen una altura inferior a la media. El valor de N se pide por teclado y debe ser entero positivo.

7.- Haz un programa que cree un array de números y otro de String de un número de elementos que pondrás en una constante, que contendrán:

El primero, notas entre 0 y 10 (debemos controlar que inserte una nota válida), pudiendo tener decimales.

En el segundo, se insertarán los nombres de alumnos.

Después, se creará un tercer array de Strings, donde se insertará la nota en forma de cadena, de la siguiente forma:

Si la nota está entre 0 y 4,99 , será un suspenso

Si está entre 5 y 6,99 , será un bien.

Si está entre 7 y 8,99 será un notable.

Si está entre 9 y 10 será un sobresaliente.

Finalmente, mostrar por pantalla el nombre de cada alumno, su nota numérica y su nota alfabética usando los tres arrays.

Entrega 2

- 1.** Haz un programa que rellene un array con los 100 primeros números enteros y los muestre en pantalla en orden inverso al orden en que se han introducido (orden inverso).
- 2.** Haz un programa que cree un array de 10 posiciones de números aleatorios entre 1 y 100. Posteriormente se pedirá por teclado una posición y se mostrará en pantalla qué valor tiene esa posición.

`Math.random() → Devuelve un numero entre 0.0 y 1.0`

`Math.floor(Math.random() * 6) → Devuelve un numero entre 0 y 5`

- 3.** Haz un programa que cree un array cuyo tamaño se pedirá por teclado y cuyo contenido serán números aleatorios entre 1 y 300. Posteriormente se creará otro array que guardará aquellos números del primer array que acaben en un dígito que se indicará por teclado (se debe controlar que se introduce un número correcto). Finalmente, mostrar por pantalla los dos arrays.

- 4.** Haz un programa que calcule la letra de un DNI. Se pedirá el DNI por teclado y devolverá la letra correspondiente al DNI. Para buscar la letra, se calcula el resto de dividir el número de dni entre 23, y con el resultado que estará entre 0 y 22, se busca en el array de caracteres la letra correspondiente en esa posición.

El orden de los caracteres es: T, R, W, A, G, M, Y, F, P, D, X, B, N, J, Z, S, Q, V, H, L, C, K, E

- 5.** Haz un programa que pida un número por teclado y después diga si el número introducido es capicúa o no.

- 6.** Haz un programa que genere 100 números aleatorios entre 0 y 10 y cree un histograma con las frecuencias de cada número. Para representar las barras del histograma se utilizará secuencias '*'.

Por ejemplo, la secuencia: 1, 1, 3, 4, 1, 3, 1, 2, generaría la siguiente salida:

1: ****

2: *

3: **

4: *

UT5 Ejercicios Matrices (Arrays Bidimensionales)

1. Crear dos matrices de dimensión 4x4 de enteros (valores aleatorios de 0 a 9) y obtener una tercera matriz correspondiente a la suma de las dos.
2. Escribir un programa que cree una matriz entera de tamaño 4x5 (valores aleatorios de 0 a 9) y un valor entero (también aleatorio), y muestre en pantalla la posición de la primera coincidencia del valor en la matriz.
3. Dada una matriz de 8x8, hacer un programa que cree una matriz idéntica y la escriba en pantalla. (*con métodos ha dicho*)
4. Escribir un programa que cree una matriz entera de tamaño 6x8 (valores aleatorios de 1 a 100) y obtenga y escriba en pantalla el valor máximo y mínimo de toda la matriz.
5. Dada una matriz de dimensión 5x5:
 - Elevar al cuadrado los elementos situados por encima de la diagonal principal.
 - Sumar los elementos situados por debajo de la diagonal principal.
 - Sumar 1 a elementos de la diagonal principal.
 - Multiplicar por 2 los elementos de la diagonal inversa.
6. Escribir un programa que lea una matriz de 4x5 de números enteros (valores aleatorios de 0 a 9), calcule la suma de cada fila y de cada columna y muestre por pantalla la nueva tabla, incluyendo las sumas de las filas como una sexta columna y de las columnas como una quinta fila. No se crea una nueva matriz más grande, solo se imprime.
7. Hacer un programa que transponga una matriz de 4x4.
 - Primero haciendo el cambio sobre la propia matriz.
 - Segundo generando una nueva matriz.

$$\boxed{\begin{array}{ll} \mathbf{A} = & \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 4 & 3 \\ 3 & 4 & 2 & 1 \end{pmatrix} \\ (matriz\ original) & \\ \\ \mathbf{V} = & \begin{pmatrix} 1 & 1 & 3 \\ 2 & 2 & 4 \\ 3 & 4 & 2 \\ 4 & 3 & 1 \end{pmatrix} \\ (transpuesta) & \end{array}}$$

8. Se dispone de tres matrices unidimensionales. La primera de ellas contiene los nombres de pila de un grupo de N personas. La segunda matriz contiene los primeros apellidos y la tercera los segundos apellidos.

Crear una nueva matriz unidimensional que contenga para cada persona, en cada celda el nombre junto con los dos apellidos.

Crea otra matriz (bidimensional ahora) en la que en la primera fila tengamos los nombres, en la segunda fila los primeros apellidos y en la tercera los segundos apellidos.

9. En un campeonato de baloncesto intervienen 20 equipos de 10 miembros cada uno. Se quiere hacer un programa que lleve a cabo las siguientes operaciones:

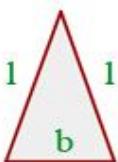
- a.** Leer y almacenar la altura de cada jugador en la estructura de datos que consideres más adecuada.
- b.** Determinar qué equipo tiene mayor altura media.
- c.** Formar una selección de 20 miembros integrada por el jugador más alto de cada equipo, guardándola en la estructura de datos que consideres más adecuada.

UT5 Ejercicios ArrayList

1.- Haz una clase para trabajar con triángulos isósceles (2 lados iguales). Dicha clase tendrá como atributos la base y la altura que serán privados. También tendrá como mínimo un constructor y dos métodos para calcular el área y el perímetro de un triángulo, y todos aquellos métodos que sean necesarios para el buen funcionamiento del programa.

Fórmula a aplicar:

Triángulo Isósceles
 $P = 2 \cdot l + b$



perímetro = $2 * \text{lado} + \text{base}$, siendo lado la medida del lado repetido

$$\text{lado}^2 = (\text{base}/2)^2 + \text{altura}^2$$

$$\text{lado} = \text{Raiz Cuadrada}((\text{base}/2)^2 + \text{altura}^2)$$

Crea una clase Principal que cree un Array/ArrayList de triángulos en el que se realicen las siguientes operaciones:

1. Añadir un triángulo.
2. Calcular e imprimir el triángulo con el área más grande.
3. Calcular e imprimir el triángulo con el perímetro más pequeño.
4. Imprimir la altura, la base, el área y el perímetro de un triángulo en concreto.
5. Imprimir la altura, la base, el área y el perímetro de todos los triángulos.

2.- Dada la clase Viaje siguiente, escribir un método denominado uneViaje que reciba como parámetros dos tipos Viaje y devuelva un nuevo objeto de esa misma clase con:

el origen del primero, el destino del segundo y como distancia la suma de las distancias de los dos viajes originales, si el destino del primero coincide con el origen del segundo, sino se cumple dicha condición se asignará -1 a la distancia.

```
class Viaje {  
    private String origen;  
    private String destino;  
    private double distancia;  
}
```

Nota: La función para comparar cadenas es: cadena1.compareTo(cadena2) devuelve 0 si las dos cadenas son iguales, <0 si la cadena1 < cadena2 y >0 si cadena1>cadena2.

Crea una clase Principal, en la que se defina un array/ArrayList de 10 objetos Viaje e imprima por pantalla el resultado de usar el método uneViaje, con una componente y la siguiente (0 y 1, 1 y 2, 2 y 3, ... ,9 y 10).

3.- Un punto en el plano se puede representar mediante un par de coordenadas x e y, ambas tomando valores en el conjunto de los números reales. En Java podemos representar un punto en el plano mediante una instancia de la siguiente clase:

```
public class Punto {  
    private double x;  
    private double y;  
  
    /* métodos */  
    /* A completar en el resto de apartados */  
}
```

En Java, el constructor de la clase sirve para inicializar los valores de los atributos a la hora de instanciarse un objeto de la misma en tiempo de ejecución. Programa un constructor para la clase Punto. Debe recibir como parámetros las dos coordenadas del punto.

El método `toString()` tiene un significado especial en los objetos Java. Es el método que se utiliza para obtener una representación como cadena de texto de dicho objeto.

Programa el método `toString()` para que devuelva una cadena de texto con la representación del punto con el siguiente formato: (x, y)

Donde x e y deben ser reemplazados por sus respectivos valores. El prototipo de dicha función se especifica a continuación:

```
public String toString() {  
    /* ... */  
}
```

Pista

Recuerda que el operador "+", aplicado a cadenas de texto, permite concatenarlas. Si se concatenan números a cadenas, Java los convierte automáticamente a cadenas de texto para realizar la concatenación.

Crea un paquete llamado *paqPunto* para guardar en él la clase punto y la clase Prueba que se te pide más adelante.

Programa una clase llamada **Prueba** que tenga un método main para probar el código anterior. Este método **debe recibir como argumentos** de línea de comandos las coordenadas x e y, crear un nuevo objeto Punto con dichas coordenadas e imprimir en su salida estándar (la pantalla en este caso) la representación textual de dicho objeto.

El programa debe comprobar que el número de argumentos de línea de comandos recibido sea el correcto.

Pista

El método `parseDouble` de la clase `Double` transforma una cadena de texto a un tipo primitivo `double`.

Es habitual que los atributos de un objeto se declaren como privados (`private`) para evitar accesos incontrolados desde otros puntos del código. Si es necesario que desde el exterior se acceda a estos atributos, se proporcionan métodos cuyo nombre empieza, por convenio, con las cadenas "get" (acceso en lectura) y "set" (acceso en escritura).

Programa en la clase `Punto` los siguientes métodos, que devuelven el valor de las coordenadas `x` e `y`. El prototipo de dichos métodos se muestra a continuación:

```
public double getX() {  
    /* ... */  
}  
public double getY() {  
    /* ... */  
}  
public void setX(double x) {  
    /* ... */  
}  
public void setY(double y) {  
    /* ... */  
}
```

Modifica el código de tu clase `Prueba` para comprobar que su comportamiento es correcto.

Cálculo de distancias.

En este apartado vamos a implementar dos métodos auxiliares que nos permitirán calcular distancias entre puntos.

Programa un método de la clase `Punto` que devuelva la distancia del punto al origen de coordenadas. El prototipo del método se muestra a continuación:

```
public double distanciaAlOrigen() {  
    /* completar */  
}
```

Modifica el código de tu clase `Prueba` para comprobar que su comportamiento es correcto.

Programa un método en `Punto` que devuelva la distancia entre el punto representado por la instancia actual del objeto y otra instancia de `Punto` que se recibe como parámetro. El prototipo del método se muestra a continuación:

```
public double calcularDistancia(Punto otroPunto) {  
    /* completar */  
}
```

Modifica el código de tu clase Prueba para comprobar que su comportamiento es correcto.

Cálculo de cuadrante.

Programa un método de la clase Punto que devuelva el cuadrante en el que se encuentra el punto.

Devuelve 0 si está en el origen de coordenadas o sobre alguno de los ejes.

Devuelve 1 si está en el primer cuadrante (x e y positivos).

Devuelve 2 si está en el segundo cuadrante (x negativo e y positivo).

Devuelve 3 si está en el tercer cuadrante (x e y negativos).

Devuelve 4 si está en el cuarto cuadrante (x positivo e y negativo).

El prototipo del método se muestra a continuación:

```
public int calcularCuadrante() {  
    /* completar */  
}
```

Modifica el código de tu clase de prueba para comprobar que su comportamiento es correcto.

Cálculo del punto más cercano.

En este apartado tienes que basarte en los métodos de los apartados anteriores.

Programa un método en Punto que reciba como parámetro un array de objetos de la clase Punto y devuelva una referencia al objeto de dicho array que esté más cercano al punto actual. El prototipo del método se muestra a continuación:

```
public Punto calcularMasCercano(Punto[] otrosPuntos) {  
    /* completar */  
}
```

Modifica el código de tu clase Prueba para comprobar que su comportamiento es correcto.

La clase Triangulo.

Un triángulo está plenamente definido por sus tres vértices. Estos vértices se pueden representar como objetos de la clase Punto.

Declaración de la clase.

En este apartado tienes que basarte en los métodos de los apartados anteriores.

Programa la clase Triangulo con sus atributos y un constructor que reciba como parámetro tres puntos del plano, así como sus geter, seter y toString. Guárdala en un paquete que llamarás paqTriangulo, en él guardarás también la clase **Prueba** donde probarás el método que se describe a continuación.

Longitud de los lados.

Programa el método calcularLongitudLados() de la clase Triangulo, que debe devolver un array de tres posiciones, cada una de las cuales debe ser la longitud de uno de los lados del triángulo. El prototipo del método se muestra a continuación:

```
public double[] calcularLongitudLados () {  
    /* completar... */  
}
```

Crea un ArrayList con algunos triángulos que pedirás al usuario para guardarlos en el programa e imprimelos. (no hace falta menú, solo es para repasar conceptos vistos hasta ahora)

EJERCICIO DE COMPOSICIÓN DE OBJETOS

4. Se pide hacer una aplicación en java que represente una finca mediante una clase.

De la finca se quiere guardar su nombre y el número de metros cuadrados que posee.

Además, se quiere guardar información sobre la casa y la parcela que la componen.

De la casa se quiere saber los metros cuadrados, el número de plantas y el número de habitaciones que posee.

De la parcela se quiere saber también el número de metros cuadrados que tiene y si se dedica a explotación agrícola y/o ganadera.

Una vez que se haya modelizado todo mediante clases, se añadirá un menú que permita:

- Crear una nueva finca y guardarla.
- Mostrar los datos de una finca dado el nombre de la misma, que se pedirá al usuario.
- Mostrar todas las fincas con todos sus datos.

El ejercicio lo harás usando un arrayList para almacenar cada una de las fincas.

5.- Realiza un programa que simule el funcionamiento de dos talleres de reparación de coches. Lo primero que hará el programa será crear un Array con dos talleres.

Cada taller tendrá nombre, CIF y dirección, que se pedirán al usuario. Cada vez que llega un coche nuevo se da de alta en uno de los dos talleres (se le pregunta al usuario a qué taller va el coche. Para cada coche se pide su matrícula y una descripción de la tarea a realizar en él, así como el NIF del dueño.

Se pide realizar un menú con las siguientes opciones:

1. Crear Array con dos talleres.
2. Introducir un taller.
3. Dar de alta coches.
4. Imprimir los talleres y sus coches.
5. Salir.

Elija opcion:

6.- Ejercicio para guardar información sobre cantantes famosos. Crearemos una clase denominada `ListaCantantes` que disponga de un atributo de tipo `ArrayList` para guardar cantantes. La clase debe tener un método que permita añadir objetos de tipo `CantanteFamoso` en la lista, otro método para eliminar un cantante y dos métodos para mostrar el disco más vendido de cada cantante y todos los discos de cada cantante (todo son opciones del menú que se muestra más abajo).

Cada objeto de tipo `CantanteFamoso` tendrá como atributos:

- nombre (String)
- discos (lista con los discos)
- y los métodos para obtener y establecer los atributos.

En el atributo `discos`, se guardará un listado de todos los discos editados por el cantante. Para cada uno de los discos se dispondrá de su título y el número de ejemplares vendidos.

Crea una clase principal que muestre un menú, con las siguientes opciones:

1. Introducir nuevo Cantante.
2. Borrar Cantante.
3. Mostrar disco más vendido de cada cantante.
4. Mostrar los cantantes y sus discos.
5. Salir.

Elija opción:

7.- Definir los atributos de las clases de objetos necesarias para almacenar la información relativa a unas muestras de alcoholemia recogidas en un análisis preventivo de la Dirección General de Tráfico, de forma que:

- a) Para cada conductor interesa tener su nombre, DNI. y una colección de referencias a las muestras que se le han tomado durante el período de estudio.
- b) Cada muestra tendrá los siguientes datos: Día y hora de la muestra, código del puesto de control preventivo (dos letras y dos números), matrícula del vehículo y la tasa de alcohol espirado en aire (entre 0 y 2.5 mg/l).

Hacer una simulación del funcionamiento de estas clases usando el siguiente menú:

1. Introducir un nuevo conductor.
2. Introducir una muestra.
3. Mostrar los datos de un conductor.
4. Mostrar las muestras de un conductor.
5. Mostrar los datos de todos los conductores.
6. Salir.

Elige una opción:

Hacerlo con Arrays y ArrayList.

Al introducir una muestra, se elige de qué conductor por su dni.
Los datos del conductor se muestran buscando por dni.

UT6 Paquetes en Java

1. Paquetes en Java

Haremos un fichero HolaMundo en java usando **paquetes**.

- **Qué son paquetes en java.**
- **La clase HolaMundo en un paquete.**
- **Estructura de directorios asociada.**
- **Compilar clases de un paquete.**
- **Ejecutar clases de un paquete.**
- **El Classpath cuando hay paquetes.**
- **Paquetes anidados.**

1.1 Que son paquetes en Java

Cuando hacemos programas más grandes, es normal que el número de clases vaya creciendo. Cada vez tenemos más y más clases. Meterlas todas en el mismo directorio no suele ser buena idea. Es mejor hacer **grupos de clases**, de forma que todas las clases que traten de un determinado tema o estén relacionadas entre sí vayan juntas. Por ejemplo, si hacemos un programa de una agenda de teléfonos que los guarde en una base de datos, podemos meter todas las clases que tratan con la base de datos en un **paquete** (grupo), todas las de ventanas en otro, las de imprimir en otro, etc.

Un **paquete** de clases es un grupo de clases que agrupamos juntas porque consideramos que están relacionadas entre sí o tratan de un tema común.

1.1.1 La clase hola mundo en un paquete

Java nos ayuda a organizar las clases en **paquetes**. En cada fichero .java que hagamos, al principio, podemos indicar a qué **paquete** pertenece la clase que hagamos en ese fichero.

Por ejemplo, si decidimos agrupar nuestros programas de pruebas, incluido el Hola Mundo, en un paquete "prueba", pondríamos esto en nuestro fichero HolaMundo.java

```
package prueba;
public class HolaMundo
{
    public static void main (String [ ] args)
    {
        System.out.println ("Hola mundo");
    }
}
```

El código es exactamente igual al de nuestro Hola Mundo original, pero hemos añadido la línea "**package prueba;**" al principio.

Hasta aquí todo correcto. Sin embargo, al hacer esto, para que todo funcione bien, java nos obliga a organizar los directorios, compilar y ejecutar de cierta forma.

1.1.2 Estructura de directorios asociada

Si hacemos que HolaMundo.java pertenezca al paquete prueba, java nos obliga a crear un subdirectorio "prueba" y meter el HolaMundo.java ahí.

Es decir, debemos tener esto así (unix y windows)

```
/<directorio_usuario>/mi_proyecto_HolaMundo/prueba/HolaMundo.java  
C:\<directorio_usuario>\mi_proyecto_HolaMundo\prueba\HolaMundo.java
```

Las mayúsculas y minúsculas son importantes. Tal cual lo pongamos en "package", debemos poner el nombre del subdirectorio.

1.1.3 Compilar una clase que está en un paquete

Para compilar la clase que está en el paquete debemos situarnos en el directorio padre del paquete y compilar desde ahí

```
$ cd /<directorio_usuario>/mi_proyecto_HolaMundo  
$ javac prueba/HolaMundo.java  
C:\> cd C:\<directorio_usuario>\mi_proyecto_HolaMundo  
C:\> javac prueba\HolaMundo.java
```

Si todo va bien, en el directorio "prueba" se nos creará un HolaMundo.class

1.1.4 Ejecutar una clase que está en un paquete

Una vez generado el HolaMundo.class en el directorio prueba, para ejecutarlo debemos estar situados en el directorio padre de "prueba". El nombre "completo" de la clase es "paquete.clase", es decir "prueba.HolaMundo". Por ello, para ejecutar, debemos hacerlo así

```
$ cd /<directorio_usuario>/mi_proyecto_HolaMundo  
$ java prueba.HolaMundo  
Hola Mundo
```

```
C:\> cd C:\<directorio_usuario>\mi_proyecto_HolaMundo
```

```
C:\> java prueba.HolaMundo  
Hola Mundo
```

Si todo va como debe, debería salir "Hola Mundo" en la pantalla

1.1.5 El classpath cuando las clases están en paquetes

Cuando las clases están en paquetes, la variable **CLASSPATH** debe ponerse de tal forma que encuentre el paquete, NO la clase. Es decir, en nuestro ejemplo:

```
$ CLASSPATH=/<directorio_usuario>/mi_proyecto_HolaMundo  
$ export CLASSPATH  
$ java prueba.HolaMundo  
Hola Mundo
```

```
C:\> set CLASSPATH="C:\<directorio_usuario>\mi_proyecto_HolaMundo"  
C:\> java prueba.HolaMundo  
Hola Mundo
```

O bien, si usamos la opción -cp en la línea de comandos

```
$ java -cp /<directorio_usuario>/mi_proyecto_HolaMundo  
prueba.HolaMundo  
Hola Mundo  
C:\> java -cp "C:\<directorio_usuario>\mi_proyecto_HolaMundo"  
prueba.HolaMundo  
Hola Mundo
```

1.1.6 Paquetes anidados

Para poder organizar mejor las cosas, java permite hacer **subpaquetes** de los paquetes y subpaquetes de los subpaquetes y así sucesivamente. Por ejemplo, si quiero partir mis clases de prueba en pruebas básicas y pruebas avanzadas, puedo poner más niveles de paquetes separando por puntos.

```
package prueba.basicas;  
package prueba.avanzadas;
```

A nivel de subdirectorios tendría que crear los subdirectorios "basicas" y "avanzadas" debajo de "prueba" y meter ahí las clases que correspondan. Para compilar, en el directorio del proyecto habría que compilar poniendo todo el path hasta llegar a la clase. Lo mismo para el **CLASSPATH**. Para ejecutar, el nombre de la clase va con todos los paquetes separados por puntos, es decir "prueba.basicas.HolaMundo" y "prueba.avanzadas.HolaMundoRemoto".

Es decir:

```
/<directorio_usuario>/mi_proyecto_HolaMundo/prueba/basicas/HolaMundo.java
$ cd /<directorio_usuario>/mi_proyecto_HolaMundo
$ javac prueba/basicas/HolaMundo.java
$ java -cp /<directorio_usuario>/mi_proyecto_HolaMundo
prueba.basicas.HolaMundo
Hola Mundo

C:<directorio_usuario>\mi_proyecto_HolaMundo\prueba\basicas\HolaMundo.java
C:> cd C:<directorio_usuario>\mi_proyecto_HolaMundo
C:> javac prueba\basicas\HolaMundo.java
C:> java -cp C:<directorio_usuario>\mi_proyecto_HolaMundo
prueba.basicas.HolaMundo
Hola Mundo
```

1.2 Qué son los ficheros jar

Cuando tenemos un programa grande, con varios paquetes y clases, ya sabemos cómo organizarlo, compilarlo y ejecutarlo. Sin embargo, si queremos usarlo en otra máquina o en otro proyecto, tenemos que llevarnos directorios enteros, con los ficheros que hay dentro y demás.

Lo ideal es meter todos estos ficheros y directorios en un único fichero comprimido. Java, con su comando jar, que está en el directorio bin de la carpeta donde tengamos java, nos permite hacer esto. Empaque todo lo que le digamos (directorios, clases, ficheros de imagen o lo que queramos) en un único fichero de extensión .jar. **Un fichero de extensión .jar es similar a los .zip de Winzip, a los .rar de Winrar o a los ficheros .tar del tar de unix. De hecho, con Winzip o Winrar se puede ver y desempaquetar el contenido de un fichero .jar.**

Java nos da además otra opción, podemos ejecutar las clases del fichero .jar sin tener que desempaquetarlo. Estará listo para ejecutar.

Vamos a ver cómo hacer todo esto:

- **Cómo crear un fichero jar.**
- **Ver qué hay dentro de un jar.**
- **Modificar el contenido de un fichero jar.**
- **Ejecutar un fichero jar.**
- **El fichero de manifiesto.**

1.2.1 Cómo crear un fichero jar

Para crear un fichero jar, en primer lugar tenemos que tener todo ya perfectamente preparado y compilado, funcionando.

Si las clases de nuestro programa **no pertenecen a paquetes**, simplemente debemos meter las clases en el fichero .jar Para ello, vamos al directorio donde estén los ficheros .class y ejecutamos el siguiente comando:

```
$ cd directorio_con_los_class  
$ jar cf fichero.jar fichero1.class fichero2.class  
fichero3.class ...
```

La opción "c" indica que queremos crear un fichero.jar nuevo. Si ya existía, se machacará, así que hay que tener cuidado. La opción "f" sirve para indicar el nombre del fichero, que va inmediatamente detrás. En nuestro caso, fichero.jar. Finalmente se pone una lista de ficheros .class (o de cualquier otro tipo) que queramos meter en nuestro jar. Se pueden usar comodines, estilo *.class para meter todos los .class de ese directorio.

Si las clases de nuestro programa pertenecen a paquetes, debemos meter en nuestro jar la estructura de directorios equivalente a los paquetes enteros. Para ello, nos vamos al directorio padre de donde empieza nuestra estructura de paquetes. En el caso de nuestro [HolaMundo con paquete](#), debemos meter el directorio **prueba** completo. El comando a ejecutar es este:

```
$ cd directorio_padre_de_prueba  
$ jar -cf fichero.jar prueba
```

Las opciones son las mismas, pero al final en vez de las clases, hemos puesto el nombre del directorio. Esto meterá dentro del jar el directorio y todo lo que hay debajo.

Otra opción sería meter los .class, pero indicando el camino relativo para llegar a ellos:

```
$ cd directorio_padre_de_prueba  
$ jar -cf fichero.jar prueba/HolaMundo.class
```

En windows la barra va al revés...

1.2.2 Ver qué hay dentro de un jar

Para comprobar si nuestro jar está bien hecho, podemos ver su contenido. El comando es este

```
$ jar tf fichero.jar
```

La opción "t" indica que queremos un listado del fichero.jar. La opción "f" es igual que antes. Esto nos dará un listado de los class (y demás ficheros) que hay dentro, indicando en qué directorio están. Deberíamos comprobar en ese listado que están todas las clases que necesitamos y la estructura de directorios concuerda con la de paquetes.

1.2.3 Modificar un jar

Para cambiar un fichero dentro de un jar o añadirle uno nuevo, la opción del comando jar es "u". Si el fichero existe dentro del jar, lo reemplaza. Si no existe, lo añade.

Por ejemplo, si hacemos un cambio en nuestro HolaMundo.class con paquete y lo recompilamos, podemos reemplazarlo así en el jar

```
$ jar uf fichero.jar prueba/HolaMundo.class
```

1.3 Cómo ejecutar un jar

Para ejecutar un jar, simplemente debemos poner el fichero jar en el CLASSPATH. Ojo, hay que poner el fichero.jar, **NO** el directorio en el que está el fichero.jar. Este suele ser un error habitual al empezar, pensar que basta con poner el directorio donde está el jar. Para los .class, basta poner el directorio, para los .jar hay que poner el fichero.jar

El path para indicar la ubicación del fichero puede ser absoluto o relativo al directorio en el que ejecutemos el comando java. El comando para ejecutar una clase dentro de un jar, en nuestro caso del HolaMundo con paquete, suponiendo que estamos en el directorio en el que está el fichero.jar, sería este:

```
$ java -cp ./fichero.jar prueba.HolaMundo
```

Hola Mundo

Simplemente, en la opción -cp del CLASSPATH hemos puesto el fichero.jar con su PATH relativo. Detrás hemos puesto el nombre de la clase, completo, con su paquete delante. Nuevamente, en Windows la barra de directorio va al revés.

1.4 El fichero de manifiesto

Ejecutar así tiene una pega. Además de acordarse de poner la opción -cp, hay que saber el nombre de la clase que contiene el método main(). Además, si nuestro programa es muy grande, tendremos varios jar, tanto nuestros como otros que nos bajemos de internet o de donde sea. La opción -cp también puede ser pesada de poner en ocasiones.

Una opción rápida que a todos se nos ocurre es crearse un pequeño fichero de script/comandos en el que se ponga esta orden. Puede ser un fichero .bat de Windows o un script de Unix. Este fichero debe acompañar al fichero.jar y suponiendo que estén en el mismo directorio, su contenido puede ser este

```
java -cp ./fichero.jar prueba.HolaMundo
```

Para ejecutarlo, se ejecuta como un fichero normal de comandos/script. Si el fichero se llama ejecuta.sh o ejecuta.bat, según sea Unix o Windows:

```
$ ./ejecuta.sh
```

Hola Mundo

```
C:\> ejecuta.bat
```

Hola Mundo

Sin embargo, java nos ofrece otra posibilidad de forma que no tengamos que hacer este fichero. Simplemente, en un fichero de texto metemos una línea en la que se ponga cuál es la clase principal. Este fichero se conoce como fichero de manifiesto y su contenido puede ser este:

```
Main-Class: prueba.HolaMundo
```

Cuando construimos el jar, debemos incluir este fichero de una forma especial. Por ejemplo, si el fichero lo llamamos manifiesto.txt y lo ponemos en el directorio donde vamos a construir el jar, el comando para hacerlo sería este:

```
$ jar cmf manifiesto.txt fichero.jar prueba/HolaMundo.class
```

En Windows, nuevamente, la barra al revés. Al comando de crear jar le hemos añadido la opción "m" para indicar que vamos a añadir un fichero de manifiesto. Hemos añadido además el fichero manifiesto.txt. El orden de las opciones "mf" es importante. El fichero de manifiesto y el fichero.jar se esperan en el mismo orden que pongamos las opciones. En el ejemplo, como hemos puesto primero la opción "m", debemos poner manifiesto.txt delante de fichero.jar. El resto de ficheros son los que queremos empaquetar.

Una vez construido, se ejecuta fácilmente. Basta con poner:

```
$ java -jar fichero.jar
```

La opción "-jar" indica que se va a ejecutar el fichero.jar que se ponga a continuación haciendo caso de su fichero de manifiesto. Como este fichero de

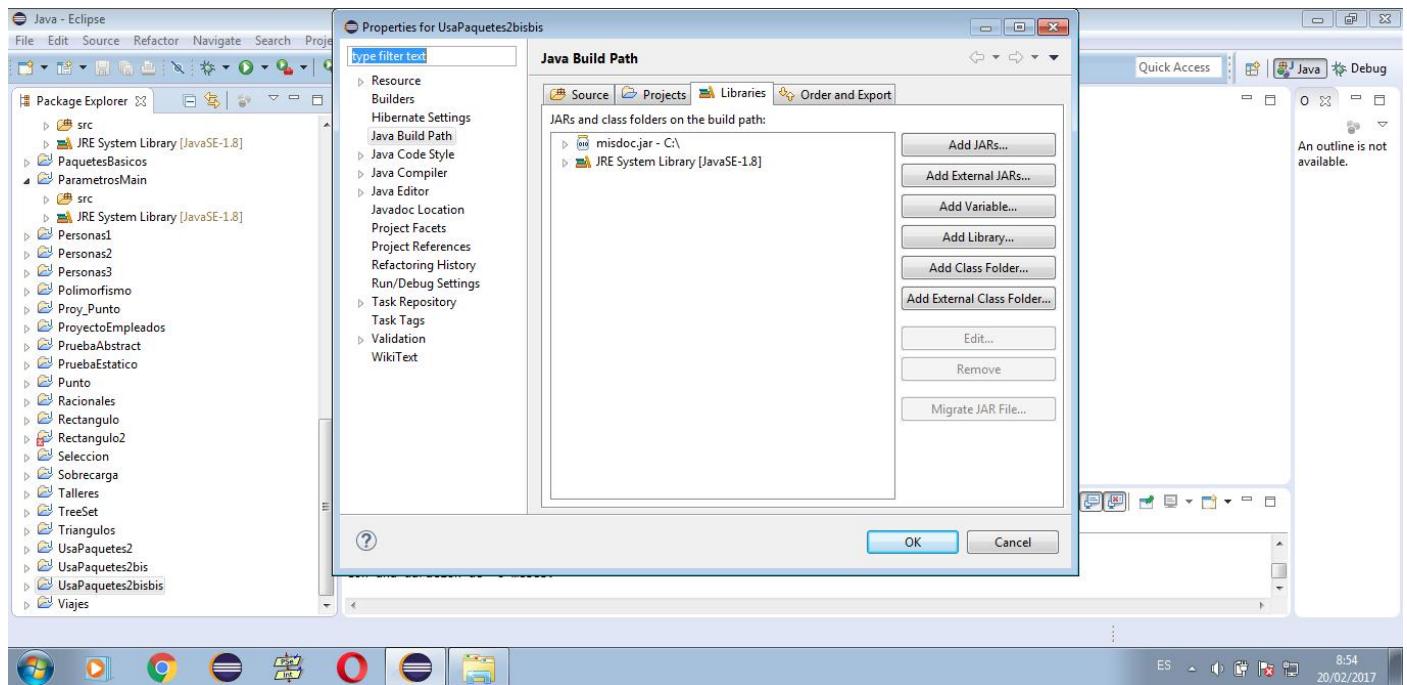
manifiesto dice que la clase principal es prueba.HolaMundo, será esta la que se ejecute.

De esta forma nos basta con entregar el jar y listo. El comando para arrancarlo es sencillo.

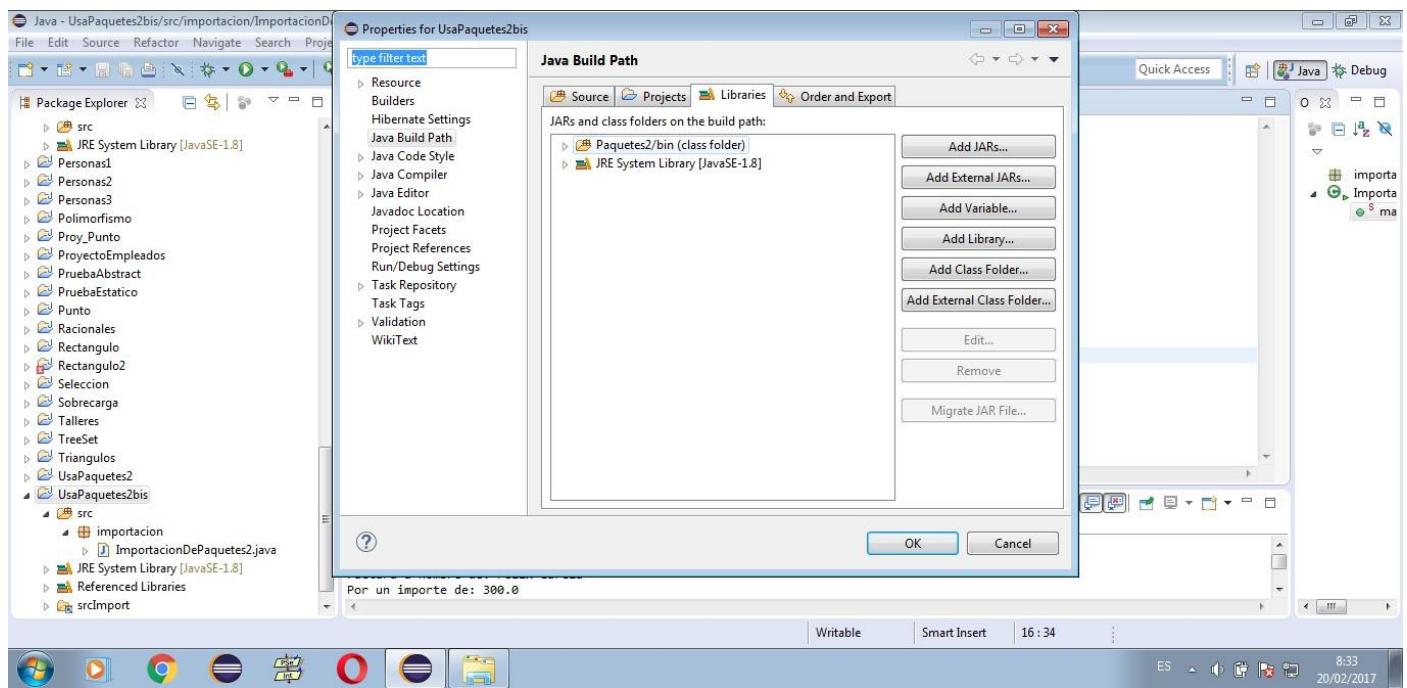
Es más, en Windows, si lo configuramos para que los ficheros jar se abran con java y la opción -jar, bastará con hacer doble click sobre ellos para que se ejecuten.

2. Uso de Paquetes desde Fuera

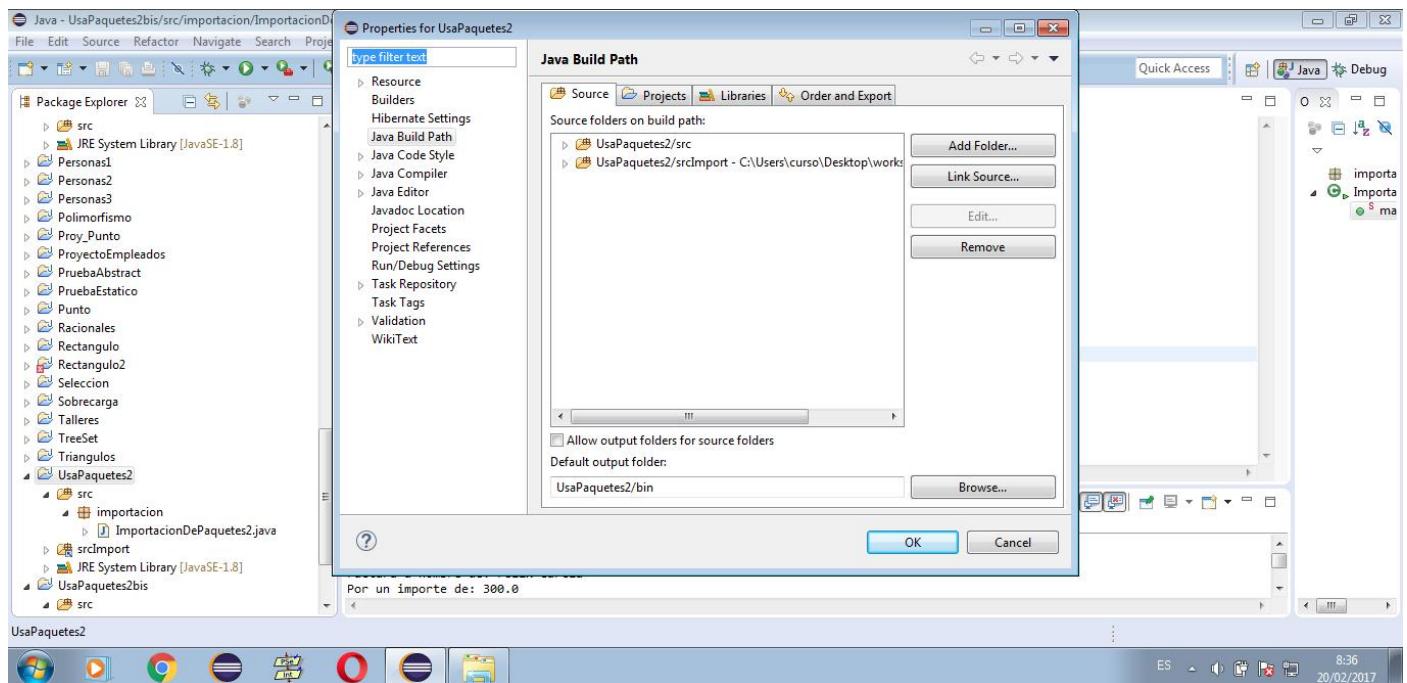
Con Jar Externo:



Incluir Class Folder:



Incluir Fuente de Paquetes:



UT6 Ejercicio para practicar paquetes

Ejercicio de aplicación de paquetes

Crea una aplicación en la que se pidan 2 números, estos se sumarán, restarán, multiplicarán y dividirán, mostrándonos el resultado por consola.

Para ello crea un proyecto llamado "**Operaciones**", en él crea 2 paquetes, uno llamado "**operaciones**" y el otro "**resultados**":

En el paquete "**operaciones**" escribe las clases:

- o **Sumar**
- o **Restar**
- o **Multiplicar**
- o **Dividir**

Cada una de ellas tendrá lo siguiente:

- Una sentencia "**package operaciones**" para que sea guardado en dicho "**package**".
- 2 atributos, uno llamado "**x**" y otro llamado "**y**" de tipo float.
- Un constructor con 2 parámetros para dar valor a los atributos.
- Un método (suma, resta, multiplica y divide, según proceda) en el cual se realiza la operación y muestra por pantalla el resultado.

Posteriormente tendremos un segundo **paquete** llamado "**resultados**" el cual tendrá las siguientes clases:

- o **IntroducirNumeros**
 - o **Calcular**
- La clase "**IntroducirNumeros**", realiza las siguientes funciones:
 - o La agregamos al package "**resultados**".
 - o En su **constructor** crea un objeto "**Scanner**" llamado teclado, y pide 2 números por consola que almacenamos en las variables "**i**" y "**j**" (que serán los atributos de esta clase).
 - La clase "**Calcular**" centrará todas las funciones que se realizan en el resto de clases (tendrá el método main()):
 - o La agregamos al package "**resultados**".
 - o Posteriormente importamos el package "**operaciones**" para poder utilizar las clases que hay en su interior.
 - o Instanciamos en el **main** la clase "**IntroducirNumeros**", para poder usar sus atributos.
 - o Finalmente creamos una instancia para cada clase del package "**operaciones**" y realizamos las operaciones matemáticas. Le pasamos a los constructores de cada una de ellas los valores que hemos introducido por teclado, a través de la clase "**introducirNumeros**" (serán los operadores de nuestras operaciones).

- o *No hacer la declaración de los atributos ni pública ni privada, si no se pone ningún modificador de acceso, la visibilidad será a nivel del paquete*

Tened en cuenta:

- Si no ponemos ningún modificador de acceso la visibilidad será a **nivel de paquete (hazlo así, sin getter ni setter)**, esto quiere decir que aunque importemos un package en otras clases, si alguna de estas clases o alguno de sus atributos o métodos tiene visibilidad de paquete, no podrán ser usados/ejecutados en otros paquetes, **SOLO** en las clases que estén en su mismo paquete.
- Para utilizar la clase Scanner del paquete **java.util** no lo importes, usa el nombre del paquete directamente.

```
Java.util.Scanner teclado = new Java.util.Scanner (System.in)
```

Haz como **segunda parte** del ejercicio la implementación de los paquetes en dos proyectos diferentes (llámalos Resultados y Operaciones), accediendo mediante acceso a fichero .jar a las clases de los paquetes que necesites.

Sobre el paquete operaciones -> click dcho -> Export Java: JAR file

He guardado el op.jar en esta ruta:

G:\DOCUMENTS\DA1D1E\Programación\Eclipse-workspace2\Operaciones\src\resultados

Creas una copia del proyecto

Borras el paquete operaciones de Operaciones2

En Operaciones2:

Project properties -> Java Build Path -> Libraries -> Add external JARs...

G:\DOCUMENTS\DA1D1E\Programación\Eclipse-workspace2\Operaciones\src\resultados

UT7 Orientación a Objetos Avanzado

0. Introducción

0.1 Conceptos

a) Encapsulamiento

Al empaquetamiento de las variables de un objeto con la protección de sus métodos se le llama **encapsulamiento**. Encapsulamiento es el ocultamiento del estado, es decir, de los datos miembro de un objeto, de manera que solo se pueda cambiar mediante las operaciones definidas para ese objeto. El encapsulamiento protege a los datos asociados de un objeto contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.

De esta forma, el usuario de la clase puede obviar la implementación de los métodos y propiedades para concentrarse solo en cómo usarlos. Por otro lado se evita que el usuario pueda cambiar su estado de maneras imprevistas e incontroladas.

El encapsulamiento de variables y métodos en un componente de software provee de dos principales **beneficios** a los desarrolladores de software:

- **Modularidad.** Esto es, el código fuente de un objeto puede ser escrito, así como darle mantenimiento, independientemente del código fuente de otros objetos.
La *modularidad* es la capacidad que tiene un sistema de ser estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo. Cada una de esas partes en que se encuentre dividido el sistema recibe el nombre de módulo.
- **Ocultamiento de la información.** Es decir, un objeto tiene una "interfaz pública" que otros objetos pueden utilizar para comunicarse con él. Pero el objeto puede mantener información y métodos privados que pueden ser cambiados en cualquier momento sin afectar a los demás objetos que dependan de ello.

Los **objetos** tienen la ventaja de la modularidad y el ocultamiento de la información. Las **clases** proveen el beneficio de la **reutilización**. Los programadores de software utilizan la misma clase, y por lo tanto el mismo código, una y otra vez para crear muchos objetos.

En las implantaciones orientadas a objetos se percibe un objeto como un paquete de datos y procedimientos que se pueden llevar a cabo con estos datos. Esto encapsula los datos y los procedimientos.

Los **atributos** se relacionan al objeto o instancia y los métodos a la clase. ¿Por qué se hace así? Los atributos son variables comunes en cada objeto de una clase y cada uno de ellos puede tener un valor para cada variable, diferente al que tienen para esa misma variable los demás objetos.

Los **métodos**, por su parte, pertenecen a la clase y no se almacenan en cada objeto, puesto que sería un desperdicio almacenar el mismo procedimiento varias veces y ello va contra el principio de reutilización de código.

b) Herencia

La **herencia** es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases, subclases y objetos.

La herencia está fuertemente ligada a la reutilización del código en la POO.

En Java, como en otros lenguajes de programación orientados a objetos, las clases pueden derivar desde otras clases. La clase derivada (la clase que proviene de otra clase) se llama **subclase**. La clase de la que está derivada se denomina **superclase**.

De hecho en Java, todas las clases deben derivar de alguna clase. Lo que nos lleva a la cuestión ¿Dónde empieza todo esto? La clase más alta, la clase de la que todas las demás descienden, es la clase **Object**, definida en `java.lang`. **Object** es la raíz de la herencia de todas las clases.

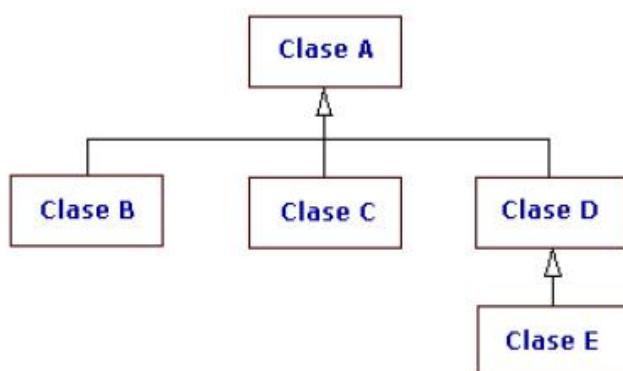
Hay dos **tipos de herencia**: Herencia Simple y Herencia Múltiple. La primera indica que se pueden definir nuevas clases solamente a partir de una clase inicial mientras que la segunda indica que se pueden definir nuevas clases a partir de dos o más clases iniciales. **Java solo permite herencia simple.**

c) Superclase y Subclases

El concepto de herencia en Java conduce a una estructura jerárquica de clases o estructura de árbol, lo cual significa que en la POO todas las relaciones entre clases deben ajustarse a dicha estructura.

En esta estructura jerárquica, cada clase tiene sólo una clase padre. La clase padre de cualquier clase es conocida como su **superclase**. La clase hija de una superclase es llamada una **subclase**.

- Una superclase puede tener cualquier número de subclases.
- Una subclase puede tener solo una superclase.



- A es la superclase de B, C y D.
- D es la superclase de E.
- B, C y D son subclases de A.
- E es una subclase de D.

d) Polimorfismo

Otro concepto de la POO es el polimorfismo. Un objeto solamente tiene una forma (la que se le asigna cuando se construye ese objeto) pero la referencia al objeto es polimórfica porque puede referirse a objetos de diferentes clases (es decir, la referencia toma múltiples formas). Para que esto sea posible **debe haber una relación de herencia entre esas clases**. Por ejemplo, considerando la figura anterior de herencia se tiene que:

- Una referencia a un objeto de la clase B también puede ser una referencia a un objeto de la clase A.
- Una referencia a un objeto de la clase C también puede ser una referencia a un objeto de la clase A.
- Una referencia a un objeto de la clase D también puede ser una referencia a un objeto de la clase A.
- Una referencia a un objeto de la clase E también puede ser una referencia a un objeto de la clase D.
- Una referencia a un objeto de la clase E también puede ser una referencia a un objeto de la clase A.

e) Abstracción

Volviendo a la figura anterior de relación de herencia entre clases, se puede pensar en una jerarquía de clases como la definición de conceptos **más abstractos en lo alto de la jerarquía** y esas ideas se convierten en algo más concreto conforme se desciende por la cadena de la superclase.

Sin embargo, las clases hijas no están limitadas al estado y conducta provistos por sus superclases; **pueden agregar variables y métodos** además de los que ya heredan de sus clases padres. Las clases hijas pueden también, **sobrescribir los métodos** que heredan por implementaciones especializadas para esos métodos. De igual manera, no hay limitación a un solo nivel de herencia por lo que se tiene un árbol de herencia en el que se puede heredar varios niveles hacia abajo y cuantos más niveles se desciende en una clase, más especializada será su conducta.

La Herencia presenta los siguientes **beneficios**:

Las subclases proveen conductas especializadas sobre la base de elementos comunes provistos por la superclase. A través del uso de herencia, los programadores pueden *reutilizar el código de la superclase* muchas veces.

Los programadores pueden implementar superclases llamadas **clases abstractas** que definen conductas "genéricas". Las superclases abstractas definen, y pueden implementar parcialmente la conducta, pero parte de la clase no está definida ni implementada. Otros programadores concluirán esos detalles con subclases especializadas.

0.2 Heredando clases en Java

El concepto de herencia conduce a una estructura jerárquica de clases o estructura de árbol, lo cual significa que en la POO todas las relaciones entre clases deben ajustarse a dicha estructura.

En esta estructura jerárquica, cada clase tiene solo una clase padre. La clase padre de cualquier clase es conocida como su superclase. La clase hija de una superclase es llamada una subclase.

De manera automática, una subclase hereda los atributos y métodos de su superclase.

Además, una subclase puede agregar nueva funcionalidad (atributos y métodos) que la superclase no tenía.

Los constructores no son heredados por las subclases.

Para crear una subclase, se incluye la palabra clave **extends** en la declaración de la clase.

```
class nombreSubclase extends nombreSuperclase {  
    .....  
}
```

En Java, la clase padre de todas las clases es la clase **Object** y cuando una clase no tiene una superclase explícita, su superclase es Object.

0.3 Sobrecarga de métodos y de constructores

Cada método tiene una "**firma**" por así decirlo, que son su nombre, el tipo y el número de sus parámetros. Existe una característica para tener dos métodos (o constructores) con el mismo nombre. Esta característica se denomina **sobrecarga de métodos**.

La **cabecera de un método** es la combinación del tipo de dato que devuelve, su nombre y su lista de argumentos.

Java diferencia los métodos sobrecargados en base al **número y tipo de argumentos** que tiene el método y **no por el tipo que devuelve**.

También existe la **sobrecarga de constructores**. Cuando en una clase existen varios constructores, se dice que hay sobrecarga de constructores.

Veamos un ejemplo en el caso de una clase llamada Publicacion y cómo los constructores nos dan un ejemplo de sobrecarga de métodos:

```

class Publicacion {
    public long idPublicacion;
    public String titulo = "";
    public String autor = "";

    public static long siguienteId = 0;
    public static String propietario = "Mari Carmen";

    Publicacion () {
        idPublicacion = siguienteId++;
    }

    Publicacion (String tituloPublicacion; String autorPublicacion) {
        this();
        titulo = tituloPublicacion;
        autor = autorPublicacion;
    }

    // ...
}

```

El compilador resolverá qué constructor debe ejecutar en cada momento en función del número de parámetros y su tipo. Si se llama al constructor sin parámetros se ejecutará el primer constructor y en caso de hacerlo con dos parámetros String, se ejecutará el segundo.

Nota: El concepto de **sobrecarga de métodos** se puede aplicar siempre que los parámetros sean diferentes, bien por su tipo, bien porque el número de parámetros de un método u otro es diferente. Hay que tener cuidado con los tipos: int, byte y short ya que aunque son tipos diferentes, si hacemos la llamada a un método con un número entero no sabría a cuál de los métodos llamar, ya que un entero puede ser considerado de las tres formas. Con double y float no pasa, porque acordaos que hemos de forzar a que Java entienda un decimal como float, por defecto lo entiende como double.

Otros ejemplos:

```

int calculaSuma(int x, int y, int z) {
...
}

int calculaSuma(double x, double y, double z) {
...
}

/* Error: los métodos siguientes no están sobrecargados */
int calculaSuma(int x, int y, int z){
...
}

double calculaSuma(int x, int y, int z){
...
}

```

Ej:

```
package usuario;
public class Usuario{

    String nombre;
    int edad;
    String direccion;

    //El constructor está sobrecargado
    public Usuario(){
        nombre = null;
        edad = 0;
        direccion = null;
    }
    public Usuario(String nombre, int edad, String direccion){
        this.nombre = nombre;
        this.edad = edad;
        this.direccion = direccion;
    }
    public Usuario(Usuario usr){
        nombre = usr.getNombre();
        edad = usr.getEdad();
        direccion = usr.getDireccion();
    }

    public void setNombre(String n) {
        nombre = n;
    }
    public String getNombre() {
        return nombre;
    }
    //El metodo setEdad() está sobrecargado
    public void setEdad(int e){
        edad = e;
    }
    public void setEdad(double e) {
        edad = (int)e;
    }
    public int getEdad() {
        return edad;
    }
    public void setDireccion(String d) {
        direccion = d;
    }
    public String getDireccion() {
        return direccion;
    }

    // Añadir un método para imprimir un usuario para eliminarlo
    // del main
}
```

```

package prueba;
import usuario.Usuario;

public class Main{
    void imprimeUsuario(Usuario usr){
        System.out.println("\nNombre: " + usr.getNombre());
        System.out.println("Edad: " + usr.getEdad());
        System.out.println("Direccion: " + usr.getDireccion());
    }
    public static void main(String args[]){
        Main prog = new Main();
        //La otra opción sería que imprimeUsuario fuera static
        Usuario usr1, usr2;

        usr1 = new Usuario();
        prog.imprimeUsuario(usr1);

        usr2 = new Usuario ("Eduardo", 24, "Mi direccion");
        prog.imprimeUsuario(usr2);

        usr1 = new Usuario(usr2);
        usr1.setEdad(50);
        usr2.setEdad(30.45);

        prog.imprimeUsuario(usr1);
        prog.imprimeUsuario(usr2);
    }
}

```

0.4 Sobreescritura de métodos

Una subclase hereda todos los métodos de su superclase que son accesibles a dicha subclase a menos que la subclase **sobreescriba los métodos**.

Una **subclase sobreescribe un método de su superclase**, cuando define un método con las mismas características (nombre, número y tipo de argumentos y retorno) que el método de la superclase.

Las subclases emplean la sobreescritura de métodos la mayoría de las veces para agregar o modificar la funcionalidad del método heredado de la clase padre.

Ej:

```

class ClaseA{
    void miMetodo(int var1, int var2){ ... }
    String miOtroMetodo(){ ... }
}

class ClaseB extends ClaseA{
    //Estos métodos sobreescriben a los métodos de la clase padre
    void miMetodo(int var1 ,int var2){ ... }
    String miOtroMetodo(){ ... }
}

```

0.5 Clases abstractas

Una clase que declara la existencia de métodos, pero no la implementación de dichos métodos, se considera una clase abstracta.

Una clase abstracta puede contener métodos no-abstractos.

Para declarar una clase o un método como abstractos, se utiliza la palabra reservada **abstract**.

```
abstract class Drawing {  
    abstract void miMetodo(int var1, int var2);  
    String miOtroMetodo() { ... }  
}
```

Una clase abstracta no se puede instanciar, no se puede hacer new Drawing, pero sí se puede heredar de ella y las clases hijas serán las encargadas de agregar la funcionalidad a los métodos abstractos. Si no lo hacen así, las clases hijas deben ser también abstractas.

Ej:

```
Public abstract class FiguraGeometrica {  
    . . .  
    abstract double Area();  
    . . .  
}  
class Circulo extends FiguraGeometrica {  
    double radio;  
    double Area() {  
        return 3.14*radio*radio;  
    }  
}
```

Se pueden crear referencias a clases abstractas como cualquier otra. No hay ningún problema en poner:

```
FiguraGeometrica figura;
```

Sin embargo, una clase abstracta no se puede instanciar, es decir, no se pueden crear objetos de una clase abstracta. El compilador producirá un error si se intenta:

```
FiguraGeometrica figura = new FiguraGeometrica();
```

Esto es coherente dado que una clase abstracta no tiene completa su implementación y encaja bien con la idea de que algo abstracto no puede materializarse.

Sin embargo, utilizando el **up-casting*** se puede escribir:

```
FiguraGeometrica figura = new Circulo(. . .);  
figura.Area();
```

(*) Es la operación en que un objeto de una clase derivada se asigna a una referencia cuyo tipo es alguna de las superclases. Cuando se realiza este tipo de operaciones, hay que tener cuidado porque la referencia es a los miembros de la clase hija, aunque la referencia apunte a un objeto de este tipo.

0.6 Interfaces (se ampliará para ver novedades -> v8)

Una interface es una variante de una clase abstracta con la condición de que todos sus métodos deben ser abstractos (lo serán aunque no se especifique). Si la interface va a tener atributos, estos deben llevar las palabras reservadas **static final** y con un valor inicial ya que funcionan como constantes.

Ej:

```
interface Nomina{  
    public static final String EMPRESA = "Patito, S. A.";  
    public void detalleDeEmpleado(Nomina obj);  
}
```

Una clase implementa una o más interfaces (separadas con comas ",") con la palabra reservada **implements**. Con el uso de interfaces se puede "simular" la herencia múltiple que Java no soporta.

```
class Empleado implements Nomina {  
    ...  
}
```

La clase que implementa una interface tiene dos opciones:

- 1) Implementar todos los métodos de la interface.
- 2) Implementar solo algunos de los métodos de la interface pero esa clase entonces debe ser una clase abstracta.

(Ver ampliación Interfaces)

0.7 Control de acceso a miembros de una clase

Los modificadores más importantes son los que permiten controlar la visibilidad y acceso a los métodos y variables que están dentro de una clase.

Uno de los beneficios de las clases, es que pueden proteger a sus variables y métodos (tanto de instancia como de clase) del acceso desde otras clases.

Java soporta cuatro niveles de acceso a variables y métodos. En orden, del más público al menos público son: público (public), protegido (protected), sin modificador (también conocido como package) y privado (private).

La siguiente tabla muestra el **nivel de acceso permitido por cada modificador**:

Visibilidad	Significado	Java	UML
Pública	Se puede acceder al miembro de la clase desde cualquier lugar.	public	+
Protegida	Sólo se puede acceder al miembro de la clase desde la propia clase o desde una clase que herede de ella.	protected	#
Por defecto	Se puede acceder a los miembros de una clase desde cualquier clase en el mismo paquete		~
Privada	Sólo se puede acceder al miembro de la clase desde la propia clase.	private	-

Como se observa en la tabla anterior:

- Una clase puede ver/usar dentro de ella misma todo tipo de variables y métodos (desde los public hasta los private);
- Las demás clases del mismo paquete (ya sean subclases o no) tienen acceso a los miembros public, protected y sin modificador.
- El resto del universo de clases (que no sean ni del mismo paquete ni subclases) pueden ver solo los miembros public.

1. Modificadores de acceso en Java

En su soporte para la **encapsulación**, la clase proporciona dos beneficios principales. *Primero*, vincula datos con el código que lo manipula. En *segundo* lugar, proporciona los medios por los que se puede controlar el acceso a los miembros.

Aunque el enfoque de Java es un poco más sofisticado, en esencia, hay dos tipos básicos de miembros de la clase: público (public) y privado (private). Se puede acceder libremente a un miembro público mediante un código definido fuera de su clase. Se puede acceder a un miembro privado solo por otros métodos definidos por su clase. A través del uso de miembros privados **el acceso está controlado**.

Restringir el acceso a los miembros de una clase es una parte fundamental de la **programación orientada a objetos**, ya que ayuda a evitar el mal uso de un objeto.

Al permitir el acceso a datos privados solo a través de un conjunto de métodos bien definidos, **podemos evitar que se asignen valores incorrectos a esos datos**, por ejemplo, realizando una verificación de rango.

- No es posible que el código fuera de la clase establezca el valor de un miembro privado directamente.
- También podemos controlar con precisión cómo y cuándo se utilizan los datos dentro de un objeto.

Por lo tanto, cuando se implementa correctamente, una clase crea una “caja negra” que se puede usar, pero cuyo funcionamiento interno no está abierto a alteraciones.

Como su nombre indica, los modificadores de acceso en Java ayudan a restringir el alcance de una clase, constructor, variable, método o miembro de datos. Hay cuatro tipos de modificadores de acceso disponibles en Java:

1. Default – No se requiere palabra clave
2. Private
3. Protected
4. Public

Tabla de Modificadores de Acceso en Java

Modificador/Acceso	Clase	Paquete	Subclase	Todos
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
default	Sí	Sí	No	No
private	Sí	No	No	No

1.1 Modificador de acceso por defecto (default)

Cuando no se especifica ningún modificador de acceso para una clase, método o miembro de datos, se dice estar teniendo **modificador de acceso default** por defecto.

Los miembros de datos, clase o métodos que no se declaran utilizando ningún modificador de acceso, es decir, que tengan un modificador de acceso predeterminado, solo son accesibles **dentro del mismo paquete**.

En este ejemplo, crearemos dos paquetes y las clases en los paquetes tendrán los modificadores de acceso predeterminados e intentaremos acceder a una clase de un paquete desde otra clase del segundo paquete.

```
// Programa Java para ilustrar el modificador default

package p1;

// La clase DemoDefault tiene modificador de acceso default

class DemoDefault {
    void mostrar()
    {
        System.out.println("Hola Mundo!");
    }
}

// Programa Java para ilustrar el error
// Al usar la clase de un paquete diferente con
// modificador default

package p2;

import p1.*;

// Esta clase tiene un modificador de acceso predeterminado

public class DemoDefaultEjecutar {
    public static void main(String args[])
    {
        DemoDefault obj = new DemoDefault();
        obj.mostrar();
    }
}
```

Salida:

```
Error:(12, 9) java: cannot find symbol
symbol: class DemoDefault
location: class p2.DemoDefaultEjecutar
```

1.2 Modificador de acceso privado (**private**)

El modificador de acceso privado se especifica con la palabra clave **private**. Los métodos o los miembros de datos declarados como privados solo son accesibles dentro de la clase en la que se declaran.

- Cualquier otra clase del mismo paquete no podrá acceder a estos miembros.
- Las clases e interfaces no se pueden declarar como privadas (**private**).

En este ejemplo, crearemos dos clases A y B dentro del mismo paquete p1. Declararemos un método en la clase A como privado e intentaremos acceder a este método desde la clase B y veremos el resultado.

```
// Programa Java para ilustrar el error
// al usar la clase desde un mismo paquete
// con modificador private

package p1;
class A {
    private void mostrar() {
        System.out.println("Ejemplo modificadores Java ");
    }
}

class B {
    public static void main(String[] args) {
        A obj= new A();
        //tratando de acceder al método privado de otra clase
        obj.mostrar();
    }
}
```

Salida:

Error:(15, 12) java: mostrar() has private access in p1.A

1.3 Modificador de acceso protegido (**protected**)

El modificador de acceso protegido se especifica con la palabra clave **protected**.

- Los métodos o miembros de datos declarados como **protected** son accesibles dentro del mismo paquete o subclases en paquetes diferentes.

En este ejemplo, crearemos dos paquetes *p1* y *p2*. La clase A en *p1* es *public*, para acceder a ella desde *p2*. El método que se muestra en la clase A está protegido y la clase B se hereda de la clase A y, a continuación, se accede a este método protegido creando un objeto de clase B.

```
// Programa Java para ilustrar
// el modificador protected

package p1;

public class A {
    protected void mostrar() {
        System.out.println("Java ejemplo");
    }
}
// Programa Java para ilustrar el modificador protected

package p2;

// importar todas las clases en el paquete p1
import p1.*;

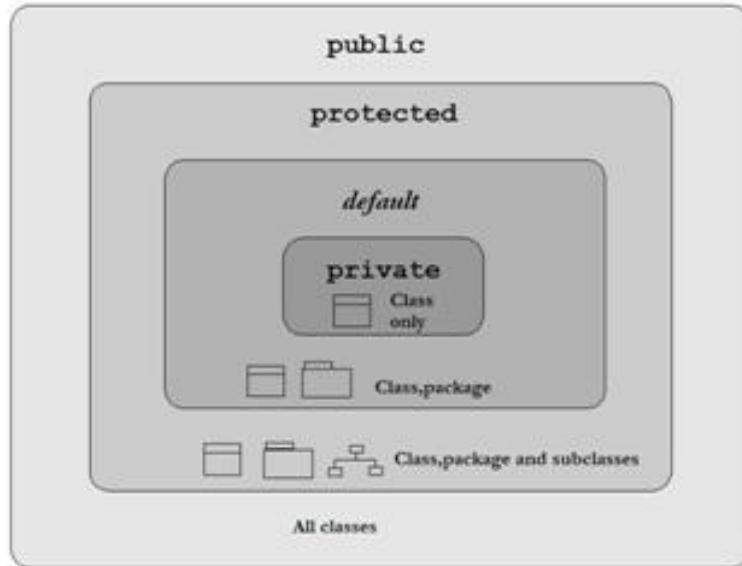
public class B extends A {
    public static void main(String[] args) {
        B obj = new B();
        obj.mostrar();
    }
}
```

Salida:

Java ejemplo

En resumen **a un miembro protected se puede acceder desde su mismo paquete y desde cualquier clase que extienda la clase en que se encuentra**, independientemente de si esta se encuentra en el mismo paquete o no. Este modificador, como private, no tiene sentido a nivel de clases o interfaces no internas.

En otras palabras, si determinada clase Hijo hereda el comportamiento de una clase Padre, la clase Hijo tendrá acceso a todos aquellos campos/métodos definidos como *protected* en Padre, pero no aquellos declarados como *private* en Padre.



1.4 Modificador de acceso público (public)

El modificador de acceso público se especifica con la palabra clave **public**.

- El modificador de acceso público tiene el alcance más amplio entre todos los demás modificadores de acceso.
- Las clases, métodos o miembros de datos que se declaran como públicos son accesibles desde cualquier lugar del programa. No hay restricciones en el alcance de los miembros de datos públicos.

```
// Programa Java para ilustrar el modificador public
package p1;
```

```
public class A {
    public void mostrar(){
        System.out.println("Java desde Cero");
    }
}
```

```
// Programa Java para ilustrar el modificador protected
package p2;
```

```
// importar todas las clases en el paquete p1
import p1.*;
```

```
public class B extends A {
    public static void main(String[] args) {
        A obj = new A();
        obj.mostrar();
    }
}
```

1.5 Modificadores que no son de acceso

En Java, tenemos 7 modificadores que no son de acceso o, a veces, también llamados especificadores. Se usan con clases, métodos, variables, constructores, etc. para proporcionar información sobre su comportamiento a la JVM. Y son:

- static
- final
- abstract
- synchronized
- transient
- volatile
- native

STATIC:

Sirve para crear miembros que pertenecen a la clase, y no a una instancia de la clase. Esto implica, entre otras cosas, que **no es necesario crear un objeto de la clase para poder acceder a estos atributos y métodos**.

- En ocasiones es necesario o conveniente generar elementos que tomen un mismo valor para cualquier número de instancias generadas o bien invocar/llamar métodos sin la necesidad de generar instancias, y es bajo estas dos circunstancias que es empleado el calificador *static*.
- Dos aspectos característicos de utilizar el calificador static en un elemento Java son los siguientes:
 - **No puede ser generada ninguna instancia** (uso de *new*) de un elemento *static* puesto que solo existe una instancia.
 - Todos los elementos definidos dentro de una estructura *static* deben ser *static* ellos mismos, o bien, poseer una instancia ya definida para poder ser invocados.

NOTA: Lo anterior no implica que no puedan ser generadas instancias dentro de un elemento *static*; no es lo mismo llamar/invocar que crear/generar.

FINAL:

Indica que una variable, método o clase no se va a modificar, lo cuál puede ser útil para añadir más semántica, por cuestiones de rendimiento, y para detectar errores.

- Si una **variable** se marca como *final*, **no se podrá asignar un nuevo valor** a la variable.
- Si una **clase** se marca como *final*, **no se podrá extender** la clase.
- Si es un **método** el que se declara como *final*, **no se podrá sobreescribir**.

Algo a tener en cuenta a la hora de utilizar este modificador es que si es un objeto lo que hemos marcado como *final*, esto no nos impedirá modificar el objeto en sí, sino tan solo usar el operador de asignación para cambiar la referencia. Por lo tanto:

```
// El siguiente código NO funcionaría:  
public class Ejemplo {  
    public static void main(String[] args) {  
        final String cadena = "Hola";  
        cadena = new String("Adios");  
    }  
}  
// El siguiente código SI funcionaría:  
public class Ejemplo {  
    public static void main(String[] args) {  
        final String cadena = "Hola";  
        cadena.concat(" mundo");  
    }  
}
```

NOTA: Una variable con modificadores *static* y *final* sería lo más cercano en Java a las constantes de otros lenguajes de programación.

ABSTRACT:

La palabra clave *abstract* indica que **no se provee una implementación para un cierto método**, sino que la implementación vendrá dada por las clases que extiendan la clase actual. **Una clase que tenga uno o más métodos abstract debe declararse como *abstract*** a su vez.

- **Mecanismos de sincronización: VOLATILE y SYNCHRONIZED.**

VOLATILE:

volatile es más simple y más sencillo que *synchronized*, lo que implica también un mejor rendimiento. Sin embargo ***volatile*, a diferencia de *synchronized*, no proporciona atomicidad** (que o se ejecuten todos los pasos o ninguno).

Una operación como el incremento, por ejemplo, no es atómica. El operador de incremento se divide en realidad en 3 instrucciones distintas (primero se lee la variable, después se incrementa, y por último se actualiza el valor) por lo que algo como lo siguiente podría causarnos problemas a pesar de que la variable sea *volatile*:

```
//Ejemplo de volatile sobre una variable
volatile int contador;

public void aumentar() {
    contador++;
}
```

Las **variables volátiles** en el lenguaje Java son declaradas anteponiendo el modificador de visibilidad de memoria ***volatile***. Estas no son guardadas en el caché del procesador, es decir, toda lectura/escritura de la misma se realiza **directamente contra la memoria principal**.

Anteponiendo *volatile* a la definición de la variable *_stop* estamos diciendo al compilador que el valor contenido en esa dirección de memoria puede modificarse en cualquier momento, y queremos que dicha modificación sea visible inmediatamente para todos los hilos que están accediendo a ella, *no queremos que su valor sea copiado en la caché del procesador*

SYNCHRONIZED:

En caso de que necesitemos atomicidad podemos recurrir a synchronized o a cosas más avanzadas, como las clases del API *java.util.concurrent* de Java 5.

synchronized se diferencia de *volatile* entre otras cosas en que este modificador **se utiliza sobre bloques de código y métodos**, y no sobre variables. Al utilizar *synchronized* sobre un bloque se añade entre paréntesis una referencia a un objeto que utilizaremos a modo de lock.

```
//Ejemplo de synchronized sobre un bloque de código
int contador;

public void aumentar() {
    synchronized(this) {
        contador++;
    }
}

//Ejemplo de synchronized sobre un metodo
int contador;

public void synchronized aumentar() {
    contador++;
}
```

1.6 Puntos importantes

Si otros programadores usan tu clase, intenta usar el nivel de acceso más restrictivo que tenga sentido para un miembro en particular. Usa private a menos que tengas una buena razón para no hacerlo. También evita los campos públicos, excepto las constantes.

<https://javadesdecero.es/poo/modificadores-de-acceso/>

<https://medium.com/@pablocastelnovo/variables-vol%C3%A1tiles-en-java-f5ae078bf8b9>

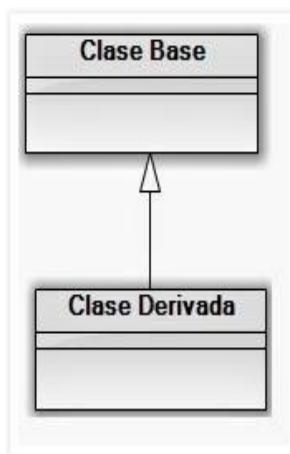
2. Herencia en Java

La herencia es una de las características fundamentales de la Programación Orientada a Objetos.

Mediante la herencia podemos **definir una clase a partir de otra ya existente**.

La clase nueva se llama **clase derivada** o subclase y la clase existente se llama **clase base** o superclase.

En UML (Unified Modelling Language) la herencia se representa con una flecha apuntando desde la clase derivada a la clase base.



La clase derivada hereda los componentes (atributos y métodos) de la clase base. La finalidad de la herencia es:

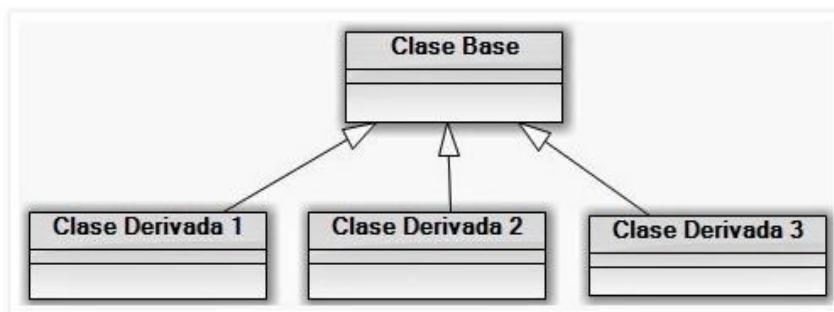
Extender la funcionalidad de la clase base: en la clase derivada se pueden **añadir** atributos y métodos nuevos.

Especializar el comportamiento de la clase base: en la clase derivada se pueden **modificar** (sobrescribir, override) los métodos heredados para adaptarlos a sus necesidades.

La herencia permite la **reutilización del código**, ya que evita tener que reescribir de nuevo una clase existente cuando necesitamos ampliarla en cualquier sentido. Todas las clases derivadas pueden utilizar el código de la clase base sin tener que volver a definirlo en cada una de ellas.

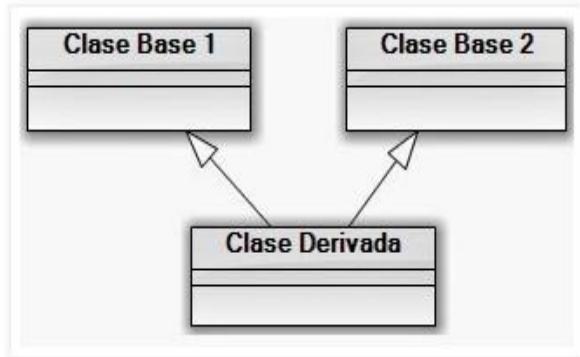
Reutilización de código: El código se escribe una vez en la clase base y se utiliza en todas las clases derivadas.

Una clase base puede serlo de tantas derivadas como se desee: Un solo parente, varios hijos.



Herencia múltiple en Java: Java no soporta la herencia múltiple. Una clase derivada solo puede tener una clase base.

Diagrama UML de herencia múltiple no permitida en Java



La herencia expresa una relación “**ES UN/UNA**” entre la clase derivada y la clase base.

Esto significa que **un objeto de una clase derivada es también un objeto de su clase base.**

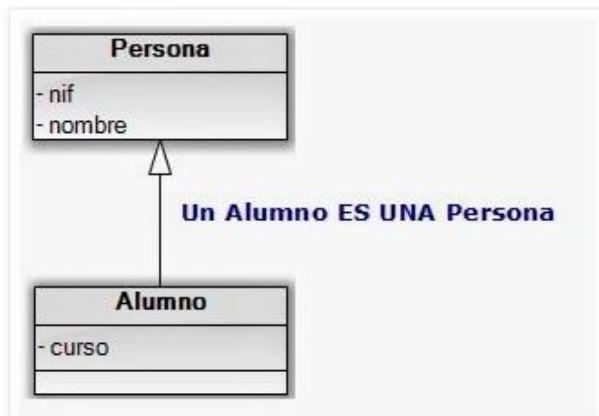
Al contrario, NO es cierto. Un objeto de la clase base no es un objeto de la clase derivada.

Por ejemplo, supongamos una clase Vehiculo como la clase base de una clase Coche. Podemos decir que un Coche es un Vehiculo pero un Vehiculo no siempre es un Coche, puede ser una moto, un camión, etc.

Un objeto de una clase derivada es a su vez un objeto de su clase base, por lo tanto, se puede utilizar en cualquier lugar donde aparezca un objeto de la clase base. Si esto no fuese posible entonces la herencia no está bien planteada.

Ejemplo de herencia bien planteada:

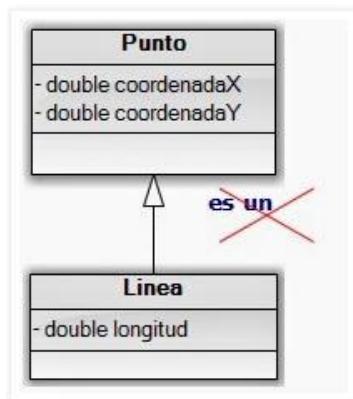
A partir de una clase Persona que tiene como atributos el nif y el nombre, podemos obtener una clase derivada Alumno. Un Alumno es una Persona que tendrá como atributos nif, nombre y curso.



Ejemplo de herencia mal planteada:

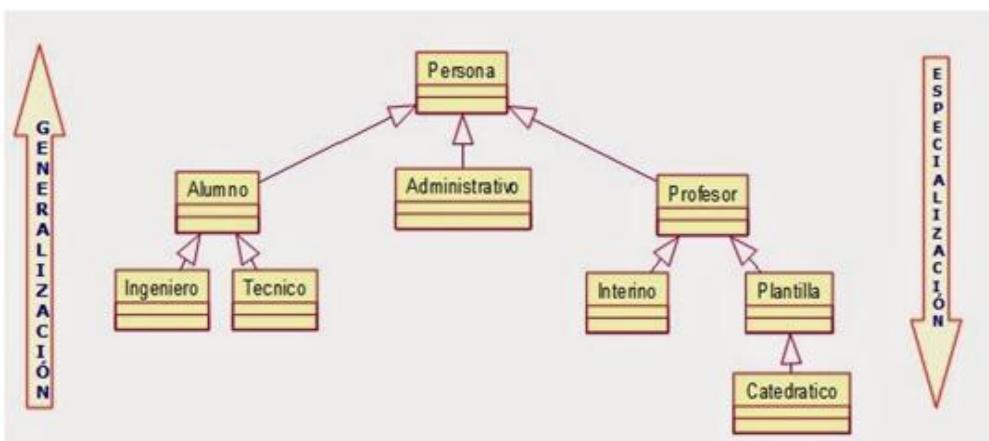
Supongamos una clase Punto que tiene como atributos coordenadaX y coordenadaY.

Se puede crear una clase Linea a partir de la clase Punto. Simplificando mucho para este ejemplo, podemos considerar una línea como un punto de origen y una longitud. En ese caso podemos crear la Clase Linea como derivada de la clase Punto, pero el planteamiento no es correcto ya que no se cumple la relación *ES UN*



Una Linea NO ES un Punto. En este caso no se debe utilizar la herencia.

Una clase derivada a su vez puede ser clase base en un nuevo proceso de derivación, formando de esta manera **una Jerarquía de Clases**.



Las clases más generales se sitúan en lo más alto de la jerarquía. Cuanto más arriba en la jerarquía, menor nivel de detalle.

Cada clase derivada debe implementar únicamente lo que la distingue de su clase base.

En java todas las clases derivan directa o indirectamente de la clase Object. **Object** es la clase base de toda la jerarquía de clases Java. Todos los objetos en un programa Java son Object.

CARACTERÍSTICAS DE LAS CLASES DERIVADAS

Una clase derivada hereda de la clase base sus componentes (atributos y métodos).

Los constructores no se heredan. Las clases derivadas deberán implementar sus propios constructores.

- Una clase derivada **puede acceder a los miembros públicos y protegidos** de la clase base como si fuesen miembros propios.
- Una clase derivada **no tiene acceso a los miembros privados** de la clase base. Deberá acceder a través de métodos heredados de la clase base.
- Si se necesita tener acceso directo a los miembros privados de la clase base se deben declarar **protected** en lugar de **private** en la clase base.

Una clase derivada puede añadir a los miembros (atributos) heredados, sus propios atributos y métodos (extender la funcionalidad de la clase). También puede modificar los métodos heredados (especializar el comportamiento de la clase base).

Una clase derivada puede a su vez, ser una clase base, dando lugar a una jerarquía de clases.

MODIFICADORES DE ACCESO JAVA

El siguiente gráfico indica el acceso a elementos de una clase según el tipo de modificador:

MODIFICADOR	PROPIA CLASE	PACKAGE	CLASE DERIVADA	RESTO
private	SI	NO	NO	NO
<Sin modificador>	SI	SI	NO	NO
protected	SI	SI	SI	NO
public	SI	SI	SI	SI

Los métodos también pueden no tener modificador. Hacer ejemplo con la clase Empleado (sacamos una de las clases derivadas fuera del paquete para comprobarlo, con el método aumentar sueldo)

El significado concreto del modificador de acceso **protected** varía ligeramente entre Java y C++: En C++, un atributo que sea declarado como “protected” en una clase será visible únicamente en dicha clase y en las subclases de la misma. En Java, un atributo que sea declarado como “protected” será visible en dicha clase, en el “package” que se encuentre la misma, y en las subclases de la misma.

HERENCIA EN JAVA. SINTAXIS

La herencia en Java se expresa mediante la palabra **extends**

Por ejemplo, para declarar una clase B que hereda de una clase A:

```
public class B extends A {  
...  
}
```

Ejemplo de herencia Java: Disponemos de una clase Persona con los atributos nif y nombre.

```
//Clase Persona. Clase Base  
  
public class Persona {  
    private String nif;  
    private String nombre;  
  
    public String getNif() {  
        return nif;  
    }  
    public void setNif(String nif) {  
        this.nif = nif;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
}
```

Queremos crear ahora una clase Alumno con los atributos nif, nombre y curso. Podemos crear la clase Alumno como derivada de la clase Persona. La clase Alumno contendrá solamente el atributo curso, el nombre y el nif son los heredados de Persona:

```
//Clase Alumno. Clase derivada de Persona  
  
public class Alumno extends Persona{  
//extends llama al constructor de persona con el nif  
//y nombre a null  
    private String curso;  
    public String getCurso() {  
        return curso;  
    }  
    public void setCurso(String curso) {  
        this.curso = curso;  
    }  
}
```

La clase alumno hereda los atributos nombre y nif de la clase Persona y añade el atributo propio curso. Por lo tanto:

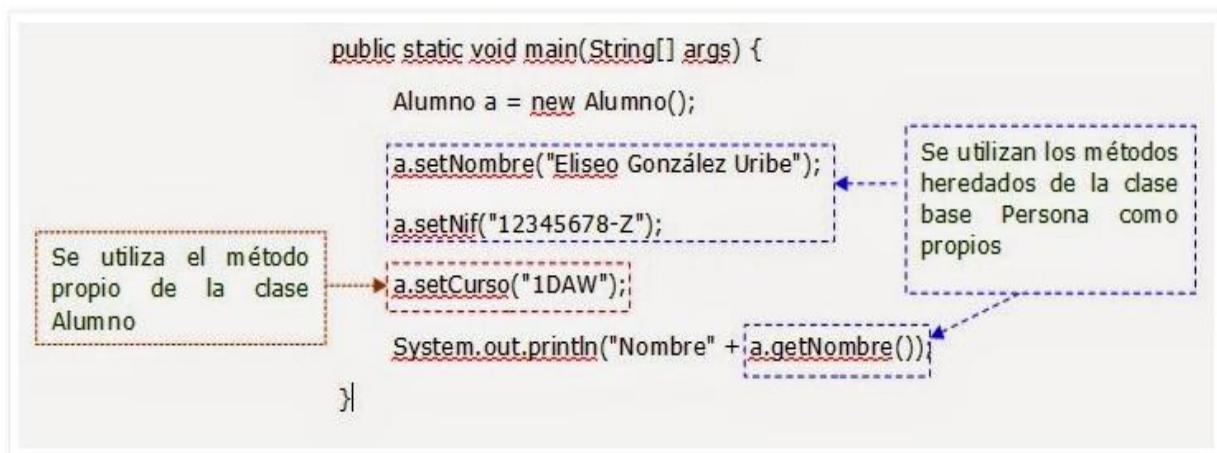
Los atributos de la clase Alumno son nif, nombre y curso.

Los métodos de la clase Alumno son: getNif(), setNif(String nif), getNombre(), setNombre(String nombre), getCurso(), setCurso(String curso).

La clase Alumno aunque hereda los atributos nif y nombre, no puede acceder a ellos de forma directa ya que son privados a la clase Persona. Se acceden a través de los métodos heredados de la clase base.

La clase Alumno puede utilizar los componentes public y protected de la clase Persona como si fueran propios.

Ejemplo de uso de la clase Alumno:



En una jerarquía de clases, cuando un objeto invoca a un método:

1. Se busca en su clase el método correspondiente.
2. Si no se encuentra, se busca en su clase base.
3. Si no se encuentra se sigue buscando hacia arriba en la jerarquía de clases hasta que el método se encuentra.
4. Si al llegar a la clase base raíz el método no se ha encontrado se producirá un error.

REDEFINIR MIEMBROS DE LA CLASE BASE EN LAS CLASES DERIVADAS

Redefinir o modificar métodos de la clase Base (Override/Sobreescribir)

Los métodos heredados de la clase base se pueden redefinir (también se le llama modificar o sobreescribir) en las clases derivadas.

El método en la clase derivada se debe escribir con el mismo nombre, el mismo número y tipo de parámetros y el mismo tipo de retorno que en la clase base. Si no fuera así estaríamos sobrecargando el método, no redefiniéndolo.

El método sobrescrito puede tener un **modificador de acceso menos restrictivo** que el de la clase base. Si, por ejemplo, el método heredado es protected se puede redefinir como public pero no como private porque sería un acceso más restrictivo que el que tiene en la clase base.

Cuando en una clase derivada se redefine un método de una clase base, se oculta el método de la clase base y todas las sobrecargas del mismo en la clase base.

Si queremos acceder al método de la clase base que ha quedado oculto en la clase derivada utilizamos:

```
super.metodo();
```

Si se quiere evitar que un método de la clase Base sea modificado en la clase derivada se debe declarar como método **final**. Por ejemplo:

```
public final void metodo () {  
.....}
```

Ejemplo:

Vamos a añadir a la clase Persona un método leer() para introducir por teclado los valores a los atributos de la clase. La clase Persona queda así:

```
public class Persona {  
    private String nif;  
    private String nombre;  
  
    public String getNif() {  
        return nif;  
    }  
    public void setNif(String nif) {  
        this.nif = nif;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
    public void leer(){  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Nif: ");  
        nif = sc.nextLine();  
        System.out.print("Nombre: ");  
        nombre = sc.nextLine();  
    }  
}
```

La clase Alumno que es derivada de Persona, heredará este método leer() y lo puede usar como propio.

Podemos crear un objeto Alumno:

```
Alumno a = new Alumno();
```

Y utilizar este método:

```
a.leer();
```

Pero utilizando este método solo se leen por teclado el nif y el nombre. En la clase Alumno se debe sobreescibir o modificar este método heredado para que también se lea el curso. El método leer() de Alumno invocará al método leer() de Persona para leer el nif y nombre y a continuación se leerá el curso.

La clase Alumno con el método leer() modificado queda así:

```
//Clase Alumno. Clase derivada de Persona

public class Alumno extends Persona{
    private String curso;

    public String getCurso() {
        return curso;
    }
    public void setCurso(String curso) {
        this.curso = curso;
    }

    @Override //indica que se modifica un método heredado
    public void leer(){
        Scanner sc = new Scanner(System.in);
        super.leer();
        //se llama al método leer de Persona para leer nif y nombre
        System.out.print("Curso: ");
        curso = sc.nextLine(); //se lee el curso
    }
}
```

Como se ha dicho antes, cuando en una clase derivada se redefine un método de una clase base, se oculta el método de la clase base y todas las sobrecargas del mismo en la clase base. Por eso para ejecutar el método leer() de Persona se debe escribir **super.leer();**

Redefinir atributos de la clase Base

Una clase derivada puede volver a declarar un atributo heredado (Atributo public o protected en la clase base). En este caso, el atributo de la clase base queda oculto por el de la clase derivada.

Para acceder a un atributo de la clase base que ha quedado oculto en la clase derivada se escribe: **super.atributo;**

CONSTRUCTORES Y HERENCIA EN JAVA. CONSTRUCTORES EN CLASES DERIVADAS

Los constructores no se heredan. Cada clase derivada tendrá sus propios constructores.

- La clase base es la encargada de inicializar sus atributos.
- La clase derivada se encarga de inicializar solo los suyos.

Cuando se crea un objeto de una clase derivada se ejecutan los constructores en este orden:

1. Primero se ejecuta el constructor de la clase base
2. Despues se ejecuta el constructor de la clase derivada.

Esto lo podemos comprobar si añadimos a las clases Persona y Alumno sus constructores por defecto y hacemos que cada constructor muestre un mensaje:

```
public class Persona {  
    private String nif;  
    private String nombre;  
    public Persona() {  
        System.out.println("Ejecutando el constructor de Persona");  
    }  
    ////////////// Resto de métodos  
}  
  
public class Alumno extends Persona{  
    private String curso;  
    public Alumno() {  
        System.out.println("Ejecutando el constructor de Alumno");  
    }  
    ////////////// Resto de métodos  
}
```

Si creamos un objeto Alumno:

```
Alumno a = new Alumno();
```

Se muestra por pantalla:

```
Ejecutando el constructor de Persona  
Ejecutando el constructor de Alumno
```

Cuando se invoca al constructor de la clase Alumno se invoca automáticamente al constructor de la clase Persona y después continúa la ejecución del constructor de la clase Alumno.

El constructor por defecto de la clase derivada llama al constructor por defecto de la clase base.

La instrucción para invocar al constructor por defecto de la clase base es:
`super();`

Todos los constructores por defecto en las clases derivadas contienen de forma implícita la instrucción `super()` como primera instrucción.

```
public Alumno() {  
    super(); //esta instrucción se ejecuta siempre. No es  
            //necesario escribirla  
    System.out.println("Ejecutando el constructor de Alumno");  
}
```

Cuando se crea un objeto de la clase derivada y queremos asignarles valores a los atributos heredados de la clase base:

1. La clase derivada debe tener un constructor con parámetros adecuado que reciba los valores a asignar a los atributos de la clase base.
2. La clase base debe tener un constructor con parámetros adecuado.
3. El constructor de la clase derivada invoca al constructor con parámetros de la clase base y le envía los valores iniciales de los atributos. Debe ser la **primera** instrucción.
4. La clase base es la encargada de asignar valores iniciales a sus atributos.
5. A continuación, el constructor de la clase derivada asigna valores a los atributos de su clase.

Ejemplo: en las clases Persona y Alumno anteriores añadimos constructores con parámetros:

```
public class Persona {  
    private String nif;  
    private String nombre;  
    public Persona() {  
        System.out.println("Ejecutando el constructor de Persona");  
    }  
    public Persona(String nif, String nombre) {  
        this.nif = nif;  
        this.nombre = nombre;  
    }  
    ////////////// Resto de métodos  
}  
public class Alumno extends Persona{  
    private String curso;  
    public Alumno() {  
        System.out.println("Ejecutando el constructor de Alumno");  
    }  
    //Constructor con parámetros. Recibe los valores de todos los atributos  
    public Alumno(String nif, String nombre, String curso) {  
        super( nif, nombre );  
        this.curso = curso;  
    }  
    ////////////// Resto de métodos  
}
```

Diagrama de flujo de ejecución:

- Al ejecutarse el constructor `Alumno()`, se llama al constructor `super()` de la clase `Persona`.
- Este llamado incluye los parámetros `nif` y `nombre` recibidos en el constructor `Alumno(String nif, String nombre)`.
- El constructor `super(nif, nombre)` asigna estos valores a los atributos `nif` y `nombre` de la clase `Persona`.
- Después, el constructor `Alumno()` asigna el valor recibido para el atributo `curso`.

Ahora se pueden crear objetos de tipo Alumno y asignarles valores iniciales. Por ejemplo:

```
Alumno a = new Alumno ("12345678-Z", "Eliseo González Manzano", "1DAW");
```

CLASES FINALES

Si queremos evitar que una clase tenga clases derivadas debe declararse con el modificador **final** delante de class:

```
public final class A{  
    .....  
}
```

Esto la convierte en clase final. Una clase final no se puede heredar.

Si intentamos crear una clase derivada de A se producirá un error de compilación:

```
public class B extends A{ //extends A producirá un error de  
compilación  
    .....  
}
```

CLASES ABSTRACTAS

Una clase abstracta es una clase que NO se puede instanciar, es decir, no se pueden crear objetos de esa clase.

Se diseñan solo para que otras clases hereden de ella.

La clase abstracta normalmente es la raíz de una jerarquía de clases y contendrá el comportamiento general que deben tener todas las subclases. En las clases derivadas se detalla la implementación.

Las clases abstractas:

- Pueden contener cero o más métodos abstractos.
- Pueden contener métodos no abstractos.
- Pueden contener atributos.

Todas las clases que hereden de una clase abstracta deben implementar todos los métodos abstractos heredados.

Si una clase derivada de una clase abstracta no implementa algún método abstracto se convierte en abstracta y tendrá que declararse como tal (tanto la clase como los métodos que siguen siendo abstractos).

Aunque no se pueden crear objetos de una clase abstracta, sí pueden tener constructores para inicializar sus atributos que serán invocados cuando se creen objetos de clases derivadas.

La forma general de declarar una clase abstracta en Java es:

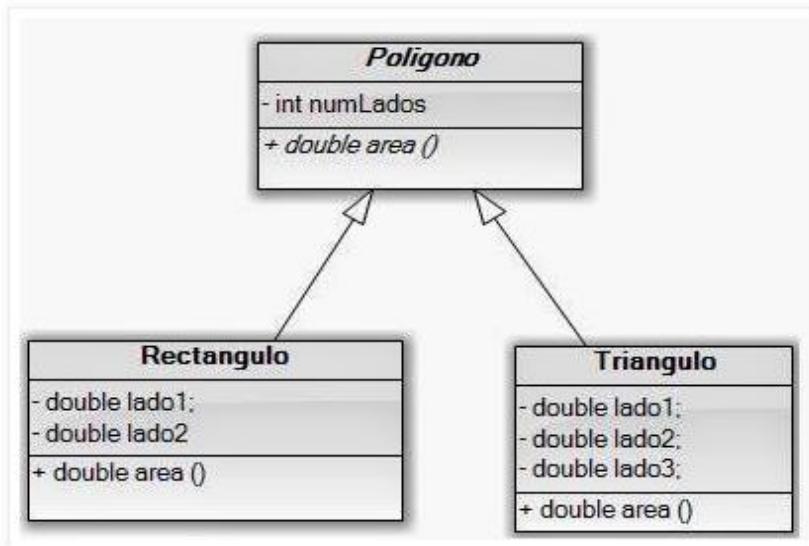
```
[modificador] abstract class nombreClase{  
}
```

Ejemplo de clase Abstracta en Java: Clase Polígono.

```
//Clase abstracta Poligono  
  
public abstract class Poligono {  
  
    private int numLados;  
  
    public Poligono() {}  
  
    public Poligono(int numLados) {  
        this.numLados = numLados;  
    }  
  
    public int getNumLados() {  
        return numLados;  
    }  
  
    public void setNumLados(int numLados) {  
        this.numLados = numLados;  
    }  
  
    //Declaración del método abstracto area()  
    public abstract double area();  
}
```

La clase Poligono contiene un único atributo *numLados*. Es una clase abstracta porque contiene el método abstracto *area()*. A partir de la clase Poligono vamos a crear dos clases derivadas Rectangulo y Triangulo. Ambas deberán implementar el método *area()*. De lo contrario también serían clases abstractas.

El diagrama UML es el siguiente:



En UML las clases abstractas y métodos abstractos se escriben con su nombre en cursiva.

```
//Clase Rectángulo
public class Rectángulo extends Polígono{

    private double lado1;
    private double lado2;

    public Rectángulo() {
    }

    public Rectángulo(double lado1, double lado2) {
        super(2);
        this.lado1 = lado1;
        this.lado2 = lado2;
    }

    public double getLado1() {
        return lado1;
    }

    public void setLado1(double lado1) {
        this.lado1 = lado1;
    }

    public double getLado2() {
        return lado2;
    }

    public void setLado2(double lado2) {
        this.lado2 = lado2;
    }

    //Implementación del método abstracto area()
    //heredado de la clase Polígono
    @Override
    public double area(){
        return lado1 * lado2;
    }
}
```

```

//Clase Triangulo
public class Triangulo extends Poligono{

    private double lado1;
    private double lado2;
    private double lado3;

    public Triangulo() {
    }

    public Triangulo(double lado1, double lado2, double lado3) {
        super(3);
        this.lado1 = lado1;
        this.lado2 = lado2;
        this.lado3 = lado3;
    }

    public double getLado1() {
        return lado1;
    }

    public void setLado1(double lado1) {
        this.lado1 = lado1;
    }

    public double getLado2() {
        return lado2;
    }

    public void setLado2(double lado2) {
        this.lado2 = lado2;
    }

    public double getLado3() {
        return lado3;
    }

    public void setLado3(double lado3) {
        this.lado3 = lado3;
    }

    //Implementación del método abstracto area()
    //heredado de la clase Polígono
    @Override
    public double area(){
        double p = (lado1+lado2+lado3)/2;
        return Math.sqrt(p * (p-lado1) * (p-lado2) * (p-lado3));
    }
}

```

Ejemplo de uso de las clases:

```

public static void main(String[] args) {
    Triangulo t = new Triangulo(3.25, 4.55, 2.71);
    System.out.printf("Área del triángulo: %.2f %n", t.area());
    Rectangulo r = new Rectangulo(5.70,2.29);
    System.out.printf("Área del rectángulo: %.2f %n", r.area());
}

```

CASTING: CONVERSIONES ENTRE CLASES

UpCasting: Conversiones implícitas

La herencia establece una relación *ES UN* entre clases. Esto quiere decir que un objeto de una clase derivada es también un objeto de la clase base.

Por esta razón:

Se puede asignar de forma implícita una referencia a un objeto de una clase derivada a una referencia de la clase base. Son tipos compatibles. También se llaman conversiones ascendentes o upcasting.

En el ejemplo anterior un triángulo en un Polígono y un cuadrado es un Polígono.

```
Poligono p;  
Triangulo t = new Triangulo(3,5,2);
```

Como un triángulo es un polígono, se puede hacer esta asignación:

```
p=t;
```

La variable p de tipo Polígono puede contener la referencia de un objeto Triangulo ya que son tipos compatibles.

Cuando manejamos un objeto **a través de una referencia a una superclase** (directa o indirecta) **solo se pueden ejecutar métodos disponibles en la superclase.**

En el ejemplo, la instrucción:

```
p.getLado1();
```

Provocará un error ya que p es de tipo Polígono y el método getLado1() no es un método de esa clase.

Cuando manejamos un objeto **a través de una referencia a una superclase** (directa o indirecta) y **se invoca a un método que está redefinido en las subclases se ejecuta el método de la clase a la que pertenece el objeto no el de la referencia.**

En el ejemplo, la instrucción

```
p.area();
```

Ejecutará el método area() de Triángulo.

DownCasting: Conversiones explícitas

Se puede asignar una referencia de la clase base a una referencia de la clase derivada, siempre que la referencia de la clase base sea a un objeto de la misma clase derivada a la que se va a asignar o de una clase derivada de ésta.

También se llaman conversiones descendentes o **downcasting**. Esta conversión debe hacerse mediante un **casting**.

Siguiendo con el ejemplo anterior:

```
Polygon p = new Triangulo(1,3,2); //upcasting  
Triangulo t;
```

```
t = (Triangulo) p; //downcasting
```

Esta asignación se puede hacer porque p contiene la referencia de un objeto triángulo.

Las siguientes instrucciones provocarán un error de ejecución del tipo **ClassCastException**:

```
Triangulo t;  
Polygon p1 = new Rectangulo(3,2);  
t = (Triangulo)p1; //----> Error de ejecución
```

p1 contiene la referencia a un objeto Rectangulo y no se puede convertir en una referencia a un objeto Triangulo. No son tipos compatibles.

EL OPERADOR instanceof

Las operaciones entre clases y en particular el downcasting requieren que las clases sean de tipos compatibles. Para asegurarnos de ello podemos utilizar el operador **instanceof**.

instanceof devuelve true si el objeto es instancia de la clase y false en caso contrario.

La sintaxis es:

```
Objeto instanceof Clase
```

Ejemplo:

```
Triangulo t;  
Polygon p1 = new Rectangulo(3,2);  
if(p1 instanceof Triangulo)  
    t = (Triangulo)p1;  
else  
    System.out.println("Objetos incompatibles");
```

3. Ejercicios Herencia

1.- Realiza un programa que sirva para representar centros educativos de diferentes tipos. Los tipos que se puede tener son: públicos, concertados y privados. Para todos ellos se quiere tener su nombre y dirección. Para los públicos se tendrá la asignación anual pública recibida. Para los concertados la asignación mensual pública recibida, así como la cuota mensual que pagan los alumnos. Para los privados se tendrá la cuota mensual que pagan los alumnos. Se pide crear tres centros, uno de cada tipo y guardarlos en un arrayList en el programa principal. Imprimir a continuación los datos de cada centro educativo usando el arrayList donde están guardados.

2.- Realiza un programa en java con las siguientes clases:

- Vivienda:
 - Atributos: metros cuadrados, calle y número.
 - Métodos para devolver/establecer valores a los atributos.
 - Constructor con valores por defecto (0,"",0)
 - Constructor con valores para los atributos
- Chalet: Una vivienda que siempre tiene 120m² y puede o no tener jardín (verdadero o falso)
- Palacio: Una vivienda de la que interesa el número de habitaciones.

En el programa principal se mostrará el siguiente menú:

```
Opcion 1- Insertar vivienda  
Opcion 2- Insertar chalet  
Opcion 3- Insertar palacio  
Opcion 4- Mostrar viviendas  
Opcion 5- Mostrar palacios  
Opcion 6- Borrar vivienda  
Opcion 7- Salir
```

Elija opcion:

En las tres primeras opciones nos pedirán los datos de cada tipo de vivienda (el tipo general corresponde a una vivienda que no es ni chalet ni palacio).

En la cuarta opción se mostrarán todas las viviendas de cualquier tipo.

En la quinta opción se mostrarán los datos de los palacios.

En la sexta opción se pide el nombre de la calle y el número de la vivienda a borrar y la eliminamos de la lista.

3.- Diseña una jerarquía de clases para los diferentes productos que se pueden encontrar en un supermercado (Alimentación, Drogería y Textil). Todos tienen un conjunto de características comunes (código, nombre y precio) y una serie de características específicas de cada producto. Los productos de alimentación tienen fecha de caducidad ([no se podrá introducir un producto si la fecha de caducidad es anterior a la fecha actual, en ese caso lanza la excepción propia FechaNoValidaException](#)), los de drogería tienen marca y los de textil color.

La fecha será introducida por el usuario pidiéndole día, mes y año por separado (harás un paquete externo al proyecto, llamado utilidadesFecha con una clase llamada ValidacionFecha que tendrá un método que reciba dia, mes y año y devuelva si es correcta la fecha, una fecha en formato LocalDate y en caso contrario una excepción denominada FechaNoValidaException)

<https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html>

--suponer que el usuario siempre mete las cosas bien

El código debe adaptarse a un tamaño de 5 caracteres de los cuales el primero y el último son letras en mayúscula y el resto números. [Utiliza expresiones regulares para controlarlo, lanzando una excepción CodigoinvalidoException que crearás si no cumple la notación.](#)

//con los codigos de las letras ASCII se puede hacer sin excepciones

Crea un carrito de la compra en el que se pueden incluir productos y emitir un ticket en el que figuren los datos de cada producto del carrito, incluyendo su precio y el importe total de la compra. Si se quiere más de una unidad de un producto, se pide varias veces, para simplificar.

Para ello, el programa al arrancar mostrará el menú:

PROCESO DE CREACIÓN DEL ALMACÉN

- 1. Introducir producto alimenticio**
- 2. Introducir producto textil**
- 3. Introducir producto droguería**
- 4. Terminar.**

Elija opción:

Con ese menú, se creará un arraylist con varios productos para poder simular los productos de un supermercado y poder hacer compras.

Entonces empezará el proceso de compras, mostrando para cada cliente el listado de productos y pidiendo al cliente que seleccione el producto que quiere comprar por nombre, hasta que ya no quiera más productos. En ese momento, se procede imprimir el ticket con los distintos productos y el precio total a pagar.

Para cada cliente nuevo se visualizará lo siguiente:

!!!Acaba de entrar en el supermercado virtual!!!!

Lista de productos

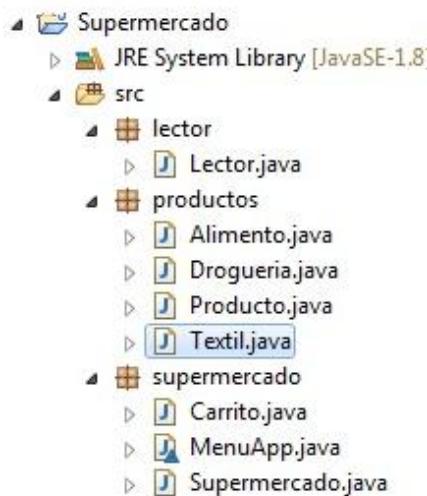
- 1 Nombre:peras Precio:2.0 fecha caducidad alimento:23/2/1989**
2 Nombre:toalla Precio:8.0 Color:azul
3 Nombre:gel Precio:2.0 Marca:Legrain

Desea comprar producto?(S/N)

Cuando el cliente termine (responde N), se calcula el importe de su compra y se empieza con un nuevo cliente (previamente se pregunta si hay un nuevo cliente)

Cuando no haya más clientes, terminaremos la ejecución del programa.

Esquema del proyecto (lector lo puedes importar como paquete externo y no incluirlo en el proyecto):



4.- Crea una clase denominada *Alarma* cuyos objetos activen un objeto de tipo *Timbre* cuando el valor medido por un sensor supere un umbral preestablecido:



Implementa en Java todo el código necesario para el funcionamiento de la alarma, suponiendo que la alarma comprueba si debe activar o desactivar el timbre cuando se invoca el método *comprobar()*.

Crea una subclase de Alarma denominada *Alarma_Luminosa*, que además de activar el timbre, encienda una luz (que representaremos con un objeto de tipo *Bombilla*). Aseguraos de que cuando se activa la alarma luminosa se enciende la luz de alarma y también suena la señal sonora asociada al timbre.

5. Tenemos que hacer un programa que gestione un centro en el que se dispone de 5 asignaturas en las que se podrán matricular los alumnos del centro. Cada alumno se matriculará de 3 asignaturas.

De dichos alumnos nos interesan las siguientes propiedades: nombre, DNI, curso, grupo y asignaturas en las que se ha matriculado (nombre), así como la nota que ha obtenido en cada una de ellas.

De cada asignatura queremos conocer el nombre y el precio.

Crearás el siguiente menú:

1. Introducir Asignaturas
2. Matricular Alumno
3. Calificar Alumno
4. Mostrar Asignaturas
5. Mostrar Alumnos
6. Salir.

Elija opción:

En la opción 1 el programa pedirá las 5 asignaturas junto con sus precios de que dispone el centro.

En la opción 2 el programa pedirá los datos de un alumno a matricular y los nombres de 3 asignaturas de las 5 disponibles.

En la opción 3 el programa pedirá los datos de un alumno a calificar en una determinada asignatura, de la que pedirá el nombre y le dará una calificación de 1 a 10 al azar.

En la opción 4 el programa muestra las asignaturas de las que se puede matricular un alumno, así como sus precios.

En la opción 5 el programa muestra todos los alumnos junto con las asignaturas de las que se ha matriculado, además de su nota en cada asignatura. (Si todavía no hay nota indicará “no disponible”) y el precio total de la matrícula.

Muestra al usuario en cada opción los datos que ya tiene almacenado el programa y que le puedan ser de ayuda para realizar la tarea pedida y considera los mensajes a mostrar en caso de no poder realizar la operación, indicando al usuario el motivo por pantalla.

4. Clases abstractas

No hay Word de clases abstractas, ver ejemplos de Eclipse

5. Interfaces

INTERFACES - (No leer, Ver ejemplos para 2eva)

5.1 ¿Qué es y para qué sirve un interfaz?

Un interfaz es un mecanismo que permite definir un conjunto de métodos y constantes, que proporcionarán una forma de acceso común a todas aquellas clases que implementen dicho interfaz. Puede verse un interfaz como un **mecanismo estándar de acceso a un conjunto de clases**.

Se trata de declarar métodos abstractos y constantes que, posteriormente puedan ser implementados de diferentes maneras según las necesidades de un programa.

Por ejemplo, el interfaz Comparable de la API de Java (puedes verlo en la documentación de Java

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html>) permite comparar objetos de forma estándar, sea cual sea la clase a la que pertenezcan, siempre y cuando dicha clase implemente **el interfaz Comparable**. En concreto, este interfaz solo contiene un método (*int compareTo(Object o)*). Ello nos asegura que, en cualquier objeto perteneciente a una de las clases que implementan el interfaz Comparable (por ejemplo, Integer, Date, Character, File, String, etc.), podemos invocar el método *compareTo* para compararlo con otro objeto.

5.2 ¿Cómo se programa un interfaz?

La API de Java proporciona bastantes interfaces ya programados. Si para tu aplicación necesitas definir un nuevo interfaz, puedes declararlo utilizando la palabra clave **interface**. En el interior de la declaración, debes declarar todas las constantes y métodos que deseas que tenga el interfaz. No puedes implementarlos: **en el interfaz solo se declaran los métodos**. Son las clases que implementan el interfaz las responsables de programar estos métodos.

```
public interface MiInterfaz {  
    int CONSTANTE = 100;  
    int metodoAbstracto( int parametro );  
}
```

Se observa que las variables (atributos) adoptan la declaración en mayúsculas, pues en realidad actuarán como constantes *final*. En ningún caso estas variables actuarán como variables de instancia.

Por su parte, los métodos tras su declaración presentan un punto y coma, en lugar de su cuerpo entre llaves. Son métodos abstractos, por tanto, métodos sin implementación.**Ej:**

```
public interface Almacen {  
    // constantes del interfaz  
    public final static int CAPACIDAD_MINIMA= 10;  
  
    // métodos del interfaz  
    public void almacenar(Object dato, int identificador);  
    public Object recuperar(int identificador);  
}
```

final static no es necesario ponerlo, por defecto son **final static**.

En el ejemplo anterior se declara un interfaz para almacenar y recuperar objetos. El interfaz, como se puede ver, solo declara los métodos, y deben ser las clases que implementen el interfaz Almacen las que programen estos dos métodos. Además de los métodos, se declara la constante CAPACIDAD_MINIMA, que puede ser accedida mediante la expresión Almacen.CAPACIDAD_MINIMA.

5.3 ¿Cómo se implementa un interfaz?

Una clase, para implementar un interfaz, debe indicarlo en su declaración, utilizando la palabra clave implements. Pero no es suficiente con indicar que implementa el interfaz, debe declarar y programar todos los métodos que indica el interfaz.

```
class ImplementaInterfaz implements MiInterfaz{  
    int multiplicando=CONSTANTE;  
    int metodoAbstracto( int parametro ){  
        return ( parametro * multiplicando );  
    }  
}
```

En este ejemplo se observa que han de codificarse todos los métodos que determina la interfaz (*metodoAbstracto()*), y la validez de las constantes (*CONSTANTE*) que define la interfaz durante toda la declaración de la clase.

Una interfaz no puede implementar otra interfaz, aunque sí extenderla (*extends*) ampliéndola.

Por ejemplo, a continuación, se presenta una clase que implementa el interfaz Almacen:

```
public class Armario implements Almacen {  
    // Declaración de atributos  
    // Otros métodos: constructores, etc.  
    // Otros métodos que no tienen nada que ver  
    // Con el interfaz  
  
    public void abrirPuertas() {  
        // Código del método  
    }  
  
    // Métodos del interfaz Almacen  
    public void almacenar(Object dato, int identificador) {  
        // Código necesario para almacenar el dato  
    }  
  
    public Object recuperar(int identificador) {  
        // Código necesario para recuperar el dato  
    }  
}
```

Es muy importante recalcar que un interfaz obliga a aquellas clases (no abstractas) que indican que lo implementan a programar todos sus métodos. Sin embargo, la clase puede, libremente, programar cualquier otro método no especificado en el interfaz.

5.4 ¿Si una clase implementa un interfaz, sus subclases lo heredan?

Dada una clase que implementa un interfaz, toda subclase de ésta, implementa implícitamente el mismo interfaz, sin necesidad de indicarlo con la palabra clave `implements`, y sin necesidad de volver a programar o declarar los métodos del interfaz.

5.5 ¿Cómo se declara una referencia a un objeto que implementa un interfaz?

Se puede declarar una referencia a un objeto que implementa un interfaz, como en el caso general, del tipo de la clase a la cual pertenece el objeto. Por ejemplo, si Armario es una clase que implementa el interfaz Almacen:

```
Armario miArmario= new Armario();  
miArmario.almacenar(new String("camisa blanca"), N_ID);
```

Pero hay situaciones en que no resulta recomendable la solución anterior. Por ejemplo, si deseamos que una misma referencia pueda apuntar a, según sea el caso, objetos de distintas clases (pero todas ellas implementando el mismo interfaz), podemos utilizar directamente el nombre del interfaz en la declaración. Por ejemplo, si tenemos la clase Caja, que también implementa

Almacen:

```
Almacen miContenedor= new Armario();
miContenedor.almacenar(new String("camisa blanca"), 56356);

...
// supongamos que ahora no se necesita más este objeto
// de la clase Armario, y se necesita uno de la clase
// Caja, utilizando la misma referencia:

miContenedor= new Caja();
miContenedor.almacenar(new Integer(34), 334233);
```

Otro caso en que resulta útil es cuando necesitamos un array de objetos que implementan un interfaz, pero cada uno de ellos puede ser de distinta clase:

```
Almacen[] contenedores= new Almacen[3];
contenedores[0]= new Armario();
contenedores[1]= new Caja();
contenedores[2]= new Armario();
```

5.6 ¿Cómo se declara un parámetro de un método para que este pueda ser un objeto de cualquier clase que implemente un determinado interfaz?

Si se declara el parámetro del método con el nombre del interfaz como tipo, entonces este parámetro podrá recibir un objeto de cualquier clase, siempre que esta implemente dicho interfaz. Por ejemplo, si las clases Armario y Caja implementan el interfaz Almacen:

```
public void guardarTodoEn(Almacen contenedor) {  
    contenedor.almacenar (objeto1, 23);  
    contenedor.almacenar (objeto2, 33);  
  
}  
  
...  
// en otra parte de la clase  
if (guardarEnCaja) guardarTodoEn(new Caja());  
else guardarTodoEn(new Armario());
```

En el ejemplo, el método guardarTodoEn puede almacenar sus objetos en un objeto de cualquier clase que implemente el interfaz Almacen. Una restricción que tiene esto es que, de esta forma, solo se puede acceder directamente en el método guardarTodoEn, a aquellos métodos que declara el interfaz, pero no al resto de los métodos que posean las clases que lo implementan. Para poder acceder habría que realizar un casting al objeto apuntado.

Todo lo explicado anteriormente es igualmente aplicable al valor de retorno de un método.

Java es un lenguaje que incorpora **herencia simple** de implementación pero que puede aportar **herencia múltiple de interfaz**. Esto posibilita la herencia múltiple en el diseño de los programas Java.

Una interfaz puede heredar de más de una interfaz antecesora.

```
interface InterfazMultiple extends Interfaz1, Interfaz2{ }
```

Una clase no puede tener más que una clase antecesora, pero puede implementar más de una interfaz:

```
class MiClase extends SuPadre implements Interfaz1, Interfaz2{ }
```

El ejemplo típico de herencia múltiple es el que se presenta con la herencia en diamante:

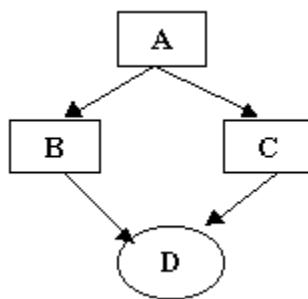


Imagen 6: Ejemplo de herencia múltiple

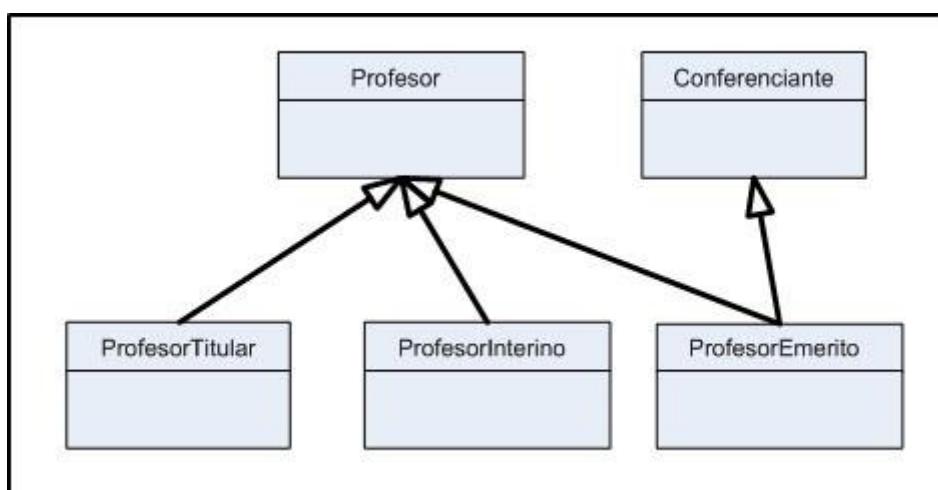
Para poder llevar a cabo un esquema como el anterior en Java es necesario que las clases A, B y C de la figura sean interfaces, y que la clase D sea una clase (que recibe la herencia múltiple):

```

interface A{ }
interface B extends A{ }
interface C extends A{ }
class D implements B,C{ }

```

Ejemplo:



Colisiones en la herencia múltiple

En una herencia múltiple, los identificadores de algunos métodos o atributos pueden coincidir en la clase que implementa varios interfaces, si dos de las interfaces padres tienen algún método o atributo que coincide en nombre. A esto se le llama *colisión*.

Esto se dará cuando las interfaces padre (en el ejemplo anterior *B* y *C*) tienen un atributo o método que se llame igual. Java resuelve el problema estableciendo una serie de reglas.

Para la colisión de nombres de atributos, se obliga a especificar a qué interfaz base pertenecen al utilizarlos.

Para la colisión de nombres en métodos:

- Si tienen el mismo nombre y diferentes parámetros: se produce sobrecarga de métodos permitiendo que existan varias maneras de llamar al mismo.
- Si solo cambia el valor devuelto: se da un error de compilación, indicando que no se pueden implementar los dos.
- Si coinciden en su declaración: se elimina uno de los dos, con lo que solo queda uno.

5.7 Diferencias entre un interface y una clase abstracta

Un interface es simplemente una lista de métodos no implementados, además puede incluir la declaración de constantes. Una clase abstracta puede incluir métodos implementados y no implementados o abstractos, miembros dato constantes y otros no constantes.

Un interface es parecido a una clase abstracta en Java, pero con las siguientes diferencias:

- Todo método es abstracto y público sin necesidad de declararlo. Por lo tanto, un interface en Java no implementa ninguno de los métodos que declara.
- Un interface se implementa (`implements`) no se extiende (`extends`) por sus subclases.
- Una clase puede implementar más de un interfaz en Java, pero solo puede extender una clase. Es lo más parecido que tiene Java a la herencia múltiple, que de clases normales está prohibida.
- Podemos declarar variables del tipo de clase del interfaz, pero para inicializarlas tendremos que hacerlo de una clase que lo implemente.

Así, por ejemplo, podemos declarar el siguiente interfaz en Java:

```
public interface Figura{  
    int area();  
}
```

Y una clase que lo implementa:

```
public class Cuadrado implements Figura {  
  
    int lado;  
    public Cuadrado (int ladoParametro) {  
        lado = ladoParametro;  
    }  
  
    public int area(){  
        return lado*lado;  
    }  
}
```

Más adelante podemos:

```
public class PruebaInterfaz{  
  
    public static void main(String args[]){  
  
        Figura figura=new Cuadrado (5);  
        // Podemos crear una referencia de interface(variable  
        // figura) y que un objeto  
  
        // que pertenezca a una clase que la implementa le sea  
        // asignada a la variable  
        System.out.println(figura.area());  
    }  
}
```

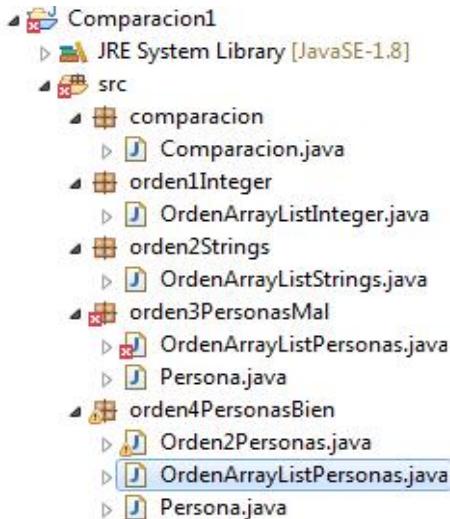
Una clase solamente puede derivar **extends** de una clase base, pero puede implementar varios interfaces. Los nombres de los interfaces se colocan separados por una coma después de la palabra reservada **implements**. El lenguaje Java no fuerza, por tanto, una relación jerárquica, simplemente permite que clases no relacionadas puedan tener algunas características de su comportamiento similares.

6. Comparator - Comparable

**ORDENACIÓN UTILIZANDO LOS INTERFACES DE JAVA
COMPARABLE Y COMPARATOR**
- este merece la pena leerlo 2^a eva

6.1 Introducción

Vamos a ver algunos ejemplos para centrarnos en cuál es el problema que tenemos que resolver (Proyecto Comparacion1).



Recordemos cómo funciona el método `compareTo`, que hemos usado muchas veces (paquete **comparacion**):

```
public class Comparacion {

    public static void main(String[] args) {
        Integer num = new Integer(6);
        Int resul = num.compareTo(new Integer(10));

        if (resul == 0)
            System.out.println("Son iguales");
        else if (resul > 0)
            System.out.println("num es mayor");
        else
            System.out.println("num es menor");

        resul = "adios".compareTo("Hola");

        if (resul == 0)
            System.out.println("Son iguales");
        else if (resul > 0)
            System.out.println("adios es mayor");
        else
            System.out.println("adios es menor");
    }
}
```

Salida:

```
num es menor  
adios es mayor
```

Fíjate que la salida te dice si ponemos adiós que es mayor que Hola y si ponemos Adios nos saldría que es menor. Esto se debe a que en la tabla ASCII se colocan primero las mayúsculas y luego las minúsculas y ese el **orden natural** que usa compareTo.

Lo mismo que para los números, el 1 siempre será menor que el 2.

En el **ejemplo** del paquete **orden1Integer** vemos que la clase **Collections tiene un método estático llamado sort()**, al que pasándole por parámetro una colección, le decimos que la ordene. Si no especificamos el orden, el criterio es el mismo que el que utiliza el método compareTo() que ya conocemos (ya veremos cómo podemos especificar el orden).

En el **ejemplo** del paquete **orden2Strings** vemos cómo podemos ordenar también un ArrayList de Strings.

En estos dos ejemplos nos fijamos en que la forma de ordenar cuando queremos hacerlo al revés que con el orden natural es añadiendo al método sort un segundo parámetro:

```
Collections.sort(arrayListInt, Collections.reverseOrder());
```

Collections.reverseOrder() nos devuelve un objeto de tipo Comparator que nos permite ordenar la colección (o Array) de forma opuesta a la natural. Por lo que vemos que ahora el orden es descendente.

En caso de tener un Array haríamos lo siguiente para ordenar de las dos formas:

```
Arrays.sort(a);  
Arrays.sort(a, Collections.reverseOrder());
```

En el **ejemplo** del paquete **orden3PersonasMal** vemos que tenemos un ArrayList de personas y que como tiene varios atributos en el objeto que queremos comparar, pues sort no sabe cómo tiene que ordenar (y no ordena, porque además da un error de compilación al intentar hacerlo).

6.2 Uso del interface Comparable y método compareTo (API Java) para comparar y ordenar objetos y colecciones

Ya vimos con los algoritmos de ordenación cómo podemos ordenar Arrays o ArrayList (en general cualquier colección).

Veremos que no es necesario incluir en nuestro código esos métodos, ya que en Java tenemos la posibilidad de ordenar de una manera más sencilla. Es el propio Java quien ordenará por nosotros, tal y como vimos en los dos ejemplos de la Introducción. Eso sí, tenemos que entender en qué se basa esa ordenación para ser capaces de utilizar la de manera adecuada.

En el ejemplo del paquete **orden3PersonasMal**, ya no nos funciona porque no hemos indicado qué criterio usamos para decidir la comparación (se trata de una clase que tiene varios atributos ahora).

Vamos a ver cómo una clase debe hacer uso de la interfaz **Comparable**. Esto nos va a permitir que en dicha clase se pueda realizar la comparación de objetos, permitiendo hacer **ordenaciones** de los mismos. Seguimos con la clase Persona de antes:

```
public class Persona {  
    public int nombre, edad;  
    public Persona( int nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
}
```

Lo primero que tenemos que pensar es que si queremos ordenar personas, tenemos que decidir cuál es el criterio para decidir que una persona es mayor, menor o igual a otra. En este caso podríamos decir que una persona es menor si tiene una edad menor, pero es una decisión que depende del caso en concreto. También podría ser válido si quisieramos hacer un listado de personas ordenar por el nombre para hacer un listado alfabético.

Vemos ahora el paquete **orden4PersonasBien**:

En **Orden2Personas.java** vemos que lo primero es implementar el interfaz Comparable:

```
public class Persona implements  
Comparable<Persona>
```

La base para realizar la ordenación va a ser la **interfaz Comparable**, en este primer ejemplo.

```
public interface  
Comparable<T>
```

Que tiene el método:

```
int compareTo(T o)
```

¿Qué significa la <T>?. Representa que se puede aplicar a cualquier Tipo de Objeto, y que hay que especificarlo como hemos visto antes.

Vamos a implementar la interfaz Comparable en la clase que representa los objetos que queremos ordenar. Por ello, estamos obligados a implementar (sobreescribir) el método de la interfaz:

```
@Override  
public int compareTo  
(Persona o)
```

Este método debe devolver un número negativo, cero o un número positivo en función de que el objeto que comparemos con el objeto “o” de referencia sea menor, igual o mayor respectivamente que ese objeto de referencia.

Por tanto, vamos a incluir el método **public int compareTo(Persona o)** en nuestra clase **Persona**.

```
@Override  
public int compareTo(Persona o) {  
    return this.nombre.compareTo(o.nombre);  
}
```

En el **main** de **Orden2Personas.java** vemos:

```
Persona p1 = new Persona("Pepe", 28);  
Persona p2 = new Persona("Juan", 32);  
Persona p3 = new Persona("Juan", 32);  
  
System.out.println(p1.compareTo(p2));  
System.out.println(p2.compareTo(p1));  
System.out.println(p2.compareTo(p3));
```

La **salida** será:

```
6  
-6  
0
```

Esa salida refleja por ejemplo, que Pepe es mayor que Juan, ya que está después en el alfabeto. En concreto, lo que refleja es que hay 6 posiciones de diferencia en el alfabeto.

Nos devuelve esto porque hemos programado en nuestro método **compareTo** que compare así el atributo nombre de las personas.

En el mismo paquete, tenemos **OrdenArrayListPersonas.java**, en el que ahora lo que tenemos es una colección de Personas. El método estático **sort** de la clase **Collections** nos va a ordenar dicha colección en base al método **compareTo** que hemos programado. Ejecútalo y observa la salida.

- 1.- ¿Cómo hacemos si queremos la ordenación alfabéticamente en orden descendente?
- 2.- ¿Cómo hacemos si queremos la ordenación por edad en orden ascendente?
- 3.- ¿Cómo hacemos si queremos la ordenación por edad en orden descendente?

RESUMEN

Si tenemos 2 Personas y queremos compararlas u ordenarlas, la pregunta a hacerse sería:

“cuándo una persona es mayor que otra o cuándo es menor, o cuándo son iguales”, atendiendo a algún tipo de atributo. Después sobreescrivimos compareTo de acuerdo al criterio establecido.

Cuando en lugar de comparar explícitamente 2 personas lo que queremos es que se ordene una colección, lo que hay que hacer es sobreescibir el método compareTo, que es el que aplica el método sort para ser capaz de ordenar una colección o un array. Esto es lo que hemos visto en **OrdenArrayListPersonas.java**, cuando hace:

```
Collections.sort(personas);
```

En el proyecto **Comparacion2**, podemos ver un ejemplo (paquete ordenArrayPersonas) de cómo se puede hacer una ordenación en un array:

```
Arrays.sort(arrayPersonas);
```

En el proyecto **Comparacion3**, vemos otro ejemplo (paquete ordenArrayListPersonas) de cómo se puede hacer una ordenación en un ArrayList:

En este caso la clase Persona quedaría de la siguiente manera:

```
public class Persona implements Comparable<Persona>{  
    public int dni, edad;  
    public Persona( int d, int e){  
        this.dni = d;  
        this.edad = e;  
    }  
}
```

Como podemos ver resaltado hemos incluido también el código **implements Comparable<Persona>**, que nos indica que vamos a implementar en la clase la interfaz Comparable para el parámetro Persona.

También hemos incluido el código necesario para implementar el método **public int compareTo(Persona o)**, donde escribiremos el criterio para comparar personas. Si consideramos que el orden será **ascendente por dni**, tenemos que escribir:

```
@Override  
public int compareTo(Persona o) {  
    return (new Integer(this.dni)).compareTo(new  
Integer(o.dni));  
}
```

O también:

```
@Override  
public int compareTo(Persona o) {  
    if (this.dni > o.dni)  
        return 1;  
    else if (this.dni < o.dni)  
        return -1;  
    else  
        return 0;  
}
```

Date cuenta que **lo que se compara en compareTo deben ser objetos**.

Ahora escribimos un programa que generará 2 Personas y las compara mostrando cuál es la relación de comparación entre ellas (en base a los dnis).

```
public class Programa {  
    public static void main(String arg[]) {  
        Persona p1 = new Persona(74999999, 35);  
        Persona p2 = new Persona(72759474, 30);  
  
        if (p1.compareTo(p2) < 0 ) {  
            System.out.println("La persona p1: es menor.");  
        }  
        else if (p1.compareTo(p2) > 0 ) {  
            System.out.println("La persona p1: es mayor.");  
        }  
        else {  
            System.out.println ("La persona p1 es igual  
                a la persona p2");  
        }  
    }  
}
```

Comprueba con varios casos si funciona bien. Si queremos ordenar un ArrayList, vemos que funcionaría perfectamente:

```
ArrayList<Persona> personas = new ArrayList<Persona>();  
  
personas.add(new Persona(74999999, 35));  
personas.add(new Persona(72759474, 30));  
personas.add(new Persona(11111111, 35));  
personas.add(new Persona(22222222, 30));  
personas.add(new Persona(33333333, 35));  
personas.add(new Persona(44444444, 30));  
personas.add(new Persona(55555555, 35));  
personas.add(new Persona(66666666, 30));  
  
// Ahora sí nos deja ordenar personas  
Collections.sort(personas);
```

Comprueba la salida.

Si queremos ordenar descendentemente por dni:

```
Collections.sort(personas, Collections.reverseOrder());
```

El reverseOrder() lo hace según lo que hayamos escrito en el método sobrescrito compareTo(),

Otra forma de ordenar descendentemente, sin usar reverseOrder() sería dar la vuelta a nuestra comparación:

```
@Override  
public int compareTo(Persona o) {  
    return (new Integer(o.dni)).compareTo(new Integer(this.dni));  
    // hemos intercambiado el orden de comparación  
}
```

Esta **implementación de la interfaz Comparable** es muy habitual ya que en general muchas veces vamos a desear que nuestras clases sean comparables no solo por el hecho de poder comparar objetos de dicha clase, sino por el uso de poder ordenar objetos de esa clase. La **ordenación** es una de las funciones más importantes y necesarias en el ámbito de la informática, ya que tener la información ordenada hace mucho más rápidas las consultas o modificaciones de datos.

¿Por qué crees que se usa la implementación de **interfaces** en Java?

Hay varios motivos, pero uno principal es que mediante la implementación de interfaces todos los programadores usan el mismo nombre de método y estructura formal para comparar objetos (o clonar, u otras operaciones).

Imagínate que estás trabajando en un equipo de programadores y tienes necesidad de utilizar una clase que ha codificado otro programador. Gracias a que se implementan interfaces Java, no necesitarás estudiar el código para ver qué método hay que invocar para comparar objetos de esa clase. Dado que todos los programadores usan la implementación de interfaces, sabemos que la

comparación de objetos (o clonación, etc.) debe hacerse invocando determinados métodos de determinada forma. Esto facilita el desarrollo de programas y la comprensión de los mismos, especialmente cuando hablamos de programas o desarrollos donde pueden intervenir cientos o miles de clases diferentes.

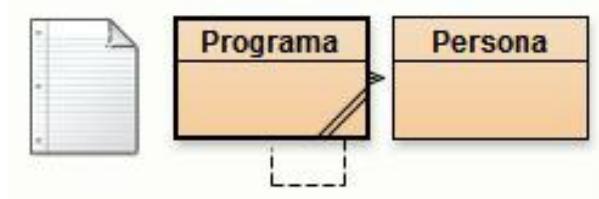
En el mismo caso de la clase Persona, vamos a cambiar el criterio de ordenación:

Ahora vamos a considerar que ordenamos a las personas por su edad, pero en el caso de que dos personas tengan la misma edad, vamos a considerar que es menor la que tenga un número de DNI más bajo. Para este caso debes introducir más personas que cumplan estas condiciones, con la misma edad.

```
@Override  
  
public int compareTo(Persona o) {  
    if (this.edad > o.edad)  
        return 1;  
    else if (this.edad < o.edad)  
        return -1;  
    else if (this.dni > o.dni)  
        return 1;  
    else if (this.dni < o.dni)  
        return -1;  
    else  
        return 0;  
}
```

Si cambiamos el criterio para que se ordene el ArrayList descendente por edad y ascendente por DNI ¿Cómo se haría?

El diagrama de clases de este ejemplo sería:



6.3 Interface Comparator (API java). Diferencias con Comparable. Clase Collections. Código ejemplo

INTERFACE COMPARATOR

A continuación procederemos a describir de manera detallada el uso y el comportamiento de la **interface Comparator** del api de Java.

Además veremos un ejemplo de uso de esta interfaz y cómo se utiliza. Señalaremos también cuáles son las **diferencias entre la interface Comparator y la interface Comparable**, dos interfaces que tienen ciertas similitudes y ciertas diferencias.

COMPARATOR

Esta interfaz es la encargada de permitirnos el poder comparar 2 elementos en una colección. Por tanto pudiera parecer que es igual a la interfaz **Comparable** (recordemos que esta interfaz nos obligaba a implementar el método **compareTo** (Object o)) que hemos visto anteriormente en el paquete java.lang.

Su diferencia es:

Mientras que **Comparable** nos obliga a implementar el método:

`compareTo (Object o)`

la interfaz **Comparator** nos obliga a implementar el método:

`compare (Object o1, Object o2)`

Si, por ejemplo para la clase Persona quisiéramos ordenar a las Personas por su altura, o por su nombre en orden alfabético podríamos recurrir a la interfaz **Comparator** para implementar el método **compare** (Object o1, Object o2) definiendo el método deseado.

EJERCICIO RESUELTO

Ahora vamos a desarrollar un ejercicio para comprender mejor su funcionamiento. Vamos a suponer una clase **Persona** similar a la vista en anteriores ocasiones sobre la cual vamos a definir un orden para poder ordenar colecciones de ese elemento.

Al igual que antes, usaremos la clase ArrayList para contener una colección de objetos.

Lo primero será definir mediante código nuestra clase Persona que será la siguiente:

```
/* Ejemplo Interface Comparable*/  
  
public class Persona implements Comparable<Persona> {  
    private int idPersona;  
    private String nombre;  
    private int altura;  
  
    public Persona (int idPersona, String nombre, int altura) {  
        this.idPersona = idPersona;  
        this.nombre = nombre;  
        this.altura = altura;}  
  
    @Override  
    public String toString() {  
        return "Persona-> ID: " +idPersona  
            +" Nombre: " +nombre  
            +"Altura: "+altura;  
    }  
  
    @Override  
    public int compareTo(Persona o) {  
        return this.nombre.compareTo(o.nombre);  
    }  
  
    public int getIdPersona() {return idPersona;}  
    public String getNombre() {return nombre;}  
    public int getAltura() {return altura;}  
}
```

En ella podemos ver que cada objeto de la clase Persona tendrá como campos un idPersona, nombre y altura. No hemos detallado los métodos set correspondientes a cada propiedad porque trataremos de centrarnos en el código que resulta de interés para este caso concreto.

Observamos también que queremos que esta clase sea ordenada naturalmente como ya hemos visto anteriormente al implementar Comparable. En este caso la ordenación se basa en el campo nombre de las Personas como queda reflejado en el método compareTo. Es decir, cuando se trate de realizar una ordenación “natural”, esta se hará por orden alfabético de nombres.

Las colecciones de Persona se ordenarán por este método siempre que se invoque una forma de ordenación que use la ordenación natural. Como ejemplo tenemos el siguiente código:

```
import java.util.ArrayList;
import java.util.Collections;

public class Programa {
    public static void main(String arg[]) {
        ArrayList<Persona> listaPersonas = new ArrayList<Persona>();
        listaPersonas.add(new Persona(1, "Maria", 185));
        listaPersonas.add(new Persona(2, "Carla", 190));
        listaPersonas.add(new Persona(3, "Yovana", 170));

        Collections.sort(listaPersonas); // Ejemplo uso ordenación natural
        System.out.println("Personas Ordenadas por orden natural:
                            "+listaPersonas);
    }
}
```

La lista de Personas ha sido ordenada usando la invocación `Collections.sort` pasando como parámetro la colección y el orden utilizado en esta invocación es el orden natural definido en su clase, es decir, por el nombre de cada Persona. En este caso: Carla, María y Yovana de acuerdo al orden alfabético C, M, Y.

La clase **Collections** que hemos utilizado es una clase similar a la clase **Arrays** del api de Java. Puede ser invocada simplemente importándola y nos aporta distintos métodos estáticos, siendo únicamente necesario pasarle determinados parámetros para obtener un resultado.

Ahora bien, imaginemos que queremos ordenar a las Personas por un orden distinto a lo que hemos denominado orden natural, en este caso hablaremos de orden total. Como orden total podríamos elegir la altura. Una forma sería modificar el método `compareTo`, como hacíamos en el apartado anterior.

Y otra forma consiste (y a partir de aquí es lo nuevo) en hacer uso de la **interfaz Comparator** de la siguiente manera:

Creamos una clase diferente para **implementar la interfaz Comparator**, donde **sobreescrivimos el método compare()** y en la clase Persona ya no implementaremos la interfaz Comparable.

```
/* Ejemplo Interface Comparator */

import java.util.Comparator;

public class OrdenarPersonaPorAltura implements Comparator<Persona> {
    @Override
    public int compare(Persona o1, Persona o2) {
        return o1.getAltura() - o2.getAltura();
        // Devuelve un entero positivo si la altura de o1 es mayor
        que la de o2
    }
}
```

Hemos definido el orden así, si $o1 < o2$ en este caso queremos que $o1$ vaya delante de $o2$ (decimos que es "más pequeño") por ordenarlo de menor altura a mayor altura. Entonces debemos devolver un número negativo, 0 si son iguales y un número positivo si $o2$ es de menor altura.

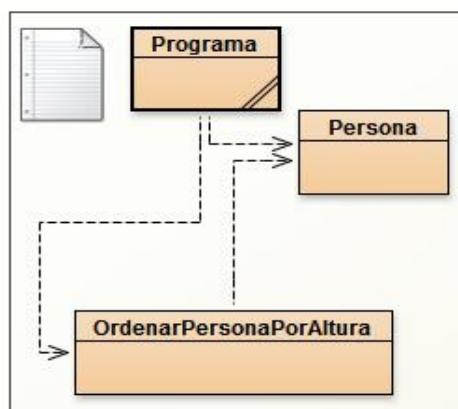
El siguiente paso es utilizar la clase OrdenarPersonaPorAltura de la siguiente manera en el programa principal:

```
public class Programa {  
    public static void main(String arg[]) {  
        ArrayList<Persona> listaPersonas = new ArrayList<>();  
  
        listaPersonas.add(new Persona(1, "Maria", 185));  
        listaPersonas.add(new Persona(2, "Carla", 190));  
        listaPersonas.add(new Persona(3, "Yovana", 170));  
  
        Collections.sort(listaPersonas,  
                         new OrdenarPersonaPorAltura());  
  
        System.out.println("Personas Ordenadas por orden total: "  
                           + listaPersonas);  
    }  
}
```

La única observación con respecto al anterior programa es que ahora la sentencia de ordenación Collections.sort lleva añadido el parámetro **new OrdenarPersonaPorAltura()**.

Esta es otra forma del método sort de la clase Collections, en este caso pasándole como parámetros un objeto colección y un objeto que implemente Comparator y defina un orden a utilizar. Al pasarle un objeto de tipo OrdenarPersonaPorAltura indicamos el orden total a utilizar, que en este caso es la altura.

El diagrama de clases de este ejemplo es:



CONCLUSIONES

La **primera conclusión** puede ser que aunque **Comparable** y **Comparator** parecen iguales, en realidad no lo son, y mientras una se define para el orden natural, la otra se define para un orden total.

El orden natural es utilizado por diversos métodos del api de Java, pero no siempre hemos de usar el orden natural. En determinados casos podremos querer ordenar objetos por un orden distinto al orden natural, y para ello nos será útil implementar la interface Comparator. Aunque vimos que dando la vuelta al método compareTo, somos capaces también de ordenar sin usar la interfaz Comparator.

La **segunda conclusión** es que gracias a la interfaz **Comparator**, podemos ordenar muy fácilmente colecciones utilizando clases que implementen el método compare **por cada tipo de ordenación que deseemos. Cosa que no podíamos hacer con la interfaz Comparable, ordenamos en el mismo programa solo de una forma, obviamente no es posible sobreescribir el método compareTo en la misma clase más de una vez.**

Prueba detenidamente los proyectos Jugadores1..Jugadores4 para terminar de comprender todo lo explicado.

En el apartado del Aula Virtual de **Excepciones, hicimos un ejercicio con Cuentas de bancos, en el que no desarrollasteis la pregunta:**

“Crea una **segunda versión** en la que las cuentas se puedan ver de dos formas distintas, ascendente por número de cuenta y descendente por saldo:”

1. Abrir cuenta
2. Ingresar dinero en cuenta
3. Sacar dinero de cuenta
4. Mostrar todas las cuentas(desc por saldo)
5. Mostrar todas las cuentas(asc por num. cuenta)
6. Mostrar una cuenta
7. Borrar una cuenta
8. Borrar todas las cuentas
9. Salir.

Elija opcion:

Lo que os pido ahora es que completéis el ejercicio con esas opciones y lo subáis al Aula virtual antes del jueves.

7. Interfaces en Java. Ejemplos de Interfaces.

Antes de Java 8 una interface era considerada como una **clase abstracta pura**: todos sus métodos son abstractos y si tiene atributos son todos constantes.

A partir de Java 8 el concepto de Interface ha cambiado. Podemos considerar una interface como una **clase abstracta** que sólo puede contener constantes, métodos abstractos, métodos por defecto, métodos estáticos y tipos anidados. Se crean utilizando la palabra clave **interface** en lugar de class.

```
[public] interface NombreInterface [extends Interface1, Interface2, ...]{  
    [métodos abstractos]  
    [métodos default]  
    [métodos static]  
    [tipos anidados]  
    [atributos constantes]  
}
```

La interface puede definirse public o sin modificador de acceso, y tiene el mismo significado que para las clases. Si tiene el modificador public el archivo .java que la contiene debe tener el mismo nombre que la interfaz. Igual que las clases, al compilar el archivo .java de la interface se genera un archivo .class

Las interfaces no pueden ser instanciadas, solo pueden ser implementadas por clases o extendidas por otras interfaces.

A diferencia de las clases que pueden heredar de una sola clase, las interfaces en Java pueden heredar de varias interfaces.

Todos los métodos que aparecen en la interfaz son públicos por lo que se puede omitir el modificador de acceso *public*.

En los métodos abstractos no es necesario escribir *abstract*.

Los métodos por defecto se especifican mediante el modificador *default*.

Los métodos estáticos se especifican mediante la palabra reservada *static*.

Todos los atributos son constates públicos y estáticos. Por lo tanto, se pueden omitir los modificadores *public static final* cuando se declara el atributo. Se deben inicializar en la misma instrucción de declaración.

Los nombres de las interface suelen acabar en *able* aunque no es necesario: configurable, arrancable, dibujable, comparable, clonable, etc.

Las interfaces juegan un papel fundamental en la creación de aplicaciones Java:

- Las interfaces definen un protocolo de comportamiento y proporcionan un formato común para implementarlo en las clases.
- Utilizando interfaces es posible que clases no relacionadas, situadas en distintas jerarquías de clases sin relaciones de herencia, tengan comportamientos comunes.

Ejemplo:

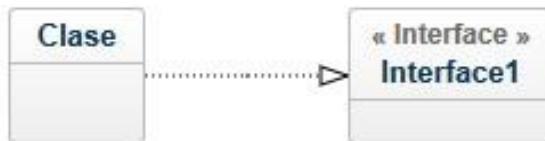
```
//Interfaz que define relaciones de orden entre objetos.  
public interface Relacionable {  
    boolean esMayorQue(Relacionable a);  
    boolean esMenorQue(Relacionable a);  
    boolean esIgualQue(Relacionable a);  
}
```

IMPLEMENTACIÓN

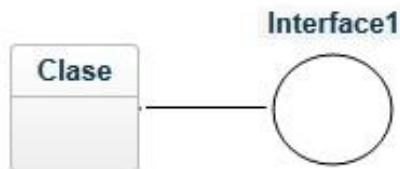
Para indicar que una clase implementa una interface se utiliza la palabra clave **implements**.

```
public class UnaClase implements Interface1{  
    .....  
}
```

En **UML** una clase que implementa una interface se representa mediante una flecha con línea discontinua apuntando a la interface:

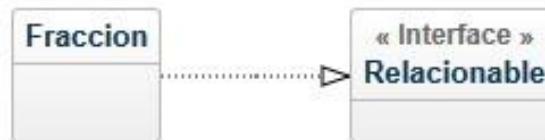


O también se puede representar de forma abreviada:



Las clases que implementan una interfaz deben **implementar todos los métodos abstractos**. De lo contrario serán clases abstractas y deberán declararse como tal.

Ejemplo: La clase Fraccion implementa la interfaz Relacionable. En ese caso se dice que la clase **Fraccion es Relacionable**



```
public class Fraccion implements Relacionable {  
  
    private int num;  
    private int den;  
  
    public Fraccion() {  
        this.num = 0;  
        this.den = 1;  
    }  
  
    public Fraccion(int num, int den) {  
        this.num = num;  
        this.den = den;  
        simplificar();  
    }  
  
    public Fraccion(int num) {  
        this.num = num;  
        this.den = 1;  
    }  
  
    public void setDen(int den) {  
        this.den = den;  
        this.simplificar();  
    }  
  
    public void setNum(int num) {  
        this.num = num;  
        this.simplificar();  
    }  
  
    public int getDen() {  
        return den;  
    }  
  
    public int getNum() {  
        return num;  
    }  
  
    //sumar fracciones  
    public Fraccion sumar(Fraccion f) {  
        Fraccion aux = new Fraccion();  
        aux.num = num * f.den + den * f.num;  
        aux.den = den * f.den;  
        aux.simplificar();  
        return aux;  
    }  
  
    //restar fracciones  
    public Fraccion restar(Fraccion f) {  
        Fraccion aux = new Fraccion();  
        aux.num = num * f.den - den * f.num;  
        aux.den = den * f.den;  
        aux.simplificar();  
        return aux;  
    }  
}
```

```

//multiplicar fracciones
public Fraccion multiplicar(Fraccion f) {
    Fraccion aux = new Fraccion();
    aux.num = num * f.num;
    aux.den = den * f.den;
    aux.simplificar();
    return aux;
}

//dividir fracciones
public Fraccion dividir(Fraccion f) {
    Fraccion aux = new Fraccion();
    aux.num = num * f.den;
    aux.den = den * f.num;
    aux.simplificar();
    return aux;
}

//Cálculo del máximo común divisor por el algoritmo de Euclides
private int mcd() {
    int u = Math.abs(num); //valor absoluto del numerador
    int v = Math.abs(den); //valor absoluto del denominador
    if (v == 0) {
        return u;
    }
    int r;
    while (v != 0) {
        r = u % v;
        u = v;
        v = r;
    }
    return u;
}

private void simplificar() {
    int n = mcd(); //se calcula el mcd de la fracción
    num = num / n;
    den = den / n;
}

@Override
public String toString() {
    //Sobreescritura del método toString heredado de Object
    simplificar();
    return num + "/" + den;
}

//Implementación del método abstracto de la interface
@Override
public boolean esMayorQue(Relacionable a) {
    if (a == null) {
        return false;
    }
    if (!(a instanceof Fraccion)) {
        return false;
    }
    Fraccion f = (Fraccion) a;
    this.simplificar();
}

```

```
f.simplificar();
if ((num / (double) den) <= (f.num / (double) f.den)) {
    return false;
}
return true;
}

//Implementación del método abstracto de la interface
@Override
public boolean esMenorQue(Relacionable a) {
    if (a == null) {
        return false;
    }
    if (!(a instanceof Fraccion)) {
        return false;
    }
    Fraccion f = (Fraccion) a;
    this.simplificar();
    f.simplificar();
    if ((num / (double) den) >= (f.num / (double) f.den)) {
        return false;
    }
    return true;
}

//Implementación del método abstracto de la interface
@Override
public boolean esIgualQue(Relacionable a) {
    if (a == null) {
        return false;
    }
    if (!(a instanceof Fraccion)) {
        return false;
    }
    Fraccion f = (Fraccion) a;
    this.simplificar();
    f.simplificar();
    if (num != f.num) {
        return false;
    }
    if (den != f.den) {
        return false;
    }
    return true;
}
}
```

Ejemplo de utilización de la clase Fraccion:

```
public static void main(String[] args) {
    //Creamos dos fracciones y mostramos cuál es la mayor y
    //cuál menor.
    Fraccion f1 = new Fraccion(3, 5);
    Fraccion f2 = new Fraccion(2, 8);

    if (f1.esMayorQue(f2)) {
        System.out.println(f1 + " > " + f2);
    } else {
        System.out.println(f1 + " <= " + f2);
    }

    //Creamos un ArrayList de fracciones y las mostramos
    //ordenadas de menor a mayor
    ArrayList<Fraccion> fracciones = new ArrayList();

    fracciones.add(new Fraccion(10, 7));
    fracciones.add(new Fraccion(-2, 3));
    fracciones.add(new Fraccion(1, 9));
    fracciones.add(new Fraccion(6, 25));
    fracciones.add(new Fraccion(3, 8));
    fracciones.add(new Fraccion(8, 3));

    Collections.sort(fracciones, new Comparator<Fraccion>() {

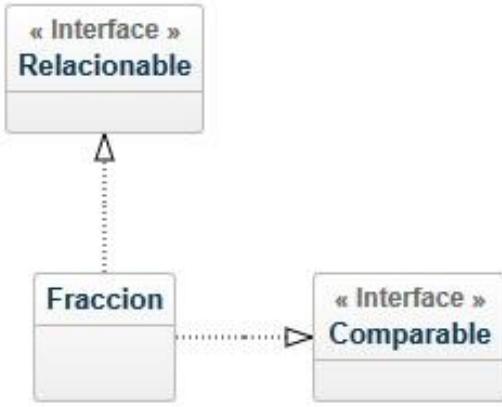
        @Override
        public int compare(Fraccion o1, Fraccion o2) {
            if(o1.esMayorQue(o2)){
                return 1;
            }else if(o1.esMenorQue(o2)){
                return -1;
            }else{
                return 0;
            }
        }
    });

    System.out.println("Fracciones ordenadas de menor a mayor");
    for(Fraccion f: fracciones){
        System.out.print(f + " ");
    }
}
```

Una clase puede implementar más de una interface. Los nombres de las interfaces se escriben a continuación de *implements* y separadas por comas:

```
public class UnaClase implements Interface1, Interface2, Interface3{
.....
}
```

En el ejemplo, para ordenar las fracciones hemos utilizado un Comparator como parámetro de Collections.sort. También podríamos ordenarlas haciendo que la clase Fraccion implemente además la interfaz Comparable:



```

public class Fraccion implements Relacionable, Comparable<Fraccion>
{
    .....
    //Código de la clase Fraccion
    .....

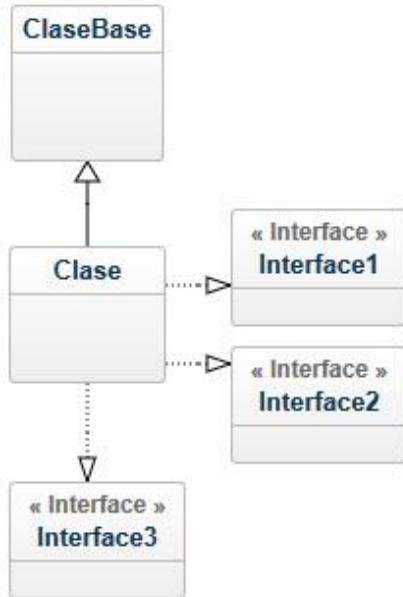
    //Añadimos a la clase el método compareTo
    @Override
    public int compareTo(Fraccion o) {
        if(this.esMenorQue(o)){
            return -1;
        }else if(this.esMayorQue(o)){
            return 1;
        }else{
            return 0;
        }
    }
} //fin de la clase Fraccion
  
```

De este modo para ordenar escribiríamos: `Collections.sort(fracciones)`
 Una clase solo puede tener una clase base pero puede implementar múltiples interfaces. El lenguaje Java no permite herencia múltiple entre clases, pero las interfaces proporcionan una alternativa para implementar algo parecido a la herencia múltiple de otros lenguajes.

Una clase que además de implementar interfaces herede de otra se declarará de esta forma:

```

public
class UnaClase extends ClaseBase implements Interface1, Interface2,
Interface3{
.....
}
  
```



Una interfaz la puede implementar cualquier clase. Por ejemplo, podemos tener una clase Linea que también implementa la interfaz Relacionable:



```

public class Linea implements Relacionable {

    private double x1;
    private double y1;
    private double x2;
    private double y2;

    public Linea(double x1, double y1, double x2, double y2) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }

    public double longitud() {
        double l = Math.sqrt((x2 - x1) * (x2 - x1) +
                             (y2 - y1) * (y2 - y1));
        return l;
    }

    //Implementación del método abstracto de la interface
    @Override
    public boolean esMayorQue(Relacionable a) {
        if (a == null) {
            return false;
        }
        if (!(a instanceof Linea)) {
            return false;
        }
        Linea linea = (Linea) a;
        if (linea.longitud() < this.longitud())
            return true;
        else
            return false;
    }
}
  
```

```

        return this.longitud() > linea.longitud();
    }

//Implementación del método abstracto de la interface
@Override
public boolean esMenorQue(Relacionable a) {
    if (a == null) {
        return false;
    }
    if (!(a instanceof Linea)) {
        return false;
    }
    Linea linea = (Linea) a;
    return this.longitud() < linea.longitud();
}

//Implementación del método abstracto de la interface
@Override
public boolean esIgualQue(Relacionable a) {
    if (a == null) {
        return false;
    }
    if (!(a instanceof Linea)) {
        return false;
    }
    Linea linea = (Linea) a;
    return this.longitud() == linea.longitud();
}

//Sobreescritura del método toString heredado de Object
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append("Coordenadas inicio linea: ");
    sb.append(x1);
    sb.append(", ");
    sb.append(y1);
    sb.append("\nCoordenadas final linea: ");
    sb.append(x2);
    sb.append(", ");
    sb.append(y2);
    sb.append("\nLongitud: ");
    sb.append(longitud());
    return sb.toString();
}
}

```

Podemos hacer un ejemplo de utilización de la clase Linea similar al que hemos hecho para la clase Fracción:

```

public static void main(String[] args) {
    Linea l1 = new Linea(2, 2, 4, 1);
    Linea l2 = new Linea(5, 2, 10, 8);
    if (l1.esMayorQue(l2)) {
        System.out.println(l1 + "\nes mayor que" + l2);
    } else {
        System.out.println(l1 + "\nes menor o igual que" + l2);
    }
}

```

```

ArrayList<Linea> lineas = new ArrayList();

lineas.add(new Linea(0, 7, 1, 4));
lineas.add(new Linea(2, -1, 3, 5));
lineas.add(new Linea(1, 9, 0, -3));
lineas.add(new Linea(15, 3, 9, 5));

Collections.sort(lineas, new Comparator<Linea>() {

    @Override
    public int compare(Linea o1, Linea o2) {
        if(o1.esMayorQue(o2)) {
            return 1;
        }else if(o1.esMenorQue(o2)) {
            return -1;
        }else{
            return 0;
        }
    }
});

System.out.println("\nLineas ordenadas por longitud de
                    menor a mayor");
for (Linea l : lineas) {
    System.out.println(l);
}
}

```

INTERFACES Y POLIMORFISMO

La definición de una interface implica una definición de un nuevo tipo de referencia y por ello **se puede usar el nombre de la interface como nombre de tipo.**

El nombre de una interfaz se puede utilizar en cualquier lugar donde pueda aparecer el nombre de un tipo de datos.

Si se define una variable cuyo tipo es una interface, se le puede asignar un objeto instancia de una clase que implementa la interface.

Volviendo al ejemplo, las clases Linea y Fraccion implementan la interfaz Relacionable. Podemos escribir las instrucciones:

```

Relacionable r1 = new Linea(2,2,4,1);
Relacionable r2 = new Fraccion(4,7);

```

En Java, utilizando interfaces como tipo se puede aplicar el polimorfismo para clases que no están relacionadas por herencia.

Por ejemplo, podemos escribir:

```

System.out.println(r1); //ejecuta toString de Linea
System.out.println(r2); //ejecuta toString de Fraccion

```

También podemos crear un array de tipo Relacionable y guardar objetos de clases que implementan la interfaz.

```
ArrayList<Relacionable> array = new ArrayList();
array.add(new Linea(15, 3, 9, 5));
array.add(new Fraccion(10, 7));
array.add(new Fraccion(6, 25));
array.add(new Linea(3, 4, 10, 15));
array.add(new Fraccion(8, 3));
array.add(new Linea(0, 7, 1, 4));
array.add(new Linea(2, -1, 3, 5));
array.add(new Fraccion(1, 9));
array.add(new Linea(1, 9, 0, -3));
array.add(new Fraccion(3, 8));
array.add(new Fraccion(-2, 3));

for (Relacionable r : array) {
    System.out.println(r);
}
```

En este caso dos clases no relacionadas, *Linea* y *Fraccion*, por implementar la misma interfaz *Relacionable* podemos manejarlas a través de referencias a la interfaz y aplicar polimorfismo.

MÉTODOS DEFAULT

A partir de Java 8 las interfaces además de métodos abstractos pueden contener métodos por defecto o métodos default.

En la interfaz se escribe el código del método. Este método estará disponible para todas las clases que la implementen, no estando obligadas a escribir su código. Solo lo incluirán en el caso de querer modificarlo.

De este modo, si se modifica una interfaz añadiéndole una nueva funcionalidad, se evita tener que modificar el código en todas las clases que la implementan.

Ejemplo: Vamos a añadir un nuevo método a la interfaz Relacionable que devuelva el nombre de la clase (String) del objeto que la está utilizando. Si lo añadimos como abstracto tendremos que modificar las clases Linea y Fraccion y añadir en cada una el nuevo método. En lugar de esto vamos a crear el método como default y de este modo las clases Linea y Fraccion lo pueden usar sin necesidad de escribirlo.

```
public interface Relacionable {

    boolean esMayorQue(Relacionable a);
    boolean esMenorQue(Relacionable a);
    boolean esIgualQue(Relacionable a);

    default String nombreClase(){ //método por defecto
        String clase = getClass().toString();
        int posicion = clase.lastIndexOf(".");
        return clase.substring(posicion+1);
    }
}
```

Ejemplo de uso:

```
ArrayList<Relacionable> array = new ArrayList();
array.add(new Linea(15, 3, 9, 5));
array.add(new Fraccion(10, 7));
array.add(new Fraccion(6, 25));
array.add(new Linea(3, 4, 10, 15));
array.add(new Fraccion(8, 3));
array.add(new Linea(0, 7, 1, 4));
array.add(new Linea(2, -1, 3, 5));
array.add(new Fraccion(1, 9));
array.add(new Linea(1, 9, 0, -3));
array.add(new Fraccion(3, 8));
array.add(new Fraccion(-2, 3));

for (Relacionable r : array) {
    System.out.println(r.nombreClase());
//usamos el método por defecto
    System.out.println(r);
    System.out.println();
}
```

Puede darse el caso de una clase que implemente varias interfaces y las interfaces contengan métodos default iguales (mismo nombre y lista de parámetros).

Ejemplo: La interface *Modificable* contiene dos métodos abstractos y un método default *nombreClase* que es igual que el contenido en la interface Relacionable:

```
public interface Modificable{
    void aumentar(int n);
    void disminuir(int n);

    default String nombreClase() {
        String clase = getClass().toString();
        int posicion = clase.lastIndexOf(".");
        return clase.substring(posicion+1);
    }
}
```

Supongamos que la clase Linea implementa las dos interfaces. En la clase Linea se debe escribir el código de los dos métodos abstractos, pero el método default provoca un error de compilación:

```
public class Linea implements Relacionable, Modificable{
    class Linea inherits unrelated defaults for nombreClase() from types Relacionable and Modifiable
    ----
    (Alt-Enter shows hints)
    private double x2;
    private double ...
```

La clase Linea *hereda* dos métodos default iguales. Esto provoca un *conflicto* que tenemos que resolver. Lo podemos hacer de dos formas. La primera forma es redefinir el método nombreClase() en la clase Linea:

```
public class Linea implements Relacionable, Modificable{
    .....
    //Código de la clase Linea
    .....
    @Override
    public void aumentar(int n) {
        this.x1+=n;
        this.y1+=n;
        this.x2+=n;
        this.y2+=n;
    }

    @Override
    public void disminuir(int n) {
        this.x1-=n;
        this.y1-=n;
        this.x2-=n;
        this.y2-=n;
    }

    //Redefinición del método default
    @Override
    public String nombreClase(){
        String clase = getClass().toString();
        int posicion = clase.lastIndexOf(".");
        return clase.substring(posicion+1);
    }

} //Final de la clase Linea
```

En este caso no se ha modificado el código del método aunque podríamos haberlo hecho. Nos hemos limitado a copiar y pegar el código cambiando el modificador *default* por *public*.

Para estos casos en los que no es necesario cambiar el método podemos usar la segunda forma de resolver el conflicto: indicar cuál de los dos métodos se tiene que ejecutar.

```
public class Linea implements Relacionable, Modificable{
    .....
    //Código de la clase Linea
    .....
    //Redefinición del método default
    //Se ejecutará el método de la interface Relacionable
    @Override
    public String nombreClase(){
        return Relacionable.super.nombreClase();
    }
}
```

La forma de indicar el método de la interface que se ejecuta es:

nombreInterface.super.metodo

MÉTODOS STATIC

A partir de Java 8 las interfaces también pueden contener métodos static. En la interfaz se escribe el código del método. Los métodos static NO pueden ser redefinidos en las clases que la implementan.

Para utilizarlos se escribe:

nombreInterface.metodoStatic

Ejemplo: Añadimos a la interfaz Relacionable un método estático *esNull*. El método comprueba si la variable *a* contiene o no la dirección de un objeto.

```
interface Relacionable {  
  
    boolean esMayorQue(Relacionable a);  
    boolean esMenorQue(Relacionable a);  
    boolean esIgualQue(Relacionable a);  
  
    default String nombreClase(){  
        String clase = getClass().toString();  
        int posicion = clase.lastIndexOf(".");  
        return clase.substring(posicion+1);  
    }  
  
    static boolean esNull(Relacionable a) {  
        return a == null;  
    }  
}
```

Ejemplo de uso de un método static en una interfaz Java: supongamos que disponemos de un Array en el que algunos elementos pueden ser null, es decir, no se les ha asignado un objeto. Utilizaremos el método *esNull* para comprobarlo y evitar que se produzca un error al intentar acceder a uno de estos elementos.

```
Relacionable[] array = new Relacionable[20];  
  
array[1] = new Linea(15, 3, 9, 5);  
array[5] = new Fraccion(10, 7);  
array[9] = new Fraccion(6, 25);  
array[11] = new Linea(3, 4, 10, 15);  
array[14] = new Fraccion(8, 3);  
array[15] = new Linea(0, 7, 1, 4);  
array[18] = new Linea(2, -1, 3, 5);  
  
for (Relacionable r : array) {  
    if (!Relacionable.esNull(r)) {  
        //usamos el método static  
        System.out.println(r.nombreClase());  
        System.out.println(r);  
        System.out.println();  
    }  
}
```

ATRIBUTOS CONSTANTES

Una interfaz también puede contener atributos constantes.

Ejemplo:

```
//Interfaz que contiene días relevantes en una aplicación de banca.
public interface IDiasOperaciones {
    int DIA_PAGO_INTERESES = 5;
    int DIA_COBRO_HIPOTECA = 30;
    int DIA_COBRO_TARJETA = 28;
}
```

Una clase que la implemente puede usar las constantes como si fuesen propias:

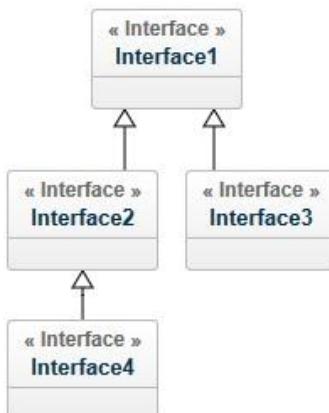
```
public class Banco implements IDiasOperaciones{
    .....
    public void mostrarInformacionIntereses(){
        System.out.println("El día " + DIA_PAGO_INTERESES
                           + " de cada mes se realiza el pago de
                           intereses");
    }
    .....
}
```

Una clase que no implemente la interfaz puede usar las constantes escribiendo antes el nombre de la interfaz.

```
public static void main(String[] args) {
    .....
    System.out.println("Los intereses se pagan el día "
                       +IDiasOperaciones.DIA_PAGO_INTERESES);
    System.out.println("La hipoteca se paga el día "
                       +IDiasOperaciones.DIA_COBRO_HIPOTECA);
    .....
}
```

HERENCIA ENTRE INTERFACES

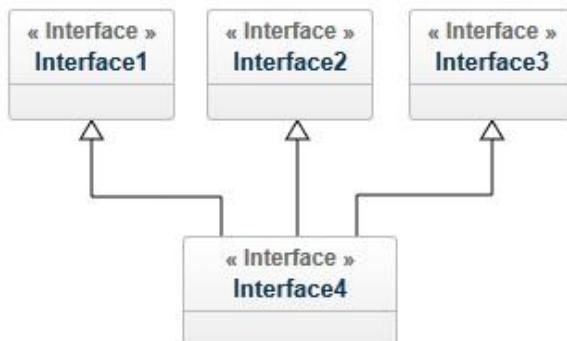
Se puede establecer una jerarquía de herencia entre interfaces igual que con las clases.



```
public interface Interface2 extends Interface1{\n....\n}\npublic interface Interface3 extends Interface1{\n....\n}\npublic interface Interface4 extends Interface2{\n....\n}
```

Cada interface hereda el contenido de las interfaces que están por encima de ella en la jerarquía y puede añadir nuevo contenido o modificar lo que ha heredado siempre que sea posible. Los métodos static no se pueden redefinir. Los métodos abstract heredados se pueden convertir en métodos default. Los métodos default se pueden redefinir o convertir en abstract.

En las interfaces sí se permite herencia múltiple:



```
public interface Interface4 extends Interface1, Interface2, Interface3{\n....\n}
```

'Interface4' hereda todo el contenido de las tres interfaces.

Extraído de: <https://www.instintoprogramador.com.mx/2018/12/interfaces-en-java-ejemplos-de.html>

8. Ejercicio interfaz (para 11-03-2021)

Construye una interfaz llamada **Relacionable** que incluya los siguientes métodos:

```
// Devuelve verdadero si a es mayor que b  
boolean esMayor(Object b) ;  
// Devuelve verdadero si a es menor que b  
boolean esMenor(Object b) ;  
// Devuelve verdadero si a es igual que b  
boolean esIgual(Object b) ;
```

Crea 3 clases:

Fracción

Línea //comparas longitud

Rectángulo //comparas el área

que implementen el interfaz Relacionable.

Crea un programa que instancie cada una de las clases y aplique sus métodos para probarlos.

$$((x_1 - x_2)^2 + (y_1 - y_2)^2)^{0.5}$$

9. Enumerados 1: Enumerados en Java

1. **Enum en Java**
2. **Cómo funcionan los Enum**
3. **Declaración de enum en Java**
4. **Ejemplo con enum en Java**
 - 4.1. **Código de Ejemplo**
 5. **Puntos importantes de enum**
 6. **Métodos values(), ordinal() y valueOf()**
 7. **enum: Constructores, métodos, variables de instancia**
 - 7.1. **Explicación del Ejemplo**
 8. **Enum y Herencia**
 - 8.1. **Uso de ordinal() y compareTo()**
 9. **enum y Métodos**

9.1 Enum en Java

¿Qué es un ENUM? En su forma más simple, **una enumeración es una lista de constantes con nombre que definen un nuevo tipo de datos**. Un objeto de un tipo de enumeración solo puede contener los valores definidos por la lista. Por lo tanto, una enumeración proporciona una manera de definir con precisión un nuevo tipo de datos que tiene **un número fijo de valores válidos**.

Por ejemplo, los 4 palos en un mazo de cartas pueden ser 4 valores llamados espadas, bastos, oros y copas, que pertenecen a un tipo enumerado llamado Baraja. Otros ejemplos incluyen tipos de enumerados naturales (como los planetas, días de la semana, meses del año, colores, direcciones, etc.).

Desde una perspectiva de programación, las enumeraciones son útiles siempre que necesites definir un **conjunto de valores que represente una colección de elementos**. Por ejemplo, se puede usar una enumeración para representar un conjunto de códigos de estado, como **éxito, espera, error y reintentos**, que indican el progreso de alguna acción.

En el pasado, dichos valores se definían como **variables finales**, pero las enumeraciones ofrecen un enfoque más estructurado. Todos los enum implícitamente extienden de la clase java.lang.Enum.

```
public abstract class Enum<E extends Enum<E>>
    extends Object implements Comparable<E>, Serializable
```

9.2 Cómo funcionan los Enum

Los enum en Java se usan cuando conocemos todos los valores posibles en tiempo de compilación.

En Java (desde 1.5), las enumeraciones se representan utilizando el tipo de datos **enum**. Las enumeraciones Java son más potentes que las enumeraciones C/C++. En Java, también podemos agregarle atributos/variables, métodos y constructores. El objetivo principal de enum es definir nuestros propios tipos de datos (**tipos de datos enumerados**).

9.3 Declaración de enum en Java

La declaración de Enum puede hacerse fuera de **una clase**, o dentro de una clase (*class*), pero **NO** dentro de un método.

EJEMPLO

```
// ejemplo donde se declara enum  
// fuera de cualquier clase (Nota la palabra enum en lugar  
// de la palabra class)  
  
enum Color  
{  
    ROJO, VERDE, AZUL;  
}  
  
public class Test  
{  
    // El método  
    public static void main(String[] args)  
    {  
        Color c1 = Color.ROJO;  
        System.out.println(c1);  
    }  
}
```

Salida:

ROJO

Una enumeración se crea usando la palabra clave **enum**.

La primera línea dentro de enum debe ser una lista de constantes y luego otras cosas como métodos, variables y constructores.

De acuerdo con las convenciones de nomenclatura de Java, se recomienda que nombremos las constantes con **mayúsculas**.

9.4 Ejemplo con enum en Java

EJEMPLO

Enumeración simple que enumera varias formas de transporte:

```
//Una enumeración de transporte
enum Transporte{
    COCHE, CAMION, AVION, TREN, BARCO;
}
```

Los identificadores *COCHE*, *CAMION*, etc. se denominan **constantes de enumeración**.

Cada uno se declara **implícitamente** como un miembro público (**public**) y estático (**static**) de *Transporte*. Además, el tipo de las constantes de enumeración es el tipo de enumeración en el que se declaran las constantes, que es *Transporte* en este caso. Por lo tanto, en el lenguaje de Java, estas constantes se llaman **auto-tipado**.

Una vez que hayas definido una enumeración, puedes crear una variable de ese tipo. Sin embargo, aunque las enumeraciones definen un tipo de clase, no creamos una instancia de una enumeración usando *new*, declaramos variables de su tipo. Por ejemplo, así declaramos *tp* como una variable del tipo de enumeración *Transporte*:

```
Transporte tp;
```

Como *tp* es de tipo *Transporte*, los únicos valores que se le pueden asignar son los definidos por la enumeración. Por ejemplo, esto asigna *tp* el valor *AVION*:

```
tp = Transporte.AVION;
```

Se pueden comparar dos constantes de enumeración utilizando el operador relacional **==**. Por ejemplo, esta declaración compara el valor en *tp* con la constante *TREN*:

```
If (tp == Transporte.TREN) // ...
```

En nuestro caso no habría igualdad.

Un valor de enumeración también se puede usar para controlar una sentencia **switch**. Por supuesto, todas las declaraciones de **case** deben usar constantes de la misma enumeración que la utilizada por la expresión de switch. Por ejemplo, este *switch* es perfectamente válido:

```
//Uso de enum para controlar una sentencia switch
switch(tp) {
    case COCHE:
        //
    case CAMION:
        //
}
```

Observa que en las sentencias *case*, los nombres de las constantes de enumeración se usan sin estar calificados por el nombre de tipo de enumeración. Es decir, se utiliza *CAMION*, no *Transporte.CAMION*. Esto se debe a que el tipo de enumeración en la expresión de *switch* ya ha especificado implícitamente el tipo de enumeración de las constantes de *case*.

No es necesario calificar las constantes en las declaraciones de **case** con su nombre de tipo enum. De hecho, intentar hacerlo provocará un error de compilación.

9.4.1 Código de Ejemplo

Cuando se muestra una constante de enumeración, como en una instrucción *println()*, se genera su nombre. Por ejemplo, dada esta declaración:

```
System.out.println(Transporte.BARCO);  
//Se muestra el nombre BARCO.
```

Prueba el siguiente programa que usa la enumeración *Transporte*:

```
enum Transporte{  
    COCHE, CAMION, AVION, TREN, BARCO;  
}  
  
class Enumerados {  
    public static void main(String[] args) {  
        Transporte tp;  
        tp=Transporte.AVION;  
  
        System.out.println("Valor de tp: "+tp);  
        System.out.println();  
  
        tp=Transporte.TREN;  
  
        //Comparación de 2 valores enum  
        if (tp==Transporte.TREN)  
            System.out.println("tp tiene el valor de TREN\n");  
  
        //enum para controlar sentencia switch  
        switch(tp){  
            case COCHE:  
                System.out.println("Un auto lleva personas.");  
                break;  
            case CAMION:  
                System.out.println("Un camión lleva carga.");  
                break;  
            case AVION:  
                System.out.println("Un avión vuela.");  
                break;  
            case TREN:  
                System.out.println("Un tren corre sobre rieles.");  
                break;  
            case BARCO:  
                System.out.println("Un barco navega en el agua.");  
                break;  
        }  
    }  
}
```

Salida:

Valor de tp: AVION
tp tiene el valor de TREN
Un tren corre sobre rieles.

Las constantes en *Transporte* las pondremos en mayúsculas. (Por lo tanto, se usa *COCHE*, *no coche*.) Sin embargo, **no es obligatorio el uso de mayúsculas**. Debido a que las enumeraciones a menudo reemplazan las variables finales, que tradicionalmente se han usado en mayúsculas, se ha llegado al acuerdo de que las **constantes de enumeración las escribamos en mayúsculas**.

9.5 Puntos importantes de enum

Esta explicación es útil para que entendáis bien los enumerados:

- Cada enum es implementado **internamente** mediante el uso de *class*.

Internamente enum Color que definimos antes se convierte en:

```
class Color {  
    public static final Color ROJO = new Color();  
    public static final Color AZUL = new Color();  
    public static final Color VERDE = new Color();  
}
```

- Cada constante enum representa un objeto de tipo enum.
- El tipo enum se puede pasar como un argumento para *switch*.

Vemos un ejemplo más:

```
// Un programa Java para demostrar el trabajo de enum
// en case de switch (Archivo Test.Java)
import java.util.Scanner;

// Una clase enum
enum Dia
{
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO;
}

// Controlador de clase que contiene un objeto de "Dia" y
// main().
public class Test
{
    Dia dia;

    // Constructor
    public Test(Dia dia)
    {
        this.dia = dia;
    }

    // Imprime una línea sobre el DIA usando switch
    public void diaEs()
    {
        switch (dia)
        {
            case LUNES:
                System.out.println("Los lunes son feos.");
                break;
            case VIERNES:
                System.out.println("Los viernes son mejores.");
                break;
            case SABADO:
            case DOMINGO:
                System.out.println("Los fines de semana son mejores.");
                break;
            default:
                System.out.println("Los días entre semana son
regulares.");
                break;
        }
    }

    // Metodo
    public static void main(String[] args)
    {
        String str = "LUNES";
        Test t1 = new Test(Dia.valueOf(str));
        t1.diaEs();
    }
}
```

Salida:

Los lunes son feos.

- Cada constante enum siempre es implícitamente public static final. Entonces, como es static, podemos acceder utilizando el nombre del *enum*. Y como es final, no podemos crear enumeraciones “*hijas*”.
- Podemos declarar el método main() dentro de enum.

Por ejemplo:

```
// Un programa Java para demostrar que podemos tener
// main() dentro de enum
enum Color
{
    ROJO, VERDE, AZUL;

    // Método
    public static void main(String[] args)
    {
        Color c1 = Color.ROJO;
        System.out.println(c1);
    }
}
```

El hecho de que enum define una clase permite que la enumeración de Java tenga poderes que las enumeraciones en otros lenguajes no tienen. Por ejemplo, pueden **crearse constructores, agregar variables y métodos de instancia**, e incluso **implementar interfaces**.

A diferencia de la forma en que se implementan las enumeraciones en algunos otros lenguajes, *Java implementa enumeraciones como tipos de clases*. Aunque no creemos una instancia de una enumeración usando new, actúa de forma muy similar a otras clases.

9.6 Métodos values(), ordinal() y valueOf()

Todas las enumeraciones tienen automáticamente dos métodos predefinidos: **values()** y **valueOf()**. Sus formas generales son:

```
public static tipo-enum[ ] values( )
public static tipo-enum valueOf(String str)
```

- Estos métodos están presentes dentro de **java.lang.Enum**.
- El método **values()** se puede usar para devolver todos los valores presentes dentro de enum.
- El **orden** es importante en las enumeraciones. Al usar el método **ordinal()**, se puede encontrar cada índice de la constante enum, al igual que el índice de matriz.
- El método **valueOf()** devuelve la constante enum del valor de cadena especificado, si existe.

El compilador agrega automáticamente el método **values** cuando se crea un enum. Este método devuelve un array conteniendo todos los valores del enumerado en el orden en que son declarados y se usa comúnmente en combinación con el ciclo for-each para iterar sobre los valores de un tipo enum.

Ejemplo de código. Compílalo y comprueba el resultado de ejecución:

```
// Programa Java para demostrar el funcionamiento de values(),
// ordinal() y valueOf()
enum Color
{
    ROJO, VERDE, AZUL;
}

public class Test
{
    public static void main(String[] args)
    {
        // Llamando a values()
        Color arr[] = Color.values();

        // enum con bucle
        for (Color col: arr)
        {
            // Llamando a ordinal() para encontrar el índice
            // de color.
            System.out.println(col + " en el índice "
                + col.ordinal());
        }

        // Usando valueOf(). Devuelve un objeto de
        // Color con la constante dada.
        // La segunda línea comentada causa la excepción
        // IllegalArgumentException
        System.out.println(Color.valueOf("ROJO"));
        // System.out.println(Color.valueOf("BLANCO"));
    }
}
```

Salida:

```
ROJO en el índice 0  
VERDE en el índice 1  
AZUL en el índice 2  
ROJO
```

9.7 enum: Constructores, métodos, variables de instancia

Es importante comprender que cada constante de enumeración es un objeto de su tipo de enumeración (como si fuera un objeto de una clase). Por lo tanto, **una enumeración puede definir constructores, agregar métodos y tener variables de instancia.**

La primera vez que llamamos a una instancia del enum se **inicializan todas** las instancias del mismo, se ejecutan los constructores para cada instancia respetando el orden definido. **Cada constante** de enumeración puede llamar a **cualquier método** definido por la enumeración. **Cada constante de enumeración tiene su propia copia de cualquier variable de instancia definida por la enumeración.**

- enum puede contener un constructor y se ejecuta por separado para cada constante enum en el momento en que se utiliza la primera.
- No podemos crear objetos enum explícitamente y, por lo tanto, no podemos invocar el constructor enum directamente.

Ejemplo:

```
enum Color{  
  
    // Enums  
    ROJO("Rojo", 3), AZUL("Azul", 5);  
  
    // Atributos  
    private String nombreColor;  
    private int numColor;  
  
    public int contador=0;  
  
    // Constructor  
    // Rojo - 3 - 1  
    // Azul - 5 - 1  
  
    Color(String nom, int col) {  
  
        this.nombreColor = nom;  
        this.numColor = col;  
        this.contador++;  
  
        System.out.println(this.nombreColor + " - "  
                           +this.numColor + " - " + this.contador)  
    }  
}
```

```

public class C1{

    public static void main(String[] args){

        //System.out.println("Inicializando: " + Color.ROJO);

        // Se inicializan todas las instancias la primera vez que se
        // usa el enum.

        Color c1;
        c1 = Color.AZUL;
        System.out.println("Inicializando: " + c1);

        Color c2;
        c2 = Color.ROJO;
        System.out.println("Inicializando: " + c2);
    }
}

```

La siguiente versión de *Transporte* ilustra el uso de un constructor, una variable de instancia y un método. Da a cada tipo de transporte una velocidad típica:

```

//Uso de un constructor, una variable de instancia y un método.
enum Transporte{

    COCHE(60), CAMION(50), AVION(600), TREN(70), BARCO(20);

    private int velocidad; //velocidad típica de cada transporte

    //Añadir un constructor
    Transporte(int s){velocidad=s;}
    //Añadir un método
    int getVelocidad(){return velocidad;}
}

class Enumerados {
    public static void main(String[] args) {
        Transporte tp;

        //Mostrar la velocidad de un avión
        System.out.println("La velocidad típica para un avión es: "+
        Transporte.AVION.getVelocidad()+" millas por hora.\n");

        //Mostrar todas las velocidades y transportes
        System.out.println("Todas las velocidades de transporte: ");

        for (Transporte t:Transporte.values())
            System.out.println(t+": velocidad típica es "+
            +t.getVelocidad()+" millas por hora.");
    }
}

```

Salida:

La velocidad típica para un avión es: 600 millas por hora.

Todas las velocidades de transporte:

COCHE: velocidad típica es 60 millas por hora.
 CAMION: velocidad típica es 50 millas por hora.
 AVION: velocidad típica es 600 millas por hora.
 TREN: velocidad típica es 70 millas por hora.
 BARCO: velocidad típica es 20 millas por hora.

9.7.1 Explicación del Ejemplo

Esta versión de *Transporte* agrega tres cosas:

La **primera** es la variable de instancia **velocidad**, que se usa para mantener la velocidad de cada tipo de transporte.

La **segunda** es el constructor **Transporte**, que pasa la *velocidad* de un transporte.

La tercera es el método **getVelocidad()**, que devuelve el valor de velocidad. Solo admite geters, que es lo que tiene sentido.

Cuando una variable de tipo *Transporte* (variable **tp** en el ejemplo) se declara en **main() y se inicializa a un valor de la enumeración**, el constructor de *Transporte* se llama una vez para cada constante que se especifica. Observa cómo se especifican los argumentos para el constructor, poniéndolos entre paréntesis, después de cada constante, como se muestra aquí:

```
COCHE(60), CAMION(50), AVION(600), TREN(70), BARCO(20);
```

Estos valores se pasan al parámetro de *Transporte()*, que luego asigna este valor a la *velocidad*. Hay algo más que notar sobre la lista de constantes de enumeración: **termina con un punto y coma**. Es decir, la última constante, *BARCO*, va seguida de un punto y coma. Cuando una enumeración contiene otros miembros, la lista de enumeración debe terminar en punto y coma.

Como cada **constante de enumeración tiene su propia copia de velocidad**, puede obtener la velocidad de un tipo de transporte especificado llamando a *getVelocidad()*. Por ejemplo, en *main()* la velocidad de un avión se obtiene mediante la siguiente llamada:

```
Transporte.AVION.getVelocidad()
```

La velocidad de cada transporte se obtiene al recorrer la enumeración mediante un [bucle for](#). Como hay una copia de velocidad para cada constante de enumeración, el valor asociado con una constante es separado y distinto del valor asociado con otra constante. Este es un concepto poderoso, que está disponible solo cuando las enumeraciones se implementan como clases, como lo hace Java.

Aunque el ejemplo anterior contiene solo un constructor, una enumeración puede ofrecer dos o más formas de sobrecargas, al igual que cualquier otra clase.

9.8 Enum y Herencia

Hay dos **restricciones** que se aplican a las enumeraciones.

- Una enumeración no puede heredar de otra clase (ya hereda implícitamente de `java.lang.Enum`).
- Una enumeración no puede ser una superclase (no se puede heredar de ella).

Esto significa que una enumeración no se puede extender. De lo contrario, enum actúa como cualquier otro tipo de clase. **La clave es recordar que cada una de las constantes de enumeración es un objeto de la clase en la que está definida.**

- Aunque no puede heredar de una superclase al declarar una enumeración, **todas las enumeraciones heredan automáticamente una: `java.lang.Enum`.** Esta clase define varios métodos que están disponibles para el uso de todas las enumeraciones.
- Hay dos métodos heredados que podemos usar: **`ordinal()` y `compareTo()`.**
- El método **`toString()`** devuelve el nombre de la constante enum.
- **enum** puede implementar interfaces.

9.8.1 Uso de `ordinal()` y `compareTo()`

El método `ordinal()` se muestra aquí:

```
final int ordinal()
```

Devuelve el valor *ordinal* de la constante invocadora. Los valores ordinales comienzan en cero. Por lo tanto, en la enumeración *Transporte*, *COCHE* tiene un valor ordinal de cero, *CAMION* tiene un valor ordinal de 1, *AVION* tiene un valor ordinal de 2, y así sucesivamente.

Se puede comparar el valor ordinal de dos constantes de la misma enumeración utilizando el método `compareTo()`. Tiene esta forma general:

```
final int compareTo(tipo-enum e)
```

Aquí, *tipo-enum* es el tipo de enumeración y *e* es la constante que se compara con la constante de invocación. Tanto la constante de invocación como *e* deben ser de la misma enumeración.

- Si la constante de invocación tiene un valor ordinal menor que *e*, entonces `compareTo()` devuelve un valor **negativo**.
- Si los dos valores ordinales son iguales, se devuelve **cero**.
- Si la constante de invocación tiene un valor ordinal mayor que *e*, se devuelve un valor **positivo**.

El siguiente programa muestra **ordinal()** y **compareTo()**:

```
package enumTransportes4;

//Demostración de ordinal() y compareTo()
enum Transporte {
    COCHE, CAMION, AVION, TREN, BARCO;
}

class Enumerados {
    public static void main(String[] args) {
        Transporte tp1, tp2, tp3;

        // Obtiene todos los valores ordinales usando ordinal() .
        System.out.println("Aqui están todas las constantes de
                            Transporte y sus valores ordinales: ");

        // Acuérdate de que Transporte.values() devuelve un array
        // con las constantes de la enumeración

        for (Transporte t: Transporte.values())
            System.out.println(t + " " + t.ordinal());

        tp1 = Transporte.AVION;
        tp2 = Transporte.TREN;
        tp3 = Transporte.AVION;

        System.out.println();

        // Uso de CompareTo()
        if (tp1.compareTo(tp2) < 0)
            System.out.println(tp1 + " esta antes que " + tp2);
        else if (tp1.compareTo(tp2) > 0)
            System.out.println(tp1 + " esta despues que " + tp2);
        else
            System.out.println(tp1 + " es igual que " + tp3);

        tp1 = Transporte.CAMION;
        tp2 = Transporte.COCHE;
        tp3 = Transporte.CAMION;

        // Uso de CompareTo()
        if (tp1.compareTo(tp2) < 0)
            System.out.println(tp1 + " esta antes que " + tp2);

        if (tp1.compareTo(tp2) > 0)
            System.out.println(tp1 + " esta despues que " + tp2);

        if (tp1.compareTo(tp3) == 0)
            System.out.println(tp1 + " es igual que " + tp3);
    }
}
```

Salida:

Aquí están todas las constantes de Transporte y sus valores ordinales:

```
COCHE 0
CAMION 1
AVION 2
TREN 3
BARCO 4
```

AVION está antes que TREN
AVION es igual que AVION

9.9 enum y Métodos

- enum puede contener métodos concretos, es decir, que no tengan ningún método abstracto (**abstract**).

Por ejemplo (fijaos bien en la ejecución):

```
// Programa Java para demostrar que los enum pueden tener
// Constructores y métodos concretos.

// Una enumeración (Note la palabra enum en lugar de class)

enum Color
{
    ROJO, VERDE, AZUL;

    // enum constructor llamado por separado para cada constante

    private Color()
    {
        System.out.println("Constructor llamado para : " + this);
    }

    // Solo métodos concretos (no abstractos) permitidos
    public void colorInfo()
    {
        System.out.println(this + " desde colorInfo()");
    }
}

public class Test
{
    // Método
    public static void main(String[] args)
    {
        Color c1 = Color.ROJO;
        // Míralo en el Proyecto, está separado y se ve más claro
        System.out.println(c1);
        c1.colorInfo();
    }
}
```

Salida:

```
Constructor llamado para: ROJO
Constructor llamado para: VERDE
Constructor llamado para: AZUL
ROJO
ROJO desde colorInfo()
```

10. Enumerados 2: Enumerados en Java

Desde java 5 tenemos la posibilidad de definir enumerados *enum*.

Un enumerado (o Enum) es una clase "especial" (tanto en Java como en otros lenguajes) **que limita la creación de objetos a los especificados explícitamente en la implementación de la clase.** La única restricción que tienen los enumerados respecto a una clase normal es que si tiene constructor, este debe ser privado para que no se puedan crear nuevos objetos.

Ejemplo sobre una clase Enum. Los futbolistas están caracterizados por una demarcación a la hora de jugar un partido de fútbol, por tanto las demarcaciones en las que puede jugar un futbolista son finitas y por tanto se pueden enumerar en: Portero, Defensa, Centrocampista y Delantero. Con esta especificación podemos crearnos la siguiente clase "Enum" llamada "Demarcación":

```
public enum Demarcacion
{
    PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO
}
```

Por convenio los nombres de los enumerados se escriben en mayúsculas.

Es muy importante entender que un "**Enum**" en java **es realmente una clase** (cuyos objetos solo pueden ser los definidos en esta clase: PORTERO..., DELANTERO) **que hereda** de la clase "**Enum(java.lang.Enum)**" y por tanto los enumerados tienen una serie de métodos heredados de esa clase padre. A continuación, vamos a mostrar algunos de los métodos más utilizados de los enumerados:

```
public enum Demarcacion{PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO}
Demarcacion delantero = Demarcacion.DELANTERO;

// Instancia de un enum de la clase Demarcación

delantero.name();
// Devuelve un String con el nombre de la constante (DELANTERO)

delantero.toString();
// Devuelve un String con el nombre de la constante (DELANTERO)

delantero.ordinal();
// Devuelve un entero con la posición del enum según está declarada
(3 en este caso).

delantero.compareTo(Enum otro);
// Compara el enum con el parámetro según el orden en el que están
declarados los enum

Demarcacion.values();
// Devuelve un array que contiene todos los enum
```

Ejemplo para probar:

```
Demarcacion delantero = Demarcacion.DELANTERO;
Demarcacion defensa = Demarcacion.DEFENSA;

// Devuelve un String con el nombre de la constante
System.out.println("delantero.name()= "+delantero.name());
System.out.println("defensa.toString()= "+defensa.toString());

// Devuelve un entero con la posición de la constante según está
// declarada.
System.out.println("delantero.ordinal()= "+delantero.ordinal());

// Compara el enum con el parámetro según el orden en el que están
// declaradas las constantes.
System.out.println("delantero.compareTo(portero)= " +
delantero.compareTo(defensa));
System.out.println("delantero.compareTo(delantero)= " +
delantero.compareTo(delantero));

// Recorre todas las constantes de la enumeración
for(Demarcacion d: Demarcacion.values()){
    System.out.println(d.toString()+" - ");
}
```

Tenemos como salida los siguientes resultados:

```
delantero.name()= DELANTERO
defensa.toString()= DEFENSA
delantero.ordinal()= 3
delantero.compareTo(defensa)= 2
delantero.compareTo(delantero)= 0
PORTERO - DEFENSA - CENTROCAMPISTA - DELANTERO
```

Como ya se ha dicho, un **enum** es una clase especial que limita la creación de objetos a los especificados en su clase (por eso, su constructor es privado, como se ve en el siguiente fragmento de código); pero estos objetos pueden tener atributos como cualquier otra clase. En la siguiente declaración de la clase, vemos un ejemplo en el que definimos un enumerado "Equipo" que va a tener dos atributos; el nombre y el puesto en el que quedaron en la liga del año 2009/2010.

```

public enum Equipo
{
    BARÇA("FC Barcelona",1), REAL_MADRID("Real Madrid",2),
    SEVILLA("Sevilla FC",4), VILLAREAL("Villareal",7);

    private String nombreClub;
    private int puestoLiga;

    private Equipo (String nombreClub, int puestoLiga) {
        this.nombreClub = nombreClub;
        this.puestoLiga = puestoLiga;
    }

    public String getNombreClub() {
        return nombreClub;
    }

    public int getPuestoLiga() {
        return puestoLiga;
    }
}

```

Como se ve BARCA, REAL_MADRID, etc. son nombres del enumerado (u objetos de la clase Equipo) que tendrán como atributos el "nombreClub" y "puestoLiga". Como se ve en la clase **definimos un constructor que es privado** (es decir que solo es visible dentro de la clase Equipo) y solo definimos los métodos "get". Para trabajar con los atributos de estos enumerados se hace de la misma manera que con cualquier otro objeto; se instancia un objeto y se accede a los atributos con los métodos get. En el siguiente fragmento de código vamos a ver cómo trabajar con enumerados que tienen atributos:

```

// Instanciamos el enumerado
Equipo villareal = Equipo.VILLAREAL;

// Devuelve un String con el nombre de la constante
System.out.println("villareal.name()= "+villareal.name());

// Devuelve el contenido de los atributos
System.out.println("villareal.getNombreClub()= "
+villareal.getNombreClub());
System.out.println("villareal.getPuestoLiga()= "
+villareal.getPuestoLiga());

```

Como salida de este fragmento de código tenemos lo siguiente:

```

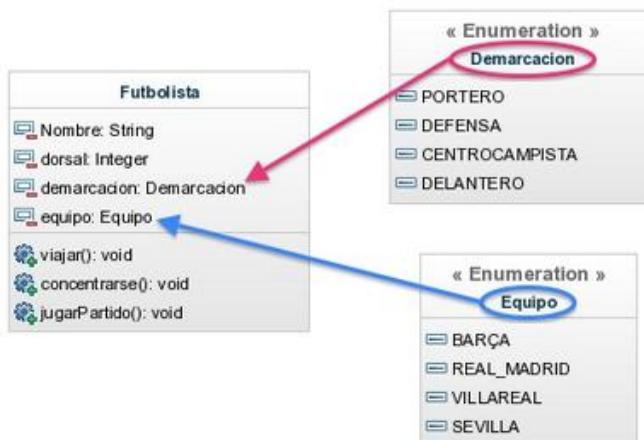
villareal.name()= VILLAREAL
villareal.getNombreClub()= Villareal
villareal.getPuestoLiga()= 7

```

Es muy importante tener claro que los enumerados no son Strings (aunque pueden serlo), sino que son objetos de una clase que solo son instanciables desde la clase que se implementa y que no se puede crear un objeto de esa clase desde cualquier otro lado que no sea dentro de esa clase. Es muy común (sobre todo cuando se está aprendiendo qué son los enumerados) que se interprete que un enumerado es una lista finita de Strings y en realidad es una

lista finita de **objetos** de una determinada clase con sus atributos, constructor y métodos getter aunque estos sean privados.

A continuación, vamos a poner un sencillo ejemplo en el que vamos a mezclar los dos enumerados anteriores (Demarcación y Equipo). En este ejemplo vamos a crear unos objetos de la clase **Futbolista**, que representarán a los jugadores de la selección española de fútbol que ganaron el mundial de fútbol de Sudáfrica en el año 2010. Esta clase va a caracterizar a los futbolistas por su nombre, su dorsal, la demarcación en la que juegan y el club de fútbol al que pertenecen; tal y como vemos en el siguiente diagrama de clases.



Como vemos, los atributos de demarcación y equipo son de la clase **Demarcacion** y **Equipo** respectivamente y son los enumerados vistos anteriormente; por tanto, un futbolista solo podrá pertenecer a uno de los cuatro equipos que forman el enumerado "Equipo" y podrá jugar en alguna de las cuatro demarcaciones que forman el enumerado "Demarcación". A continuación, mostramos la implementación de la clase "Futbolista":

```
package Main;

public class Futbolista {

    private int dorsal;
    private String Nombre;
    private Demarcacion demarcacion;
    private Equipo equipo;

    public Futbolista() {
    }

    public Futbolista(String nombre, int dorsal, Demarcacion
        demarcacion, Equipo equipo) {
        this.dorsal = dorsal;
        Nombre = nombre;
        this.demarcacion = demarcacion;
        this.equipo = equipo;
    }

    // Metodos getter y setter
    .....
}
```

```

@Override
public String toString() {
    return this.dorsal + " - " + this.Nombre + " - "
        + this.demarcacion.name() + " - "
        + this.equipo.getNombreClub();
}
}

```

Dada esta clase podemos crearnos ya objetos de la clase futbolista, como mostramos a continuación:

```

Futbolista casillas = new Futbolista("Casillas", 1, Demarcacion.PORTERO,
Equipo.REAL_MADRID);
Futbolista capdevila = new Futbolista("Capdevila", 11,
Demarcacion.DEFENSA, Equipo.VILLAREAL);
Futbolista iniesta = new Futbolista("Iniesta", 6,
Demarcacion.CENTROCAMPISTA, Equipo.BARCA);
Futbolista navas = new Futbolista("Navas", 22, Demarcacion.DELANTERO,
Equipo.SEVILLA);

```

Como vemos la demarcación y el equipo al que pertenecen solo pueden ser los declarados en la clase enumerado. Si llamamos al método "toString()" declarado en la clase futbolista, podemos imprimir por pantalla los datos de los futbolistas. Dado el siguiente código:

```

System.out.println(casillas.toString());
System.out.println(capdevila.toString());
System.out.println(intiesta.toString());
System.out.println(navas.toString());

```

Y dado el siguiente método "toString()"

```

@Override
public String toString() {
    return this.dorsal + " - " + this.Nombre + " - "
        + this.demarcacion.name() + " - "
        + this.equipo.getNombreClub();
}

```

Tenemos como salida lo siguiente:

```

1 - Casillas - PORTERO - Real Madrid
11 - Capdevila - DEFENSA - Villareal
6 - Iniesta - CENTROCAMPISTA - FC Barcelona
22 - Navas - DELANTERO - Sevilla FC

```

En resumen: Un enumerado está formado por objetos definidos en la misma clase con constructor privado y si tiene atributos, estos solo tienen que tener métodos "getter" para obtener el valor del atributo.

Mirar:

http://www.aprenderaprogramar.es/index.php?option=com_content&view=article&id=647:tipos-enumerados-enum-java-ejemplos-de-codigo-error-enum-types-must-not-be-local-ejercicio-cu00681b&catid=68:curso-aprender-programacion-java-desde-cero&Itemid=188

http://chuwiki.chuidiang.org/index.php?title=Enumerados_en_java

11. Enumerados en Java: Ejemplos a implementar

Ejercicio 1 que nos va a dar un error. Se propone que compruebes y expliques el error y encuentres una solución basada en la excepción que se produce.

```
public class PruebaValueOf {  
  
    enum Animal {  
        PERRO, GATO  
    };  
  
    public static void main(String[] args) {  
  
        Animal animal = Animal.valueOf(args[0]);  
  
        switch (animal) {  
            case PERRO:  
                System.out.println("El perro ladra.");  
                break;  
            case GATO:  
                System.out.println("El gato maulla.");  
                break;  
            default:  
                System.out.println("No es un tipo aceptado.");  
        }  
    }  
}
```

Ejercicio 2 sobre una clase Enum. Los futbolistas están caracterizados por una demarcación a la hora de jugar un partido de fútbol, por tanto las demarcaciones en las que puede jugar un futbolista son finitas y por tanto se pueden enumerar en: Portero, Defensa, Centrocampista y Delantero. Con esta especificación podemos crearnos la siguiente clase "Enum" llamada "Demarcación":

```
public enum Demarcacion
{
    PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO
}
```

Por convenio los nombres de los enumerados se escriben en mayúsculas.

Es muy importante entender que un "**Enum**" en java **es realmente una clase** (cuyos objetos solo pueden ser los definidos en esta clase: PORTERO,..., DELANTERO) **que hereda** de la clase "**Enum (java.lang.Enum)**" y por tanto los enumerados tienen una serie de métodos heredados de esa clase padre. A continuación, vamos a mostrar algunos de los métodos más utilizados de los enumerados:

```
public enum Demarcacion{PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO}

// Instancia de un enum de la clase Demarcación.
Demarcacion delantero = Demarcacion.DELANTERO;

// Devuelve un String con el nombre del enumerado (DELANTERO) .
delantero.name();

// Devuelve un String con el nombre del enumerado (DELANTERO) .
delantero.toString();

// Devuelve un entero con la posición del enum según está declarada (3).
delantero.ordinal();

// Compara el enum que invoca el método con el enum parámetro, según el orden en el que están declarados los enum.
delantero.compareTo(Enum otro);

// Devuelve un array que contiene todos los enum.
Demarcacion.values();

// Devuelve el enumerado que corresponde con la cadena pasada por
// parámetro
System.out.println(Demarcacion.valueOf("DELANTERO"));
```

Ejemplo para probar:

```
Demarcacion delantero = Demarcacion.DELANTERO;
Demarcacion defensa = Demarcacion.DEFENSA;

// Devuelve un String con el nombre de la constante
System.out.println("delantero.name()= " + delantero.name());
System.out.println("defensa.toString()= " + defensa.toString());

// Devuelve un entero con la posición de la
// constante según está declarada.
```

```

System.out.println("delantero.ordinal()= " + delantero.ordinal());
// Compara el enum con el parámetro según el orden en el que están
// declaradas las constantes.

System.out.println("delantero.compareTo(portero)= "
+delantero.compareTo(defensa));
System.out.println("delantero.compareTo(delantero)= "
+delantero.compareTo(delantero));

// Recorre todas las constantes de la enumeración
for(Demarcacion d: Demarcacion.values()){
    System.out.println(d.toString() + " - ");
}

```

Tenemos como salida los siguientes resultados:

```

delantero.name()= DELANTERO
defensa.toString()= DEFENSA
delantero.ordinal()= 3
delantero.compareTo(defensa)= 2
delantero.compareTo(delantero)= 0
PORTERO - DEFENSA - CENTROCAMPISTA - DELANTERO

```

Como ya se ha dicho, un **enum** es una clase especial que **limita la creación de objetos a los especificados en su clase** (por eso, su constructor es privado, como se ve en el siguiente fragmento de código); pero estos objetos pueden tener atributos como cualquier otra clase. En la siguiente declaración de la clase, vemos un ejemplo en el que definimos un enumerado "Equipo" que va a tener dos atributos; el nombre y el puesto en el que quedaron en la liga del año 2009/2010.

```

public enum Equipo
{
    BARÇA("FC Barcelona",1), REAL_MADRID("Real Madrid",2),
    SEVILLA("Sevilla FC",4), VILLAREAL("Villareal",7);

    private String nombreClub;
    private int puestoLiga;

    private Equipo (String nombreClub, int puestoLiga){
        this.nombreClub = nombreClub;
        this.puestoLiga = puestoLiga;
    }

    public String getNombreClub() {
        return nombreClub;
    }

    public int getPuestoLiga() {
        return puestoLiga;
    }
}

```

Como se ve BARCA, REAL_MADRID, etc. son el nombre del enumerado (u objetos de la clase Equipo) que tendrán como atributos el "nombreClub" y "puestoLiga". Como se ve en la clase **definimos un constructor que es privado** (es decir que solo es visible dentro de la clase Equipo) y solo definimos los métodos "get". Para trabajar con los atributos de estos

enumerados se hace de la misma manera que con cualquier otro objeto; se instancia un objeto y se accede a los atributos con los métodos `get`. En el siguiente fragmento de código vamos a ver cómo trabajar con enumerados que tienen atributos:

```
// Instanciamos el enumerado  
Equipo villareal = Equipo.VILLAREAL;  
  
// Devuelve un String con el nombre de la constante  
System.out.println("villareal.name()= "+villareal.name());  
  
// Devuelve el contenido de los atributos  
System.out.println("villareal.getNombreClub()= "+villareal.getNombreClub());  
System.out.println("villareal.getPuestoLiga()= "+villareal.getPuestoLiga());
```

Como salida de este fragmento de código tenemos lo siguiente:

```
villareal.name()= VILLAREAL  
villareal.getNombreClub()= Villareal  
villareal.getPuestoLiga()= 7
```

Es muy importante tener claro que los enumerados no son Strings (aunque pueden serlo), sino que son objetos de una clase que solo son instanciables desde la clase que se implementa y que no se puede crear un objeto de esa clase desde cualquier otro lado que no sea dentro de esa clase. Es muy común (sobre todo cuando se está aprendiendo que son los enumerados) que se interprete que un enumerado es una lista finita de Strings y en realidad es una lista finita de **objetos** de una determinada clase con sus atributos, constructor y métodos getter aunque estos sean privados.

A continuación, vamos a poner un sencillo ejemplo en el que vamos a mezclar los dos enumerados anteriores (Demarcación y Equipo). En este ejemplo, vamos a crear unos objetos de la clase **Futbolista**, que representarán a los jugadores de la selección española de fútbol que ganaron el mundial de fútbol de Sudáfrica en el año 2010. Esta clase va a caracterizar a los futbolistas por su nombre, su dorsal, la demarcación en la que juegan y el club de fútbol al que pertenecen; tal y como vemos en el siguiente diagrama de clases.



Como vemos los atributos de demarcación y equipo son de la clase **Demarcacion** y **Equipo** respectivamente y son los enumerados vistos anteriormente; por tanto un futbolista solo podrá pertenecer a uno de los cuatro equipos que forman el enumerado "Equipo" y podrá jugar en alguna de las cuatro demarcaciones que forman el enumerado "Demarcación". A continuación, se muestra la implementación de la clase "Futbolista":

```
package Main;

public class Futbolista {

    private int dorsal;
    private String Nombre;
    private Demarcacion demarcacion;
    private Equipo equipo;

    public Futbolista() {
    }

    public Futbolista(String nombre, int dorsal,
        Demarcacion demarcacion, Equipo equipo) {
        this.dorsal = dorsal;
        Nombre = nombre;
        this.demarcacion = demarcacion;
        this.equipo = equipo;
    }

    // Metodos getter y setter
    .....
    @Override
    public String toString() {
        return this.dorsal + " - " + this.Nombre + " - "
            + this.demarcacion.name()
            + " - " + this.equipo.getNombreClub();
    }
}
```

Dada esta clase podemos crearnos ya objetos de la clase futbolista, como mostramos a continuación:

```
Futbolista casillas = new Futbolista("Casillas", 1, Demarcacion.PORTERO,
Equipo.REAL_MADRID);

Futbolista capdevila = new Futbolista("Capdevila", 11,
Demarcacion.DEFENSA, Equipo.VILLAREAL);

Futbolista iniesta = new Futbolista("Iniesta", 6,
Demarcacion.CENTROCAMPISTA, Equipo.BARCA);

Futbolista navas = new Futbolista("Navas", 22, Demarcacion.DELANTERO,
Equipo.SEVILLA);
```

Como vemos la demarcación y el equipo al que pertenecen solo pueden ser los declarados en la clase enumerado. Si llamamos al método "toString()" declarado en la clase futbolista, podemos imprimir por pantalla los datos de los futbolistas. Dado el siguiente código:

```
System.out.println(casillas.toString());
System.out.println(capdevila.toString());
System.out.println(iniesta.toString());
System.out.println(navas.toString());
```

Y dado el siguiente método "toString()":

```
@Override
public String toString() {
    return this.dorsal + " - " + this.Nombre + " - "
           + this.demarcacion.name() + " - " +
           this.equipo.getNombreClub();
}
```

Tenemos como salida lo siguiente:

```
1 - Casillas - PORTERO - Real Madrid
11 - Capdevila - DEFENSA - Villareal
6 - Iniesta - CENTROCAMPISTA - FC Barcelona
22 - Navas - DELANTERO - Sevilla FC
```

En **resumen**: Un enumerado está formado por objetos definidos en la misma clase con constructor privado y si tiene atributos, estos solo tienen que tener métodos "getter" para obtener el valor del atributo.

UT8 Gestión de Excepciones

1. Excepciones. Categorías.

Las excepciones son el mecanismo por el cual pueden controlarse en un programa Java las condiciones de error que se producen. Estas condiciones de error pueden ser errores en la lógica del programa como un índice de un array fuera de su rango, una división por cero o errores disparados por los propios objetos que denuncian algún tipo de estado no previsto, o condición que no pueden manejar.

La idea general es que cuando un objeto encuentra una condición que no sabe manejar crea y dispara una excepción que deberá ser capturada por el que le llamó o por alguien más arriba en la pila de llamadas. Las excepciones son objetos que contienen información del error que se ha producido y que heredan de la clase `Throwable` o de la clase `Exception`. Si nadie capture la excepción interviene un manejador por defecto que normalmente imprime información que ayuda a encontrar quién produjo la excepción.

Existen dos categorías de excepciones:

- **Excepciones verificadas:** El compilador obliga a verificarlas. Son todas las que son lanzadas explícitamente por objetos de usuario.
- **Excepciones no verificadas:** El compilador no obliga a su verificación. Son excepciones como divisiones por cero, excepciones de puntero nulo, o índices fuera de rango.

2. Generación de excepciones

Supongamos que tenemos una clase `Empresa` que tiene un array de objetos `Empleado` (clase vista en capítulos anteriores). En esta clase podríamos tener métodos para contratar un `Empleado` (añadir un nuevo objeto al array), despedirlo (quitarlo del array) u obtener el nombre a partir del número de empleado. La clase podría ser algo así como lo siguiente:

```
public class Empresa {  
    String nombre;  
    Empleado [] listaEmpleados;  
    int totalEmpleados = 0;  
    . . .  
    Empresa(String n, int maxEmp) {  
        nombre = n;  
        listaEmpleados = new Empleado [maxEmp];  
    }  
    . . .  
    void nuevoEmpleado(String nombre, int sueldo) {  
        if (totalEmpleados < listaEmpleados.length ) {  
            listaEmpleados [totalEmpleados++] =  
                new Empleado(nombre, sueldo);  
        }  
    }  
}
```

Obsérvese en el método nuevoEmpleado que se comprueba que hay sitio en el array para almacenar la referencia al nuevo empleado. Si lo hay se crea el objeto. Pero si no lo hay el método no hace nada más. No da ninguna indicación de si la operación ha tenido éxito o no. Se podría hacer una modificación para que, por ejemplo el método devolviera un valor booleano true si la operación se ha completado con éxito y false si ha habido algún problema.

Otra posibilidad es generar una excepción verificada (Una excepción no verificada se produciría si no se comprobara si el nuevo empleado va a caber o no en el array). Vamos a ver como se haría esto.

Las excepciones son clases, que heredan de la clase genérica Exception. Es necesario por tanto asignar un nombre a nuestra excepción. **Se suelen asignar nombres que den alguna idea del tipo de error que controlan**. En nuestro ejemplo le vamos a llamar CapacidadEmpresaExcedida.

Para que un método lance una excepción:

- Debe declarar el tipo de excepción que lanza con la cláusula throws, en su declaración.
- Debe lanzar la excepción, en el punto del código adecuado con la sentencia throw.

En nuestro ejemplo:

```
void nuevoEmpleado(String nombre, int sueldo) throws  
CapacidadEmpresaExcedida {  
    if (totalEmpleados < listaEmpleados.length) {  
        listaEmpleados [totalEmpleados++] =  
            new Empleado (nombre, sueldo);  
    }  
    else throw new CapacidadEmpresaExcedida (nombre);  
}
```

Además, necesitamos escribir la clase CapacidadEmpresaExcedida. Sería algo así:

```
public class CapacidadEmpresaExcedida extends Exception {  
    CapacidadEmpresaExcedida (String nombre) {  
        super ("No es posible añadir el empleado " + nombre);  
    }  
    . . .  
}
```

La sentencia throw crea un objeto de la clase CapacidadEmpresaExcedida. El constructor tiene un argumento (el nombre del empleado). El constructor simplemente llama al constructor de la superclase pasándole como argumento un texto explicativo del error (y el nombre del empleado que no se ha podido añadir).

La clase de la excepción puede declarar otros métodos o guardar datos de depuración que se consideren oportunos. El único requisito es que extienda la

clase Exception. Consultar la documentación del API para ver una descripción completa de la clase Exception.

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Throwable.html>

De esta forma se pueden construir métodos que generen excepciones.

3. Captura de excepciones

Con la primera versión del método nuevoEmpleado (sin excepción) se invocaría este método de la siguiente forma:

```
Empresa em = new Empresa("La Mundial");
em.nuevoEmpleado("Adán Primero", 500);
```

Si se utilizara este formato en el segundo caso (con excepción) el compilador produciría un error indicando que no se ha capturado la excepción verificada lanzada por el método nuevoEmpleado. Para capturar la excepción es utiliza la construcción try / catch, de la siguiente forma:

```
Empresa em = new Empresa("La Mundial");
try {
    em.nuevoEmpleado("Adán Primero", 500);
} catch (CapacidadEmpresaExcedida exc) {
    System.out.println(exc.toString());
    System.exit(1);
}
```

- Se encierra el código que puede lanzar la excepción en un bloque try / catch.
- A continuación del catch se indica que tipo de excepción se va a capturar.
- Después del catch se escribe el código que se ejecutará si se lanza la excepción.
- Si no se lanza la excepción el bloque catch no se ejecuta.

El formato general del bloque try / catch es:

```
try {
    . . .
} catch (Clase_Excepcion nombre) { . . . }
catch (Clase_Excepcion nombre) { . . . }
. . .
```

Obsérvese que se puede capturar más de un tipo de excepción declarando más de una sentencia catch. También se puede capturar una excepción genérica (clase Exception) que engloba a todas las demás.

En ocasiones el código que llama a un método que dispara una excepción tampoco puede (o sabe) manejar esa excepción. Si no sabe qué hacer con ella puede de nuevo lanzarla hacia arriba en la pila de llamada para que la gestione quien le llamo (que a su vez puede capturarla o reenviarla). Cuando un método

no tiene intención de capturar la excepción debe declararla mediante la cláusula throws, tal como hemos visto en el método que genera la excepción.

Supongamos que, en nuestro ejemplo es el método main de una clase el que invoca el método nuevoEmpleado. Si no quiere capturar la excepción debe hacer lo siguiente:

```
public static void main(String [] args) throws
//nunca se pone el throws en el main lolCapacidadEmpresaExcedida {
    Empresa em = new Empresa("La Mundial");
    em.nuevoEmpleado("Adán Primero", 500);
}
```

4. Cláusula finally

La cláusula finally forma parte del bloque **try / catch** y sirve para especificar un bloque de código que se ejecutará tanto si se lanza la excepción como si no. Puede servir para limpieza del estado interno de los objetos afectados o para liberar recursos externos (descriptores de fichero, por ejemplo). La sintaxis global del bloque **try / catch / finally** es:

```
try {
    . . .
} catch (Clase_Excepcion nombre) { . . . }
    catch (Clase_Excepcion nombre) { . . . }
    . . .
finally { . . . }
```

UT8 Matches

El método matches de String nos permite comprobar si un String cumple una expresión regular pasada como parámetro. Si es cierta devuelve true, sino false.

Una expresión regular es una expresión textual que utiliza símbolos especiales para hacer búsquedas avanzadas.

Las expresiones pueden contener:

- **Caracteres.**
- **Caracteres de control**, por ejemplo, \s, \d, etc. Recuerda que añadir un \ más al introducirlo en una cadena en java. Los más usados son:
 - \d, dígito, es igual que [0-9]
 - \D, no dígito, es igual que [^0-9] // ^ que no esté en este rango
 - \s, carácter en blanco, es igual que [\t\n\x0B\f\r]
 - \S, no carácter en blanco, es igual que [^\s]
 - \w, carácter alfanumérico, es igual que [a-zA-Z0-9]
 - \W, no carácter alfanumérico, es igual que [^\w], [^a-zA-Z0-9]
- **Opciones de caracteres**, se usa el corchete. Por ejemplo, [afgd] significa que puede contener a, f, g o d.
- **Negación de caracteres**, funciona al revés que el anterior, se usa ^. Por ejemplo, [^afgd]
- **Rangos**, se usa para que incluya un rango de caracteres. Por ejemplo, para que incluya los caracteres entre a y z [a-z]
- **Intersección**: permite unir dos condiciones, es como el operador &&.
- **Cualquier carácter**: se usa un punto.
- **Opcional**: se usa el símbolo ?, indica que un carácter puede o no aparecer.
- **Repetición**: se usa el símbolo *, indica que un conjunto de caracteres se puede repetir o no.
- **Repetición obligada**: se usa el símbolo +, es como el anterior, pero debe aparecer mínimo una vez.
- **Repetición un número exacto de veces**: después de una expresión abrimos llaves {} con un número dentro, indica el número de veces que debe repetirse un carácter o expresión. Si después del número escribimos una coma, indica que debe repetirse como mínimo el número que indiquemos y como máximo los que queramos. Si después de la coma

escribimos un número, indica que debe repetirse entre los números que le indiquemos como si fuera un rango.

Página para probar expresiones regulares: [regex101: build, test, and debug regex](https://www.regex101.com/)

1. Símbolos comunes en expresiones regulares

.	Un punto indica cualquier carácter
^expresión	El símbolo ^ indica el principio del String. En este caso el String debe contener la expresión al principio.
expresión\$	El símbolo \$ indica el final del String. En este caso el String debe contener la expresión al final.
[abc]	Los corchetes representan una definición de conjunto. En este ejemplo el String debe contener las letras a ó b ó c.
[abc] [12]	El String debe contener las letras a ó b ó c seguidas de 1 ó 2
[^abc]	El símbolo ^ dentro de los corchetes indica negación. En este caso el String debe contener cualquier carácter excepto a ó b ó c.
[-A Z1-9]	Rango. Indica las letras minúsculas desde la a hasta la z (ambas incluidas) y los dígitos desde el 1 hasta el 9 (ambos incluidos)` . Solo 1.
A B	El carácter es un OR. A ó B
AB	La concatenación. A seguido de B

2. Meta caracteres en expresiones regulares

Expresión	Descripción
\ D	Dígito. Equivale a [0-9]
\ D	No dígito. Equivale a [^0-9]
\ s	Espacio en blanco. Equivale a [\t\n\x0b\r\f]
\ S	No espacio en blanco. Equivale a [^\s]
\ W	Una letra mayúscula o minúscula, un dígito o el carácter _Equivale a [a-zA-Z0-9_]
\ W	Equivale a [^\w]
\ B	Límite de una palabra.

3. Cuantificadores en expresiones regulares

Expresión	Descripción
{X}	Indica que lo que va justo antes de las llaves se repite X veces
{X, Y}	Indica que lo que va justo antes de las llaves se repite mínimo X veces y máximo Y veces. También podemos poner {X,} indicando que se repite un mínimo de X veces sin límite máximo.
*	Indica 0 ó más veces. Equivale a {0,}
+	Indica 1 ó más veces. Equivale a {1,}
?	Indica 0 ó 1 veces. Equivale a {0,1}

4. Cómo usar los símbolos especiales de expresiones regulares en Java

En Java debemos usar una doble barra invertida \\. Por ejemplo para utilizar \w tendremos que escribir \\w.

Si queremos indicar que la barra invertida es un carácter de la expresión regular tendremos que escribir \\\\".

5. Expresiones Regulares con Matches de Java

Para usar expresiones regulares en Java se usa el package java.util.regex. Contiene las clases Pattern y Matcher y la excepción PatternSyntaxException

6. Algunos ejemplos en código de Expresiones Regulares con Matches de Java

```
// Comprobar si el String cadena contiene exactamente el patrón
(matches) "abc"

Patternpat Pattern.compile ("abc");
Matchermat = pat.matcher (Cadena);

if (mat.matches ()) {
    System.out.println ("SI");
} else {
    System.out.println ("NO");
}
```

UT8 Ejercicios de Excepciones

1.- Realiza un programa que calcule el valor del factorial de un número que se pide al usuario. Si el número es negativo debe saltar una excepción llamada FactorialNegExc creada por el programador que indique que no se puede calcular el factorial de un número negativo. En la cláusula finally escribirá el resultado, siempre que haya sido posible calcularlo.

2.- Realiza un programa en Java que permita crear cuentas bancarias pidiendo la cantidad inicial al usuario y el número de cuenta (no se permite repetir número de cuenta), así como realizar las operaciones ingresar y sacar dinero de esas cuentas. En el caso de intentar sacar de la cuenta corriente más dinero del que hay, se mostrará el mensaje asociado a una excepción denominada NoHayDineroExcepcion que crearás.

El programa constará de un menú con las siguientes opciones:

1. Abrir cuenta
2. Ingresar dinero en cuenta
3. Sacar dinero de cuenta
4. Visualizar todas las cuentas con su saldo actual
5. Mostrar una cuenta
6. Borrar una cuenta
7. Borrar todas las cuentas
8. Salir

El orden al mostrar todas las cuentas será por saldo de mayor a menor, y si dos cuentas tienen el mismo saldo por orden creciente de cuenta. Utiliza el interfaz **Comparable**.

Para realizar el programa deberás distribuirlo en varios archivos:

Cuenta.java para desarrollar métodos y atributos propios de una cuenta corriente

CuentaApp.java que contendrá el método main y mostrará el menú con las opciones indicadas, realizando llamadas a los métodos que se muestran a continuación:

```

switch (opcion)
{
    case 1:
        b.guardarCuenta(b.crearCuenta(sc));
        break;
    case 2:
        b.ingresarEnCuenta(sc);
        break;
    case 3:
        b.sacarDeCuenta(sc);
        break;
    case 4:
        b.listarCuentas();
        break;
    case 5:
        b.muestraCuenta(sc);
        break;
    case 6:
        b.borraCuenta(sc);
        break;
    case 7:
        b.borraTodasCuentas(sc);
        break;
}

```

Banco.java que contendrá los métodos llamados en el menú, así como el arraylist de cuentas corrientes.

Con la clase **Lector.java**, que se te proporciona con el enunciado crearás un paquete externo que incorporarás en el proyecto.

Añadirás los archivos/clases necesarios para llevar a cabo los requerimientos pedidos.

Crea una **segunda versión** en la que las cuentas se puedan ver de dos formas distintas, ascendente por número de cuenta y descendente por saldo:

1. Abrir cuenta
2. Ingresar dinero en cuenta
3. Sacar dinero de cuenta
4. Mostrar todas las cuentas(desc por saldo)
5. Mostrar todas las cuentas(asc por num. cuenta)
6. Mostrar una cuenta
7. Borrar una cuenta
8. Borrar todas las cuentas
9. Salir.

Elija opcion:

Ejercicio X. Cambios DGT Excepciones

7.- Definir los atributos de las clases de objetos necesarias para almacenar la información relativa a unas muestras de alcoholemia recogidas en un análisis preventivo de la Dirección General de Tráfico, de forma que:

- a) Para cada conductor interesa tener su nombre, DNI. y una colección de referencias a las muestras que se le han tomado durante el período de estudio.
- b) Cada muestra tendrá los siguientes datos: Día y hora de la muestra, código del puesto de control preventivo (dos letras y dos números), matrícula del vehículo y la tasa de alcohol espirado en aire (entre 0 y 2.5 mg/l).

Hacer una simulación del funcionamiento de estas clases usando el siguiente menú:

1. Introducir un nuevo conductor.
2. Introducir una muestra.
3. Mostrar los datos de un conductor.
4. Mostrar las muestras de un conductor.
5. Mostrar los datos de todos los conductores.
6. Salir.

Elige una opción:

Hacerlo con Arrays y ArrayList.

Al introducir una muestra, se elige de qué conductor por su dni.

Los datos del conductor se muestran buscando por dni.

Cambios pedidos:

1. Mostrar los conductores con su número de orden para identificar de qué conductor se quieren introducir muestras.
2. Hacer el control de formato de código y de alcohol con dos nuevas excepciones que crearás. (se vuelve a pedir en caso de saltar la excepción) La fecha y hora de las muestras será obtenida por programa como las actuales, no se piden al usuario ya.

Ejercicio Vehículos

1. Se quiere implementar un simulador de vehículos. Existen dos clases de **Vehiculos: Coche y Camion.**

- a) De todos los **Vehiculos** queremos almacenar la matrícula y la velocidad. En el momento de crearlos, la matricula se recibe como parámetro y la velocidad se pone a 0. En su correspondiente método **toString()** se devolverá la matrícula y la velocidad. Además se puede **acelerar** pasando como parámetro la cantidad en km/h que se tiene que acelerar.
- b) Los coches tienen además un atributo para el número de puertas que se recibe también como parámetro a la hora de crearlo. Y contiene un método que **devuelve** el número de puertas.
- c) Los camiones tienen un atributo del tipo de **Remolque** que se inicializa a null en el momento de crear el camión para indicar que aún no tiene remolque. Definimos en la clase dos métodos, **ponRemolque** que recibe un Remolque por parámetro y otro **quitaRemolque** que pone el atributo a null. Cuando se muestre la información que lleve remolque, además de matrícula y velocidad debe aparecer información del remolque.
- d) En la clase Camion hay que sobrescribir el método de **acelerar**, de forma que si el camión lleva remolque y la velocidad supera los 100 km/h debe levantarse una excepción del tipo **DemasiadoRapidoExpcion**.
- e) Hay que implementar la clase Remolque que tiene un atributo de tipo entero que es el peso del remolque y su valor se le asigna a la hora de crear el Remolque. Tendrá su método **toString()** como todas las demás clases.
- f) Se implementará la clase **DemasiadoRapidoExpcion**.
- g) Crearemos la clase **PruebaVehiculo** con un método main donde se tenga un menú con las siguientes operaciones (utiliza un ArrayList para guardar los vehículos que se creen desde el menú):

- 1.- Añadir coche**
- 2.- Añadir camión**
- 3.- Añadir un remolque (de peso en kg el que indique el usuario) a todos los camiones**
- 4.- Muestre la información de todos los vehículos por pantalla.**
- 5.- Borre todos los camiones que tengan un remolque de más de 1000 kg**
- 6.- Salir**

Para practicar ordenación con interfaces de Java, prueba a mostrar los vehículos ordenados por todos los criterios que se te ocurran.

UT9 Ficheros en Java

1. Conceptos Básicos de Ficheros
2. Operaciones sobre ficheros
3. Tipos de fichero
4. Conceptos Básicos de Entrada/Salida
5. Ficheros de Texto
6. Ficheros Binarios
7. Ficheros de Acceso Aleatorio
8. Serialización

1. Conceptos Básicos de Ficheros

Hasta ahora todos los datos que creábamos en nuestros programas solamente existían durante la ejecución de los mismos. Cuando salíamos del programa, todo lo que habíamos generado se perdía.

A veces nos interesa que la vida de los datos vaya más allá de la de los programas que los generaron. Es decir, que al salir de un programa los datos generados queden guardados en algún lugar que permita su recuperación desde el mismo u otros programas. Esto que quiere decir que queremos que los datos sean *persistentes*.

En este tema veremos el uso básico de **archivos/ficheros** en Java para conseguir persistencia de datos. Para ello, presentaremos conceptos básicos sobre archivos y algunas de las clases de la biblioteca estándar de Java para su creación y manipulación.

Además, el uso de esas bibliotecas nos obligará a introducir algunos conceptos "avanzados" de programación en Java para transformar nuestros datos a vectores de bytes.

Cuando se desea guardar información más allá del tiempo de ejecución de un programa, lo habitual es organizar esa información en uno o varios **ficheros** almacenados en algún soporte de almacenamiento persistente.

Otras posibilidades como el uso de **bases de datos**, también utilizan archivos como soporte para el almacenamiento de la información.

Desde el punto de vista de más bajo nivel, podemos definir un archivo (o fichero) como:

Un conjunto de bits almacenados en un dispositivo, y accesible a través de un camino de acceso (pathname) que lo identifica. Es decir, un conjunto de 0s y 1s que reside fuera de la memoria del ordenador, ya sea en el disco duro, un pendrive, un CD...

Esa versión de bajo nivel, si bien es completamente cierta, desde el punto de vista de la programación de aplicaciones, es demasiado simple. Por ello definiremos varios criterios para distinguir diversas subcategorías de archivos.

Estos tipos de archivos se diferenciarán desde el punto de vista de la programación: Cada uno de ellos proporcionará diferentes funcionalidades (métodos) para su manipulación.

1.1 Objeto file

[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java.io/File.html](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/File.html)

En el paquete java.io se encuentra la clase **File** pensada para poder realizar operaciones de información sobre archivos. No proporciona métodos de acceso a los archivos, sino operaciones **a nivel de sistema** de archivos (listado de archivos, crear carpetas, borrar ficheros, cambiar nombre,...).

Aclaro que quiero decir con “No proporciona métodos de acceso a los archivos”: no sirven para crear un fichero y poder acceder a él para escribir o leer su contenido, sino que sirven para ver información del fichero desde el punto de vista del sistema operativo.

Vamos a realizar una serie de ejercicios sobre este concepto.

Un **objeto File** puede representar un archivo o un directorio y sirve para obtener información (permisos, tamaño,...) y también para navegar por la estructura de archivos.

Su *constructor* puede recibir un único parámetro: una cadena que representa una ruta (*path*) en el sistema de archivos. O también puede recibir, opcionalmente, un segundo parámetro con una ruta segunda que se define a partir de la posición de la primera.

Si el archivo o carpeta que se intenta examinar no existe, la clase **File** **no devuelve** una excepción. Por lo que si necesitamos saberlo, hay que utilizar el método *exists*. Este método **devuelve true si la carpeta o archivo existe**. Vamos a ver un ejemplo.

Ojo, que **new File()** no crea el fichero físicamente, ¡es un mero descriptor que apunta a la información del fichero al que apunta o apunte en el futuro!

```

public class InfoFichero0 {

    public static void main(String args[]) {
        if (args.length == 1) {
            File f = new File(args[0]);
            System.out.println("Nombre: " + f.getName());
            System.out.println("Camino: " + f.getPath());
            if (f.exists()) {
                System.out.print("Fichero existente ");
                System.out.print((f.canRead() ? " y se puede Leer" : " y no se puede leer"));
                System.out.print((f.canWrite() ? " y se puede Escribir" : " y no se puede escribir"));
                System.out.println(".");
                System.out.println("La longitud del fichero es de " + f.length() + " bytes");
            } else
                System.out.println("El fichero no existe.");
        } else
            System.out.println("Debe indicar un fichero.");
    }
}

```

Según la ayuda del api, los **constructores** son:

File (File padre, String hijo)	Crea una instancia de File a partir de un File padre y una ruta que se une a la ruta del hijo (String)
File (String ruta)	Crea una instancia de File usando la cadena recibida como ruta.
File (String padre, String hijo)	Crea una instancia de File usando las 2 cadenas recibidas y la ruta es la concatenación de la primera con la segunda
File (URI uri)	Crea una instancia de File convirtiendo la URI en una ruta

```

public class EjemploFile {

    public static void main(String[] args) {

        try {
            File f1 = new File("C:\\\\Pruebas");
            System.out.println(f1.getName());

            File f2 = new File("C:\\\\Pruebas\\\\prueba.txt");
            System.out.println(f2.getName());

            File dir = new File("C:\\\\Pruebas");
            File f3 = new File(dir, "prueba.txt");
            System.out.println(f3.getName());

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Arriba tienes ejemplo de los 3 primeros constructores, pruébalo tecleando el programa en tu Eclipse (**ejemplos029_ficheros -> CreaFichero2.java**). Un ejemplo de la última, que quizás sea la que puedes ver menos clara, sería:

```
package IO.infoFile;
import java.io.File;
import java.net.URI;
import java.net.URISyntaxException;

public class EjemploURI {
    public static void main(String[] args) {
        File aFile;
        try {
            //aFile = new File(new URI("file:///c:/a.bat"));
            aFile = new File("https://elpais.com");
            System.out.println(aFile.getName()); // false
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2.2 Creación de directorios/carpetas

Esta necesita que exista la carpeta contenedora:

boolean java.io.File.mkdir()

Creates the directory named by this abstract pathname.

Returns:

true if and only if the directory was created; false otherwise

Throws:

[SecurityException](#) - If a security manager exists and its [java.lang.SecurityManager.checkWrite\(java.lang.String\)](#) method does not permit the named directory to be created

Esta crea las carpetas contenedoras si no existen:

boolean java.io.File.mkdirs()

Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories.

Returns:

true if and only if the directory was created, along with all necessary parent directories; false otherwise

Throws:

[SecurityException](#) - If a security manager exists and its [java.lang.SecurityManager.checkRead\(java.lang.String\)](#) method does not permit verification of the existence of the named directory and all necessary parent directories; or if the [java.lang.SecurityManager.checkWrite\(java.lang.String\)](#) method does not permit the named directory and all necessary parent directories to be created

Borrado de ficheros/directorios:

● `boolean java.io.File.delete()`

Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted.

Note that the [java.nio.file.Files](#) class defines the [delete](#) method to throw an [IOException](#) when a file cannot be deleted. This is useful for error reporting and to diagnose why a file cannot be deleted.

Returns:

`true` if and only if the file or directory is successfully deleted; `false` otherwise

Throws:

[SecurityException](#) - If a security manager exists and its [java.lang.SecurityManager.checkDelete](#) method denies delete access to the file

Se borra el fichero si la ruta es de un fichero, o la carpeta si se refiere a la ruta de una carpeta. No borra el resto de carpetas: ¡Ojo, que **solo se borra la carpeta si está vacía!** Haz la prueba.

Creación de ficheros, crea el fichero si no existe:

● `boolean java.io.File.createNewFile() throws IOException`

Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. The check for the existence of the file and the creation of the file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the file.

Note: this method should *not* be used for file-locking, as the resulting protocol cannot be made to work reliably. The [FileLock](#) facility should be used instead.

Returns:

`true` if the named file does not exist and was successfully created; `false` if the named file already exists

Throws:

[IOException](#) - If an I/O error occurred

[SecurityException](#) - If a security manager exists and its [java.lang.SecurityManager.checkWrite](#) ([java.lang.String](#)) method denies write access to the file

Since:

1.2

Ejemplo de creación de un fichero (**ejemplos029_ficheros -> CreaFichero2.java**):

```
import java.io.*;

public class CreaFichero1 {
    public static void main(String args[]){
        // Crea un objeto File dada la ruta completa
        File f1 = new File("C:\\Ficheros\\nuevo.txt");
        try {
            // A partir del objeto File creamos el
            // fichero físicamente
            if (f1.createNewFile())
                System.out.println("El fichero se ha creado
                                    correctamente");
            else
                System.out.println("No ha podido ser creado
                                    el fichero");
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }

        // Crea un objeto File dada la ruta del directorio y el nombre
        // del fichero por separado

        File f2 = new File("C:\\Ficheros", "nuevo2.txt");

        // Crea un objeto File dado el directorio y el nombre
        // del fichero por separado
        try {
            // A partir del objeto File creamos el
            // fichero físicamente

            if (f2.createNewFile())
                System.out.println("El fichero se ha creado
                                    correctamente");
            else
                System.out.println("No ha podido ser creado el
                                    fichero");
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }

        File dir = new File("C:\\Ficheros");
        File f3 = new File(dir, "nuevo3.txt");

        try {
            // A partir del objeto File creamos el
            // fichero físicamente
            if (f3.createNewFile())
                System.out.println("El fichero se ha creado
                                    correctamente");
            else
                System.out.println("No ha podido ser creado
                                    el fichero");
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

Ejemplo de creación de un fichero para escritura:

```

package ejemplos029_ficheros3_readerWriter;
import java.io.*;

public class CreaFichero2 {
    public static void main(String []args) {
        String frase="Esto es un ejemplo de escritura de ficheros de texto";
        try{
            //Crear un objeto File se encarga de crear o abrir
            // acceso a un archivo que se especifica en su constructor
            File archivo = new File("texto.txt");
            // Crear objeto FileWriter que será el que nos ayude a
            // escribir sobre archivo
            FileWriter escribir = new FileWriter(archivo);
            // Crea el fichero en la carpeta del proyecto
            // FileWriter escribir = new FileWriter(archivo,true);
            // Para añadir
            // Se escribe en el archivo con el metodo write
            // true: si ya existe el fichero no lo machaco,
            // continuo escribiendo donde lo deje escribir.write(frase);
            escribir.close(); // Se cierra la conexión
        }
        catch(Exception e){ System.out.println("Error al escribir"); }
    }
}

```

2. Operaciones sobre ficheros

Los tipos de operaciones son:

- a) Operación de Creación
- b) Operación de Apertura
 - Varios modos:
 - b.1) Solo lectura
 - b.2) Solo escritura
 - b.3) Lectura y escritura
- c) Operaciones de lectura / escritura
- d) Operaciones de inserción / borrado
- e) Operaciones de renombrado / eliminación
- f) Operación de desplazamiento dentro de un fichero
- g) Operación de cierre

Las operaciones para el manejo habitual de un fichero:

- 1.- Crearlo (solo si no existía previamente)
- 2.- Abrirlo
- 3.- Operar sobre él (lectura/escritura, inserción, borrado, etc.)
- 4.- Cerrarlo

3. Tipos de ficheros

La clasificación de los ficheros **según el acceso** a la información almacenada es:

- Acceso secuencial:** Para acceder a los datos es necesario pasar por todos los anteriores. **Ej:** Cinta de Cassete.
- Acceso directo o aleatorio:** Se puede acceder a un dato sin pasar por todos los anteriores. **Ej:** Disco Duro, arrays en Java.

Clasificación de los ficheros **según el contenido**:

Sabemos que es diferente manipular números que Strings, aunque en el fondo ambos acaben siendo bits en la memoria del ordenador. Por eso, cuando manipulamos archivos, distinguiremos dos clases de archivos dependiendo del tipo de datos que contienen:

- Los **archivos de caracteres (o de texto)**. Almacenan **caracteres alfanuméricos** en un formato estándar (ASCII, Unicode, UTF8, UTF16, etc.).
- Los **archivos de bytes (o binarios)**. Almacenan **secuencias de dígitos binarios** (ej: ficheros que almacenan enteros, floats, imágenes...). *Más adecuado para copiarlos*

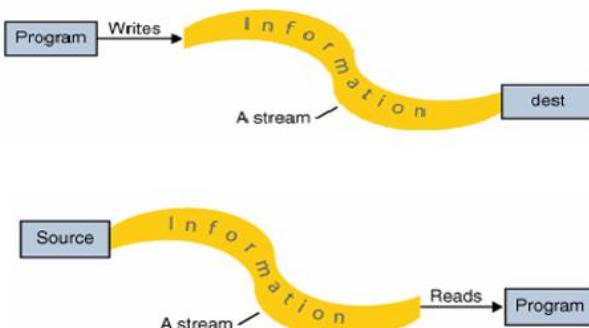
Un **fichero de texto** es aquel formado exclusivamente por **caracteres** y que, por tanto, **puede crearse y visualizarse** usando un **editor (plano)**. Las operaciones de lectura y escritura trabajarán con **caracteres**. **Por ejemplo, los ficheros con código java son ficheros de texto.**

En cambio, un **fichero binario** ya no está formado por **caracteres**, sino que los **bytes** que contiene pueden representar otras cosas como **números, imágenes, sonido, etc.**

4. Conceptos Básicos de Entrada/Salida

En Java se define la abstracción de **stream** (flujo) para tratar la comunicación de información entre el programa y el exterior. Entre una fuente y un destino fluye una secuencia de datos.

Los flujos actúan como **interfaz con el dispositivo o clase asociada** y proporcionan una **operación independiente del tipo de datos y del dispositivo**. Nos permiten usar diversidad de dispositivos (fichero, pantalla, teclado, red...) y diversidad de formas de comunicación.



Flujos estándar:

Entrada estándar - habitualmente el teclado

Salida estándar - habitualmente la consola

Salida de error - habitualmente la consola

En Java se accede a la E/S estándar a través de atributos estáticos de la clase `java.lang.System`:

System.in implementa la entrada estándar *atributo estatico de la clase System*

System.out implementa la salida estándar

System.err implementa la salida de error

La entrada/salida estándar (normalmente el teclado y la pantalla, respectivamente) se define mediante dos objetos que puede usar el programador sin tener que crear flujos específicos.

La clase **System** tiene un miembro denominado **in** que es una instancia de la clase **InputStream** que representa al teclado o flujo de entrada estándar [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java.io/InputStream.html](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStream.html) Sin embargo, el miembro **out** de la clase **System** es un objeto de la clase **PrintStream**, que imprime texto en la pantalla (la salida estándar).



Flujos estándar:

System.in

Instancia de la clase `InputStream`: flujo de bytes de entrada

Métodos para la lectura de datos:

- `read()` permite leer un byte de la entrada como entero
- `skip(n)` ignora n bytes de la entrada
- `available()` número de bytes disponibles para leer en la entrada

System.out

Instancia de la clase `PrintStream`: flujo de bytes de salida

Métodos para impresión de datos:

- `print()`, `println()`
- `flush()` vacía el buffer de salida escribiendo su contenido

System.err

Funcionamiento similar a `System.out`

Se utiliza para enviar mensajes de error

(por ejemplo a un fichero de log o a la consola)

Para **leer un carácter** solamente tenemos que llamar a la función **read** desde **System.in** (recuerda que System.in que es una instancia de la clase **InputStream**).

//pulse cualquier tecla para continuar

```
try{
    System.in.read();
}catch (IOException ex) { }
```

Ejemplo: package ejemplos029_ficheros3_leeLineaEntradaEstandar;

```
import java.io.*;

// Ejemplo de lectura de líneas de la entrada estándar
// carácter a carácter
// Contamos los caracteres que hay en una línea

class LecturaDeLinea {
public static void main( String args[] ) throws IOException {

    //el throws en el main no debe hacerse,
    //no nos enteramos cuando el main la relanza

    int c;
    int contador = 0;

    // Se lee hasta encontrar el fin de línea
    while((c = System.in.read()) != '\n')
    {
        contador++;
        System.out.print( (char) c );
    }
    System.out.println(); // Se escribe el fin de línea
    System.err.println( "Contados "+ contador
                        +" bytes/caracteres en total." );
    }

}
```

Utilización de los flujos:

FLUJOS DE ENTRADA Y SALIDA EN **JAVA**. ... El **flujo (stream)** es una secuencia ordenada de datos que tiene una fuente (**flujos** de entrada) o un destino (**flujos** de salida). Los streams soportan varios tipos de datos: bytes simples, tipos de datos primitivos, caracteres localizados, y objetos. Los flujos se implementan en las clases del paquete `java.io`. Esencialmente todos funcionan igual, independientemente de la fuente de datos.

Lectura:

1. Abrir un flujo a una fuente de datos (creación del objeto stream)
 - Teclado (como hemos visto en el ejemplo **leerUnaLineaEntrEstanda.java**)
 - Fichero
 - Socket remoto
2. Mientras existan datos disponibles:
 - Leer datos
3. Cerrar el flujo (método close())

Escritura:

1. Abrir un flujo a una fuente de datos (creación del objeto stream)
 - Pantalla (como hemos visto en el ejemplo **leerUnaLineaEntrEstanda.java**)
 - Fichero
 - Socket local
2. Mientras existan datos disponibles:
 - Escribir datos
3. Cerrar el flujo (método close())

Nota: para los flujos estándar ya se encarga el sistema de abrirlos y cerrarlos

Un fallo en cualquier punto produce la excepción IOException, es obligatorio capturarla.

Clasificación de flujos

Según la representación de la información:

Flujos de bytes: clases *InputStream* y *OutputStream*
Flujos de caracteres: clases *Reader* y *Writer*

Recuerda que System.in es un *InputStream*

Se puede pasar de un flujo de bytes a uno de caracteres con las clases *InputStreamReader* y *OutputStreamWriter*

Según el propósito:

Entrada: *InputStream*, *Reader*

Salida: *OutputStream*, *Writer*

Lectura/Escritura: *RandomAccessFile* (acceso directo, no aleatorio)

Transformación de los datos:

Realizan algún tipo de procesamiento sobre los datos (p.e. buffering, conversiones, filtrados): *BufferedReader*, *BufferedWriter*.

Según el tipo de acceso:

Secuencial - (Para acceder a un elemento hay que pasar por todos los demás)

Aleatorio - (*RandomAccessFile*) (Se puede acceder directamente a un elemento)

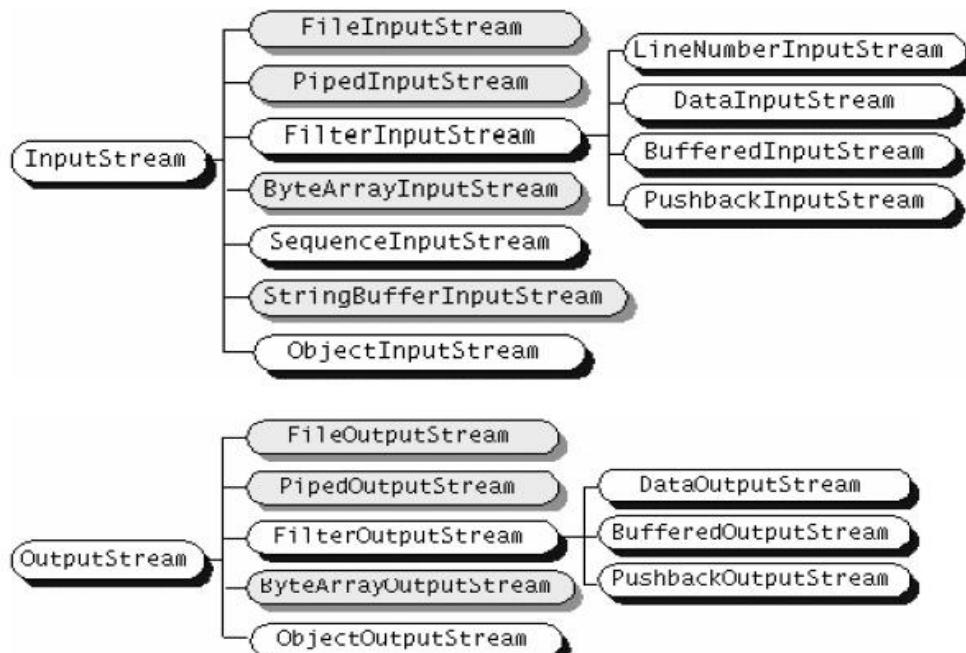
Los flujos están agrupados en el paquete `java.io` y se dividen en dos jerarquías de clases independientes, una para lectura/escritura binaria (bytes) y otra para lectura/escritura de caracteres de texto (char).

Jerarquía de flujos de bytes (Streams)

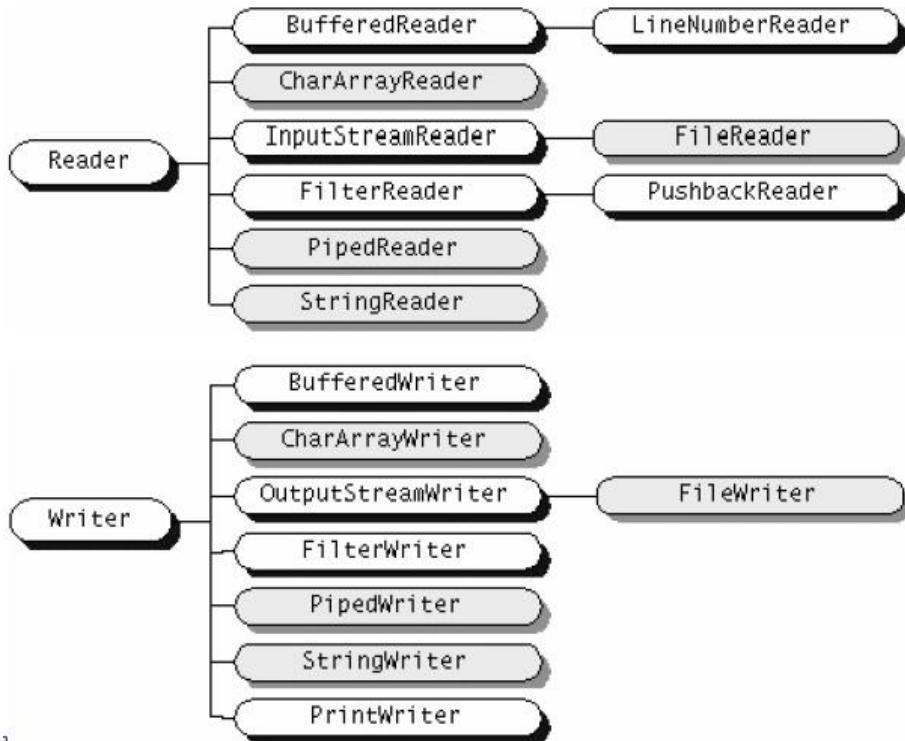
Son para leer ficheros binarios

En el siguiente gráfico vemos la relación entre la clase `InputStream` y sus hijas. Existen para facilitarnos las distintas operaciones que tenemos que hacer. Veremos distintos ejemplos con las más importantes. Recuerda que tratamos con estas clases cuando leamos bytes o conjuntos de bytes directamente.

Con estas clases (que tratan con Streams) se podría leer cualquier tipo de fichero, aunque cuando se trate de ficheros de texto puro (de los que se editan en un editor de ficheros planos, como notepad) es más fácil tratarlos con las clases que se muestran en el segundo gráfico (Jerarquía de flujos de caracteres).



Jerarquía de flujos de caracteres



Puedes investigar sobre las principales clases de ambos tipos en la ayuda de Java:

<https://docs.oracle.com/javase/8/docs/api/java/io/Writer.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/Writer.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/Reader.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/Reader.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStream.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/OutputStream.html>

Vamos a ir viendo cómo se producen las distintas operaciones de lectura/escritura con las distintas clases. Se utilizarán unas u otras según las necesidades de nuestros programas.

Entrada de caracteres (ambas clases son descendientes de la clase Reader, flujo de caracteres)

InputStreamReader

Lee bytes de un flujo InputStream y los convierte en caracteres Unicode.

Métodos de utilidad:

- `read()` lee un único carácter
- `ready()` indica cuando está listo el flujo para lectura

[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java.io/InputStreamReader.html](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStreamReader.html)

BufferedReader

Entrada mediante búfer, mejora el rendimiento.

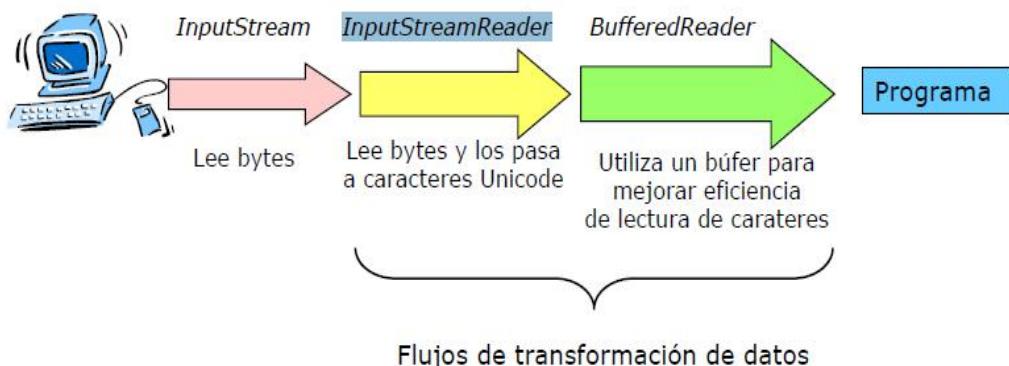
Método de utilidad:

- `readLine()` lectura de una línea como cadena

Ejemplo (*antes de tener Scanner para leer una línea había que hacer esto*):

```
InputStreamReader entrada = new  
InputStreamReader(System.in);  
  
BufferedReader teclado = new BufferedReader  
(entrada);  
  
String cadena = teclado.readLine();
```

Los flujos se pueden combinar para obtener la funcionalidad deseada:



Ejemplo de combinación de flujos:

ejemplos029_ficheros3_leeLineaEntradaEstandar;

```
import java.io.*;
public class Eco {
    public static void main (String[] args) {

        BufferedReader entradaEstandar =
        new BufferedReader (new InputStreamReader(System.in));

        String mensaje;

        System.out.println ("Introducir una línea de texto:");
        mensaje = entradaEstandar.readLine();

        System.out.println("Introducido: " + mensaje);
    }
}
```

Ejemplo de entrada de texto desde un fichero:

class LeeFicheroLineas3: ejemplos029_ficheros3_bufferedReader

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/BufferedReader.html>
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/FileReader.html>

```
try {
    BufferedReader reader =
    new BufferedReader(new FileReader("nombrefichero"));

    String linea = reader.readLine();

    while(linea != null) {
        // procesar el texto de la línea
        linea = reader.readLine();
    }

    reader.close();
}

catch(FileNotFoundException e) {
    // No se encontró el fichero
}
catch(IOException e) {
    // Algo fue mal al leer o cerrar el fichero
}
```

Ejemplo (como la clase Scanner):

ejemplos029_ficheros3_leeLineaEntradaEstandar;

```
import java.io.*;
class Teclado {

    /** variable de clase asignada a la entrada est ndar
     * del sistema
    */

    public static BufferedReader entrada =
    new BufferedReader(new InputStreamReader(System.in));

    /** lee una cadena desde la entrada est ndar
     * @return cadena de tipo String
    */

    public static String leerString() {
        String cadena="";
        try {
            cadena = new String(entrada.readLine());
            //antes la clase String no estaba en Java
            //implicitamente
        }catch (IOException e) {
            System.out.println("Error de E/S");
        }
        return cadena;
    }

    /** lee un numero entero desde la entrada estandar
     * @return numero entero de tipo int
    */

    public static int leerInt() {
        int entero = 0;
        boolean error = false;

        do {
        try {
            error = false;
            entero = Integer.valueOf(entrada.readLine());
        }catch (NumberFormatException e1) {
            error = true;
            System.out.println("Error en el formato del n mero,
                intentelo de nuevo.");
        }catch (IOException e) {
            System.out.println("Error de E/S");
        } while (error);

        return entero;
    }
}
```

5. Ficheros de Texto

Existen dos clases que manejan caracteres que son **FileWriter** y **FileReader**.

Veamos su jerarquía:

java.lang.Object
java.io.Writer
java.io.OutputStreamWriter
java.io.FileWriter

java.lang.Object
java.io.Reader
java.io.InputStreamReader
java.io.FileReader

La construcción de objetos del tipo **FileReader** se hace con un parámetro que puede ser un objeto **File** o un **String** que representarán a un determinado archivo.

La construcción de objetos **FileWriter** se hace igual, aunque también se le puede añadir un **segundo parámetro** booleano, que tomará valor **true** si se abre el archivo para añadir datos o **false** para crearlo por primera vez; si tuviese datos estos se borrarían en este segundo caso.

Para escribir se utiliza el método **write** que recibe como parámetro lo que se desea escribir en formato int o String y para leer se utiliza el método **read**, que devuelve un int, el valor en código ASCII del carácter leído. En el momento en que se haya alcanzado el **final del fichero** (no haya más datos que leer) el método **read** retorna el valor -1.

Métodos básicos de **Reader**:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

Métodos básicos de **Writer**:

```
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

Ejemplo de lectura de un fichero (ya existente) carácter a carácter:

```
package ejemplos029_ficheros3_readerWriter;

//Ejemplo de lectura de un fichero carácter a carácter.

import java.io.*;
public class LeeFicheroCaracter{

    public static void main(String arg[]){
        // Se define un int que va a contener los
        // caracteres del archivo

        int c;
        try{
            // Se crea un objeto FileReader que
            // obtiene lo que tenga el archivo

            FileReader lector=new FileReader("texto.txt");
            c=lector.read();
            while(c!=-1){ //c=-1 indica el fin de fichero
                System.out.println((char) c);
                c=lector.read();
            }

            lector.close();
        }
        catch(Exception e){
            System.out.println("Error al leer");
        }
    }
}
```

Las clases **BufferedReader** y **BufferedWriter** leen y escriben un texto desde un stream de salida o de entrada. Proporcionan un buffer de almacenamiento temporal.

La **escritura** se realiza con el método **write** que permite grabar caracteres, Strings y arrays de caracteres en el fichero. Permite utilizar el método **newLine()** que escribe un salto de línea en el archivo.

java.lang.Object
java.io.Writer
java.io.BufferedWriter

java.lang.Object
java.io.Reader
java.io.BufferedReader

Para **leer**, se tiene el método **readLine()** que lee una línea de texto entera cada vez. En este caso, cuando se haya alcanzado el final del fichero el método retorna el valor null.

No es posible usarlas por sí mismas, se deben envolver en un Reader o en un Writer dentro de un BufferedReader o de un BufferedWriter respectivamente. Esto es así porque sus constructores necesitan un parámetro de tipo Reader y Writer respectivamente. Busca estas clases en la ayuda del jdk y compruébalo.

```

public class LeeLineas{
    public static void main(String arg[]){
        // Creamos un String que va a ir conteniendo
        // todo el texto del archivo
        String texto="";
        try{
            // Se crea un objeto FileReader que
            // obtiene lo que tenga el archivo
            FileReader lector=new FileReader("texto.txt");
            // El contenido del objeto se envuelve
            // en un BufferedReader
            BufferedReader contenido=new BufferedReader(lector);
            texto= contenido.readLine();
            while(texto!=null){
                System.out.println(texto);
                texto= contenido.readLine();
            }
            contenido.close(); // Se puede omitir
            lector.close();
        }catch(Exception e){
            System.out.println("Error al leer");
        }
    }
}

```

Otra opción de escritura es usar la clase PrintWriter:

```

salida =new PrintWriter(new FileWriter(nombre))
FileWriter (nombre, añadir)
// así escribimos al final del fichero

```

Con los métodos:

```

println()
print()
close():IOException

```

Ejemplo:

```
package ejemplos029_ficheros5_ficheroTextoBuffered;

import java.io.*;
public class PruebaEscritura {

    public static void main(String[] args) {

        try {
            FileWriter connection =
            new FileWriter("C:\\test.txt", true);
            PrintWriter file = new PrintWriter(connection);
            file.println("Hola");
            file.println("Hola");
            file.close();
            writeAgain();
        }

        catch (IOException e) {
            System.out.println("IOException");
        }
    }

    public static void writeAgain() throws IOException {
        FileWriter connection =
        new FileWriter("C:\\test2.txt", true);
        BufferedWriter file = new BufferedWriter(connection);
        file.write("Adios");
        file.write("Adios");
        file.close();
    }
}
```

BufferedWriter vs PrintWriter:

PrintWriter: es el objeto que utilizamos para escribir directamente sobre el archivo de texto.

BufferedWriter: objeto que reserva un espacio en memoria donde se guarda la información antes de ser escrita en un archivo.

Comparado con las clases **FileWriter**, los **BufferedWriters** escriben relativamente grandes cantidades de información a un archivo, lo cual minimiza el número de veces que las operaciones de escritura de archivos se llevan a cabo

Es más óptimo el bufferWriter porque hace menos accesos a disco

- **FileWriter**: es un objeto que tiene como función escribir datos en un archivo.
- **BufferedWriter**: objeto que reserva un espacio en memoria (buffer) donde se guarda la información antes de ser escrita en un archivo.
- **PrintWriter**: Es el objeto que utilizamos para escribir directamente sobre el archivo de texto.

6. Ficheros binarios

Definición y uso

Un **fichero binario** es un fichero que **almacena una secuencia de datos codificados en binario** (secuencias de datos short, int, double, boolean, char, etc.)

Para trabajar con este tipo de ficheros, es decir, para escribir o leer en ellos la información binaria que almacenan se necesitan herramientas apropiadas y conocer el patrón que guía su disposición en el fichero. **En un fichero binario no hay líneas.** Vamos a ver **dos formas de usarlos:**

FileOutputStream y FileInputStream:

java.lang.Object
java.io.OutputStream
java.io. FileOutputStream

java.lang.Object
java.io.InputStream
java.io. FileInputStream

Es una pareja de clases que **nos permite leer grupos de bytes de un tamaño definido**, por ejemplo nos serviría para copiar un fichero de imagen.

void	write(byte[] b) Writes b.length bytes from the specified byte array to this file output stream.
void	write(byte[] b, int off, int len) Writes len bytes from the specified byte array starting at offset off to this file output stream.
void	write(int b) Writes the specified byte to this file output stream.

Pero como puedes ver en los ejemplos, también valdría para hacer una copia de un fichero de texto:

EjFileInputStream1.java y EjFileInputStream2.java

DataOutputStream y DataInputStream:

java.lang.Object
java.io.OutputStream
java.io.FilterOutputStream
java.io. DataOutputStream

java.lang.Object
java.io.InputStream
java.io.FilterInputStream
java.io. DataInputStream

La clase **DataOutputStream** permite crear un objeto que se asocia a un objeto **FileOutputStream** y facilita métodos para escribir o almacenar secuencialmente información codificada en binario en el fichero asociado a dicho objeto. La escritura de un nuevo dato se produce de forma secuencial.

La escritura física de datos en el fichero no se produce uno a uno, sino que se gestiona en bloques mediante un **buffer** o almacén intermedio. El programador debe gestionar la captura y tratamiento de IOException.

Los **métodos** más usados son:

```
void writeBoolean(boolean val)
void writeChar(int val)
void writeInt(int val)
void writeDouble(double val)
void writeChars(String str)
//DEPRECATED -> obsolete (substitute: writeUTF)
void writeUTF(String str)
```

Ej: Crea un fichero binario compuesto por una secuencia de pares (Nmat, nota) solicitados por teclado.

```
import java.io.*;
import java.util.*;

public class Binarios1 {

    public static void main(String args[]) {

        int nm;
        double nota;
        Scanner teclado = new Scanner(System.in); // para leer

        try {
            FileOutputStream fos=new FileOutputStream("notas.dat");
            DataOutputStream dos= new DataOutputStream(fos);
            System.out.print("Introduzca un NMat (0 para acabar): ");
            nm = teclado.nextInt();
            while (nm != 0) {
                dos.writeInt(nm);
                System.out.print("Introduzca una nota: ");
                nota = teclado.nextDouble();
                dos.writeDouble(nota);
                System.out.print("Introduzca un NMat (0 para acabar): ");
                nm = teclado.nextInt();
            }
            dos.close();
            fos.close();
        }
        catch (IOException ex) {
            System.out.println("Error: " + ex.getMessage());
        }
    }
}
```

La clase **DataInputStream** permite crear un objeto que se asocia a un objeto **FileInputStream** y facilita métodos para leer de él secuencialmente información codificada en binario.

La lectura de un nuevo dato se produce de forma secuencial, se lee la secuencia, información almacenada en el fichero detrás de la última información leída.

El programador debe gestionar la captura y tratamiento de IOException. El intento de lectura de un dato cuando ya ha sido leído todo el fichero lanza la excepción EOFException. Su gestión es clave en la programación de operaciones de lectura de un fichero binario.

Los **métodos** más usados son:

```
boolean readBoolean()
byte readByte()
char readChar()
int readInt()
double readDouble()
String readUTF()
```

Ej: Muestra el fichero binario compuesto por una secuencia de pares (NMAT, nota)

```
public class Binarios2 {

    public static void main(String args[]) {

        int nm;
        double nota;

        try {
            FileInputStream fis = new FileInputStream("notas.dat");
            DataInputStream dis = new DataInputStream(fis);
            System.out.println(" NMAT      Nota");
            try {
                while (true){
                    nm = dis.readInt();
                    nota = dis.readDouble();
                    System.out.printf("%6d %5.1f \n", nm, nota);
                    //nº matrícula ocupa: 6, nota: 5 enteros y 1 decimal
                }
            }catch (EOFException ex) {
                System.out.println("FINAL DE FICHERO " +
                    ex.getMessage());
            }
            dis.close();
            fis.close();
        }
        catch (IOException ex) {
            System.out.println("Error: " + ex.getMessage());
        }
    }
}
```

También podríamos haber usado para leer el fichero la condición: while (dis.available() > 0)

Evitando leer el fin de fichero

7. Ficheros de acceso aleatorio- acceso directo (binarios)

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/RandomAccessFile.html>
implementa autoclosable (se puede poner en un try con recursos)
Hasta ahora los archivos se han leído o escrito secuencialmente, es decir, desde el inicio hasta el final; pero es posible, leer datos o escribir datos en una zona concreta del archivo.

La clase **RandomAccessFile** permite acceder al archivo en forma aleatoria. Es decir, se permite leer o escribir cualquier posición del archivo en cualquier momento. Esta clase implementa las **interfaces DataInput y DataOutput** que sirven para leer y escribir datos.

La construcción requiere de una cadena que contenga un archivo File y un segundo parámetro obligatorio denominado **modo**, que va a especificar el tipo de operación que se realizará sobre el fichero. El modo es una cadena que puede contener una **r** (lectura), **w** (escritura) o ambas, **rw**.

```
java.lang.Object  
java.io.RandomAccessFile
```

Ej:

```
File f=new File("prueba.txt");  
RandomAccessFile archivo = new RandomAccessFile(f, "rw");
```

Como ocurría en las clases anteriores, hay que capturar la excepción. Los métodos fundamentales son:

void seek(long pos). Permite colocar el puntero en una posición concreta, contada en bytes, en el archivo. Lo que se coloca es la señal que marca la posición a leer o escribir.

long getFilePointer(). Posición actual del puntero de acceso

long length(). Devuelve el tamaño del archivo

int skipBytes(int desplazamiento). Desplaza el puntero desde la posición actual, el número de bytes indicado por desplazamiento

Los **métodos de lectura**, leen un dato del tipo indicado y son equivalentes a los disponibles en la clase **DataInputStream**.

readBoolean, readByte, readInt, readDouble, readFloat, readUTF, readLine.

Los **métodos de escritura**, reciben como parámetro el dato a escribir y escriben encima de lo hubiese ya escrito, por lo que para añadir hay que colocar el puntero de acceso al final del archivo.

writeBoolean, writeByte, writeBytes, writeInt, writeDouble, writeFloat, writeUTF, writeLine.

Ej: Crear un fichero binario, de acceso aleatorio, con los 100 primeros números y mostrarlo en pantalla.

```
import java.io.*;
public class Aleatorio{
    public static void main(String args[]) {
        File f = new File("prueba.dat");
        RandomAccessFile raf=null;
        try {
            raf = new RandomAccessFile(f,"rw");
            for ( int i=1; i <= 100 ; i++ )
                raf.writeInt( i );
            // un entero ocupa 4 bytes; total 400 bytes
            // total 400 bytes
            try{
                System.out.println( " El fichero ocupa " +
                    raf.length() + " bytes." );
                raf.seek(0); // La primera posición empieza en 0
                System.out.print(" En la posicion " +
                    raf.getFilePointer());
                System.out.println(" está el número " +
                    +raf.readInt() );
                raf.skipBytes( 9*4 );
                // Salto 9 => Elemento 10 más allá
                // ya ha leído la 1era, así que salto 10
                System.out.print(" 10 elementos más allá, está el ");
                System.out.println(raf.readInt());
                raf.close();
            }
            catch(IOException e){
                System.out.println();
            }
        }catch (FileNotFoundException e) {
            System.out.println();
        }
        catch(IOException e){
            System.out.println();
        }
    }
}
```

Ej: Crea un fichero con los 10 primeros números enteros y después modificarlo multiplicando por 2 las componentes pares (valores escritos en el fichero).

```
import java.io.*;  
  
public class Aleatorios2{  
  
    public static void main(String args[]) {  
  
        File f = new File("prueba.dat");  
        RandomAccessFile raf=null;  
  
        try {  
            raf = new RandomAccessFile(f, "rw");  
  
            for ( int i=1; i <= 10 ; i++ )  
                raf.writeInt( i );  
  
            try{  
                raf.seek(0); // La primera posición empieza en 0  
  
                while(true){  
                    int valor=raf.readInt();  
                    System.out.println(valor);  
                    if(valor%2==0){  
                        valor=valor*2;  
                    System.out.println("Escribo "+valor+  
                        " En la posicion " +  
                        ((raf.getFilePointer()-4)/4 + 1));  
                    // restamos lo que ocupa un int: 4  
                    raf.seek(raf.getFilePointer()-4);  
                    //Me posiciono 4 bytes atrás  
                    raf.writeInt(valor);  
                }  
            }  
  
            }catch(EOFException e){  
  
            }  
  
        }  
        catch(IOException e){  
            System.out.println(e);  
        } finally {  
  
        try {  
            if (raf != null) {  
                raf.close();  
            }  
        } catch (IOException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

8. Serialización

Un objeto **serializable** es un objeto que se puede convertir en una secuencia de bytes. Es una forma automática de guardar y cargar el estado de un objeto.

Se basa en la interfaz **Serializable** (en el paquete `java.io`) que es la que permite esta operación. Si una clase implementa esta interfaz, sus objetos pueden ser guardados y restaurados directamente en un archivo.

Cuando se desea utilizar un objeto para ser almacenado con esta técnica, la clase debe implementar la interfaz **Serializable**, incluyendo en la cabecera de la clase **implements Serializable**. Esta interfaz no posee métodos, pero es un requisito obligado para hacer que un objeto sea serializable.

Las clases **ObjectInputStream** y **ObjectOutputStream** se encargan de realizar los procesos de lectura o escritura del objeto de o a un fichero. Son herederas de **InputStream** y **OutputStream** y son casi iguales a **DataInputStream**/**DataOutputStream** solo que añaden los métodos **readObject** y **writeObject** que son los que permiten utilizar directamente los objetos.

<code>java.lang.Object</code>	<code>java.lang.Object</code>
<code>java.io.OutputStream</code>	<code>java.io.InputStream</code>
<code>java.io. ObjectOutputStream</code>	<code>java.io. ObjectInputStream</code>

Escribir objetos al flujo de salida **ObjectOutputStream** es muy simple y requiere los siguientes **pasos**:

- Se crea el objeto de la clase serializable.
- Se crea un objeto de la clase **FileOutputStream**, con el nombre de fichero y opcionalmente, se le puede añadir el valor true si en el fichero se van a añadir datos.
- Se vincula el objeto de la clase **FileOutputStream** al del flujo de salida de la clase **ObjectOutputStream** que es el que va a procesar los datos.

Ej:

```
FileOutputStream fos      = new FileOutputStream("fichero.dat", true);
ObjectOutputStream oos  = new ObjectOutputStream(fos);
```

- Con el método **writeObject** se escriben los objetos al flujo de salida y los guarda en el archivo.
- Finalmente hay que cerrar los flujos.

Ejemplo:

```
import java.io.*;
public class Persona implements Serializable{

    String nombre;
    int edad;
    public Persona(String n,int e) {
        nombre=n;
        edad=e;
    }

    void CambiarEdad(int e){
        edad=e;
    }

    void Escribir(){
        System.out.println("Nombre "+nombre+" Edad "+edad);
    }
}
```

El **proceso de lectura** es similar al proceso de escritura y es muy simple. Los **pasos** a seguir son los siguientes:

- a) Se crea un flujo de entrada a disco, pasándole el nombre del archivo.
- b) El flujo de entrada `ObjectInputStream` es el que procesa los datos y se vincula a un objeto de la clase `FileInputStream`.
- c) El método `readObject` lee los objetos del flujo de entrada, en el mismo orden en el que ha sido escritos. Es necesario hacer un cast al tipo de la clase. Cuando no haya más datos que leer saltará la excepción `EOFException`.
- d) Finalmente, se cierra los flujos

Ejemplo: `ejemplos030_ficherosSerializable`

```
import java.io.*;
public class PruebaPersona {
    public static void main(String[] args) {
        Persona p1=new Persona("Pablo", 20);
        Persona p2=new Persona("Rosa", 35);
        try {
            FileOutputStream fos =
            new FileOutputStream("datos.obj");
            ObjectOutputStream oos=new ObjectOutputStream(fos);
            oos.writeObject(p1);
            oos.writeObject(p2);
            oos.close();
            fos.close();
        }
        catch (IOException ex) {
            System.out.println(ex);
        }
        try{
            FileInputStream fis= new FileInputStream("datos.obj");
            ObjectInputStream ois=new ObjectInputStream(fis);
            try{
                while(true){
```

```
        p1=(Persona)ois.readObject();
        System.out.println("Nombre "+p1.nombre+
                           " Edad "+p1.edad);
    }
} catch(EOFException e){ }
ois.close();
fis.close();
} catch (IOException ex) {
    System.out.println(ex);
} catch (ClassNotFoundException ex) {
//en el JDK buscar que excepciones provoca el metodo
    System.out.println(ex);
}
}
```

Cuando un atributo de una clase contiene información que no se desea que se pueda guardar, hay técnicas para protegerla.

Cuando dicha información es privada, el atributo tiene el modificador `private`, pero una vez que se ha enviado al flujo de salida cualquiera puede leerla en el archivo en disco o interceptarla en la red. El modo más simple de proteger la información sensible, como una contraseña es la de poner el modificador `transient` delante del atributo que la guarda.

La *salida* del programa es:

Nombre: Pepe
Cuenta: No disponible

Lo que indica que la información que contiene el atributo Cuenta y que tiene el modificador transient no se ha guardado en el archivo. En la reconstrucción del objeto con la información guardada en el archivo el dato Cuenta toma el valor null.

Cuando se quieren añadir registros al fichero surge un problema, pues cada vez que se comienza a escribir se añade una cabecera, por lo que al leer posteriormente los registros salta un error cuando encuentra esas cabeceras. Para solucionar este problema, tenemos dos posibilidades:

1.- Crear un fichero auxiliar en el que se copien todos los registros que se tienen y a continuación los que se quieren añadir. Posteriormente, se borrará el primer fichero y el auxiliar se renombrará con el nombre del primer fichero.

2.- Crear una nueva clase que herede de la clase ObjectOutputStream y que el método writeStreamHeader() lo sobrescriba dejando en blanco.

Ej: Usando la 1^a manera:

Ejemplo: ejemplos030_ficherosSerializable

```
import java.io.*;

public class PruebaPersona2 {
    public static void main(String[] args) {
        Persona p1;
        try{
            //Se crea el fichero con dos registros
            FileOutputStream fos= new FileOutputStream("datos.obj");
            ObjectOutputStream oos=new ObjectOutputStream(fos);
            p1=new Persona("Pepe",40);
            oos.writeObject(p1);
            p1=new Persona("Carmen",15);
            oos.writeObject(p1);
            oos.close();
            fos.close();
        }catch (IOException ex) {
            System.out.println(ex);
        }

        try{
            //Se añaden dos registros más usando el primer método.
            File f=new File("auxiliar.obj");
            FileOutputStream fos= new FileOutputStream(f);
            ObjectOutputStream oos=new ObjectOutputStream(fos);
            File f1=new File("datos.obj");
            FileInputStream fis= new FileInputStream(f1);
            ObjectInputStream ois=new ObjectInputStream(fis);
            try{
                while(true){
                    p1=(Persona)ois.readObject();
                    oos.writeObject(p1);
                }
            }
            catch(EOFException e){ }
            catch(ClassNotFoundException e){ }
            // Hemos leído los objetos del fichero datos.obj
            // y los hemos pasado
            // al nuevo fichero temporal
            //Añadimos dos nuevos objetos

            p1=new Persona("Juan",50);
            oos.writeObject(p1);
            p1=new Persona("Maria",35);
            oos.writeObject(p1);

            ois.close();
            fis.close();
            oos.close();
            fos.close();
            f1.delete();
            f.renameTo(f1);
            // auxiliar pasa a ser el fichero datos.obj
        }catch (IOException ex) {
            System.out.println(ex);
        }

        //Ahora leemos el fichero con todos los datos
        try{
            FileInputStream fis= new FileInputStream("datos.obj");
            ObjectInputStream ois=new ObjectInputStream(fis);
```

```
try{
    while(true) {
        p1=(Persona)ois.readObject();
        System.out.println("Nombre "+p1.nombre+
                           " Edad "+p1.edad);
    }
}
catch(EOFException e){ }
ois.close();
fis.close();
}
catch (IOException ex) {
    System.out.println(ex);
}
catch (ClassNotFoundException ex) {
    System.out.println(ex);
}
}
```

Ej: A ese fichero que ya tiene 4 registros se le van a añadir 2 más usando la 2^a manera.

Ejemplo: [ejemplos030_ficherosSerializable](#)

```
import java.io.*;

public class PruebaPersona3{
    public static void main(String[] args) {
        Persona p1;
        try{
            // Se añaden dos registros más usando el segundo método.
            // El fichero tiene que existir previamente para que esto
            // funcione
            File f=new File("datos.obj");
            ClaseOutput co;

            FileOutputStream fos= new FileOutputStream(f,true);
            co=new ClaseOutput(fos);
            //clase que hereda de ObjectOutputStream
            p1=new Persona("Pepi",53);
            co.writeObject(p1);
            p1=new Persona("Julia",17);
            co.writeObject(p1);
            co.close();
            fos.close();
        } catch (IOException ex) {
            System.out.println(ex);
        }

        try{
            //Se lee el fichero con todos los datos
            FileInputStream fis= new FileInputStream("datos.obj");
            ObjectInputStream ois=new ObjectInputStream(fis);
            try{
                while(true){
                    p1=(Persona) ois.readObject();
                    if(p1 instanceof Persona){ //aseguramos
                        p1.Escribir();
                    }
                }
            }catch(EOFException e){}
            ois.close();
            fis.close();
        }catch (IOException ex){
            System.out.println(ex);
        }catch (ClassNotFoundException ex){
            System.out.println(ex);
        }
    }
}
```

Primero se crea la clase que hereda de ObjectOutputStream y que reescribirá el método writeStreamHeader() dejándolo en blanco. Será la clase que se use para escribir en el fichero.

¿Qué haríamos para que el segundo fichero funcione tanto con un fichero vacío como un fichero con datos ya introducidos?

Recapitulando:

Flujos de bytes especiales

File streams:

Para escribir y leer datos en ficheros

Filter streams:

Permiten filtrar datos mientras se escriben o leen (Se construyen sobre otro flujo)

Permiten manipular tipos de datos primitivos.

Implementan las interfaces DataInput y DataOutput y pueden heredar de las clases FilterInputStream y FilterOutputStream.

El mejor ejemplo son las clases DataInputStream y DataOutputStream para leer y escribir datos de tipos básicos

Object streams:

Para escribir y leer objetos

Implementa lo que se denomina **serialización de objetos** (guardar un objeto con una representación de bytes).

Uso de filter streams

Para leer tipos de datos primitivos

Se puede utilizar un *DataInputStream*

```
FileInputStream ficheroEntrada = new FileInputStream("precios.cat");
DataInputStream entrada = new DataInputStream(ficheroEntrada);
double precio= entrada .readDouble();
entrada.close();
```

Para escribir tipos de datos primitivos

Se puede utilizar un *DataOutputStream*

```
FileOutputStream ficheroSalida = new
FileOutputStream("precios.cat");
DataOutputStream salida = new DataOutputStream(ficheroSalida);
salida.writeDouble(234.56);
salida.flush(); //Fuerza la escritura de datos
salida.close();
```

Métodos básicos de InputStream:

```
int read()
int read(byte cbuf[])
int read(byte cbuf[], int offset, int length)
```

Métodos básicos de OutputStream:

```
int write(int c)
int write(byte cbuf[])
int write(byte cbuf[], int offset, int length)
```

Los Streams se abren automáticamente al crearlos, pero es necesario cerrarlos explícitamente llamando al método **close()** cuando se dejan de usar.

PrintStream / PrintWriter se utilizan para escribir cadenas de texto.

DataInputStream / DataOutputStream se utilizan para escribir/leer tipos básicos (int, long, float,...).

Según el acceso a ficheros se utilizan unas clases u otras.

a) Acceso **Secuencial**: El más común. Puede ser:

a.1) Acceso **binario**: FileInputStream / FileOutputStream

a.2) Acceso a **caracteres** (texto): FileReader / FileWriter

b) Acceso **Aleatorio**: Se utiliza la clase RandomAccessFile

UT9 Cómo convertir un Array en una Lista en Java

1. Crear una lista vacía y añadir todos los elementos

Es un método muy trivial y obvio. Podemos crear una lista vacía, hacer un bucle con todos los elementos del array y añadirlos a la lista.

El siguiente ejemplo ilustra esto:

```
import java.util.stream.*;
import java.util.*;

public class MyClass {
    public static void main(String args[]) {
        String[] myArray = new String[] { "1", "2", "3" };
        List<String> myList = new ArrayList<>();
        for (int i=0; i<myArray.length; i++) {
            myList.add(myArray[i]);
        }
        System.out.println(myList);
    }
}
```

2. Usa Arrays.asList() para convertir un array en una lista en Java

Podemos usar el método incorporado que proporciona la clase Arrays para convertir un array en una lista - Arrays.asList(arr). Importamos java.util.* en el código.

El siguiente ejemplo lo ilustra.

```
import java.util.*;

public class MyClass {
    public static void main(String args[]) {
        String[] myArray = new String[] { "1", "2", "3" };
        List<String> myList = Arrays.asList(myArray);
        System.out.println(myList);
    }
}
```

Sin embargo, este método da como resultado una lista de tamaño fijo, y no podemos añadir más elementos a ella. Daría una excepción.

3. Usa new ArrayList<>(Arrays.asList(arr)) para convertir un Array en una lista en Java

Este método permite convertir un array en una lista, y añadir también puede añadir elementos adicionales a la lista resultante. El siguiente ejemplo ilustra esto:

```
import java.util.*;  
  
public class MyClass {  
    public static void main(String args[]) {  
        String[] myArray = new String[] { "1", "2", "3" };  
        List<String> myList =  
            new ArrayList<>(Arrays.asList(myArray));  
        System.out.println("After conversion from array to list: "  
                           + myList);  
        myList.add("4");  
        System.out.println("After adding a new element: "  
                           + myList);  
    }  
}
```

UT9 Resumen Ficheros en Java

1. La clase File

```
File(String nombre)
File(String directorio, String nombre)
File(File directorio, String nombre)
File fichero=new File(nombre)
```

Métodos:

```
exists()
getName()
length()
lastModified()
list()
delete()
```

2. Ficheros secuenciales de texto

BufferedReader y PrintWriter

Escritura

Sintaxis:

```
PrintWriter salida;
salida =new PrintWriter(new FileWriter(nombre))
FileWriter (nombre, añadir)
```

Excepciones que lanza el constructor FileWriter: IOException

Métodos:

```
println()
print()
close():IOException
```

Lectura

Sintaxis:

```
BufferedReader entrada;
entrada= new BufferedReader(new FileReader(nombre));
```

Excepciones que lanza el constructor FileReader: FileNotFoundException

Métodos:

```
readLine(): null
read(): -1
BufferedReader entrada=new BufferedReader(
new FileReader(nombre));
char car=(char) (entrada.read());
```

Excepciones que lanzan los métodos: IOException

```
close():IOException
```

Scanner y PrintWriter (a partir de la versión 5.0 de Java)

Escritura

Sintaxis:

```
PrintWriter salida;  
salida =new PrintWriter(new FileWriter(nombre))  
FileWriter (nombre, añadir)
```

Excepciones que lanza el constructor FileWriter: IOException

Métodos:

```
println()  
print()  
printf()  
close():IOException
```

Lectura

Sintaxis:

```
Scanner entrada;  
entrada= new Scanner (new FileReader(nombre)); o  
entrada=new Scanner (new File(nombre));  
entrada=new Scanner (nombre);
```

Excepciones que lanza el constructor: FileNotFoundException

Métodos:

```
useLocale (Locale.US)  
next (),nextLine()  
nextInt (), nextDouble (), nextFloat (),...  
hasNextInt (), hasNextDouble (), hasNextFloat (), ....  
close()
```

Escritura

BufferedWriter

```
BufferedWriter bw;  
bw= new BufferedWriter(new FileWriter("E:\\fichero1.txt"));
```

Métodos:

```
flush();  
write();  
newline();  
close();
```

3. Ficheros secuenciales binarios

Byte a byte

FileOutputStream y FileInputStream

Escritura

Sintaxis:

```
FileOutputStream salida;  
salida = new FileOutputStream(nombre);  
FileOutputStream(File Objeto_File) ;  
FileOutputStream(String nombre_fichero, boolean añadir);
```

Excepciones que lanza el constructor:

FileNotFoundException

Métodos:

```
write(int i): IOException  
close(): IOException
```

Lectura

Sintaxis:

```
FileInputStream entrada;  
entrada=new FileInputStream(nombre);  
entrada=FileInputStream(objeto_File);
```

Excepciones que lanza el constructor:

FileNotFoundException

Métodos:

```
read(): IOException  
close(): IOException
```

Datos pasados a byte

DataOutputStream y DataInputStream

Escrutura

Sintaxis:

```
DataOutputStream salida;  
salida=new DataOutputStream(new FileOutputStream(nombre));
```

```
DataOutputStream salida=new DataOutputStream  
(new BufferedOutputStream (new FileOutputStream(nombre)));
```

Excepciones que lanza el constructor: Las del FileOutputStream

Métodos:

```
writeInt(variable_tipo_entero)  
writeUTF(objeto_tipo_cadena)  
writeDouble(variable_tipo_doble)  
writeFloat(variable_tipo_float)  
writeChar(variable_tipo_carácter)  
writeBoolean, writeByte, writeLong, writeShort, etc.
```

Excepciones que lanzan: IOException

```
close():IOException
```

Lectura

Sintaxis:

```
DataInputStream entrada;  
entrada=new DataInputStream(new FileInputStream(nombre));
```

```
DataInputStream entrada=new DataInputStream(new  
BufferedInputStream(new FileInputStream(nombre)));
```

Excepciones que lanza el constructor: Las del FileInputStream

Métodos:

```
readChar(), readDouble(), readInt(), readFloat(), readUTF()  
readBoolean(), readByte(), readShort(), readLong(), available()  
etc.
```

Excepciones que lanzan los métodos: EOFException y IOException

```
close():IOException
```

NOTA: BufferedInputStream y BufferedOutputStream son similares a
BufferedReader y BufferedWriter

4. Ficheros de acceso directo

Sintaxis

```
RandomAccessFile(File objeto_fichero, String modo)  
RandomAccessFile(String nombre, String modo)
```

modo: r (read) y rw (read-write)

Excepciones que lanza el constructor: FileNotFoundException

Métodos:

```
void seek(long posición)  
long getFilePointer()  
int skipBytes(int desplazamiento)  
long length()
```

Excepciones que lanzan los métodos: IOException
close(): IOException

Escritura

```
RandomAccessFile salida;  
salida=new RandomAccessFile(nombre,"rw");
```

Métodos:

```
writeInt(entero), writeDouble(doble), writeBytes(cadena)  
writeUTF(String), etc
```

Excepciones que lanzan los métodos: IOException
close(): IOException

Lectura

```
RandomAccessFile entrada;  
entrada=new RandomAccessFile(nombre, "r");
```

Métodos:

```
readInt(), readDouble(), readUTF(), readFloat(), readShort(), etc.
```

Excepciones que lanzan los métodos: EOFException y IOException

Movimiento en un fichero:

```
posicion=n*l_registro;
```

5. Ficheros y objetos

```
class Ejemplo implements Serializable {  
--- Código para la clase Ejemplo ---  
}
```

Creación de un **stream** de objetos para salida:

```
ObjectOutputStream salida;  
salida=new ObjectOutputStream(new FileOutputStream(nombre));
```

Creación de un **stream** de objetos para entrada:

```
ObjectInputStream entrada;  
entrada=new ObjectInputStream(new FileInputStream(nombre));
```

Métodos:

writeObject (Objeto)
readObject ()

Ejemplo:

```
ObjectOutputStream salida;  
  
salida=new ObjectOutputStream (new FileOutputStream(nombre));  
salida.writeObject (obj1);  
  
ObjectInputStream entrada;  
entrada=new ObjectInputStream(new FileInputStream(nombre));  
obj2=(Ejemplo) entrada.readObject();
```

UT9 Ejercicios Ficheros de Texto

1.- Con un programa en java crea un fichero denominado parrafo.txt. Escribe en él las líneas que vaya introduciendo el usuario, terminando cuando escriba FIN.

2.- Con un programa en java lee el fichero de texto anterior parrafo.txt, y genera otro fichero que sea el inicial pero sin las vocales. Muestra por pantalla ambos ficheros.

3.- Crea manualmente dos ficheros f1.txt y f2.txt, que tendrán un solo carácter por línea y que estarán ordenados alfabéticamente, y escribe un programa en java que genere el fichero f3.txt que será la fusión de f1 y f2 manteniendo su ordenación. Escribir por pantalla el fichero resultante. (Supuestamente los 2 ficheros están ordenados)

Fichero1	Fichero2	Fichero3
A	B	A
F	D	B
G		D
		F
		G

4.- Haz programa en java que lea el fichero parrafo.txt y lo codifique sumándole a cada letra un valor que se pedirá por teclado, guardarlo en otro fichero denominado codificado.txt. Sacar por pantalla el nuevo fichero creado.

5.- Genera un fichero llamado nombres.txt en el que hay una lista de personas, con su nombre y apellidos (datos de cada persona en una línea diferente). Poner fin para terminar.

6.- Lee el fichero nombres.txt, generado en el ejercicio anterior, y calcula la longitud media de los nombres completos (nombre + apellido). Muestra el contenido de dicho fichero por pantalla de forma que para cada pareja de nombre y apellido se muestre su longitud y al final la media de las longitudes de los nombres+apellido.

7.- Como el 6, pero ahora calcula la longitud media de los nombres y apellidos por separado (se lean juntos en una línea y se separan usando el método split()).

UT9 Ejercicios Ficheros Binarios

1.- Crea una aplicación que pida por teclado la cantidad de números aleatorios enteros positivos que se van a guardar en un fichero binario. El rango de los números aleatorios estará entre 0 y 100, incluyendo el 100.

Cada vez que se ejecute la aplicación se añadirán números al fichero sin borrar los anteriores.

Muestra en pantalla el fichero creado, después de realizar las operaciones de escritura sobre él.

2.- Crea una aplicación que almacene los datos básicos de un vehículo: matrícula (String), marca (String), tamaño de depósito (double) y modelo (String) en un fichero binario.

Los datos se pedirán por teclado. Se irán añadiendo al fichero los datos de nuevos vehículos cada vez que ejecutemos la aplicación (no se sobrescriben los datos).

Cada vez que se modifica el fichero, se muestran por pantalla los datos de cada coche.

Utiliza la clase JOptionPane para leer los datos de entrada (showInputDialog) y mostrarlos (showMessageDialog).

3.- Realiza un programa que lea de teclado los siguientes datos y los almacene en un fichero binario llamado "datosbeca.bin".

- Nombre y apellido de un supuesto becario.
- Género (M-F)
- Edad (20-60)
- Número de suspensos del curso anterior (0-4).
- Residencia familiar (SI o NO)
- Ingresos anuales de la familia.

4.- Realiza un programa que, partiendo del fichero binario "datosbeca.bin", calcule la cuantía de la beca (en caso de que la haya). El total de la beca se calcula como sigue:

- Base fija de 1500€
- Si los ingresos anuales de la familia son menores o iguales a la media (12.000€), la beca se incrementa en 500€, en caso contrario no lleva complementos.
- Si la edad de la persona es inferior a 23 años, 200€ de gratificación, si es mayor no hay gratificación.
- Si no hay suspensos en el curso anterior, hay una gratificación de 500€, 1 suspenso 200€. Si hay 2 o más suspensos no hay beca.
- Si vive de alquiler (no residencia familiar), 1000€ más de gratificación.

Visualiza el nombre de cada uno de los becarios y su cuantía total (solo los que tienen beca).

UT9 Ejercicios Ficheros Serializables

1.- Realiza un programa en JAVA en el que se le pida al usuario notas de asignaturas y se guarden en un fichero cuando ya se hayan introducido todas las notas.

Posteriormente se leerá el fichero y se calculará la nota media del curso.

Cada una de las asignaturas será un objeto que se encuentra en una lista, y cuyos atributos serán el nombre y la nota.

El usuario establecerá el nombre de la asignatura mediante un método que pedirá el nombre de la asignatura. El atributo nota, también será el usuario quien lo introduzca mediante un método que controle que la nota tenga un valor entre 0 y 10.

Ejemplo de ejecución:

Por favor, introduzca el nombre de la asignatura: Programación.

Introduzca la nota de Programación: 8,5

¿Desea introducir otra asignatura? SI

Por favor, introduzca el nombre de la asignatura: Lenguajes de Marcas.

Introduzca la nota de Lenguajes de Marcas: 9

¿Desea introducir otra asignatura? SI

Por favor, introduzca el nombre de la asignatura: Bases de Datos

Introduzca la nota de Bases de Datos: 8

¿Desea introducir otra asignatura? NO

***** Notas almacenadas en la lista *****

..... Volcando la lista al fichero.....

***** Volcado finalizado con éxito *****

.....Leyendo el fichero (mostramos los datos) y calculando media.....

Su nota media del curso es de: 8,5

2.- Se trata de hacer una aplicación en Java que gestione los clientes de una empresa. Los datos de los clientes, se almacenarán en un fichero serializado, denominado clientes.dat.

Los datos que se almacenarán sobre cada cliente son:

- NIF.
- Nombre.
- Teléfono.
- Dirección.
- Deuda.

Mediante un menú se podrán realizar determinadas operaciones:

Añadir cliente. Esta opción pedirá los datos del cliente y añadirá el registro correspondiente en el fichero.

Listar clientes. Recorrerá el fichero mostrando los clientes almacenados en el mismo.

Buscar clientes. Pedirá al usuario el nif del cliente a buscar, y comprobará si existe en el fichero, mostrando sus datos.

Borrar cliente. Pedirá al usuario el nif del cliente a borrar, y si existe, lo borrará del fichero.

Borrar fichero de clientes completamente. Elimina del disco el fichero clientes.dat

Salir de la aplicación.

3. Escribe un programa que permita almacenar en un fichero serializable una **agenda telefónica**. Cada registro de la agenda deberá contener los siguientes datos: Nombre, Apellidos, Nº de Teléfono, E-mail.

El programa deberá permitir las siguientes operaciones, mediante un menú:

- a)** Leer los datos de un nuevo registro (dar de alta), almacenándolo en el fichero.
- b)** Buscar una persona de la agenda leyendo de teclado su nombre y apellidos. Se visualizarán los datos de la persona.
- c)** Modificar el teléfono y/o e_mail de una persona que se pedirá por teclado (por su nombre y apellido).
- d)** Eliminar una persona de la agenda telefónica dando su nombre y apellidos. Todas las personas que se eliminan del fichero binario, se guardarán en un fichero de texto, llamado Eliminados.txt, en el que se guardará un registro por línea.
- e)** Ordenar el fichero alfabéticamente por nombre y apellido
- f)** Mostrar el contenido del fichero
- g)** Finalmente, cuando se salga del menú, se escribirá en consola el contenido del fichero de texto Eliminados.txt.

UT9 Ejercicios Ficheros Aleatorios

1.- Utilizando las clases de acceso directo a ficheros, realiza un programa que le muestre al usuario un menú con las siguientes opciones:

- 1.- Añadir números de tipo double al principio del fichero.
- 2.- Añadir números de tipo double al final del fichero.
- 3.- Mostrar el fichero creado.
- 4.- Sustituir un número indicado por el usuario por otro número que también te indique el usuario.
- 5.- Ordenar los números en el fichero descendente (ayúdate de un ArrayList).

Nota: Un double en JAVA ocupa 8 bytes (en Java 8 existe un método para obtener el tamaño).

UT10 Acceso a Bases de Datos (BBDD)

Lo primero que se necesita para conectarnos con una base de datos es un **driver** o **Conector**. Ese driver es el elemento que, de alguna forma, sabe cómo hablar con la base de datos.

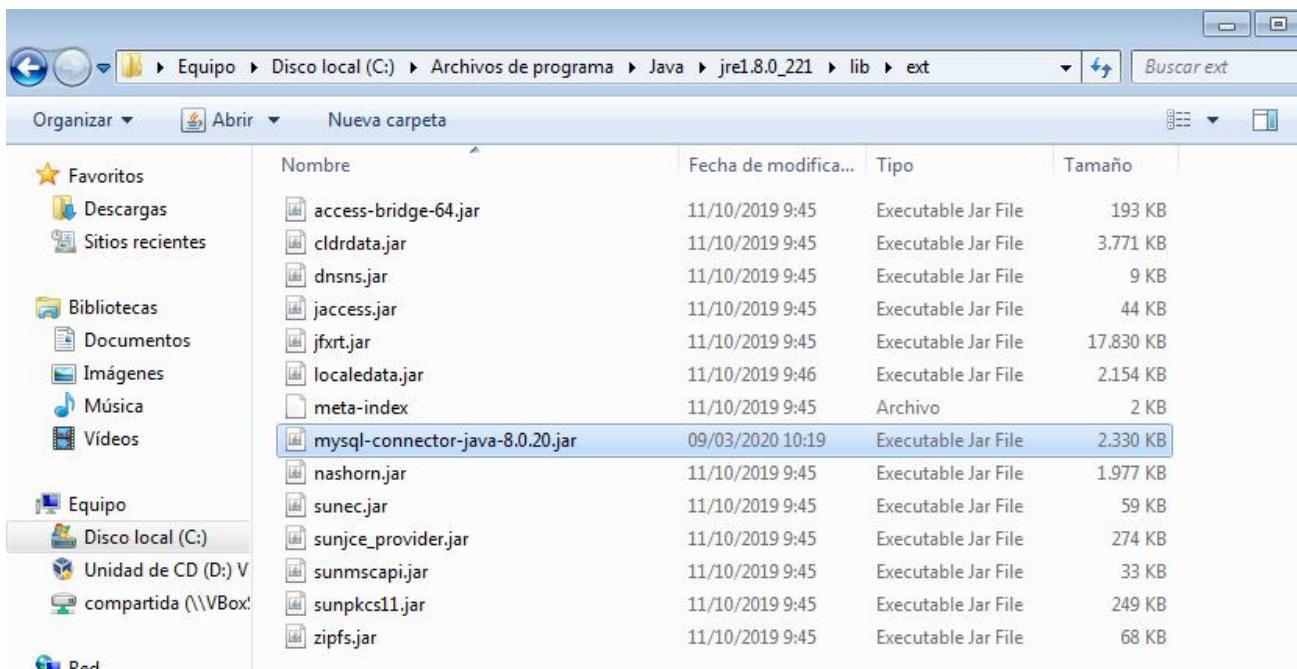
Según el gestor de base de datos que se utilice, se usará un **driver** u otro.

Para **MySQL** se usa ***mysql-connector-java-X.X.X-bin.jar***.

Para **Oracle** se usa ***ojdbc6.jar***.

Para poder conectar MySQL con Java vamos a utilizar la tecnología JDBC de Java. JDBC nos permite el acceso a los datos desde un programa Java.

El fichero que contiene el **driver mysql-connector-java-X.X.X.jar** se encontrará dentro del **classpath** de la aplicación, ya que iremos a dicho **classpath** para cargarlo y utilizarlo en nuestro programa.



1. Cargar el Driver MySQL

- Si estás utilizando una **versión anterior a JDBC 4**, es decir, si utilizas un compilador **inferior a JavaSE 7**, lo primero que necesitarás será cargar el driver. Para cargar el driver acudiremos a la clase `Class.forName()`, la cual recibirá como parámetro el nombre del driver.

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

- Si estás utilizando **Java SE 7 o superior** puedes evitar esta línea de código ya que el compilador cargará el Driver automáticamente por nosotros.

2. Conexión a MySQL desde JDBC

Una vez tengamos el Driver cargado deberemos conectarnos a la base de datos mediante la clase **Connection** y su método **.getConnection()**. Dicho método espera una **cadena de conexión** a la base de datos. La cadena de conexión a MySQL tiene la siguiente estructura:

```
jdbc:mysql://[host1]:[port1], [host2]:[port2] / [databaseName] ? [property1=value1] & [property2=value2]
```

Veremos que el segundo y tercer parámetro es el usuario/password de la conexión.

En resumen:

Los **pasos** a seguir para trabajar con la base de datos desde Java son:

1º.- Importar el paquete java.sql.*

2º.- Cargar el driver para lo cual se pone Class.forName("Nombre del driver") si es necesario.

Para el caso del MySql, será: Class.forName("com.mysql.jdbc.Driver");
import java.sql.DriverManager

3º.- Conectarse a la base de datos invocando el método getConnection().

La **sintaxis** es:

```
Connection conn = DriverManager.getConnection(url, usr, pwd);
```

Donde **url** es una cadena compuesta por el protocolo + driver + lugar donde se encuentra la base de datos + nombre de la bbdd, **usr** es el usuario para acceder a la bbdd y **pwd** es la clave de acceso.

Para el caso de MySql sería (si no hemos puesto ninguna contraseña):

```
url: "jdbc:mysql://localhost/Nombre_de_la_Base_de_Datos"  
usuario:"root"  
password:""
```

Todas estas instrucciones estarán recogidas dentro de un try-catch para que recoja la excepción en caso de no poder realizarse la conexión, tener un error de sql o cualquier otra excepción.

Ejemplo:

```
try{  
    Class.forName("com.mysql.jdbc.Driver");  
    Connection conn=  
    DriverManager.getConnection("jdbc:mysql://localhost/bbdd","root","");
    .....  
}catch(ClassNotFoundException cnfe){  
    System.out.println("Driver JDBC no encontrado");  
}catch(SQLException sqle){  
    System.out.println("Error al conectarse a la BD");  
}catch(Exception e){  
    System.out.println("Error general");
```

4º.- Una vez creada la conexión se crean las sentencias SQL, a través de los métodos execute(), executeUpdate() y executeQuery() de la interface Statement.

Statement st=conn.createStatement();

El método **execute()** se utiliza para las operaciones **DDL** sobre tablas (**Create, Alter o Drop**).

El método **executeUpdate()** se utiliza para operaciones **DML**, sobre los registros (**Update, Insert o Delete**). Devuelve un número entero que indica la cantidad de registros afectados.

El método **executeQuery()** se utiliza para las consultas **SELECT**. Devuelve un conjunto de registros que se almacenan en un objeto **ResultSet**.

Ejemplo **execute**:

```
st.execute("CREATE TABLE Libros (id NUMBER(11) primary key, title  
VARCHAR2(64));");
```

Ejemplo **executeUpdate**:

```
int nr;  
String cnssSQL;  
cnssSQL="INSERT INTO autor VALUES(127,'Peter','Norton');";  
nr=st.executeUpdate(cnssSQL);  
cnssSQL="UPDATE autor SET nombre='Pedro' WHERE nombre='Peter'";  
nr=st.executeUpdate(cnssSQL);
```

Ejemplo **executeQuery**:

```
ResultSet rs=st.executeQuery("SELECT * FROM autor");
```

5º.- ResultSet también tiene los métodos **next()** y **getXXX()**.

next() recorre los registros devueltos.

getXXX() recupera los valores de las columnas por su nombre o posición.

Ejemplo:

```
while(rs.next()) {  
    System.out.println("Autor "+rs.getString(2));  
}
```

Cuando se lanza un método **getXXX** sobre un objeto **ResultSet**, el driver JDBC convierte el dato que se quiere recuperar al tipo Java especificado y entonces devuelve un valor Java adecuado. La conversión de tipos se puede realizar gracias a la clase `java.sql.Types`. En esta clase se definen lo que se denominan tipos de datos JDBC, que se corresponde con los tipos de datos SQL.

6º.- Por último habrá que cerrar todas las conexiones con el método **close()**.

```
st.close();  
conn.close();
```

3. Correspondencia datos Java-SQL:

SQL	Java
BIGINT	getLong()
BINARY	getBytes()
BIT	getBoolean()
CHAR	getString()
DATE	getDate()
DECIMAL	getBigDecimal()
DOUBLE	getDouble()
FLOAT	getDouble()
INTEGER	getInt()
LONGVARBINARY	getBytes()

SQL	Java
LONGVARCHAR	getString()
NUMERIC	getBigDecimal()
OTHER	getObject()
REAL	getFloat()
SMALLINT	getShort()
TIME	getTime()
TIMESTAMP	getTimestamp()
TINYINT	getByte()
VARBINARY	getBytes()
VARCHAR	getString()

Ejemplo:

```
import java.sql.*;  
  
public class Programa {  
    public static void main(String args[]) {  
        try {  
            Class.forName("com.mysql.jdbc.Driver");  
            Connection conexion;  
            conexion=DriverManager.getConnection  
                ("jdbc:mysql://localhost/ejemplo","root","");
            Statement instruccion = conexion.createStatement();
            ResultSet tabla;
            tabla = instruccion.executeQuery("SELECT cod , nombre  
                FROM datos");
            System.out.println("Codigo\tNombre");
              
            while(tabla.next())
                System.out.println(tabla.getInt(1)+"\t"+
                    tabla.getString(2));
              
            tabla.close();
            instruccion.close();
            conexion.close();
        }catch(ClassNotFoundException e) {
            System.out.println(e);
        }catch(SQLException e) {
            System.out.println(e);
        }catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Probamos a usar el try con recursos para que se cierre automáticamente la base de datos.

CONSIDERACIONES

- Si la sentencia SQL ejecutada con el método executeQuery() no devolviera un conjunto de registros resultado (p.ej.INSERT INTO...), obtendríamos una excepción SQLException.
- Por defecto, un objeto Statement tiene solamente un ResultSet abierto. Por tanto, si queremos tener más de un ResultSet abierto simultáneamente, deberemos utilizar distintos objetos Statement's.

"Esto del MetaData no lo vamos a ver"

Existe un objeto **ResultSetMetaData** que proporciona varios métodos para obtener información sobre los datos que están dentro de un objeto ResultSet. Estos métodos permiten entre otras cosas obtener de manera dinámica el número de columnas en el conjunto de resultados, así como el nombre y el tipo de cada columna.

NOTA:

La última versión del driver de mySQL, da un error si realizamos la conexión como se ha indicado antes. Debemos añadir la zona horaria. ([comprobar](#))

```
conexion=DriverManager.getConnection  
("jdbc:mysql://localhost/Prueba?useUnicode=true&use"+  
"JDBCCCompliantTimezoneShift&useLegacyDatetimeCode=false&ServerTimez  
one=UTC","root","");
"");
```

4. La interfaz PreparedStatement

La interfaz **PreparedStatement** hereda de **Statement** y difiere de esta en dos maneras:

- Las instancias de **PreparedStatement** contienen una sentencia SQL que ya ha sido compilada. Esto es lo que hace que se le llame 'preparada'.
- La sentencia SQL contenida en un objeto **PreparedStatement** puede tener uno o más parámetros IN. Un parámetro IN es aquel cuyo valor no se especifica en la sentencia SQL cuando se crea. En vez de ello, la sentencia tiene un interrogante ('?') como un 'ancla' para cada parámetro IN. Debes suministrar un valor para cada interrogante mediante el método apropiado, que puede ser: **setInt**, **setString**, etc, antes de ejecutar la sentencia.

Un objeto PreparedStatement se usa para sentencias SQL que toman uno o más parámetros como argumentos de entrada (parámetros IN).

Conviene que nos aseguremos de que el servidor está capacitado para trabajar con sentencias precompiladas:

```
Connection conexion =  
    DriverManager.getConnection(  
        "jdbc:mysql://servidor/basedatos?useServerPrepStmts=true",  
        "usuario", "password");
```

PreparedStatement tiene un **grupo de métodos** que fijan los valores de los parámetros IN, los cuales son enviados a la base de datos cuando se procesa la sentencia SQL.

Las Instancias de PreparedStatement extienden, es decir, heredan de Statement y por tanto heredan los métodos de Statement. Un objeto PreparedStatement es potencialmente más eficiente que un objeto Statement porque este ha sido precompilado y almacenado para su uso futuro.

Son muy útiles cuando una sentencia SQL se ejecuta muchas veces cambiando sólo algunos valores.

- Se utiliza para enviar sentencias SQL precompiladas con uno o más parámetros.
(El sistema gestor de BBDD es un pequeño sistema operativo.)
- Se crea un objeto PreparedStatement especificando la plantilla y los lugares donde irán los parámetros.

- Los parámetros son especificados después utilizando los métodos **setXXX(.)** indicando el número de parámetro y el dato a insertar en la sentencia.
- La sentencia SQL y los parámetros se envían al gestor base de datos cuando se llama al método: **executeXXX()**

Los objetos PreparedStatement contienen órdenes SQL **precompiladas** que pueden por tanto ejecutarse muchas veces.

Ejemplos:

El siguiente fragmento de código, donde **conexión** es un objeto Connection, crea un objeto **PreparedStatement** que contiene una instrucción SQL:

```
//Creamos un objeto PreparedStatement desde el objeto Connection
PreparedStatement ps = conexion.prepareStatement(
"select * from Propietarios where DNI=? AND NOMBRE=? AND EDAD=?");

///"Seteamos" los datos al prepared statement de la siguiente forma
//(los segundos parámetros son variables con valores):

ps.setString(1, dni); // 1 1era interrogación, 2 2da ?...
ps.setString(2, nombre);
ps.setInt(3, edad);

//Ejecutamos el PreparedStatement, en este caso con executeQuery():

ResultSet rs= ps.executeQuery();
// no se le manda la sentencia, la sentencia va asociada al ps
```

Ejemplo de PreparedStatement de consulta. Por ejemplo, supongamos que hay un campo de texto en el que el usuario puede introducir su dirección de correo electrónico y con este dato se desea buscar al usuario:

```
Connection con = DriverManager.getConnection(url);

String consulta = "SELECT usuario FROM registro WHERE email like ?";
PreparedStatement pstmt = con.prepareStatement(consulta);
pstmt.setString(1 , campoTexto.getText());

ResultSet resultado = pstmt.executeQuery();
```

Ejemplo de PreparedStatement de modificación. En el siguiente ejemplo se va a insertar un nuevo registro en una tabla:

```
Connection con = DriverManager.getConnection(url);

String insercion = "INSERT INTO registro(usuario , email , fechaNac)
values ( ? , ? , ? )";
PreparedStatement pstmt = con.prepareStatement(insercion);

String user = . . . ;
String email = . . . ;
Date edad = . . . ; //O int edad;
pstmt.setString(1, user);
pstmt.setString(2, email);
pstmt.setDate(3, edad); // setInt(3, edad);
pstmt.executeUpdate();
```

Como los objetos **PreparedStatement** están precompilados, su ejecución es más rápida que los objetos **Statement**. Consecuentemente, una sentencia SQL que se ejecute muchas veces a menudo se crea como **PreparedStatement** para incrementar su eficacia.

Siendo una subclase de **Statement**, **PreparedStatement**, como ya se ha mencionado hereda toda la funcionalidad de **Statement**. Además, se añade un conjunto completo de métodos necesarios para fijar los valores que van a ser enviados a la base de datos en el lugar de las ‘anclas’ para los parámetros IN.

También se modifican los tres métodos **execute**, **executeQuery** y **executeUpdate** de tal forma que **no toman argumentos**. Los formatos de **Statement** de estos métodos (los formatos que toman una sentencia SQL como argumento) no deberían usarse nunca con objetos **PreparedStatement**, a pesar de estar disponibles para ellos.

5. Los métodos de **PreparedStatement**

Ejecución de órdenes: **executeQuery()**

```
public abstract ResultSet executeQuery() throws SQLException
```

Devuelve la tabla resultado de ejecutar la orden precompilada. No devuelve nunca un valor null.

Ejecución de órdenes de actualización: **executeUpdate()**

```
public abstract int executeUpdate() throws SQLException
```

Devuelve el número de filas afectadas por órdenes del tipo INSERT, UPDATE, DELETE, o sentencias de definición de datos.

Parámetros nulos: **setNull()**

```
public abstract void setNull(int indiceParametro, int tipoSQL) throws SQLException
```

Asigna un valor nulo a un parámetro. No puede omitirse el tipo SQL.

Asignación de valores a los parámetros: setXXX()

```
public abstract void setXXX(int indiceParametro, tipoJava valor) throws  
SQLException
```

Asigna el valor valor del tipo Java tipoJava al parámetro de índice indiceParametro, que es transformado por el controlador JDBC en un tipo SQL correspondiente para pasarlo a la base de datos.

Método setXXX()	Tipo Java	Tipo SQL
setBoolean	boolean	BIT
setByte	byte	TINYINT
setShort	short	SMALLINT
setInt	int	INTEGER
setLong	long	BIGINT
setFloat	float	FLOAT
setDouble	double	DOUBLE
setBigInt	BigInt	NUMERIC
setString	String	VARCHAR o LONGVARCHAR
getBytes	Array de bytes	VARBINARY o LONGVARBINARY
 setDate	Date	DATE
 setTime	Time	TIME
 setTimestamp	Timestamp	TIMESTAMP
 setAsciiStream*	InputStream	LONGVARCHAR
 setUnicodeStream*	InputStream	LONGVARCHAR
 setByteStream*	InputStream	LONGVARBINARY

```
public abstract void setXXX(int indiceParametro, inputStream corriente,  
int longitud) throws SQLException
```

Se usan para valores muy grandes ASCII, UNICODE o binarios.

Borrado de parámetros: clearParameters()

```
public abstract void clearParameters() throws SQLException
```

Borra todos los parámetros.

Asignación de objetos: setObject()

```
void setObject(int indiceParametro, Object objeto [, int tiposQL][, int decimales]) throws SQLException
```

Asigna un objeto a un parámetro para ser convertido al tipo SQL especificado.

Si el tipo SQL no se especifica, se emplean las tablas de conversión del estándar JDBC para entregarlo a la base de datos como el tipo adecuado.

Ordenes con múltiples resultados: execute()

```
public abstract boolean execute() throws SQLException
```

Se emplea para enviar órdenes que pueden devolver múltiples resultados.

Nota: El índice de los parámetros comienza en 1.

<http://tutorials.jenkov.com/jdbc/preparedstatement.html>

6. Anexo:

¿Qué es un DAO?

Cuando se pretende modelar con objetos de un modelo de datos, es decir las tablas y sus relaciones, es muy común utilizar una propuesta simple que consiste en armar una clase por cada tabla existente.

Dicha clase tendrá los métodos de acceso a la tabla correspondiente, entre ellos la inserción, modificación y la eliminación. Este tipo de clases suelen conocerse como DAOs, ya que objetos de esta clase se utilizarán para realizar operaciones de datos.

De esta forma, si existe la tabla autos, es posible construir una clase denominada Auto, que siguiendo la especificación de un DAO, se debería tener los métodos insertar, modificar, eliminar y métodos con las consultas que resulten necesarias.

<https://www.ecodeup.com/patrones-de-diseno-en-java-mvc-dao-y-dto/>

UT10 Aclaración sobre el uso de los métodos Execute de JDBC

1. JDBC

Diferencia entre execute, executeQuery y executeUpdate en JDBC.

El método **execute** se puede usar con cualquier tipo de sentencia SQL, devuelve un boolean. Un true indica que devuelve un objeto result set, cuyos datos se pueden recuperar usando el método `getResultSet()`. Un false indica que la sentencia devuelve un int o nada. El método `execute()` se puede usar con sentencias select y insert/update.

El método **executeQuery** ejecuta sentencias que devuelven un result set con datos extraídos de la base de datos. Solo sirve para sentencias select.

El método **executeUpdate** ejecuta sentencias SQL insertan/actualizan/borran datos en la base de datos. Este método devuelve un valor entero con el número de registros afectados. Devuelve 0 si la sentencia no devuelve nada. Es válido para sentencias que no son select.

UT10 Iniciación MySQL - Tutorial de MySQL

MySQL es un gestor de base de datos rápido y sencillo de usar. También **es uno de los motores de base de datos más usados en Internet**, la principal razón de esto es que **es gratis para aplicaciones no comerciales**.

Las características principales de MySQL son:

Es un gestor de base de datos. Una base de datos es un conjunto de datos y un gestor de base de datos es una aplicación capaz de manejar este conjunto de datos de manera eficiente y cómoda.

- **Es una base de datos relacional.** Una base de datos relacional es un conjunto de datos que están almacenados en tablas entre las cuales se establecen unas relaciones para manejar los datos de una forma eficiente y segura. Para usar y gestionar una base de datos relacional se usa el lenguaje estándar de programación SQL.
- **Es Open Source.** El código fuente de MySQL se puede descargar y está accesible a cualquiera. Por otra parte, usa la licencia GPL para aplicaciones no comerciales.
- **Es una base de datos muy rápida, segura y fácil de usar.** Gracias a la colaboración de muchos usuarios, la base de datos se ha ido mejorando optimizándose en velocidad. Por eso es una de las bases de datos más usadas en Internet.
- **Existe una gran cantidad de software que la usa.**

El objetivo de este tutorial es mostrar el uso del programa cliente mysql para crear y usar una sencilla base de datos. MySQL (algunas veces referido como "monitor mysql") es un programa interactivo que permite conectarnos a un servidor MySQL, ejecutar algunas consultas, y ver los resultados. MySQL puede ser usado también en modo batch: es decir, se pueden colocar toda una serie de consultas en un archivo, y posteriormente decirle a MySQL que ejecute dichas consultas.

Este tutorial asume que MySQL está instalado en alguna máquina y que disponemos de un servidor MySQL al cual podemos conectarnos.

Para ver la lista de opciones proporcionadas por mysql, lo invocamos con la opción --help:

```
shell> mysql --help
```

También nos podemos conectar a mysql y después teclear help. A continuación, se describe el proceso completo de creación y uso de una base de datos en MySQL.

Para conectarse al servidor, usualmente necesitamos de un nombre de usuario (login) y de una contraseña (password), y si el servidor al que nos deseamos conectar está en una máquina diferente de la nuestra, también necesitamos

indicar el nombre o la dirección IP de dicho servidor. Una vez que conocemos estos tres valores, podemos conectarnos de la siguiente manera:

```
shell> mysql -h NombreDelServidor -u NombreDeUsuario -p
```

Cuando ejecutamos este comando, se nos pedirá que proporcionemos también la contraseña para el nombre de usuario que estamos usando.

Nosotros vamos a usar **XAMPP** para trabajar en clase con MySql, por defecto no tenemos password para el usuario root, pero podemos cambiarlo a posteriori. Si la conexión al servidor MySQL se pudo establecer de manera satisfactoria, recibiremos el mensaje de bienvenida y estaremos en el prompt de mysql:

```
shell>mysql -h casita -u blueman -p  
Enter password: *****
```

```
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 5563 to server version: 3.23.41
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

```
mysql>
```

Este prompt nos indica que mysql está listo para recibir comandos.

Algunas instalaciones permiten que los usuarios se conecten de manera anónima al servidor corriendo en la máquina local. Si es el caso de nuestra máquina, debemos de ser capaces de conectarnos al servidor invocando a mysql sin ninguna opción:

```
shell> mysql
```

Después de que nos hemos conectado de manera satisfactoria, podemos desconectarnos en cualquier momento al escribir "quit", "exit", o presionar CONTROL+D.

La mayoría de los ejemplos siguientes asume que estamos conectados al servidor, lo cual se indica con el prompt de mysql.

En este momento debemos de haber podido conectarnos ya al servidor MySQL, aun cuando no hemos seleccionado alguna base de datos para trabajar. Lo que haremos a continuación es escribir algunos comandos para irnos familiarizando con el funcionamiento de mysql.

```
mysql> SELECT VERSION(), CURRENT_DATE;  
+-----+-----+  
| VERSION() | CURRENT_DATE |  
+-----+-----+  
| 3.23.41 | 2002-10-01 |  
+-----+-----+  
1 row in set (0.03 sec)
```

```
mysql>
```

Este comando ilustra distintas cosas acerca de mysql:

- Un comando normalmente consiste en una sentencia SQL seguida por un punto y coma.
- Cuando emitimos un comando, mysql lo manda al servidor para que lo ejecute, nos muestra los resultados y vuelve a poner el prompt indicando que está listo para recibir más consultas.
- mysql muestra los resultados de la consulta como una tabla (filas y columnas). La primera fila contiene etiquetas para las columnas. Las filas siguientes muestran los resultados de la consulta. Normalmente, las etiquetas de las columnas son los nombres de los campos de las tablas que estamos usando en alguna consulta. Si lo que estamos recuperando es el valor de una expresión (como en el ejemplo anterior) las etiquetas en las columnas son la expresión en sí.
- mysql muestra cuántas filas fueron devueltas y cuánto tiempo tardó en ejecutarse la consulta, lo cual puede darnos una idea de la eficiencia del servidor, aunque estos valores pueden ser un tanto imprecisos ya que no se muestra la hora de la CPU, y porque pueden verse afectados por otros factores, tales como la carga del servidor y la velocidad de comunicación en una red.
- Las palabras clave pueden ser escritas usando mayúsculas y minúsculas.

Las siguientes consultas son equivalentes:

```
mysql> SELECT VERSION(), CURRENT_DATE;
mysql> select version(), current_date;
mysql> SeLeCt vErSiOn(), current_DATE;
```

Aquí está otra consulta que demuestra cómo se pueden escribir algunas expresiones matemáticas y trigonométricas:

```
mysql> SELECT SIN(PI()/4), (4+1)*5;
+-----+-----+
| SIN(PI()/4) | (4+1)*5 |
+-----+-----+
| 0.707107 | 25 |
+-----+-----+
```

Aunque hasta este momento se han escrito sentencias sencillas de una sola línea, es posible escribir más de una sentencia por línea, siempre y cuando estén separadas por punto y coma:

```
mysql> SELECT VERSION(); SELECT NOW();
+-----+
| VERSION() |
+-----+
| 3.23.41 |
+-----+
1 row in set (0.01 sec)

+-----+
| NOW() |
+-----+
| 2002-10-28 14:26:04 |
+-----+
1 row in set (0.01 sec)
```

Un comando no necesita ser escrito en una sola línea, así que los comandos que requieran de varias líneas no son un problema. mysql determinará en dónde finaliza la sentencia cuando encuentre el punto y coma, no cuando encuentre el fin de línea.

Aquí está un ejemplo que muestra una consulta simple escrita en varias líneas:

```
mysql> SELECT
-> USER(),
-> CURRENT_DATE;
+-----+-----+
| USER() | CURRENT_DATE |
+-----+-----+
| blueman@localhost | 2002-09-14 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

En este ejemplo debe notarse como cambia el prompt (de mysql> a ->) cuando se escribe una consulta en varias líneas. Esta es la manera en cómo mysql indica que está esperando a que finalice la consulta. Sin embargo si deseamos no terminar de escribir la consulta, podemos hacerlo al escribir \c como se muestra en el siguiente ejemplo:

```
mysql> SELECT
-> USER(),
-> \c
mysql>
```

De nuevo, volvemos al comando prompt mysql> que nos indica que mysql está listo para una nueva consulta.

Prompt	Significado
mysql>	Listo para una nueva consulta.
->	Esperando la línea siguiente de una consulta multi-línea.
'>	Esperando la siguiente línea para completar una cadena que comienza con una comilla sencilla (').
">	Esperando la siguiente línea para completar una cadena que comienza con una comilla doble (").

Los comandos multi-línea comúnmente ocurren por accidente cuando tecleamos ENTER, pero olvidamos escribir el punto y coma. En este caso mysql se queda esperando para que finalicemos la consulta:

```
mysql> SELECT USER()
->
```

Si esto llega a suceder, muy probablemente mysql estará esperando por un punto y coma, de manera que si escribimos el punto y coma podremos completar la consulta y mysql podrá ejecutarla:

```
mysql> SELECT USER()
-> ;
+-----+
| USER()           |
+-----+
| root@localhost |
+-----+
1 row in set (0.00 sec)

mysql>
```

Los prompts '>' y ">" ocurren durante la escritura de cadenas. En mysql podemos escribir cadenas utilizando comillas sencillas o comillas dobles (por ejemplo, 'hola' y "hola"), y mysql nos permite escribir cadenas que ocupen múltiples líneas. De manera que cuando veamos el prompt '>' o ">" , mysql nos indica que hemos empezado a escribir una cadena, pero no la hemos finalizado con la comilla correspondiente.

Aunque esto puede suceder si estamos escribiendo una cadena muy grande, es más frecuente que obtengamos alguno de estos prompts si inadvertidamente escribimos alguna de estas comillas.

Por ejemplo:

```
mysql> SELECT * FROM mi_tabla WHERE nombre = "Lupita AND edad < 30;
">
```

Si escribimos esta consulta SELECT y entonces presionamos ENTER para ver el resultado, no sucederá nada. En lugar de preocuparnos porque la consulta ha tomado mucho tiempo, debemos notar la pista que nos da mysql cambiando el prompt. Esto nos indica que mysql está esperando que finalicemos la cadena iniciada ("Lupita).

En este caso, ¿qué es lo que debemos hacer?. La cosa más simple es cancelar la consulta. Sin embargo, no basta con escribir \c, ya que mysql interpreta esto como parte de la cadena que estamos escribiendo. En lugar de esto, debemos escribir antes la comilla correspondiente y después \c :

```
mysql> SELECT * FROM mi_tabla WHERE nombre = "Lupita AND edad < 30;  
"> "\c  
mysql>
```

El prompt cambiará de nuevo al ya conocido mysql>, indicándonos que mysql está listo para una nueva consulta.

Es sumamente importante conocer lo que significan los prompts '>' y '>', ya que si en algún momento nos aparece alguno de ellos, todas las líneas que escribamos a continuación serán consideradas como parte de la cadena, inclusive cuando escribimos QUIT. Esto puede ser confuso, especialmente si no sabemos que es necesario escribir la comilla correspondiente para finalizar la cadena, para que podamos escribir después algún otro comando, o terminar la consulta que deseamos ejecutar.

Ahora que conocemos como escribir y ejecutar sentencias, es tiempo de acceder a una base de datos.

Supongamos que tenemos diversas mascotas en casa (nuestro pequeño zoológico) y deseamos tener registros de los datos acerca de ellas. Podemos hacer esto al crear tablas que guarden esta información, para que posteriormente la consulta de estos datos sea bastante fácil y de manera muy práctica. Esta sección muestra cómo crear una base de datos, crear una tabla, incorporar datos en una tabla, y recuperar datos de las tablas de diversas maneras.

La base de datos "zoológico" será muy simple, pero no es difícil pensar en situaciones del mundo real en las cuales una base de datos similar puede ser usada.

Primeramente usaremos la sentencia SHOW para ver cuáles son las bases de datos existentes en el servidor al que estamos conectados:

```
mysql> SHOW DATABASES;  
+-----+  
| Database |  
+-----+  
| mysql   |  
| test    |  
+-----+  
2 rows in set (0.00 sec)  
  
mysql>
```

Es probable que la lista de bases de datos que veamos sea diferente en nuestro caso, pero seguramente las bases de datos "mysql" y "test" estarán entre ellas. En particular, la base de datos "mysql" es requerida, ya que ésta tiene la información de los privilegios de los usuarios de MySQL. La base de datos "test"

es creada durante la instalación de MySQL con el propósito de servir como área de trabajo para los usuarios que inician en el aprendizaje de MySQL.

Es posible que no veamos todas las bases de datos si no tenemos el privilegio SHOW DATABASES. Se recomienda revisar la sección del manual de MySQL dedicada a los comandos GRANT y REVOKE.

Si la base de datos "test" existe, hay que intentar acceder a ella:

```
mysql> USE test  
Database changed  
mysql>
```

Observar que **USE**, al igual que **QUIT**, no requiere el uso del punto y coma, aunque si se usa, no hay ningún problema. El comando **USE** es especial también de otra manera: éste debe ser usado en una sola línea.

Podríamos usar la base de datos "test" (si tenemos acceso a ella) para los ejemplos que vienen a continuación, pero cualquier cosa que hagamos puede ser eliminada por cualquier otro usuario que tenga acceso a esta base de datos. Por esta razón, es recomendable que preguntemos al administrador MySQL acerca de la base de datos que podemos usar. Supongamos que deseamos tener una base de datos llamada "zoologico" (nótese que no se está acentuando la palabra) a la cual sólo nosotros tengamos acceso, para ello el administrador necesita ejecutar un comando como el siguiente:

```
mysql> GRANT ALL on zoologico.* TO MiNombreUsuario@MiComputadora  
-> IDENTIFIED BY 'MiContraseña';
```

En donde **MiNombreUsuario** es el nombre de usuario asignado dentro del contexto de MySQL, **MiComputadora** es el nombre o la dirección IP de la computadora desde la que nos conectamos al servidor MySQL, y **MiContraseña** es la contraseña que se nos ha asignado, igualmente, dentro del ambiente de MySQL exclusivamente. Ambos, nombre de usuario y contraseña no tienen nada que ver con el nombre de usuario y contraseña manejados por el sistema operativo (si es el caso).

Una vez que estamos en la base de datos, si no nos acordamos de qué base de datos es:

```
select database();
```

Si el administrador creó la base de datos en el momento de asignar los permisos, podemos hacer uso de ella. De otro modo, nosotros debemos crearla:

```
mysql> USE zoologico  
ERROR 1049: Unknown database 'zoologico'  
mysql>
```

El mensaje anterior indica que la base de datos no ha sido creada, por lo tanto necesitamos crearla.

```
mysql> CREATE DATABASE zoologico;
Query OK, 1 row affected (0.00 sec)
```

```
mysql> USE zoologico
Database changed
mysql>
```

Bajo el sistema operativo Unix, los nombres de las bases de datos son sensibles al uso de mayúsculas y minúsculas (no como las palabras clave de SQL), por lo tanto debemos de tener cuidado de escribir correctamente el nombre de la base de datos. Esto es cierto también para los nombres de las tablas.

Al crear una base de datos no se selecciona ésta de manera automática; debemos hacerlo de manera explícita, por ello usamos el comando USE en el ejemplo anterior.

La base de datos se crea sólo una vez, pero nosotros debemos seleccionarla cada vez que iniciamos una sesión con MySQL. Por ello es recomendable que se indique la base de datos sobre la que vamos a trabajar al momento de invocar al monitor de MySQL. Por ejemplo:

```
shell>mysql -h casita -u blueman -p zoologico
```

```
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 17 to server version: 3.23.38-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer

mysql>
```

Observar que "zoologico" no es la contraseña que se está proporcionando desde la línea de comandos, sino el nombre de la base de datos a la que deseamos acceder. Si deseamos proporcionar la contraseña en la línea de comandos después de la opción "-p", debemos de hacerlo sin dejar espacios (por ejemplo, -phola123, no como -p hola123). Sin embargo, escribir nuestra contraseña desde la línea de comandos no es recomendado, ya que es bastante inseguro.

Crear la base de datos es la parte más fácil, pero en este momento la base de datos está vacía, como lo indica el comando SHOW TABLES:

```
mysql> SHOW TABLES;
Empty set (0.00 sec)
```

La parte un tanto complicada es decidir la estructura que debe tener nuestra base de datos: qué tablas se necesitan y qué columnas estarán en cada tabla.

En principio, necesitamos una tabla que contenga un registro para cada una de nuestras mascotas. Ésta puede ser una tabla llamada mascotas, y debe contener por lo menos el nombre de cada uno de nuestros animales. Ya que el nombre en sí no es muy interesante, la tabla debe contener alguna otra información. Por ejemplo, si más de una persona en nuestra familia tiene una mascota, es probable que tengamos que guardar la información acerca de

quién es el dueño de cada mascota. Así mismo, también sería interesante contar con alguna información más descriptiva tal como la especie, y el sexo de cada mascota.

¿Y qué sucede con la edad? Esto puede ser también de interés, pero no es una buena idea almacenar este dato en la base de datos. La edad cambia conforme pasa el tiempo, lo cual significa que debemos de actualizar los registros frecuentemente. En vez de esto, es una mejor idea guardar un valor fijo, tal como la fecha de nacimiento. Entonces, cuando necesitemos la edad, la podemos calcular como la diferencia entre la fecha actual y la fecha de nacimiento. MySQL proporciona funciones para hacer operaciones entre fechas, así que no hay ningún problema.

Al almacenar la fecha de nacimiento en lugar de la edad tenemos algunas otras ventajas:

Podemos usar la base de datos para tareas tales como generar recordatorios para cada cumpleaños próximo de nuestras mascotas. Podemos calcular la edad en relación a otras fechas que la fecha actual. Por ejemplo, si almacenamos la fecha en que murió nuestra mascota en la base de datos, es fácil calcular que edad tenía nuestro animal cuando falleció. Es probable que estemos pensando en otro tipo de información que sería igualmente útil en la tabla "mascotas", pero para nosotros será suficiente por ahora contar con información de nombre, propietario, especie, nacimiento y fallecimiento.

Usaremos la sentencia CREATE TABLE para indicar como estarán conformados los registros de nuestras mascotas.

```
mysql> CREATE TABLE mascotas(  
-> nombre VARCHAR(20), propietario VARCHAR(20),  
-> especie VARCHAR(20), sexo CHAR(1), nacimiento DATE,  
-> fallecimiento DATE);
```

Query OK, 0 rows affected (0.02 sec)

mysql>

VARCHAR es una buena elección para los campos nombre, propietario, y especie, ya que los valores que almacenarán son de longitud variable. No es necesario que la longitud de estas columnas sea la misma, ni tampoco que sea de 20. **Se puede especificar cualquier longitud entre 1 y 255**, lo que se considere más adecuado. Si resulta que la elección de la longitud de los campos que hemos hecho no resultó adecuada, MySQL proporciona una sentencia ALTER TABLE que nos puede ayudar a solventar este problema.

El campo sexo puede ser representado en una variedad de formas, por ejemplo, "m" y "f", o tal vez "masculino" y "femenino", aunque resulta más simple la primera opción.

El uso del tipo de dato DATE para los campos nacimiento y fallecimiento resulta obvio.

Ahora que hemos creado la tabla, la sentencia SHOW TABLES debe producir algo como:

```
mysql> SHOW TABLES;
+-----+
| Tables_in_zoologico |
+-----+
| mascotas           |
+-----+
1 row in set (0.00 sec)

mysql>
```

Para verificar que la tabla fue creada como nosotros esperábamos, usaremos la sentencia DESCRIBE:

```
mysql> DESCRIBE mascotas;
+-----+-----+-----+-----+-----+-----+
| Field      | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| nombre     | varchar(20) | YES  |     | NULL    |       |
| propietario | varchar(20) | YES  |     | NULL    |       |
| especie    | varchar(20) | YES  |     | NULL    |       |
| sexo       | char(1)     | YES  |     | NULL    |       |
| nacimiento | date       | YES  |     | NULL    |       |
| fallecimiento | date   | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
6 rows in set (0.01 sec)

mysql>
```

Podemos hacer uso de la sentencia DESCRIBE en cualquier momento, por ejemplo, si olvidamos los nombres o el tipo de las columnas en la tabla.

Después de haber creado la tabla, ahora podemos incorporar algunos datos en ella, para lo cual haremos uso de las sentencias INSERT y LOAD DATA.

Supongamos que los registros de nuestras mascotas pueden ser descritos por los datos mostrados en la siguiente tabla.

Nombre	Propietario	Especie	Sexo	Nacimiento	Fallecimiento
Fluffy	Arnoldo	Gato	f	1999-02-04	
Mau	Juan	Gato	m	1998-03-17	
Buffy	Arnoldo	Perro	f	1999-05-13	
FanFan	Benito	Perro	m	2000-08-27	
Kaiser	Diana	Perro	m	1998-08-31	1997-07-29
Chispa	Omar	Ave	f	1998-09-11	
Wicho	Tomás	Ave		2000-02-09	
Skim	Benito	Serpiente	m	2001-04-29	

Debemos observar que MySQL espera recibir fechas en el formato YYYY-MM-DD, que puede ser diferente a lo que nosotros estamos acostumbrados.

Ya que estamos iniciando con una tabla vacía, la manera más fácil de poblarla es crear un archivo de texto que contenga un registro por línea para cada uno de nuestros animalitos para que posteriormente carguemos el contenido del archivo en la tabla únicamente con una sentencia (¡Ojo! Eso no es SQL).

Por tanto, debemos de crear un archivo de texto "mascotas.txt" que contenga un registro por línea con valores separados por tabuladores, cuidando que el orden de las columnas sea el mismo que utilizamos en la sentencia CREATE TABLE. Para valores que no conoczamos podemos usar valores nulos (NULL). Para representar estos valores en nuestro archivo debemos usar \N.

Usaremos el archivo mascotas.txt.

Para cargar el contenido del archivo en la tabla mascotas, usaremos el siguiente comando:

```
mysql> LOAD DATA LOCAL INFILE "/mascotas.txt" INTO TABLE mascotas  
LINES TERMINATED BY '\r\n';  
o  
mysql> LOAD DATA LOCAL INFILE "\\\mascotas.txt" INTO TABLE mascotas  
LINES TERMINATED BY '\r\n';
```

La sentencia LOAD DATA (de Mysql) nos permite especificar cuál es el separador de columnas, y el separador de registros, por defecto el tabulador es el separador de columnas (campos), y el salto de línea es el separador de registros, que en este caso son suficientes para que la sentencia LOAD DATA lea correctamente el archivo "mascotas.txt".

Tendremos que indicar el directorio donde tengamos el archivo de texto. Por defecto cogerá el path donde se almacena el esquema de la base de datos:

```
C:\\xampp\\mysql\\data\\zoologico
```

La sentencia quedaría:

```
LOAD DATA LOCAL INFILE "C:/xampp/mysql/data/zoologico/mascotas.txt"  
INTO TABLE mascotas LINES TERMINATED BY '\r\n';
```

Si lo que deseamos es añadir registro a registro, entonces debemos hacer uso de la sentencia INSERT (de SQL). En la manera más simple, debemos proporcionar un valor para cada columna en el orden en el cual fueron listados en la sentencia CREATE TABLE. Supongamos que Diana compra un nuevo hámster llamado Pelusa. Podemos usar la sentencia INSERT para agregar su registro en nuestra base de datos.

```
mysql> INSERT INTO mascotas  
-> VALUES('Pelusa','Diana','Hamster','f','2000-03-30',NULL);
```

Los valores de cadenas y fechas deben estar encerrados entre comillas. También, con la sentencia INSERT podemos insertar el valor NULL directamente para representar un valor nulo, un valor que no conocemos. En este caso no se usa \N como en el caso de la sentencia LOAD DATA.

La sentencia SELECT de SQL es usada para obtener la información guardada en una tabla. La forma general de esta sentencia es:

```
SELECT LaInformaciónQueDeseamos FROM DeQueTabla WHERE  
CondiciónASatisfacer
```

Aquí, LaInformaciónQueDeseamos es la información que queremos ver. Esta puede ser una lista de columnas, o un * para indicar "todas las columnas". DeQueTabla indica el nombre de la tabla de la cual vamos a obtener los datos. La cláusula **WHERE** es opcional. Si está presente, la CondiciónASatisfacer especifica las condiciones que los registros deben satisfacer para que puedan ser mostrados.

Seleccionando todos los datos

La manera más simple de la sentencia **SELECT** es cuando se recuperan todos los datos de una tabla:

```
mysql> SELECT * FROM mascotas;  
+-----+-----+-----+-----+-----+-----+  
| nombre | propietario | especie | sexo | nacimiento | fallecimiento |  
+-----+-----+-----+-----+-----+-----+  
| Fluffy | Arnoldo | Gato | f | 1999-02-04 | NULL |  
| Mau | Juan | Gato | m | 1998-03-17 | NULL |  
| Buffy | Arnoldo | Perro | f | 1999-05-13 | NULL |  
| FanFan | Benito | Perro | m | 2000-08-27 | NULL |  
| Kaiser | Diana | Perro | m | 1998-08-31 | 1997-07-29 |  
| Chispa | Omar | Ave | f | 1998-09-11 | NULL |  
| Wicho | Tomás | Ave | NULL | 2000-02-09 | NULL |  
| Skim | Benito | Serpiente | m | 2001-04-29 | NULL |  
| Pelusa | Diana | Hamster | f | 2000-03-30 | NULL |  
+-----+-----+-----+-----+-----+-----+  
9 rows in set (0.00 sec)
```

Esta forma del **SELECT** es útil si deseamos ver los datos completos de la tabla, por ejemplo, para asegurarnos de que están todos los registros después de la carga de un archivo.

Por ejemplo, en este caso que estamos tratando, al consultar los registros de la tabla, nos damos cuenta de que hay un error en el archivo de datos (mascotas.txt): parece que Kaiser ha nacido después de que ha fallecido. Al revisar un poco los datos de Kaiser encontramos que la fecha correcta de nacimiento es el año 1989, no 1998.

Hay por lo menos un par de maneras de solucionar este problema:

Editar el archivo "mascotas.txt" para corregir el error, eliminar los datos de la tabla mascotas con la sentencia **DELETE**, y cargar los datos nuevamente con el comando **LOAD DATA**:

```
mysql> DELETE FROM mascotas;  
mysql> LOAD DATA LOCAL INFILE "mascotas.txt" INTO TABLE mascotas;
```

Sin embargo, si hacemos esto, debemos introducir a mano los datos de Pelusa, la mascota de Diana.

La segunda opción consiste en corregir sólo el registro erróneo con una sentencia **UPDATE**:

```
mysql> UPDATE mascotas SET nacimiento="1989-08-31"  
WHERE nombre="Kaiser";
```

Como se mostró anteriormente, es muy fácil recuperar los datos de una tabla completa. Pero típicamente no deseamos hacer esto, particularmente cuando las tablas son demasiado grandes. En vez de ello, estaremos más interesados en responder preguntas particulares, en cuyo caso debemos especificar algunas restricciones para la información que deseamos ver.

Podemos seleccionar solo registros particulares de una tabla. Por ejemplo, si deseamos verificar el cambio que hicimos a la fecha de nacimiento de Kaiser, seleccionamos solo el registro de Kaiser de la siguiente manera:

```
mysql> SELECT * FROM mascotas WHERE nombre="Kaiser";  
+-----+-----+-----+-----+-----+  
| nombre | propietario | especie | sexo | nacimiento | fallecimiento |  
+-----+-----+-----+-----+-----+  
| Kaiser | Diana | Perro | m | 1989-08-31 | 1997-07-29 |  
+-----+-----+-----+-----+-----+  
1 row in set (0.00 sec)
```

La salida mostrada confirma que el año ha sido corregido de 1998 a 1989.

La comparación de cadenas en MySQL es normalmente no sensitiva, así que podemos especificar el nombre como "kaiser", "KAISER", etc. El resultado de la consulta será el mismo.

Podemos además especificar condiciones sobre cualquier columna, no sólo el "nombre". Por ejemplo, si deseamos conocer qué mascotas nacieron después del 2000, tendríamos que usar la columna "nacimiento":

```
mysql> SELECT * FROM mascotas WHERE nacimiento >= "2000-1-1";  
+-----+-----+-----+-----+-----+  
| nombre | propietario | especie | sexo | nacimiento | fallecimiento |  
+-----+-----+-----+-----+-----+  
| FanFan | Benito | Perro | m | 2000-08-27 | NULL |  
| Wicho | Tomás | Ave | NULL | 2000-02-09 | NULL |  
| Skim | Benito | Serpiente | m | 2001-04-29 | NULL |  
| Pelusa | Diana | Hamster | f | 2000-03-30 | NULL |  
+-----+-----+-----+-----+-----+  
4 rows in set (0.00 sec)
```

Hay que tener cuidado en especificar la fecha entre dobles comillas. Podemos también combinar condiciones, por ejemplo, para localizar a los perros hembras:

```
mysql> SELECT * FROM mascotas WHERE especie="Perro" AND sexo="f";  
+-----+-----+-----+-----+-----+  
| nombre | propietario | especie | sexo | nacimiento | fallecimiento |  
+-----+-----+-----+-----+-----+  
| Buffy | Arnoldo | Perro | f | 1999-05-13 | NULL |  
+-----+-----+-----+-----+-----+  
1 row in set (0.00 sec)
```

La consulta anterior usa el operador lógico **AND**. Hay también un operador lógico **OR**:

```
mysql> SELECT * FROM mascotas WHERE especie = "Ave" OR especie = "Gato";
+-----+-----+-----+-----+-----+
| nombre | propietario | especie | sexo | nacimiento | fallecimiento |
+-----+-----+-----+-----+-----+
| Fluffy | Arnoldo | Gato | f | 1999-02-04 | NULL |
| Mau | Juan | Gato | m | 1998-03-17 | NULL |
| Chispa | Omar | Ave | f | 1998-09-11 | NULL |
| Wicho | Tomás | Ave | NULL | 2000-02-09 | NULL |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

El operador **AND** y el operador **OR** pueden ser intercambiados. Si hacemos esto, es buena idea usar paréntesis para indicar como deben ser agrupadas las condiciones:

```
mysql> SELECT * FROM mascotas WHERE (especie = "Gato" AND sexo = "m")
      -> OR (especie = "Perro" AND sexo = "f");
+-----+-----+-----+-----+-----+
| nombre | propietario | especie | sexo | nacimiento | fallecimiento |
+-----+-----+-----+-----+-----+
| Mau | Juan | Gato | m | 1998-03-17 | NULL |
| Buffy | Arnoldo | Perro | f | 1999-05-13 | NULL |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Si no deseamos ver los registros completos de una tabla, entonces tenemos que usar los nombres de las columnas en las que estamos interesados separándolas por coma. Por ejemplo, si deseamos conocer la fecha de nacimiento de nuestras mascotas, debemos seleccionar la columna "nombre" y "nacimiento":

```
mysql> SELECT nombre, nacimiento FROM mascotas;
+-----+-----+
| nombre | nacimiento |
+-----+-----+
| Fluffy | 1999-02-04 |
| Mau | 1998-03-17 |
| Buffy | 1999-05-13 |
| FanFan | 2000-08-27 |
| Kaiser | 1989-08-31 |
| Chispa | 1998-09-11 |
| Wicho | 2000-02-09 |
| Skim | 2001-04-29 |
| Pelusa | 2000-03-30 |
+-----+-----+
9 rows in set (0.00 sec)
```

Para conocer quién tiene alguna mascota, usaremos la siguiente consulta:

```
mysql> SELECT propietario FROM mascotas;
+-----+
| propietario |
+-----+
| Arnoldo    |
| Juan       |
| Arnoldo    |
| Benito     |
| Diana      |
| Omar       |
| Tomás      |
| Benito     |
| Diana      |
+-----+
9 rows in set (0.00 sec)
```

Sin embargo, debemos notar que la consulta recupera el nombre del propietario de cada mascota, y algunos de ellos aparecen más de una vez. Para minimizar la salida, agregaremos la palabra clave **DISTINCT**:

```
mysql> SELECT DISTINCT propietario FROM mascotas;
+-----+
| propietario |
+-----+
| Arnoldo    |
| Juan       |
| Benito     |
| Diana      |
| Omar       |
| Tomás      |
+-----+
6 rows in set (0.03 sec)
```

Se puede usar también una cláusula **WHERE** para combinar selección de filas con selección de columnas. Por ejemplo, para obtener la fecha de nacimiento de los perritos y los gatitos, usaremos la siguiente consulta:

```
mysql> SELECT nombre, especie, nacimiento FROM mascotas
    -> WHERE especie = "perro" OR especie = "gato";
+-----+-----+-----+
| nombre | especie | nacimiento |
+-----+-----+-----+
| Fluffy | Gato   | 1999-02-04 |
| Mau    | Gato   | 1998-03-17 |
| Buffy  | Perro  | 1999-05-13 |
| FanFan | Perro  | 2000-08-27 |
| Kaiser | Perro  | 1989-08-31 |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

Se debe notar en los ejemplos anteriores que las filas devueltas son mostradas sin ningún orden en particular. Sin embargo, frecuentemente es más fácil examinar la salida de una consulta cuando las filas son ordenadas en alguna forma útil. Para ordenar los resultados, tenemos que usar una cláusula **ORDER BY**.

Aquí aparecen algunos datos ordenados por fecha de nacimiento:

```
mysql> SELECT nombre, nacimiento FROM mascotas ORDER BY nacimiento;
+-----+-----+
| nombre | nacimiento |
+-----+-----+
| Kaiser | 1989-08-31 |
| Mau | 1998-03-17 |
| Chispa | 1998-09-11 |
| Fluffy | 1999-02-04 |
| Buffy | 1999-05-13 |
| Wicho | 2000-02-09 |
| Pelusa | 2000-03-30 |
| FanFan | 2000-08-27 |
| Skim | 2001-04-29 |
+-----+-----+
9 rows in set (0.00 sec)
```

En las columnas de tipo carácter, la ordenación es ejecutada normalmente de forma no sensitiva, es decir, no hay diferencia entre mayúsculas y minúsculas. Sin embargo, se puede forzar un ordenamiento sensitivo al usar el operador **BINARY**.

```
INSERT INTO mascotas
-> VALUES('pelusa','Diana','Hamster','f','2000-03-30',NULL);

SELECT nombre, nacimiento FROM mascotas ORDER BY binary(nombre);
```



```
SELECT nombre, nacimiento FROM mascotas ORDER BY nombre;
```

Para ordenar en orden inverso, debemos agregar la palabra clave **DESC** al nombre de la columna que estamos usando en el ordenamiento:

```
mysql> SELECT nombre, nacimiento FROM mascotas ORDER BY
-> nacimiento DESC;
+-----+-----+
| nombre | nacimiento |
+-----+-----+
| Skim | 2001-04-29 |
| FanFan | 2000-08-27 |
| Pelusa | 2000-03-30 |
| Wicho | 2000-02-09 |
| Buffy | 1999-05-13 |
| Fluffy | 1999-02-04 |
| Chispa | 1998-09-11 |
| Mau | 1998-03-17 |
| Kaiser | 1989-08-31 |
+-----+-----+
9 rows in set (0.00 sec)
```

Podemos ordenar múltiples columnas. Por ejemplo, para ordenar por tipo de animal, y poner al inicio los animales más pequeños de edad, usaremos la siguiente consulta:

```
mysql> SELECT nombre, especie, nacimiento FROM mascotas
-> ORDER BY especie, nacimiento DESC;
+-----+-----+-----+
| nombre | especie | nacimiento |
+-----+-----+-----+
| Wicho | Ave | 2000-02-09 |
| Chispa | Ave | 1998-09-11 |
| Fluffy | Gato | 1999-02-04 |
| Mau | Gato | 1998-03-17 |
| Pelusa | Hamster | 2000-03-30 |
| FanFan | Perro | 2000-08-27 |
| Buffy | Perro | 1999-05-13 |
| Kaiser | Perro | 1989-08-31 |
| Skim | Serpiente | 2001-04-29 |
+-----+-----+-----+
9 rows in set (0.00 sec)
```

Notar que la palabra clave **DESC** aplica sólo a la columna nombrada que le precede.

UT10 ResultSet scrollable y updatable

1. ResultSet scrollable y updatable

Cuando hacemos una consulta a base de datos con una `statement()`, obtenemos un `ResultSet` en el que podemos ir leyendo los resultados de la consulta. Este `ResultSet` normalmente es sólo de lectura (no nos permite modificar la base de datos) y se va recorriendo secuencialmente, desde la primera fila de resultados a la última. Pues bien, podemos cambiar esta configuración al crear el `Statement`, de forma que obtengamos un `ResultSet` por el que podamos avanzar y retroceder, además de que nos permita modificar los datos. Sólo un detalle importante a tener en cuenta: Todo esto funciona si la consulta es una consulta simple a una única tabla de base de datos.

La forma de hacerlo es la siguiente:

```
Statement st =  
conexion.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CON  
CUR_UPDATABLE);  
ResultSet rs = st.executeQuery("select * from tabla_prueba");
```

El primer parámetro, `ResultSet.TYPE_SCROLL_INSENSITIVE`, indica que el `ResultSet` debe ser navegable hacia delante y hacia atrás, de forma que una vez leídos los resultados, podamos retroceder para volver a leerlos. La parte `_INSENSITIVE` indica que no verá futuras modificaciones hechas por terceros en la base de datos, es decir que, si otro usuario cambia los datos en la base de datos, nuestro `ResultSet` no volverá a consultar la base de datos cada vez que le hagamos avanzar o retroceder, y no nos enteraremos de esos futuros cambios.

El segundo parámetro `ResultSet.CONCUR_UPDATABLE` nos indica que el `ResultSet` debe permitir modificar los datos en la base de datos desde el mismo `ResultSet`. La opción por defecto es solo lectura: `ResultSet.CONCUR_READ_ONLY`.

El siguiente trozo de código, por ejemplo, lee los resultados de la forma normal, avanzando con el método `next()`, y cuando llega al final vuelve a recorrer los resultados, pero esta vez hacia atrás.

```
while (rs.next()) {  
    System.out.println(rs.getObject(1));  
    System.out.println(rs.getObject(2));  
}  
  
while (rs.previous()) {  
    System.out.println(rs.getObject(1));  
    System.out.println(rs.getObject(2));  
}
```

Por supuesto, tenemos más métodos que nos permiten ir directamente a determinadas posiciones, como `afterLast()`, `last()`, `beforeFirst()` y `first()`.

2. Modificar filas

También podemos modificar el contenido de la base de datos. Para ello, vamos recorriendo el `ResultSet` hasta llegar a la posición que nos interesa y una vez en esa posición, usamos los métodos `update()` para modificar el contenido. Primero se modifican las columnas concretas que nos interesan y finalmente se llama a `updateRow()` para hacer los cambios. En el siguiente código suponemos que ya nos hemos ido a la fila concreta que nos interesa del `ResultSet`. Ejecutamos entonces

```
rs.updateString(2, "nuevo texto");
rs.updateRow();
```

3. Borrar filas

Se puede también borrar una fila concreta. Para ello vamos recorriendo el `ResultSet` hasta llegar a ella y hacemos la llamada a:

```
rs.deleteRow();
```

4. Insertar filas

Y finalmente, también es posible insertar nuevas filas. El `ResultSet` tiene un método especial que nos lleva a una fila "vacía" preparada para insertar. Este método es `moveToInsertRow()`. Una vez en ella, llamamos a los `update()` necesarios para dar valores a cada columna y finalmente llamamos a `insertRow()` para hacer efectivos los cambios. El siguiente trozo de código es un ejemplo de todo esto

```
rs.moveToInsertRow();
rs.updateString(2, "nueva fila");
rs.insertRow();
```

UT10 Ejercicios de Acceso a BBDD

1º.- Crea la base de datos Instituto, cuyas tablas y valores se encuentran en el fichero instituto.sql. Una vez creada la base de datos haz un programa en Java que cree una nueva tabla llamada **NotasFinales** que tendrá la siguiente estructura:

```
NotasFinales(Mat, Cod, NotaMedia);
```

Y cuyos valores se sacarán de la tabla Notas.

Por último, se imprimirá un listado de todos los alumnos con el siguiente formato:

Nombre Alumno	Nombre Asignatura	Nota 1	Nota 2	Nota 3	NotaMedia
---------------	-------------------	--------	--------	--------	-----------

2º.- Se tiene la base de datos América, compuesta por las tablas Personas y Países. Haz un programa en Java que cree la tabla PersonasPaises que tendrá los siguientes atributos:

```
Id, Nombre, Apellido, Edad, NombrePais y Tamaño.
```

La información que va almacenar es la sacada de las otras dos tablas. Tras crear dicha tabla, actualizarla sumando 1 a la edad de las personas de Costa Rica. Finalmente sacar un listado con toda la información de la nueva tabla.

3º.- Realiza un programa en Java para trabajar con la base de datos Empresa (empleados y departamentos), que haga lo siguiente:

- a. Conexión** a la base de datos (carga del driver y establecimiento de conexión).
- b. Inserción de un departamento.** Escribe un método llamado **insertarDepto** que recibirá tres argumentos (número, nombre y localidad del departamento).
- c. Inserción de un departamento.** El mismo nombre de método que b, pero recibiendo un solo argumento: un objeto de la clase Departamento. Será necesario por tanto, crear una clase Departamento, con sus atributos y métodos getter y setter.
- d. Método (listarDepartamentos)** que **devuelva un ArrayList de objetos Departamento** a partir de una consulta de todas las columnas de todos los departamentos de la tabla departamentos.
- e. Método (borrarDepartamento)** que reciba un número de departamento y lo dé de baja.
- f. Método (actualizarDepartamento)** que reciba un número de departamento y una localidad y actualice su localidad, con ese nuevo valor.
- g. Método (actualizarDepartamento)** que reciba un objeto departamento y actualice el departamento con el número de departamento indicado a los valores dados en el objeto.
- h. Método (devolverDepartamento)** que reciba un número de departamento y devuelva un objeto con sus datos
- i. Método (subirSalario)** que reciba una cantidad y un número de departamento e incremente el sueldo de todos los empleados de ese departamento en esa cantidad.

(Ampliación:)

- j. Método** que imprima el gestor de base de datos empleado, el driver utilizado y el usuario conectado.
- k. Método** que imprima del esquema actual todas las tablas y vistas que contiene, indicando además del nombre, si se trata de una tabla o una vista.
- l. Método** que reciba una consulta (p.ej. SELECT * FROM departamentos) e imprima el número de columnas recuperadas, y por cada columna el nombre, tipo, tamaño y si admite o no nulos.

EJERCICIO 4: TIENDA

Utilizando la aplicación **PHPMyAdmin** crea una base de datos llamada Tienda con las siguientes tablas:



Escribe un programa en Java para conectarse a esta base de datos que:

- a.-** Cree una tabla de nombre **ArtFab** con los campos: nombre del artículo, nombre del fabricante, precio e iva (con 2 decimales)
- b.-** Rellene la tabla **ArtFab** con los valores correspondientes. Para el campo IVA lo harás en función del precio, teniendo en cuenta:
- Si es > 500 se aplica un 14%
 - Entre 500 y 200 (no incluido) se aplica un 12%
 - Entre 200 y 100 (no incluido) se aplica un 10%
 - 100 ó menos se aplica un 8%
- c.-** Muestre el contenido de la nueva tabla **ArtFab**.

UT11 Gráficos - Alertas y dialogos en Java

La clase JOptionPane es una clase que hereda de JComponent. Esta clase nos permitirá crear alertas o cuadros de diálogo simples para poder solicitar o mostrar información al usuario.

Los métodos que veremos son:

1. JOptionPane.showMessageDialog(...);
2. JOptionPane.showConfirmDialog(...);
3. JOptionPane.showInputDialog(...);
4. JOptionPane.showOptionDialog(...);

Cada uno de estos métodos presenta una particularidad distinta pero todos ellos nos muestran una ventana pop up que nos permitirá captar información del usuario.

1. showMessageDialog

```
JOptionPane.showMessageDialog(Component componentePadre, Object mensaje, String titulo, int tipoDeMensaje)
```

Nos sirve para mostrar información, por ejemplo alguna alerta que queremos hacerle al usuario. Veamos cuales son los principales argumentos del método.

componentePadre: es el Frame desde el cual lo llamamos. Si queremos lo podemos poner null de momento. Si hubiera un padre, el dialogo se mostraría sobre él.

mensaje: es lo que queremos que diga el cuadro de dialogo.

titulo: el título de la ventana.

tipoDeMensaje: son constantes que le dirán a java qué tipo de mensaje queremos mostrar. De acuerdo a esto serán los iconos que se mostrarán en el cuadro de dialogo.

Las opciones son:

- ERROR_MESSAGE
- INFORMATION_MESSAGE
- WARNING_MESSAGE
- QUESTION_MESSAGE
- PLAIN_MESSAGE

Si quisiéramos mostrar un ícono personalizado, podemos agregarlo al final como un argumento más.

Ejemplo:

```
JOptionPane.showMessageDialog(null, "Acceso Denegado",  
"Error", JOptionPane.ERROR_MESSAGE);
```

2. showConfirmDialog

Este método sirve para pedirle al usuario una confirmación. Por ejemplo, una confirmación de salida del sistema

```
Int respuesta =JOptionPane. showConfirmDialog(Component  
componentePadre, Object mensaje, String titulo, int tipoDeOpcion);
```

Lo anterior es la versión corta de los argumentos del método. La versión larga incluye el tipo de mensaje y el ícono, por si queremos personalizarlo.

Los argumentos son idénticos a los del método anterior. Excepto por el tipo de opción, que es otra constante y los valores pueden ser:

- DEFAULT_OPTION
- YES_NO_OPTION
- YES_NO_CANCEL_OPTION
- OK_CANCEL_OPTION

Como vemos, el método devuelve un entero que nos permitirá captar cual es la opción elegida por el usuario. Los valores serán **0** para **Sí**, **1** para **No**, **2** para **Cancelar** y **-1** para el **cierre de la ventana**. Así podremos preguntar cuál es el valor devuelto y realizar la acción que deseamos.

Ejemplo:

```
int i = JOptionPane.showConfirmDialog(this,"¿Realmente Desea  
Salir de Hola Swing?", "Confirmar  
Salida", JOptionPane.YES_NO_OPTION);  
  
if(i==0){  
    System.exit(0);  
}
```

3. ShowInputDialog

Este método nos muestra una ventana donde podremos insertar un String. Por ejemplo, cuando queremos que el usuario inserte su nombre. La versión corta del método es:

```
String respuesta = JOptionPane.showInputDialog(Object mensaje)
```

Este método devolverá un String para poder utilizarlo después. La versión larga de los argumentos del método es similar a los anteriores.

Ejemplo:

```
String nombre = JOptionPane.showInputDialog("Inserte su Nombre");
```

También podemos crear un cuadro de dialogo que contenga un combo con las opciones predeterminadas que le queremos dar al usuario.

Ejemplo:

```
Object[] valoresPosibles = {"Pedro", "Juan", "Carlos" };

Object jefe = JOptionPane.showInputDialog(null, "Seleccione cual es
su Jefe Inmediato", "Seleccionar Jefe", JOptionPane.
QUESTION_MESSAGE, null, valoresPosibles, valoresPosibles[0]);
```

El array de valores posibles nos muestra en un combo cuáles serán los jefes que podemos mostrar. El último argumento del método nos muestra cuál será la opción seleccionada por defecto

4. showOptionDialog

Con este método podemos crear cuadros de diálogos con botones personalizados. Está bien para personalizar los cuadros de dialogo.

El método tiene la forma:

```
int res = JOptionPane.showOptionDialog(Component componentePadre, Object mensaje, String titulo, int tipoDeOpcion, int tipoMensaje, Icon icono, Object[] botones, Object botonDefault)
```

Aquí lo único que varía con el resto, es el array de botones que vamos a tener, debemos destacar que no hace falta que creamos botones, solo tenemos que poner cuál será el texto que saldrá en él.

```
Object[] botones = {"No, nada", "Un poquito", "Me estalla" };

int i=JOptionPane.showOptionDialog(null, "¿Te duele la cabeza?", "ventanita",

JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE,
null, botones, botones[0]);

System.out.println(i);
```

Después podemos tomar la respuesta como lo hacíamos con el confirmDialog.

Se puede consultar el API y

<https://serprogramador.es/programando-mensajes-de-dialogo-en-java-parte-1/>

UTX Varios

1. Clases y tipos genéricos en Java

En **Java**, cuando definimos una nueva clase, **debemos conocer el tipo de dato** con el que trabajaremos. Si queremos realizar una operación específica dentro de esta nueva clase, **sea cual sea el tipo de datos** que va a recibir, podemos hacer uso de los **tipos genéricos**. Este tipo genérico asumirá el tipo de dato que realmente le pasaremos a la clase.

Mejor con un ejemplo:

```
class ClaseGenerica<T> {
    T obj;

    public ClaseGenerica(T o) {
        obj = o;
    }

    public void classType() {
        System.out.println("El tipo de T es " + obj.getClass().getName());
    }
}

public class MainClass {
    public static void main(String args[]) {
        // Creamos una instancia de ClaseGenerica para Integer.
        ClaseGenerica<Integer> intObj = new ClaseGenerica<Integer>(88);
        intObj.classType();

        // Creamos una instancia de ClaseGenerica para String.
        ClaseGenerica<String> strObj = new ClaseGenerica<String>("Test");
        strObj.classType();
    }
}
```

Notas:

- T es el tipo genérico que será reemplazado por un tipo real.
- T es el nombre que damos al parámetro genérico.
- Este nombre se sustituirá por el tipo real que se le pasará a la clase.

El resultado será el siguiente:

```
1 El tipo de T es java.lang.Integer
2 El tipo de T es java.lang.String
```

Hay que tener en cuenta que **los generics de java solo funcionan con objetos**. El código siguiente nos mostrará un error:

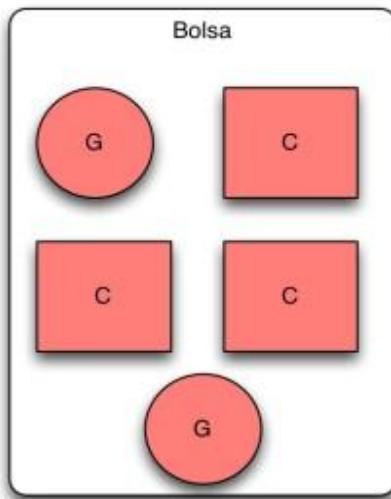
```
ClaseGenerica<int> myOb = new ClaseGenerica<int>(53);
// Error, can't use primitive type
```

Existen una serie de **convenciones para nombrar a los genéricos**:

- E - Element (usado bastante por Java Collections Framework)
- K - Key (Llave, usado en mapas)
- N - Number (para números)
- T - Type (Representa un tipo, es decir, una clase)
- V - Value (representa el valor, también se usa en mapas)
- S,U,V etc. - usado para representar otros tipos.

1.1 Uso de Java Generics

Veremos cómo se usan los Java Generics o llamadas simplemente clases Genéricas. Construiremos la clase Bolsa que es una clase sencilla que nos permitirá almacenar objetos de varios tipos.



Esta clase tendrá un límite de objetos a almacenar. Alcanzado el límite no se podrán añadir más. Vamos a ver su código fuente:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Bolsa implements Iterable{
    private ArrayList lista;
    private int tope;

    public Bolsa(int tope) {
        //super();
        lista= new ArrayList();
        this.tope = tope;
    }

    public void add(Object objeto ) {
        if (lista.size()<=tope) {
            lista.add(objeto);
        }else {
            throw new RuntimeException("no caben mas");
        }
    }

    public Iterator iterator() {
        return lista.iterator();
    }
}
```

En nuestro caso vamos a disponer de dos clases con las cuales rellenar la bolsa. La clase Golosina y la clase Chocolatina.



Vamos a ver su código fuente:

```
public class Golosina {  
    private String nombre;  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
}  
  
public class Chocolatina {  
    private String marca;  
  
    public String getMarca() {  
        return marca;  
    }  
    public void setMarca(String marca) {  
        this.marca = marca;  
    }  
    public Chocolatina(String marca) {  
        super();  
        this.marca = marca;  
    }  
}
```

Creamos un sencillo programa que llene la Bolsa de Chocolatinas y Golosinas para luego recorrer los elementos que están en la bolsa y sacarlos por pantalla.

```
public class Principal {  
  
    public static void main(String[] args) {  
        Bolsa bolsa= new Bolsa(5);  
        Chocolatina c= new Chocolatina("milka");  
        Chocolatina c1= new Chocolatina("milka");  
        Chocolatina c2= new Chocolatina("ferrero");  
        Golosina g1= new Golosina("gominola");  
        Golosina g2= new Golosina("chicle");  
  
        bolsa.add(c);  
        bolsa.add(c1);  
        bolsa.add(c2);  
        bolsa.add(g1);  
        bolsa.add(g2);  
  
        for (Object o: bolsa) {  
  
            if( o instanceof Chocolatina) {  
                Chocolatina chocolatina= (Chocolatina)o;  
                System.out.println(chocolatina.getMarca());  
            }else {  
                Golosina golosina= (Golosina)o;  
                System.out.println(golosina.getNombre());  
            }  
        }  
    }  
}
```

El programa funcionará correctamente, pero nos podremos dar cuenta que **resulta bastante poco amigable la estructura if /else en la cual se chequean cada uno de los tipos a la hora de presentarlo por pantalla.**

1.2 Java Generics

Para solventar este problema podemos construir una clase Genérica. Este tipo de clase nos permitirá definir una Bolsa de un tipo concreto. Puede ser una bolsa de Golosinas o una bolsa de Chocolatinas **pero NO de las dos cosas a la vez**. Esto en un principio puede parecer poco flexible pero si nos ponemos a pensar cuando programamos solemos imprimir una lista de Facturas o una lista de Compras no una lista mixta. Así pues el enfoque parece razonable. Vamos a ver el código fuente y comentarlo:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Bolsa<T> implements Iterable<T>{
    private ArrayList<T> lista= new ArrayList<T>();
    private int tope;

    public Bolsa(int tope) {
        super();
        this.tope = tope;
    }

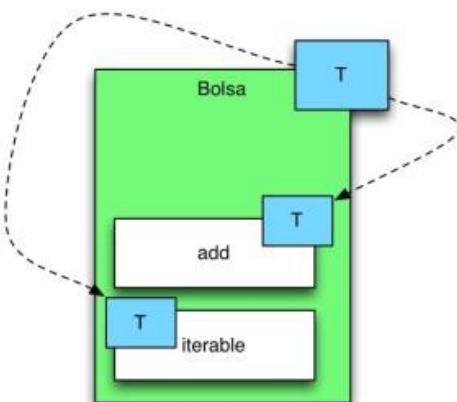
    public void add(T objeto ) {
        if (lista.size()<=tope) {

            lista.add(objeto);
        }else {

            throw new RuntimeException("no caben mas");
        }
    }

    public Iterator<T> iterator() {
        return lista.iterator();
    }
}
```

La clase es un poco peculiar ya que al no saber de entrada de qué tipo va a ser la bolsa **debemos declarar un tipo Genérico T a nivel de clase y que será repetido en cada uno de los métodos que lo usen.**



De esta manera cuando construyamos un objeto de esta clase será el momento de especificar el tipo de Bolsa que deseamos. En el siguiente ejemplo hemos elegido “Chocolatina” como tipo para la Bolsa. De esta manera la bolsa solo admitirá este tipo de objetos.

```
public class Principal {  
    public static void main(String[] args) {  
        Bolsa<Chocolatina> bolsa= new Bolsa<Chocolatina>();  
  
        Chocolatina c= new Chocolatina("milka");  
        Chocolatina c1= new Chocolatina("milka");  
        Chocolatina c2= new Chocolatina("ferrero");  
  
        bolsa.add(c);  
        bolsa.add(c1);  
        bolsa.add(c2);  
  
        for (Chocolatina chocolatina : bolsa) {  
            System.out.println(chocolatina.getMarca());  
        }  
    }  
}
```

<http://www.arquitecturajava.com/uso-de-java-generics/>

2. Copiar Arrays en Java

2.1 Copiar arrays en Java

Muchas veces nos vamos a encontrar ante la disyuntiva de tener que copiar los elementos de un array en otros. Normalmente será para manipular el contenido del mismo guardando en uno de los arrays los datos originales.

Asumida ya la situación, lo primero que se nos ocurriría, independientemente del lenguaje en el que nos encontremos, será el montar un algoritmo que recorriendo el primer array vaya copiando dichos elementos en el segundo.

En Java nos quedaría un código como este:

```
for (int x=0;x<aOrigen.length;x++)
    aDestino[x] = aOrigen[x];
```

En este sentido nada que objetar, ya que es muy buena práctica de programación. Pero todo buen programador tiene que tener en mente el concepto de "reutilización". No "reutilización" cómo copia del código, sino "reutilización" pensando en que alguien ya puede haberse encontrado el problema y haberle dado ya una solución.

Para poder reutilizar tenemos que ser conscientes de lo que el entorno en el que estamos nos ofrece. Y en el caso de Java, es la librería del sistema la que nos ofrece una función para la copia de arrays. Como vemos en el siguiente código:

```
System.arraycopy(aOrigen, inicioArrayOrigen, aDestino,
                 inicioArrayDestino, numeroElementosACopiar);
```

Hay que tener cuidado la función arrayCopy ya que esta nos puede devolver las siguientes excepciones: IndexOutOfBoundsException si intentamos copiar fuera del área reservado para el array, ArrayStoreException si intentamos copiar arrays de diferente tipo o NullPointerException si alguno de los array es nulo (es decir, no inicializado).

2.2 Copiar dos arrays en uno con Java

A partir de dos arrays, copiar el contenido de ambos dentro de un tercer array.

Lo primero será definir los dos arrays de origen:

```
int a1 [] = {1, 2, 3, 4, 5};  
int a2 [] = {6, 7, 8, 9, 10};
```

Y posteriormente el array destino. Hay que tener en cuenta que el tamaño del array destino tiene que ser lo suficientemente grande como para albergar el contenido de los dos arrays origen.

Para ello le damos como tamaño la suma de los dos arrays origen:

```
int a [] = new int [a1.length+a2.length];
```

En la copia de arrays nos apoyaremos en el método Java `arrayCopy`.

Los parámetros que recibe el método `arrayCopy` son:

- Array origen.
- Posición inicial del array origen.
- Array destino.
- Posición inicial en el array de destino.
- Número de elementos a copiar del array origen al array destino.

El siguiente paso será copiar el primer array en el array destino:

```
System.arraycopy(a1, 0, a, 0, a1.length);
```

Vemos que del array origen copiamos desde la posición 0, al igual que el array destino. El número de elementos a copiar es igual al tamaño del array de origen.

El tercer paso será copiar array en el array destino. Hay que tener cuidado, ya que en el array destino ya tenemos cargado el primer array.

```
System.arraycopy(a2, 0, a, a1.length, a2.length);
```

Lo que vemos es que del array de origen se copia desde el primer elemento, el cero. En el caso del array destino nos tendremos que posicionar en el elemento siguiente al último elemento del primer array. Esto nos lo da el tamaño del primer array. Es por ello que utilizamos `a1.length`. El número de elementos sigue siendo los elementos del segundo array.

3. Listas en Java

Una **lista** es una secuencia de elementos dispuestos en un cierto orden, en la que cada elemento tiene como mucho un predecesor y un sucesor. El número de elementos de la lista no suele estar fijado, ni suele estar limitado por anticipado. Representaremos la estructura de datos de forma gráfica con cajas y flechas. Las cajas son los elementos y las flechas simbolizan el orden de los elementos.



La estructura de datos deberá permitirnos determinar cuál es el primer elemento y el último de la estructura, cuál es su predecesor y su sucesor (si existen de cualquier elemento dado). Cada uno de los elementos de información suele denominarse nodo.

La lista también puede representarse de forma simbólica escribiendo sus elementos separados por comas y encerrados entre corchetes. Por ejemplo:

```
[ "rojo", "verde", "azul", "amarillo" ]
```

Las listas admiten ciertas operaciones como son insertar un nodo adicional, borrar un nodo, etc. En función de la forma de insertar nuevos elementos y acceder a los existentes tendremos distintos tipos de listas. Veamos ahora qué operaciones básicas se pueden realizar sobre las listas.

Hay ciertas operaciones básicas sobre una lista como:

- **EsVacia** Averiguar si la lista está vacía.
- **Insertar** Añade un elemento al principio de la lista.
- **Primero** Obtener el valor del primer elemento de la lista, también llamado cabeza.
- **Resto** Devuelve el trozo de lista resultado de eliminar el primer elemento de la lista.
- **Borrar** Borrar el primer elemento de la lista.
- **etc.**

4. Pilas

Una **pila** (stack en inglés) es una lista ordenada o estructura de datos que permite almacenar y recuperar datos, el modo de acceso a sus elementos es de tipo **LIFO** (del inglés **Last In, First Out**, «último en entrar, primero en salir»).

