INTRODUCCIÓN

Descripción

Java es un potente lenguaje de programación de propósito general, lo que significa que sirve para el desarrollo de aplicaciones de todo tipo. Sus principales características son las seguridad y la flexibilidad. Java es un lenguaje orientado a objetos, que es un paradigma que hace más fácil e intuitivo el desarrollo de aplicaciones.

El lenguaje **Java** también es el lenguaje utilizado para la programación de dispositivos móviles con sistema operativo Android, así como programación de servidores de Internet y servidores empresariales, lo que hace que dicho lenguaje sea múy útil y solicitado. **Java** también está fuertemente vinculado al gestor de bases de datos **Oracle**.

Arquitectura y funcionamiento de un programa Java

La máquina virtual Java (**JVM**) es un entorno de ejecución para aplicaciones **Java**. Su principal finalidad es adaptar los programas Java compilados a las características del sistema operativo donde se vayan a ejecutar.

Todo programa realizado en Java está organizado en clases (archivos con extensión .java). Cada archivo **.java** puede contener una o varias clases, aunque lo normal y recomendable es que solo contengan una sola clase. Cuando se compila un archivo .java, se obtiene un archivo **.class** (los bytecodes, que es un código "neutro" independiente de la arquitectura).

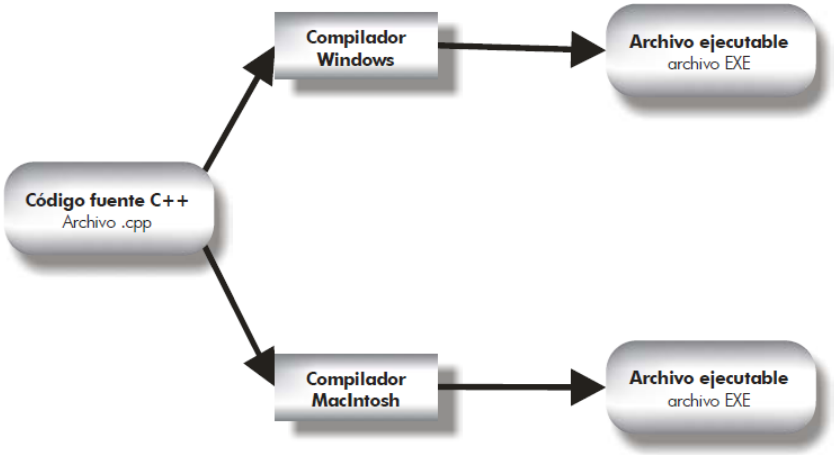
Estos bytecodes no pueden ser directamente ejecutados por un sistema operativo. Para ello, el intérprete (máquina virtual del sistema operativo donde se va a ejecutar la aplicación Java), interpreta dichos bytecodes, ejecutando así la aplicación Java en dicho sistema operativo.

Hay una máquina virtual Java (**JVM**) para cada sistema operativo. Cada JVM de cada sistema operativo lee los bytecodes de la misma manera, pero cada una de ellas realiza la interpretación (traducción) correspondiente al sistema operativo para el que ha sido diseñada, es decir, para el sistema operativo donde se va a realizar dicha interpretación.

Es normal hoy en día, encontrar máquinas virtuales para cualquier sistema operativo existente. En muchos casos, dicha máquina virtual forma parte del propio sistema operativo.

Un programa C o C++ es totalmente ejecutable y eso hace que no sea independiente de la plataforma, ya que habría que cambiar el código del programa y compilarlo de nuevo para ejecutarlo en las diferentes plataformas, y que además, su tamaño normalmente es muy grande, ya que dentro del código final hay que incluir las librerías para dicha plataforma.

Ejemplo del proceso de compilación de un programa C++:



Los programas Java no son ejecutables (**EXE**), no se compilan como los programas en C o C++. En su lugar son interpretados por una aplicación conocida como la máquina virtual de Java (JVM). Gracias a ello no tienen porque incluir todo el código y librerías propias de cada máquina final o sistema.

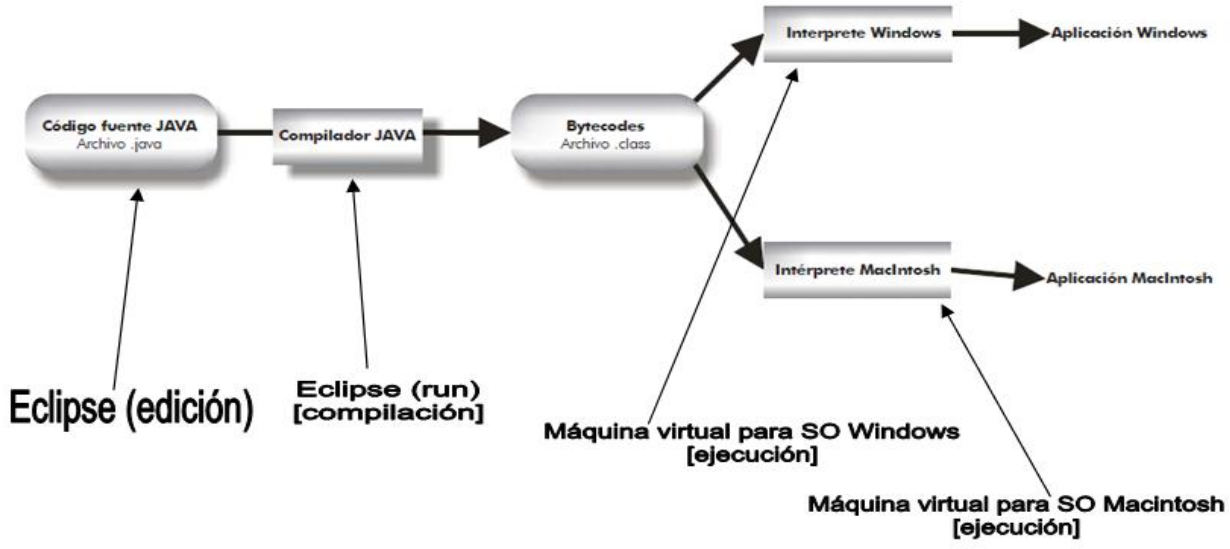
Previamente, el código fuente en Java (archivos **.java**) se tiene que precompilar generando un código previo (que no es directamente ejecutable) conocido como bytecode o J-code (código Java). Dicho código (bytecode), genera archivos con extensión .class (dichos archivos class son los que contienen las instrucciones bytecodes) son los archivos que son ejecutados por la máquina virtual de Java (JVM) que interpreta las instrucciones de dichos bytecodes contenidos en los archivos .class, ejecutando así el código de la aplicación.

El **bytecode** se puede ejecutar en cualquier plataforma. Lo único que se requiere es que dicha plataforma posea un intérprete adecuado a dicha plataforma (es decir, que posea la máquina virtual para dicha plataforma). La máquina virtual de Java, además es un programa muy pequeño y que se distribuye gratuitamente para prácticamente todos los sistemas operativos.

Por consiguiente, habrá una máquina virtual Java (JVM) para cada sistema operativo donde se desee ejecutar la aplicación Java; por ejemplo, habrá una máquina virtual Java para el SO Windows, otra máquina virtual Java para el SO Macintosh, otra máquina virtual para un dispositivo electrónico en concreto, otra máquina virtual para un electrodoméstico específico, etc. A este método de ejecución de programas en tiempo real se le llama Just in Time (**JIT**).

En Java la unidad fundamental del código es la clase (compilada en archivo .class). Son las clases compiladas (en archivos .class) las que se distribuyen (ya que contienen el bytecode de Java). Estas clases (los archivos .class) se cargan dinámicamente durante la ejecución del programa Java (la máquina virtual, carga dinámicamente los archivos .class durante la ejecución del programa Java).

Proceso de compilación de un programa en Java, suponiendo que se utilice el entorno Eclipse para crear, compilar y ejecutar el programa:



- El **JRE** proporciona solo el entorno de ejecución para las aplicaciones Java. Se descagará el JRE si solo se desea ejecutar aplicaciones Java ya construídas, pero no servirá para el desarrollo de aplicaciones Java.
- La **JVM** es el programa que ejecuta el código Java previamente compilado (bytecode) mientras que las librerias de clases estándar son las que implementan el API de Java. Ambas (JVM y API) deben ser consistentes entre sí, de ahí que sean distribuidas de modo conjunto (mediante el JRE).
- Un usuario sólo necesita el JRE para ejecutar las aplicaciones desarrolladas en lenguaje Java, mientras que para desarrollar nuevas aplicaciones en dicho lenguaje es necesario el entorno de desarrollo JDK, que además del JRE (mínimo imprescindible) incluye, entre otros, un compilador para Java.

Lo necesario para ejecutar y desarrollar programas Java

Para ejecutar aplicaciones Java

El **JRE** es lo mínimo que debe contener un sistema para poder ejecutar una aplicación Java.

Por lo que, para ejecutar una aplicación Java, solo se necesita el JRE, que contiene la máquina virtual de Java y las bibliotecas de Java.

JRE (Java Runtime Enviroment [Entorno de ejecución para Java])
=
JVM (Java Virtual Machine [Máquina Virtual Java])
+
Bibliotecas de Java.

Para desarrollar aplicaciones Java

JDK (Java Development Kit [Kit de desarrollo Java])
=
JRE (Java Runtime Enviroment [Entorno de ejecución para Java])
+
Herramientas para la compilación y ejecución de programas Java.
+
Paquetes de clases del JSE.

La API de Java

La **API** (Application Programming Interface [Interfaz de Programación de Aplicaciones]) contiene todo lo necesario para dotar a los programadores de Java de los medios para desarrollar aplicaciones Java.

La **API** de Java provee de un conjunto de clases que permiten realizar toda clase de tareas necesarias para la realización de aplicaciones Java.

La **API** de Java está organizada en paquetes lógicos (como carpetas), donde cada paquete contiene un conjunto de clases relacionadas por su utilidad.

Para consultar la API de Java, se puede consultar en la siguiente página:

http://docs.oracle.com/javase/7/docs/api/

Tipos de programas Java

En el lenguaje Java se pueden desarrollar programas de diversos tipos, según el objetivo que se desee:

- Aplicación autónoma: aplicación que funciona de forma independiente en cualquier ordenador o dispositivo, y que necesita ser instalada previamente a su utilización.
- Applet: son programas incrustados en otras aplicaciones, normalmente una página Web que se muestra en un navegador.
- Servlets: son componentes de la parte del servidor de Java EE, encargados de generar respuestas a las peticiones recibidas de los programas clientes.
- Aplicaciones gráficas: Swing es la biblioteca para la interfaz gráfica de usuario avanzada de la plataforma Java SE, que permite hacer aplicaciones con gráficas, con ventanas, botones, etc.

Ediciones o plataformas de la tecnología Java

Las tres Ediciones (o Plataformas) de la tecnología Java son las siguientes:

- **JSE** (Java Standard Edition). Está formado por un grupo de paquetes de clases de uso general: tratamiento de cadenas, colecciones, acceso a datos, aplicaciones gráficas, Applets para internet,

etc. Éstas clases son las necesarias para cualquier tipo de aplicación de propósito general: consola, interfaz gráfica o applets.
- **JEE** (Java Enterprise Edition). Está formado por los paquetes y las tecnologías necesarias para la creación de aplicaciones empresariales multicapa (organización del código de la aplicación en varias capas); entre ellas, las aplicaciones Web, así como aplicaciones del lado del servidor.
- **JME** (Java Micro Edition). Está formado por los paquetes y las especificaciones necesarias para crear aplicaciones Java para dispositivos móviles.

POO y trabajo con objetos

La **Programación Orientada a Objetos (POO)**, en inglés OOP (Objetc Oriented Programming), es un paradigma (esquema formal de organización) de programación, que utiliza objetos, así como interacciones entre dichos objetos, para realizar el diseño de programas informáticos.

La **POO** utiliza diversas técnicas para el desarrollo de programas orientados a objetos. Surge en los años 70. y su uso se hizo popular en los principios de 1990.

Actualmente, hay multitud de lenguajes de programación que soportan la orientación a objetos, es decir, que su escritura se realiza con las técnicas de la POO.

La programación orientada a objetos es un paradigma que utiliza objetos como elementos fundamentales en la construcción de la solución de un programa informático.

Un **objeto** es una abstracción (extracción de las partes importantes de algo) de algún hecho o ente del mundo real que tiene atributos que representan sus características o propiedades y métodos que representan su comportamiento o acciones que realizan. Todas las propiedades y métodos comunes a los objetos se encapsulan o se agrupan en **clases**. Una **clase** es una plantilla o un prototipo para crear objetos, por eso se dice que los objetos son instancias de clases.

Respecto al trabajo con objetos, la POO está basada se basa en la analogía de los objetos existentes en la vida real, con el objetivo de la construcción de programas informáticos; de esta forma, la programación se realiza de una manera más natural e intuitiva.

Cada objeto de la vida real tiene: un nombre, unos atributos (propiedades o características) y puede realizar una o varias operaciones (acciones). Por ejemplo, en la vida real, existe un objeto llamado Grapadora. Veamos sus cualidades:

- Nombre: Grapadora.
- Atributos: modelo, peso, color, longitud, capacidad (de grapas), precio, etc.
- Operaciones: grapar, cargar grapas, extraer grapas, vaciar grapas.

Nota: en realidad, la carga, extracción y el vaciado de las grapas las realizaría el ser humano, pero son operaciones que van ligadas al objeto Grapadora. Éste objeto Grapadora, sería uno de los muchos objetos que formaría parte de todos los objetos de una oficina.

Bien, pues si se traslada este ejemplo a un programa informático, imaginar que se nececesita realizar un programa que tenga como objetivo gestionar una tienda. Una de las cosas que hay que realizar en la tienda es realizar facturas. Pues entonces nuestro programa orientado a objetos, necesitará un objeto llamado factura, que podría tener las siguientes cualidades:

- Nombre: Factura.
- Atributos: fecha, concepto, etc.
- Operaciones: crear-factura.

Por supuesto, en la tienda hay que realizar mas cosas, aparte de crear facturas. Por ejemplo, también hay que realizar envíos, por ejemplo de paquetería. Pues podría haber un objeto Envío, con las siguientes cualidades:

- Nombre: Envío.
- Atributos: fecha de envío, destino, peso, etc.
- Operaciones: enviar, embalar, etc.

Y así, se debería analizar, con mucha meditación, todos los objetos necesarios en la tienda, ya que después, una vez analizados y creados todos los objetos, dichos objetos interactuarán unos con otros para realizar el objetivo, que es el programa informático que controla el negocio de la tienda.
Por ejemplo, en el programa informático de la tienda, cada vez que haya que hacer un envío, se ejecutará el objeto Envío y se especificarán los datos de dicho envío; después, automáticamente, el objeto envío se comunicará con el objeto Factura, para emitir la factura correspondiente a dicho envío.

Por lo que en la "Programación Orientada a Objetos", un programa informático en funcionamiento está compuesto por un conjunto de objetos que cada uno de ellos hace su labor específica y todos interactúan entre ellos.

Entornos de desarrollo Java (IDE)

Cuando se desarrolla una aplicación Java, utilizar el método de trabajo mediante la línea de comandos es muy lento y pesado, además de dificultar la detección de errores al compilar o ejecutar las aplicaciones.

Es mucho más práctico y extendido utilizar un **IDE** (Integrated Development Environment [Entorno de desarrollo integrado]).

Un **IDE** proporciona los siguientes mecanismos y ventajas para el desarrollo de programas:

- Codificación.
- Compilación.
- Depuración.
- Ejecución.

Todo ello, dentro de un entorno cómodo y eficiente, así como relativamente fácil de utilizar.

Los **IDE** de Java utilizan (de forma interna) las herramientas básicas del JDK en la realización de todas las operaciones. La ventaja de ello, es que el programador no tendrá que utilizar la consola para todas estas operaciones, dado que el propio IDE permite realizar dichas operaciones con una interfaz gráfica cómoda de utilizar.

En un IDE, también es sencillo la escritura de código. Los IDE suelen tener un sistema para el resaltado de las palabras reservadas del lenguaje corresponiente, para diferenciarlas así del resto del código, así como de otras muchas ventajas. También algunos IDE, como por ejemplo "Eclipse" tienen la característica Intellisense, que permite la autoescritura del código, es decir, mostrar la lista completa de métodos de un objeto, cuando se escribe el punto (.), y después poder seleccionar cualquier método de dicha lista.

Para el desarrollo en Java existen disponibles muchos entornos (IDE), como por ejemplo:

- **Eclipse**. (Es es más flexible y utilizado).
- Netbeans.
- Jdeveloper.
- Etc.

La mecánica de cualquier IDE es bastante similar. Cualquiera de ellos se basa en el concepto de proyecto. Dicho proyecto es un conjunto de clases y paquetes bien organizados, que forman la aplicación Java. De esta forma, en cualquier IDE, lo primero que se suele hacer cuando se comienza una aplicación Java es crear un proyecto.

Cuando se crea un proyecto, los IDE permiten escoger entre un conjunto de plantillas o tipos de proyecto, que cada uno de ellos se suele corresponder con un tipo de aplicación en Java: aplicación web, aplicación de escritorio (de interfaz gráfica o consola), aplicación para dispositivo móvil, etc.

INFRAESTRUCTURA PARA EL DESARROLLO EN JAVA

Descripción

Para realizar desarrollo en lenguaje Java, se necesitará lo siguiente:

- **JDK** (Java Depelopment Kit): toda la arquitectura necesaria para el desarrollo en Java.
- **IDE Eclipse**: entorno de desarrollo **IDE** para facilitar el desarrollo en lenguaje Java.
    - Además de Eclipse, existen otros entornos de desarrollo para Java, como por ejemplo: NetBeans.

INFRAESTRUCTURA PARA EL DESARROLLO EN JAVA: JDK

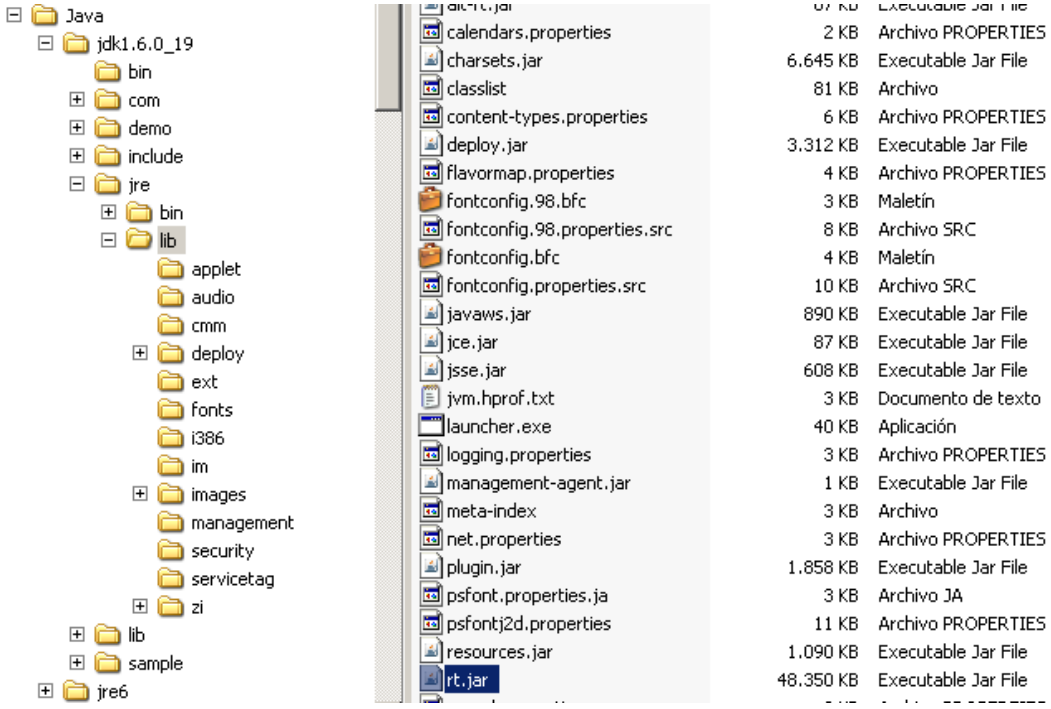
Descripción

El Java Development Kit (**JDK**), Kit de desarrollo Java, es el conjunto de herramientas para el desarrollo de aplicaciones Java.

Java Depelopment Kit (Kit de desarrollo de Java) es el Kit de Desarrollo de Java, todo lo correspondiente al lenguaje de programación Java.

El **JDK** (una vez instalado) proporciona lo siguiente:

- Implementación de la máquina virtual (JRE) para el sistema operativo que se haya indicado en la descarga de la web.
- Herramientas para la compilación y ejecución de programas Java. Todo esto se deberá utilizar desde la consola de comandos, si no se dispone de un IDE como por ejemplo: Eclipse, Netbeans, , etc.
- Paquetes de clases del JSE. Por ejemplo, si se tuviese instalado la versión 6 de Java, se tendría lo siguiente: dentro del directorio de instalación del JDK, en la carpeta: jre/lib, se encuentra el archivo: rt.jar, donde están contenidas todas las clases que componen JSE.



Para obtener el JDK se podrá descargar de forma gratuíta de la web. En dicha web, se podrá descargar, respecto al JDK, entre otras cosas, lo siguiente:

- JDK. Para el desarrollo de aplicaciones Java estándar. (JDK y SDK es lo mismo).
- JDK with JEE. Además del JDK, contiene las librerías de JEE para el desarrollo de aplicaciones empresariales.
- JDK with Netbeans. Además del JDK, el entorno de desarrollo (IDE) Netbeans, para la construcción de aplicaciones Java.
- JRE (Java Runtime Enviroment). Se descagará el JRE si solo se desea ejecutar aplicaciones Java ya construídas, pero no servirá para el desarrollo de aplicaciones Java. El JRE es un conjunto de utilidades que permite la ejecución de programas Java. En su forma más simple, el entorno en tiempo de ejecución de Java (JRE) está conformado por lo siguiente:
    - Una Máquina Virtual de Java (JVM).
    - Un conjunto de bibliotecas Java.

- Otros componentes necesarios para que una aplicación escrita en lenguaje Java pueda ser ejecutada.

Descarga de JDK

NOTA:

- Algún detalle de este proceso puede variar, debido a que las webs se modifican regularmente, pero lo importante es el proceso general.

Para descargarse el JDK, se hará mediante el siguiente vínculo:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

En la página, aparecerá un vínculo como el siguiente:

Java Platform (JDK) 7u40

Pulsando en él, aparecerá una ventana similar a la siguiente, que contiene una sección similar a la siguiente:

Java SE Development Kit 7u40

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux ARM v6/v7 VFP Hard Float ABI	67.62 MB	jdk-7u40-linux-arm-vfp-hflt.tar.gz
Linux ARM v6/v7 VFP Soft Float ABI	67.62 MB	jdk-7u40-linux-arm-vfp-sflt.tar.gz
Linux x86	115.55 MB	jdk-7u40-linux-i586.rpm
Linux x86	132.83 MB	jdk-7u40-linux-i586.tar.gz
Linux x64	116.83 MB	jdk-7u40-linux-x64.rpm
Linux x64	131.63 MB	jdk-7u40-linux-x64.tar.gz
Mac OS X x64	183.35 MB	jdk-7u40-macosx-x64.dmg
Solaris x86 (SVR4 package)	139.84 MB	jdk-7u40-solaris-i586.tar.Z
Solaris x86	95.29 MB	jdk-7u40-solaris-i586.tar.gz
Solaris x64 (SVR4 package)	24.43 MB	jdk-7u40-solaris-x64.tar.Z
Solaris x64	16.17 MB	jdk-7u40-solaris-x64.tar.gz
Solaris SPARC (SVR4 package)	139.06 MB	jdk-7u40-solaris-sparc.tar.Z
Solaris SPARC	98.07 MB	jdk-7u40-solaris-sparc.tar.gz
Solaris SPARC 64-bit (SVR4 package)	23.74 MB	jdk-7u40-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	18.18 MB	jdk-7u40-solaris-sparcv9.tar.gz
Windows x86	123.46 MB	jdk-7u40-windows-i586.exe
Windows x64	125.25 MB	jdk-7u40-windows-x64.exe

Pulsar en la parte superior el botón de radio correspondiente a: aceptar acuerdos, etc. y pulsar en el vínculo correspondiente al sistema operativo deseado, por ejemplo, para Windows 32 bits, sería el penúltimo vínculo: **Windows x86**.

Se descargará el fichero ejecutable.

Instalación de JDK

Para instalar JDK, pulsar doble clic en el archivo descargado.

Cuando comience el proceso de instalación, aparecerá una ventana inicial del asistente, entonces pulsar al botón **Next** en todas las ventanas que aparezcan. De este modo se dejará todo por defecto en el proceso de instalación.

Cuando la instalación finalice, en la última ventana que aparece, pulsar el botón **Close**.

NOTA:

- CON ESTO, HA TERMINADO LA INSTALACIÓN DE **Java JDK**.

INFRAESTRUCTURA PARA EL DESARROLLO EN JAVA: IDE ECLIPSE

Descripción

Eclipse es un **IDE** (entorno de desarrollo) utilizado para el lenguaje Java, así como para otros lenguajes de programación.

Eclipse permite: codificar, compilar, depurar y ejecutar programas Java.

Eclipse también permite la instalación de Plug-ins, que son aplicaciones complementarias para Eclipse que permiten realizar tareas específicas complementarias.

Descarga de Eclipse

NOTA:

- Algún detalle de este proceso puede variar, debido a que las webs se modifican regularmente, pero lo importante es el proceso general.

1. Acceder al sitio web **www.eclipse.org**.
2. En la parte superior, pulsar en el vínculo **Downloads**.
    - Se mostrará una página con todos los tipos de Eclipse.
3. Pulsar en la primera opción **Eclipse Standard x.x.x**.
    - Donde x.x.x será la última versión descargable de Eclipse.
    - Se mostrará una página con las características de dicho Eclipse.
4. En la sección derecha **Download Links**, pulsar en el vínculo **Windows 32-bit**.
    - Se mostrará una página con los lugares desde donde se puede descargar dicho Eclipse.
    - La primera opción de arriba es la recomendable. Muestra una flecha verde grande.
5. Pulsar en dicha primera opción de dicha flecha verde grande.
    - Comenzará la descarga.
    - Cuando termine la descarga, se mostrará una página con el mensaje **Thank you for downloding Eclipse**.
    - Cuando la descarga termine, se habrá descargado un fichero comprimido, de extensión **.zip**.

Ubicación de Eclipse

Una vez descargado Eclipse, no se necesita realizar una instalación, sino simplemente una descompresión del archivo descargado.

1. Descomprimir el archivo **.zip** que se descargó.
    - Se crea automáticamente la carpeta **eclipse**.
2. Mover la carpeta **eclipse** al disco duro **C**.

NOTA:

- CON ESTO, HA TERMINADO DESCARGA Y UBICACIÓN DE **Eclipse**.

IDE ECLIPSE: EJECUCIÓN

Descripción

Consiste en ejecutar el IDE **Eclipse** para su utilización.

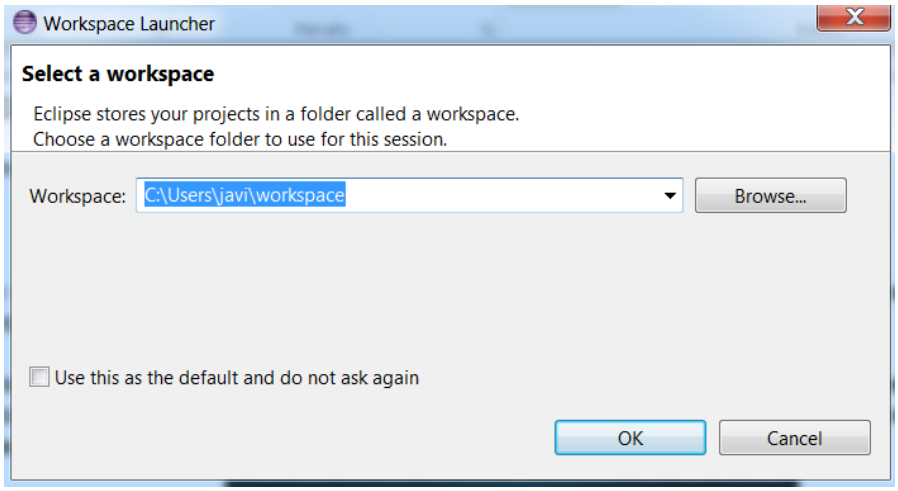
Ejecutar Eclipse

Para ejecutar **Eclipse**, dentro de la carpeta **eclipse**, pulsar doble clic en su archivo ejecutable **eclipse.exe**.

También se puede crear un acceso directo en el escritorio, para ejecutar Eclipse con más rapidez y comodidad.

Carpeta "workspace". Espacio de trabajo

Cuando se ejecuta **Eclipse** por primera vez, aparecerá la siguiente ventana:



Esta ventana está pidiendo la ruta para la carpeta **workspace**.

La carpeta **workspace** es el **espacio de trabajo**, es decir, donde se guardará el proyecto o diferentes proyectos que se vayan creando con **Eclipse**.

Se puede dejar esa ruta por defecto, pulsando directamente el botón **OK**.

Para que esta ventana no pregunte esto de nuevo, marcar la casilla **Use this as the default…**.

De todas formas, si se desea, dicha ruta se podría cambiar posteriormente.

Después de pulsar el botón OK, aparece Eclipse y su entorno. En su barra de título aparece **Java - Eclipse**.

También aparece una pantalla de bienvenida que se puede cerrar con la **X** de su pestaña **Welcome** (o botón derecho del ratón en dicha pestaña **Welcome - Close**), dejando ver así el entorno completo de **Eclipse**.

IDE ECLIPSE: ENTORNO GENERAL

Descripción

El entorno general de **Eclipse** dispone de diversas opciones que permiten controlar los proyectos que se crean con el lenguaje Java.

Entorno general

La interfaz general de **Eclipse** muestra algunos paneles por defecto, donde el más importante es el **Package Explorer**.

Package Explorer

El **Package Explorer** está situado por defecto en la parte izquierda.

En el **Package Explorer** es donde se verá toda la estructura de los proyectos de las aplicaciones Java que se tengan abiertas.

Mostrar vistas (paneles)

Para mostrar los diferentes paneles existentes en Eclipse:

1. Pulsar el menú **Window**.
2. Pulsar la opción **Show View**.
3. Seleccionar la vista deseada.

IDE ECLIPSE: PROYECTOS

Descripción

Un proyecto es el conjunto de archivos estructurados de forma organizada y jerárquica, que forman una aplicación completa de **Java**.

Crear un nuevo proyecto Java

1. Pulsar el menú **File**.
2. Pulsar la opción **New**.
3. Pulsar la opción **Java Project**.
    - Aparece la ventana **New Java Project**.
4. En la opción **Project name**, escribir el nombre deseado para el proyecto.
    - Este será el nombre real de la aplicación final.
    - NOTA: el resto de opciones se puede dejar por defecto.
5. Pulsar el botón **Finish**.
    - Se crea el proyecto nuevo, que se muestra en el panel **Package Explorer** de la zona izquierda.

Exportar un proyecto Java

Para exportar un proyecto **Java**, será necesario copiar la carpeta que se encuentra en el disco duro (en el **workspace** [espacio de trabajo]) correspondiente al proyecto deseado.

1. Para saber en que carpeta del disco duro se encuentra el **workspace** (ya que está ahí la carpeta del proyecto): Menú **File** - **Switch Workspace** - **Other**.
    - Aparece la ventana **Workspace Launcher**, que en la opción **Workspace** muestra la ruta del **workspace**. Fijarse en dicha ruta (que suele ser: "C:\usuarios\pepe**workspace**") y cerrar la ventana **Workspace Launcher**.
2. En el **Explorador de Windows** acceder a la carpeta correspondiente a dicha ruta del **workspace**.
    - Se apreciará que dentro de la carpeta **workspace** se encuentra la carpeta correspondiente al proyecto (que tiene el mismo nombre que dicho proyecto).
3. En el **Explorador de Windows** pulsar con el botón derecho del ratón dentro de la carpeta del proyecto - **Copiar**.
4. Pegar la carpeta en el lugar donde se desee guardar una copia de dicha carpeta correspondiente al proyecto de **Java**.

Importar a Eclipse un proyecto Java existente

La opción **Importar** de Eclipse sirve para copiar dentro de Eclipse un proyecto ya hecho, ya sea por nosotros hace tiempo, o que nos ha pasado un compañero, o descargado de Internet. Ésto es muy útil, ya que de esta forma se puede importar un proyecto, para continuar trabando en dicho proyecto, así como reutilizar parte de dicho proyecto en otro proyecto, etc..

1. Pulsar el menú **File**.
2. Pulsar la opción **Import**.
    - Aparece la ventana Import, mostrando la pantalla **Select**.
    - Se muestran un conjunto de carpetas, que contienen diferentes posibilidades de importación.
3. Abrir la carpeta **General**.
4. Pulsar en la opción **Existing Projects into Workspace**.
    - Esta opción permite seleccionar un proyecto del sistema de archivos.
5. Pulsar el botón **Next**.
    - Aparece la pantalla **Import Projects**.
6. Dejar seleccionada la opción **Select root directory**.
    - Esto permite seleccionar del sistema de archivos la carpeta al proyecto que se desea importar.
7. Pulsar el botón de su derecha **Browse…**.
    - Aparece la ventana **Buscar carpeta**.

8. En la ventana **Buscar carpeta** seleccionar del sistema de archivos la carpeta correspondiente al proyecto que se desea importar.
9. Pulsar el botón **Aceptar**.
    - En la sección **Projects** aparece el proyecto y entre paréntesis su ruta.
    - Si se especificó una ruta que contiene varios proyectos, en la sección **Projects** se mostrarán todos los dichos proyectos.
    - En dicha lista, se pueden marcar todos los proyectos que se desean importar.
10. Pulsar en la opción **Copy projects into workspace**.
    - Esta opción no es obligatorio marcarla, pero es recomendable porque se copia la carpeta correspondiente al proyecto que se está importando en el espacio de trabajo (workspace). Si no se marca esta opción, se trabajaría con el proyecto en el lugar del disco donde se encontrara, lo cual es posibie, pero siempre es recomendable copiarlo en el workspace, para trabajar con el proyecto en dicho workspace.
11. Pulsar el botón **Finish**.
    - El proyecto importado se mostrará en la vista **Package Explorer**.

Eliminar un proyecto en Eclipse

1. En el **Package Explorer**, pulsar con el botón derecho del ratón en el icono raiz del proyecto que se desea eliminar.
    - Aparece un menú de opciones.
2. Pulsar la opción **Delete**.
    - Aparece la ventana **Delete Resources**.
3. Marcar la opción **Delete project contents on disk (cannot be undone)**.
    - Marcando esta opción, no solo se elimina el proyecto en Eclipse, sino también su carpeta de archivos asociada a dicho proyecto, que se encuentra en la carpeta **workspace**.
4. Pulsar el botón **OK**.

EJERCICIO: Trabajo con proyectos en Eclipse.

1. Arrancar **Eclipse**.
2. Crear un proyecto Java llamado **Miproyecto**.
3. Cerrar **Eclipse**.
4. Abrir **Eclipse**.
5. Eliminar el proyecto **Miproyecto**.
6. En el **Explorador de archivos** de **Windows**, crear una carpeta dentro el disco duro **C**, llamada **Copia-Seguridad**.
7. En Eclipse, exportar el y guardarlo en la carpeta creada anteriormente **Copia-Seguridad**.
8. En Eclipse, eliminar el proyecto **Miproyecto**.
    - Quedará el entorno de Eclipse vacío (sin proyectos).
9. En Eclipse, importar el proyecto **Miproyecto** ubicado en la carpeta creada anteriormente **Copia-Seguridad**.
    - El proyecto **Miproyecto** quedará cargado en el **Package Explorer** de **Eclipse**.
10. En Eclipse, eliminar el proyecto **Miproyecto**.
11. Cerrar **Eclipse**.
12. En el **Explorador de archivos** de **Windows**, en el disco duro **C**, eliminar la carpeta llamada **Copia-Seguridad**.

ESTRUCTURA DE UNA APLICACIÓN JAVA

Descripción

Cuando se crea un proyecto, automáticamente se crea una estructura de carpetas. Esta estructura se puede observar en la vista **Package Explorer**.

El elemento raíz de un proyecto Java es una carpeta que representa al proyecto y que tiene como nombre, el nombre que se le dio al proyecto en la opción **Project Name**, cuando se creó el proyecto **Java**.

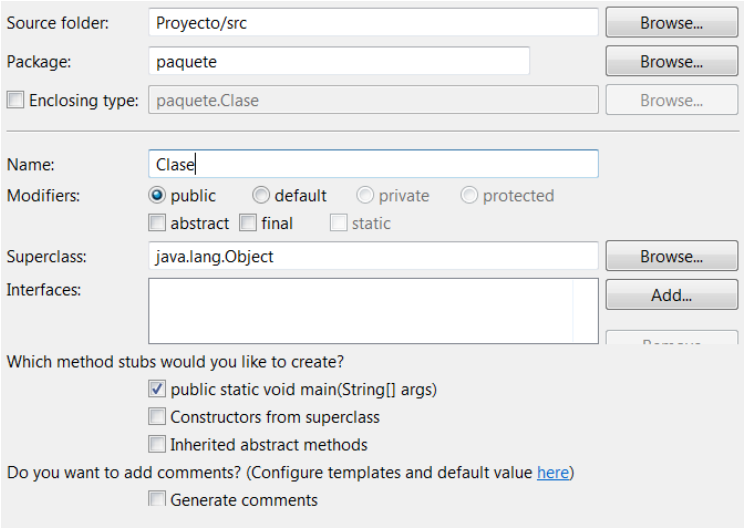
Estructura de la aplicación

Dentro del elemento raiz del proyecto **Java**, por defecto, aparecerá lo siguiente:

- **src**: (abreviatura de **source** [código]). Esta carpeta contiene los paquetes, que a su vez, contienen la clase o clases que contienen el código de la aplicación.
    - Cuando se crea un proyecto nuevo, esta carpeta aparece vacía.
    - En el momento que se crea, importa o pega una clase, se crea un paquete por defecto (con el mismo nombre del proyecto) dentro de dicha carpeta **src** donde se aloja dicha clase.
- **JRE System Library**: contiene las librerías de clases, necesarias para el desarrollo de aplicaciones Java.

Crear una clase

1. Pulsar con el botón derecho del ratón en el icono de la raíz del proyecto o en la carpeta **src**.
2. Pulsar la opcion **New**.
3. Pulsar la opción **Class**.
    - Aparece la ventana **New Java Class**.
4. En la opción **Package** escribir el nombre del paquete donde se almacenarán las clases de la aplicación.
    - Por defecto, aparecerá vacío o se llamará igual que el proyecto.
    - Si se deja vacío, el paquete se llamará **default package**.
    - Por buena costumbre, el nombre de los paquetes se pone todo en minúscula. (Por ejemplo: **paquete**.
5. En la opción **Name** escribir el nombre deseado para la clase.
    - Por buena costumbre, el nombre de las clases tiene la inicial en mayúscula. (Por ejemplo: **Clase**).
6. En la sección **Which method stubs would you like to create?**, seleccionar las opciones deseadas:
    - public static void main(String args[]): añade a la clase un método **main**, con lo que la clase queda configurada como clase inicial de la aplicación, es decir, la clase que arrancará la aplicación. Solo puede ser una clase la que arranque la aplicación.
    - Constructors for superclass: añade a la clase un método **constructor**.
        - El método constructor es un método que se ejecuta automáticamente cuando se ejecuta la clase y normalmente se utiliza para inicializar variables.
7. En la sección **Do you want to add comments?...**, seleccionar si se desea la opción:
    - Generate comments: añade a la clase los comentarios necesarios.
8. Pulsar el botón **Finish**.



Este ejemplo, creará una clase con el siguiente código:

```
package paquete;

public class Clase {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

	}

}
```

Copiar y pegar una clase (archivo java) en el proyecto

Se puede copiar un archivo java (extensión .java) y pegarlo directamente en el Package Explorer, en el proyecto deseado, en el paquete deseado.

ESTRUCTURA DE UNA CLASE

Descripción

Una clase de Java está formada por partes bien diferenciadas.

Partes de una clase

Hay que recordar que una aplicación Java suele estar formada por un conjunto de clases, en donde en cada una de dichas clases se codifica el comportamiento de cada clase en particular.

A continuación se muestran las partes de una clase básica que arranca una aplicación Java (tiene el método **main**). Si la clase no es la que arranca la aplicación, no llevaría dicho método **main**:

```
package paquete;

public class Clase
{
	public Clase()
	{

	}

	public static void main(String[] args)
	{

	}

}
```

Ahora con comentarios:

```
/*
 * Declaramos que esta clase (Miclase) pertenece al paquete (package) "paquete".
 * La clausula package debe ser la primera sentencia del archivo java
 */
package paquete;

/*
 * Cabecera de la clase Miclase
 */
public class Miclase
{
	/*
	 * Declaración de atributos de la clase.
	 */

	/*
	 * Método constructor.
	 * Es el primer método que se ejecuta cuando se
	 * ejecuta la clase.
	 * Se llama igual que el archivo de la clase (.java)
	 * y también igual que la clase
	 * (el nombre de la cabecera de la clase).
	 * El método constructor puede recibir parámetros.
	 */
	public Miclase()
	{
		/*
```

```
		 * Aquí se suelen inicializar los atributos de la clase.
		 * Lo primero que ocurre automáticamente al ejecutarse una clase
		 * es ejecutar su método constructor.
		 */
	}

	/*
	 * Éste es el método "main".
	 * Éste método se ejecuta al lanzarse la aplicación,
	 * es decir, es el primer método que se ejecuta
	 * cuando la aplicación arranca.
	 * La clase que tiene este método main
	 * es la clase que arranca la aplicación.
	 * (Ya que hay que tener en cuenta que una aplicación
	 * está compuesta normalmente por varias clases).
	 * El resto de clases de la aplicación, lo normal es que no
	 * tengan este método "main".
	 */
	public static void main(String[] args)
	{
		/*
		 * Aquí se pone todo lo que se desea que se ejecute
		 * cuando arranca la aplicación.
		 * Normalmente, aquí se ponen las llamadas
		 * a otra u otras clases de la aplicación.
		 */

		/*
		 * En este ejemplo, se muestra el texto "Hola amigos".
		 * Para ello, se hace referencia al paquete "System",
		 * a su clase "out" y a su método "println"
		 * que permite mostrar un texto por la consola.
		 */
		System.out.println("Hola amigos");
	}

	/*
	 * Declaración de métodos de la clase.
	 */

}	//Fin de la clase
```

El método main

Toda aplicación Java debe tener una clase que contenga el método **main**.

La clase que decidamos que arrancará la aplicación tendrá el modificador de acceso **public** (escrito en su cabecera), y dentro de dicha clase deberá existir un un método llamado **main()** que será de tipo estático (con el modificador **static**) y también con su modificador de acceso **public**.

El formato del método **main**, deberá ser el siguiente:

```
	public static void main(String[] args)
	{
		/*
		 * CÓDIGO DEL MÉTODO "main".
		 */
	}
```

El método main debe cumplir las siguientes características:

- Ha de ser un método público (llevar su modificador **public**).
- Ha de ser un método estático (llevar su modificador **static**).
- Es un método que no puede devolver ningún resultado (es un método con tipo de devolución **void**, es decir, no devuelve nada).
- En su lista de parámetros, debe declarar un array de cadenas de caracteres en la lista de parámetros o un número variable de argumentos.

El método **main** es el punto de arranque de una aplicación (programa) **Java**.

Cuando se ejecuta una aplicación Java, la **JVM** (máquina virtual de Java [que es la parte de la arquitectura Java que ejecuta la aplicación]) busca la clase (entre todas las clases de la aplicación) que contiene un método estático llamado **main**() con el formato adecuado (que cumpla todo lo indicado previamente).

Cuando lo encuentra, lo ejecuta como punto de arranque de la aplicación. Dentro del código **main**() pueden crearse objetos de otras clases e invocar a sus métodos, en general, se puede incluir cualquier tipo de lógica que respete las restricciones indicadas para los métodos estáticos.

EJECUCIÓN DE LA APLICACIÓN

Descripción

Una vez que la aplicación consta de lo básico ya se puede ejecutar para probarla. La aplicación se puede ejecutar desde la línea de comandos de Java, pero disponiendo de un IDE como Eclipse, es más cómodo, rápido y efectivo ejecutar dicha aplicación desde **Eclipse**.

Ejecutar la aplicación

Si es la primera vez que se ejecuta la aplicación:

1. En el **Package Explorer**, pulsar con el botón derecho del ratón en el elemento raíz del proyecto correspondiente a la aplicación que se desea ejecutar.
2. Pulsar la opción **Run As**.
3. Pulsar la opción **Java Application**.
    - Se ejecuta la aplicación y se muestra el resultado de dicha ejecución en la vista **Console**.
    - Al no ser una aplicación gráfica (con ventanas, botones, etc.), esta aplicación sería de consola.

Si NO es la primera vez que se ejecuta la aplicación:

1. Pulsar el botón **Run**.
    - También se puede pulsar la combinación de teclas: **Ctrl+F11**.

Ejecutar la clase deseada de un proyecto

Esta clase deberá tener el método **main** (para arrancar la aplicación).

1. En el **Package Explorer**, pulsar con el botón derecho del ratón en la clase deseada.
2. Pulsar la opción **Run As**.
3. Pulsar la opción **Java Application**.
    - Si esta clase no tiene el método **main**, no aparecerá esta opción, por lo que no se permitirá su ejecución.

RESUMEN DE CREACIÓN DE UNA APLICACIÓN JAVA

Descripción

La creación de una aplicación Java, implica una proceso compuesto por varios pasos.

Creación del proyecto

Lo primero será crear el proyecto en Eclipse.

Codificación

Se crearán las clases de la aplicación. Dichas clases se guardan con la extensión ".java" (estoy lo hace Eclipse automáticamente).

El nombre del fichero **.java** tiene que ser exactamente el mismo que el nombre de la clase. Con esto ya se tendrían todos los archivos fuente, es decir, todas las clases que forman la aplicación **Java**, en forma de archivos con extensión **.java**.

Codificar todos los archivos java convenientemente.

Compilación

Cuando ya se tienen los ficheros fuente **.java**, Eclipse se encargará de compilarlos. De hecho, cuando en Eclipse se ejecuta la aplicación se produce una **compilación y ejecución**.

En el proceso de compilación se transforman los ficheros fuente **.java** en los bytecodes de Java (archivos **.class**. La compilación se realizará con el compilador de Java (javac.exe), que forma parte del JDK (Todo esto lo hace Eclipse automáticamente de forma transparente para el programador).

- NOTA: lo habitual y mas recomendable es que cada fichero fuente **java** contenga una sola clase, pero si contuviese más de una, al realizar la compilación, se generarían tantos ficheros **.class** como clases tuviese dicho fichero fuente **java**.

En caso de existir errores sintácticos en el código fuente, el compilador informará de ello en el momento de compilar, se interrumpiría dicha compilación y por consiguiente, no se generaría el fichero **.class** necesario para ejecutar correctamente la aplicación.

Ejecución

Cuando ya se tienen los bytecodes (fichero **.class**) producto de la compilación, se ejecutará dicho fichero compilado **.class**. La ejecución se realizará mediante el intérprete de Java (java.exe), que forma parte del JDK.

En resumen, tras ejecutarse la compilación (con javac.exe) y la interpretación (con java.exe), el intérprete de Java (JRE) ejecutará (interpretará) los bytecodes (ficheros ".class"), mostrándose el programa Java correspondiente.

- NOTA: la primera clase que se ejecuta, deberá contener el método **main**. Éste método **main** actuará como inicio de la aplicación. Dicho método **main** es el método que se ejecutará al lanzarse la aplicación.
- La llamada al intérprete Java (java.exe), insta a la máquina virtual de Java (JVM) a buscar en la clase indicada un método **main**() y proceder a su ejecución.

SINTAXIS BÁSICA

Descripción

La sintaxis básica del lenguaje Java implica una serie de fundamentos básicos de la sintaxis del lenguaje.

Sensibilidad a mayúsculas/minúsculas

El lenguaje Java distingue entre mayúsculas y minúsculas.

El lenguaje Java es **Case sensitive** (sensible a mayúscula y minúscula). El compilador Java hace distinción entre mayúsculas y minúsculas, no es lo mismo escribir **public** que **Public**, de hecho, utilizar la segunda forma en vez de la primera provocaría un error de compilación al ser **Public** una palabra reservada.

La distinción entre mayúsculas y minúsculas no solo se aplica a las palabras reservadas del lenguaje, también a los nombres de variables y métodos. Es decir, una variable llamada **minumero** no se le puede hacer referencia de la forma **Minumero**.

Finalización de las sentencias

Toda sentencia en Java debe terminarse con el carácter **punto y coma** "**;**".

Delimitación de los bloques de instrucción

Los bloques de instrucción se delimitan con llaves "**{ }**".

Hay que delimitar con llaves lo siguiente:

- El código de la clase.
- El código del método.
- Dentro de los métodos las porciones de código deseado, sentencias condicionales, bucles, etc.

Comentarios

Los comentarios sirven para documentar el programa y comentarlo adecuadamente. Los comentarios no son compilados, por lo que no ocuparán espacio en los archivos compilados finales.

Los comentarios son muy útiles para que el programador documente y comente los aspectos del código, así como para que otros programadores también entiendan por qué el código se codificó de cierta manera.

Hay dos tipos de comentarios en Java:

- Comentarios de una línea.
- Comentarios de varias líneas.

Los comentarios de una línea debe comenzar con ***//***.

Ejemplo:

```
//Comentario de una sola línea
```

Los comentarios de varias líneas debe comenzar con ***/**** y terminar con *****/***.

Ejemplo:

```
/*
Comentario de varias líneas.
Y sigo escribiendo todas las líneas
que me apetece.
```

```
*/
```

El IDE Eclipse hace mas elegantes los comentarios de varias líneas, poniendo automáticamente un asterisco en cada una de las líneas intermedias, quedando de la siguiente forma:

```
/*
 * Comentario de varias líneas.
 * Y sigo escribiendo todas las líneas
 * que me apetece.
 * (Los asteriscos de estas 4 líneas comentadas, no son obligatorios,
 * pero eclipse los pone automáticamente).
 */
```

En **Eclise**, para poner comentarios:

1. Seleccionar las líneas de código deseadas.
2. Botón derecho del ratón dentro de dicho código - Source:
    - Toggle Comment: comentarios de una línea.
    - Add Block Comment: bloque de comentarios de varias líneas.

SECUENCIAS DE ESCAPE

Descripción

Las secuencias de escape son pequeñas constantes que nos brindan una gran utilidad, ya que permiten ciertas modificaciones en el código para hacer más cómodas ciertas acciones.

Las secuencias de escape, en su forma directa, se representan entre comillas simples. Entre dichas comillas simples puede aparecer:

- Un carácter Unicode. Ejemplos: 'a' , 'B', '{', 'ñ'.

Las secuencias de escape son combinaciones del símbolo contrabarra **** seguido de una letra, y sirven para representar caracteres que no tienen una equivalencia en forma de símbolo.

Las secuencias de escape también pueden aparecer dentro de las comillas dobles de una cadena tipo String.

Ejemplo: "Hola, buenas**\n**tardes"

Secuencias de escape

\n Nueva Linea. (Finaliza una línea y comienza una nueva línea).
\t Tabulador.
\r Retorno (Retroceso) de Carro. (Mueve el cursor al primer carácter de la siguiente línea).
**** El carácter barra inversa (\ ).
\' El carácter comilla simple (').
\" El carácter comilla doble (").

Ejemplos de secuencias de escape

Escribir texto en tres líneas (**println** añade automáticamente una secuencia de escape de nueva línea):

```
System.out.println("En un lugar");
System.out.println("de la");
System.out.println("Mancha.");
```

Resultado:

```
En un lugar
de la
Mancha.
```

Secuencia de escape de **Nueva línea**:

```
System.out.println("En un lugar\nde la\nMancha.");
```

Resultado:

```
En un lugar
de la
Mancha.
```

Secuencia de escape **Tabulador**:

```
System.out.println("\tEn un lugar de la Mancha.");
```

Resultado:

```
En un lugar de la Mancha.
```

Secuencias de escape de caracteres varios:

```
System.out.println("\\En un \"lugar\" de la \'Mancha\'.\\");
```

Resultado:

```
\En un "lugar" de la 'Mancha'.\
```

TIPOS DE DATOS

Descripción

El tipo de dato de una variable determina los valores que puede contener dicha variable, además de las operaciones que se puede realizar sobre ella.

Tipos de datos según su origen

En Java existen dos tipos principales de datos, atendiendo a su origen:

- Tipos de datos **Primitivos** (o simples).
  - A los datos almacenados en variables con tipos de datos **Primitivos** se accede accede directamente.
- Tipos de datos: **Referencias** (a objetos).
  - A los datos almacenados en variables con tipos de datos **Referencias** se accede a través de un puntero, es decir, no almacenan un valor sino una dirección de memoria. (El puntero es una variable que almacena una dirección de memoria, no un valor).

Tipos de datos primitivos

Los tipos de datos primitivos (o simples) son aquellos que pueden utilizarse directamente en un programa, sin necesidad del uso de clases (POO).

A diferencia de otros lenguajes de programación como **C**, en **Java** los tipos de datos primitivos no dependen de la plataforma, microprocesador o sistema operativo. En Java, un dato entero de tipo **int** siempre tendrá 4 bytes (32 bits), por lo que no tendremos sorpresas al migrar un programa de un sistema operativo a otro. Es más, ni siquiera hay que volverlo a compilar.

En otros lenguajes de programación, el formato o el tamaño de los tipos primitivos dependen del microprocesador o del sistema operativo en el que se están ejecutando. En cambio, Java pretende ser independiente de la plataforma y mantiene los formatos sin cambios.

Para los caracteres alfanuméricos utiliza la codificación **UNICODE** de 16 bits, para permitir la inclusión de varios alfabetos.

Un tipo primitivo está predefinido por el lenguaje y utiliza una palabra clave **reservada** (**byte**, **short**, **int**, **long**, etc.).

En Java, los tipos de datos primitivos son 8. Son los siguientes:

- **byte**: es un entero de 8 bits. Su valor mínimo es -128 y el máximo 127, ambos inclusive. El tipo de datos byte se puede utilizar para ahorrar memoria en grandes arrays, donde el ahorro de memoria realmente importa. También se pueden utilizar en lugar de int donde sus límites ayudan a aclarar el código. De hecho, el rango de una variable es limitado puede servir como una forma de documentación.
- **short**: es un entero de 16 bits. Su valor mínimo es -32.768 y el máximo 32.767 ambos inclusive. Se puede utilizar short para ahorrar memoria en grandes arrays, en situaciones en las que el ahorro realmente importa.
- **int**: es un entero de 32 bits. Su valor mínimo es -2.147.483.648 y el máximo 2.147.483.647, ambos inclusive. Generalmente este tipo es la elección predeterminada para valores enteros a no ser que haya una razón (como las mencionadas anteriormente, de ahorro de memoria) para elegir otro. Este tipo de dato normalmente será lo suficientemente grande para los números que casi cualquier programa vaya a utilizar, pero si necesita un rango más amplio, utilice "long".
- **long**: es un entero de 64 bits. Su valor mínimo es -9.223.372.036.854.775.808 y el máximo 9.223.372.036.854.775.807, ambos inclusive. Utilice este tipo de dato cuando necesite un rango de valores más amplio que el proporcionado por "int".
- **float**: utilizado para valores decimales (y también enteros, por ejemplo, que durante la ejecución del programa, puedan cambiar a decimal). Es un dato real (entero, decimal ó 0) en coma flotante (o "punto flotante", es una forma de notación científica usada principalmente en términos

computacionales, con la cual se pueden representar números reales extremadamente grandes y pequeños de una manera muy eficiente y compacta, y con la que se pueden realizar operaciones aritméticas). Su capacidad es de 32 bits y se le denomina de "precisión simple". Su rango de valores es: +/- 3,4 * 10^{-38} y +/- 3,4 * 10^{38}, ambos inclusive. Al igual que con byte y short, se recomienda usar un float (en vez de un double) si necesita ahorrar memoria en grandes array de números decimales. Este tipo de dato nunca debería ser usado para valores precisos como, por ejemplo, divisas.
    - Si se desean almacenar valores muy precisos, como divisas, es aconsejable usar la clase "BigDecimal" ("java.math.BigDecimal").
- **double**: del mismo uso que "float", pero su capacidad es de 64 bits y se le denomina de "precisión doble". Su rango de valores es: +/- 1,7 * 10^{-308} y +/- 3,4 * 10^{308}. Normalmente este tipo de dato es la opción predeterminada para valores decimales, por lo que si no se require tan altísima precisión, es mejor utilizar el tipo "float".
    - Al igual que se mencionó con el tipo "float", si se desean almacenar valores muy precisos, como divisas, es aconsejable usar la clase "BigDecimal" ("java.math.BigDecimal").
- **char**: almacena un carácter "Unicode". Es un solo carácter "Unicode" de 16 bits. Tiene un valor mínimo de 0 y un valor máximo de 65.535, ambos inclusive.
    - Los valores de tipo "char" van entre comillas simples. Por ejemplo: 'a'.
- **boolean**: solamente tiene dos valores posibles: "true" (verdadero) y "false" (falso). Utilice este tipo de datos como conmutadores para la evaluación de condiciones del tipo "verdadero/falso". Este tipo de dato representa solamente 1 bit de información.
    - No tienen correlación con ningún número, como ocurre en el lenguaje C, que teníamos al 0 como false y al 1 como true.

NOTA: estos datos primitivos (simples) también pueden ser utilizados en Java como objetos, envolviendo dichos datos primitivos en sus clases equivalentes (java.lang.Integer, java.lang.Double, java.lang.Byte, etc.), tratando de esta forma a dichos datos primitivos (simples) como si fueran objetos.

Tipo de datos simple	Clase equivalente
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean

Tipos de datos referencias

Se les llama referencias porque en realidad lo que se almacena en los mismos son punteros a zonas de memoria donde se encuentran almacenadas las estructuras de datos que los soportan.

Al declarar un objeto, perteneciente a una determinada clase, se está reservando una zona de memoria donde se almacenarán los atributos y otros datos pertenecientes a dicho objeto. Lo que se almacena en dicha zona de memoria respecto al objeto, es un puntero (referencia) a dicha zona de memoria.

En realidad, el momento en el que se realiza la reserva física del espacio de memoria es cuando se instancia el objeto, realizando la llamada a su constructor, y no en el momento de la declaración.

Existe un tipo de dato referencia (referencial) especial nominado por la palabra reservada "null", que puede ser asignado a cualquier variable de cualquier clase y que indica que el puntero no tiene referencia a ninguna zona de memoria (el objeto no está inicializado). Sirve para declarar una variable de tipo referencia, sin todavía asignarle un objeto.

Además, todos los tipos de datos primitivos (simples) vistos anteriormente, pueden ser declarados como referenciales (objetos), ya que existen clases que los engloban.

Dentro de este grupo de los datos referencias (referenciales), se encuentran:

- Objetos (instancias de una clase).
- Interfaces.
- Vectores.
- Strings.

Las interfaces, los vectores y los strings, son unas clases especiales.

Todos los tipos de datos que no son los tipos de datos primitivos, son considerados tipos de datos referencias (o referenciales), y son principalmente las clases, en las que se basa la programación orientada a objetos.

Strings en Java

A diferencia de otros lenguajes de programación, los **Strings** (cadenas) en Java, no son un tipo primitivo (simple) de datos, sino un dato de tipo referencia, que es un **objeto**, por lo que cuando se trabaje con **Strings** (cadenas), se trabajará con la clase **String**.

Mencionar que los valores de tipo **String** van entre comillas dobles, por ejemplo "Aquí estamos", mientras que los valores de tipo **char** van entre comillas simples, por ejemplo: 'a'.

Unicode

Para los caracteres alfanuméricos, Java utiliza la codificación **Unicode** de 16 bits, para permitir la inclusión de varios alfabetos.

Unicode es un estándar de codificación de caracteres diseñado para facilitar el tratamiento informático, transmisión y visualización de textos de múltiples lenguajes y disciplinas técnicas, además de textos clásicos de lenguas muertas. El término **Unicode** proviene de los tres objetivos perseguidos: universalidad, uniformidad y unicidad.

Unicode especifica un nombre e identificador numérico único para cada carácter o símbolo, el code point o punto de código, además de otras informaciones necesarias para su uso correcto: direccionalidad, mayúsculas y otros atributos.

Unicode trata los caracteres alfabéticos, ideográficos y símbolos de forma equivalente, lo que significa que se pueden mezclar en un mismo texto sin la introducción de marcas o caracteres de control.

El objetivo de **Unicode** es reemplazar los esquemas de codificación de caracteres existentes (por ejemplo "ASCII"), muchos de los cuales están muy limitados en tamaño y son incompatibles con entornos plurilingües.

Unicode se ha vuelto el más extenso y completo esquema de codificación de caracteres, siendo el dominante en la internacionalización y adaptación local del software informático. El estándar ha sido implementado en un número considerable de tecnologías recientes: XML, Java, sistemas operativos modernos, etc.

Literales

Son los valores de cadena (que pueden contener cualquier carácter), que pueden asignarse a variables y que pueden ser utilizados en expresiones del lenguaje.

Conversiones entre los diferentes tipos de datos

Pueden realizarse conversiones entre los distintos tipos de datos (incluso entre los tipos primitivos y referencias), bien de forma implícita (realizada automáticamente por el programa) o de forma explícita (realizada por el programador).

VARIABLES

Descripción

Una variable es un espacio que se reserva en la memoria del ordenador, para almacenar un dato, con el objeto de utilizar dicho dato posteriormente.

Las variables es la manera en la que indicamos al compilador el espacio en memoria que debe de reservar para almacenar los datos. Podemos acceder a un dato almacenado en la memoria por medio de una variable.

Toda variable tiene un nombre para poder identificarla y realizar operaciones con dicha variable.

Identificadores: reglas

Los identificadores son los nombres que se les da a las variables, así como a otros elementos del programa (clases, interfaces, atributos, métodos, etc.).

Reglas para la creación de identificadores:

- **Java** hace distinción entre mayúsculas y minúsculas, por lo tanto, nombres o identificadores como **var1**, **Var1** y **VAR1** son distintos.
- Pueden estar formados por cualquiera de los caracteres del código **Unicode**, por lo tanto, se pueden declarar variables como los siguientes ejemplos: **añoDeCreación**, **datoReciente**, etc.
    - Se acabó la época de los nombres de variable como **ano_de_creacion**.
- El primer carácter no puede ser un dígito numérico.
- No puede tener espacios.
- No puede tener caracteres que coincidan con los operadores del lenguaje.
    - Por ejemplo: *, /.... etc.
- La longitud máxima es prácticamente ilimitada.
    - Aunque se recomienda que los nombres no sean muy largos por comodidad y legibilidad.
- No puede ser una palabra reservada del lenguaje.
    - Por ejemplo: if, else, null, true, false, etc.
- No pueden existir dos identificadores con el mismo nombre y en el mismo ámbito.
    - Por ejemplo, puede haber dos variable "año", si cada una está en ámbitos diferentes, como por ejemplo, que estén en métodos diferentes.

Identificadores: recomendaciones

- Se recomienda sencillez y claridad. Es recomendable que al ver un identificador en el código se intuya rápidamente el dato que contiene (que sea fácil su interpretación).
    - Ejemplo: **presupuesto-2010**.
    - Hay que evitar identificadores del tipo: **a1**, **c4**, **me-56**. Ya que cuesta interpretar el dato que pueden contener.
- Se recomienda identificadores no muy cortos si se pierde legibilidad y no muy largos que cuesta mas leerlos en las expresiones. Se recomienda lo justo y necesario para que se entienda lo que almacena sin ser muy largo.
    - El nombre ideal para un identificador es aquel que no se excede en longitud (lo más corto posible) siempre que califique claramente el concepto que intenta representar.
- Por convenio, los nombres de las variables y los métodos deberían empezar por una letra minúscula. Ejemplo: **dato1**, **mes**, **calcular**.
- Por convenio, si los nombres de las variables y los métodos tienen varias palabras, el identificador debería empezar por una letra minúscula y el nombre de cada palabra que le siga, con mayúscula. Por ejemplo: **verDato**, **cargarFicheroContabilidad**.
- Por convenio, el nombre de las clases, comienza con mayúscula. Ejemplo: **Calculadora**, **Contabilidad**.

Datos de una variable

Cada variable debe tener un tipo de dato. Esto determina el tipo de dato que puede almacenar, así como el rango de valores que puede almacenar y las operaciones que se pueden realizar con dicho dato.

Por ejemplo, una variable de tipo entero (**int**) puede almacenar números sin decimales y puede realizar operaciones aritméticas, pero no puede contener texto.

Declaración de variables

Antes de trabajar con alguna variable debemos declararla en el programa.

Ejemplo:

```
int valor;
```

Aquí estamos reservando memoria para una variable de tipo entero (**int**) y la identificamos con el nombre (identificador) **valor**. Desde este momento, si en el programa hacemos referencia a la palabra **valor**, estamos haciendo referencia a esa porción de memoria y al valor que contiene. (En este ejemplo, la variable valor, de momento, no contendría ningún dato, ya que solo la hemos declarado, pero todavía no hemos almacenado ningún valor en ella).

Sintaxis para declarar una variable:

Tipo_Dato Nombre_Variable [= Valor];

Definimos el tipo de dato, el nombre y opcionalmente su valor.

La declaración de una variable se realiza de la misma forma que en el lenguaje C (ya que el lenguaje Java, deriva del lenguaje C). Siempre contiene el nombre (identificador de la variable) y el tipo de dato al que pertenece. El ámbito de la variable depende de la localización en el programa donde es declarada.

Asignación

Para asignar un dato a una variable, se realiza de la siguiente forma:

```
valor = 15;
```

Por supuesto, primero se ha tenido que declarar la variable, lo que todo (declaración + asignación) nos quedará como sigue:

```
// Declaración
int valor;
// Asignación
valor = 15;
```

Declaración y asignación

Se puede realizar una asignación en el momento de la declaración:

```
int valor = 15;
```

Declaración y asignación múltiple

Declaración múltiple sin inicializar y asignación posterior:

```
// Declaración de tres variables
int dato1, dato2, dato3;

// Asignación de dato a las tres variables previamente declaradas
dato1 = 3;
```

```
dato2 = 5;
dato3 = 7;
```

Declaración múltiple con inicialización:

```
// Declaración y asignación de tres variables en la misma línea del código
int dato1 = 3, dato2 = 5, dato3 = 7;
```

Uso de las variables

Las variables se utilizan en las expresiones del lenguaje. En el siguiente ejemplo, se utiliza una variable en una expresión que muestra el contenido de dicha variable:

```
// Declaración
int valor;
// Asignación
valor = 15;

// Se muestra el contenido de la variable, mediante la orden print:
System.out.println(valor);
```

Cambiar el valor de las variables

Un variable puede cambiar su valor en el transcurso del programa.

Ejemplo:

```
// Declaración
int valor;

// Asignación
valor = 15;

// Se muestra el contenido de la variable, mediante la orden print:
System.out.println(valor);

// Se modifica el contenido de la variable
valor = 50;

// Se muestra el nuevo contenido de la variable, que ya mostrará el nuevo dato
System.out.println(valor);
```

Inicialización de variables

Siempre se debe inicializar una variable que se va a usar, es decir, se le asignará un dato, ya sea en la misma declaración, o porteriormente, pero siempre antes de usar la variable en una expresión.

Al compilar el programa, el compilador de **Java** cuando llegue al nombre de una variable, leerá el contenido de dicha variable y siempre verificará que tenga un dato asignado. Si dicha variable no tiene un dato asignado, el programa no compilará y mostrará un error del tipo:

```
"The local variable valor may not have been initialized"
```

Si se declara una variable, no se inicializa, pero después no se utiliza dicha variable en el programa, la compilación no mostrará error, pero los IDEs de desarrollo, suelen mostrar un aviso de que la variable no está siendo utilizada:

```
"The value of the local variable valor is not used"
```

Por lo que, siempre se recomienda inicializar variables, de cualquiera de las siguientes formas:

```
// Declaración y asignación
int valor = 15;
```

o también de la forma:

```
// Declaración
int valor;
/*
Asignación.
Esto se escribiría en el lugar deseado del programa,
pero siempre antes de que la variable sea utilizada, por ejemplo,
en una expresión.
*/
valor = 15;
```

Uso de tipos de variables

Ejemplos del uso de los tipos de variables primitivas:

NOTA: si no se respetan los rangos, la compilación provocará error.

```
// Declaración y asignación de variable tipo byte (valores de -128 a 127)
byte mibyte = 60;

// Declaración y asignación de variable tipo short (valores de -32.768 a 32.767)
short mishort = 5700;

// Declaración y asignación de variable tipo int
// (valores de -2.147.483.648 a 2.147.483.647)
int miint = 125900;

// Declaración y asignación de variable tipo long
// (valores de -9.223.372.036.854.775.808 a 9.223.372.036.854.775.808)
// Hay que poner sufijo: L ó l
long milong = 12000000000L;

// Declaración y asignación de variable tipo float
//(+/- 3,4 * 10^-38 y +/- 3,4 * 10^38)
//Hay que poner sufijo: F ó f
float mifloat = 25.38F;

// Declaración y asignación de variable tipo double
// (+/- 1,7 * 10^-308 y +/- 3,4 * 10^308)
// Se puede poner opcionalmente sufijo (D ó d)
double midouble = 250000.38847363;

// Declaración y asignación de variable tipo "char"
// (Un carácter "UNICODE")
char micharCaracter = 'A';

// Declaración y asignación de variable tipo "boolean"
// ("true" o "false")
boolean miboolean = true;
```

El tipo String no es un tipo de dato primitivo. Pero se comporta como si lo fuera. Para indicar una cadena de caracteres se encierran entre comillas dobles.

Una cadena de tipo String va encerrada entre comillas dobles y no puede dividirse en varias líneas (para dividirlo se puede utilizar el operador de concatenación **+**).

Ejemplos:

```
// Declaración
String micadena;

// Asignación
micadena = "Buenas tardes";
```

```
// Declaración e inicialización a dato nulo (cadena vacía)
String micadena = "";
```

```
// Declaración e inicialización
String micadena = "Buenas tardes";
```

Ejemplo de concatenación de Strings:

```
// Declaración e inicialización
String micadena = "";

// Concatenación de cadenas
micadena = "Hola amigo, " + "estoy en casa," + " nos vemos."
```

Literales

Es la forma en la que se representa un valor en Java.

Un literal es la forma de representar un dato que se asigna a una variable.

El literal es el **dato** que se almacena en la **variable**.

Ejemplos:

LITERAL	TIPO DE DATO
"Gato"	String
"G"	String
'G'	char
123	int
8000L	long
3.14	double
3.14D	double
3.14f	float
'c'	char

"true"	String
true	boolean

Sufijos

En los literales numéricos puede forzarse un tipo determinado con un sufijo:

- Entero largo: L ó l.
- Float: F ó f.
- Double: D ó d.

Ejemplos:

```
long midato = 5L;
float mivalor = 5F;
```

En los sufijos que tenemos para identificar el tipo de dato no importa si son en minúscula o mayúscula. Por convención se utilizan en mayúscula, pues se puede confundir al leer el código una ele minúscula (l) con un uno (1). Hay que tener cuidado de no olvidarnos los sufijos cuando sean necesarios.

El uso se hace más evidente cuando los primitivos son empleados para realizar operaciones matemáticas con mayor precisión.

Por ejemplo, si tenemos un número definido como 14**D**, aunque el número catorce bien pudiera ser un entero (byte o int), éste se esta definiendo como un número double a través del sufijo **D**. Esto permite que el número sea manipulado con otros primitivos de tipo **double** para el trabajo con decimales (sin perder precisión).

Valores por defecto de una variable

Las únicas variables que se pueden dejar que se inicialicen por defecto son las variables de instancia (atributos de una clase), declarados en la parte superior de la clase y que tienen un ámbito de toda la clase.

En estas variables, no siempre es necesario asignar un valor cuando se declara. De esta forma, se declaran pero no se inicializan, por lo que quedará establecido un valor por defecto predeterminado por el compilador. En general, este valor suele ser cero o nulo, dependiendo del tipo de datos.

Algunos tipos de variables se inicializan a un valor por defecto:

Numeros: **0**
boolean: **false**
Objetos: **null**

Las variables locales deben inicializarse antes de utilizarse o se producirá un error en tiempo de compilación.

Constantes

Una constante es una variable del sistema que mantiene un valor inmutable a lo largo de toda la vida del programa. Una vez que se asigne un valor a dicha constante, este valor no puede ser modificado.

Tiene sentido utilizar una constante cuando sabemos que su valor no cambiará su valor durante toda la ejecución del programa: DIAS_SEMANA, HORAS_DIA, PI, etc.

Por convención, los nombres de las constantes se escriben en mayúscula.

No es necesario iniciar el valor de una constante al momento de declararla , pero una vez que se asigne un valor a dicha constante, este valor no puede ser modificado.

Las constantes en Java se definen mediante el modificador **final**.

Para declarar una constante se utiliza la sintaxis:

final tipo-de-dato identificador

Ejemplos de declaración e inicialización:

```
// Declarar una constante sin inicializar
final int MIVALOR;

// Inicializar posteriormente la constante
MIVALOR = 5;
```

```
// Declarar e inicializar una constante
final int MIVALOR = 12;
```

Ejemplo de intento de reasignación (modificación del valor) de una constante:

```
// Declarar e inicializar una constante
final int MIVALOR = 12;

// Intentamos dar un nuevo valor a la constante
MIVALOR = 7;     //DA ERROR ANTES DE COMPILAR.
                 //EN UNA CONSTANTE NO SE PUEDE REASIGNAR SU VALOR
```

VARIABLES: ÁMBITO

Descripción

El ámbito de una variable (scope) es el área del programa donde la variable existe y puede ser utilizada. Fuera de ese ámbito la variable no puede ser usada.

Cuando se declara una variable, dependiendo del lugar y la manera de declarar dicha variable, definirá el ámbito de dicha variable.

Una vez que se termina el ámbito de una variable y se sale de dicho lugar, la variable no puede ser usada y su valor se pierde.

En el caso de los objetos, el ámbito de una variable miembro (variable que pertenece a un objeto) es el de la usabilidad de un objeto. Un objeto es utilizable desde el momento en que se crea y mientras existe una referencia que apunte a él. Cuando la última referencia que lo apunta sale de su ámbito el objeto queda "perdido" y el espacio de memoria ocupado por el objeto puede ser recuperado por la **JVM** cuando lo considere oportuno. Esta recuperación de espacio en memoria se denomina **recogida de basura**.

NOTA IMPORTANTE: no es buena práctica usar variables con un ámbito mayor del que se va a necesitar, ya que puede originar un código dificil de mantener.

Tipos de ámbito

Dependiendo del lugar donde esté declarada una variable en una aplicación Java, estos son los tipos de ámbitos (de menor ámbito a mayor ámbito):

- Bloque: el bloque de código delimitado entre una pareja de llaves (apertura y cierre). Por ejemplo, una estructura de control **if** con sus llaves, las llaves de un método, así como una simple pareja de llaves puestas en cualquier lugar del código.
    - Así como todos los bloques anidados dentro de dicho bloque.
    - También se puede decir que la variable tiene un ámbito **local** respecto al bloque en el que está declarada.
    - El ámbito de la variable comenzará después de la llave de abrir (o llave izquierda) **{** y terminará con la llave de cerrar (o llave derecha) **}**.
- Método: el método de una clase.
    - Así como todos los bloques que haya dentro de dicho método.
    - También se puede decir que la variable tiene un ámbito **local** respecto al método en el que está declarada.
- Argumento de Método: el método donde se recibe dicho argumento. (el mismo nivel de ámbito que el anterior de **Método**).
    - Así como todos los bloques que haya dentro de dicho método.
- Clase: la clase.
    - Así como todos los métodos de dicha clase y todos los bloques y bloques anidados que haya dentro de dichos métodos. Es decir, la clase completa. Por lo que también tendrá el ámbito completo, pero independientemente para cada uno de los objetos que se creen de dicha clase, pero no será común (global) a todos los objetos creados de dicha clase.
    - A esta variable de clase, también se le llama **variable de instancia** o **variable de objeto**, ya que cada objeto creado con dicha clase tiene una (cada objeto creado con dicha clase tiene dicha variable). Son creadas (normalmente al principio de la clase) y permanecerán en memoria tanto tiempo como dure la instancia (el objeto) de la clase. Estas variables almacenan el estado del Objeto. Su ámbito ocupa toda la clase entera. Estos valores existen y conservan su valor desde que la clase es inicializada hasta que la clase es reinicializada o no se hace referencia a ella mas. Las variables de instancia deben ser declaradas en la clase (al principio de la clase), no en un método; de esta forma todos los métodos de dicha clase tendrán acceso a dicha variable.
- Global de objeto: (esto es una variable global a todos los objetos creados de la misma clase).
    - Ejemplo. Se creará en la clase (en la parte superior): `static int mivalor = 6;`
    - A esta variable global de clase, también se le llama **variable de clase**, ya que todos los objetos creado con dicha clase, comparten dicha variable global. Y al ser global para todos

los objetos creados con dicha clase, desde cualquiera de dichos objetos, se puede tener acceso a dicha variable global.

NOTAS:

- Como puede observarse, NO existen las variables globales de aplicación completa, como ocurre en otros lenguajes.
- Esto no es un defecto del lenguaje, sino todo lo contrario. La utilización de variables globales resulta peligrosa, ya que puede ser modificada en cualquier parte del programa y por cualquier procedimiento.
- Además, a la hora de utilizarlas hay que buscar dónde están declaradas para conocerlas y dónde son modificadas para evitar sorpresas en los valores que pueden contener.

ENTRADA Y SALIDA EN JAVA

Descripción

Java permite la entrada y salida de datos por la consola del sistema.

Salida normal

Es la salida normal de **Java** y se realiza mediante **print** y **println**.

Con **print** se imprime en la misma linea y con **println** se imprime en una línea nueva.

Ejemplo de **print** y de **println**:

```
// Se imprime en la misma línea
System.out.print("Hola amigos, ");
System.out.print("aquí estamos. ");
System.out.print("Adiós.");

// Se añaden dos líneas vacías de separación
System.out.println();
System.out.println();

// Se imprimen tres líneas
System.out.println("Hola amigos, ");
System.out.println("aquí estamos. ");
System.out.println("Adiós.");
```

Ejemplo que imprime varios datos de salida:

```
// Declaración de variables
int edad = 46;
String nombre = "Luisa";

// Se muestra un mensaje en la consola, concatenando cadenas y variables
System.out.println("Hola, " + nombre + " tiene " + edad + " años.");
```

EJERCICIO: Trabajo con la salida en Java, mediante print.

1. Crear un proyecto Java llamado **Cadenas**.
2. Crear dentro del proyecto una clase llamada **Miclase**.
3. Dentro de la clase, escribir el código que haga lo siguiente:
    - Declarar las variables: **nombre**, **apellidos**, **edad**, **población**, asignando a cada variable su tipo de dato correspondiente, y los valores respectivos: **Juana**, **Ruano Martos**, **36**, **Madrid**.
    - Construir la expresión necesaria para que al ejecutar el programa, aparezca la siguiente expresión: **Hola, soy Juana Ruano Martos; tengo 36 años y soy natural de Madrid**.
4. Ejecutar la aplicación y asegurarse que en la consola sale exactamente la expresión que se pide.
5. Añadir los comentarios deseados.
6. Enviar la clase **Miclase** al tutor.
    - NOTA: para enviar la clase al tutor, en el Package Explorer de Eclipse, pulsar con el botón derecho - **Copy**. Después, pegar en el lugar deseado (y ya será un archivo de texto normal y corriente).

Salida con formato: printf

Con **printf** se puede formatear la salida, ofreciendo mas flexibilidad que las órdenes **print** y **println**.

La sintaxis de printf es la siguiente:

```
System.out.printf(Formato de la salida, variable1, variable2, ...);
```

- Formato de salida: se especificará el formato deseado de la cadena, incluyendo los códigos correspondientes a cada tipo de variable:
  - **d%**: impresión del dato de variable numérica.
  - **s%**: impresión del dato de variable String (cadena).
- Variables: aquí se pondrán las variables en el orden en el que aparecen en la zona del formato de salida.

Ejemplo que muestra datos mediante **printf**:

```
// Declaración de variables
int edad = 46;
String nombre = "Luisa";

// Se muestra un mensaje en la consola, concatenando cadenas y variables
System.out.printf("Hola %s, tu edad es de %d años", nombre, edad);
```

EJERCICIO: Trabajo con la salida en Java, mediante printf.

1. Realizar el mismo ejercicio anterior, pero utilizando **printf**.
2. Enviar la clase **Miclase** al tutor.

Entrada de datos: clase Scanner

La entrada básica por terminal se realiza creando un objeto de la clase **Scanner**.

Mediante la clase **Scanner** se pueden pedir datos al usuario en tiempo de ejecución de un programa que se muestre por la consola. Por ejemplo, si el programa nos pide nuestros datos para introducirlos en una base de datos, o nos pide una contraseña para acceder al sistema, deberemos programar que dichos datos se le pidan al usuario en el momento oportuno.

Para pedir dichos datos al usuario se utiliza la clase **Scanner**, que nos permitirá recoger dicho dato y almacenarlo en una variable, para después utilizar dicha variable para trabajar con dicho dato.

Cuando el programa encuentre la clase **Scanner**, se detendrá y pedirá dicho dato al usuario. Cuando el usuario escriba dicho dato y pulse la tecla **Intro**, continuará la ejecución del programa.

La clase **Scanner** se encuentra en el paquete java.util.

La clase **Scanner** permite leer valores de varios tipos.

Para utilizar la clase Scanner deberemos importar dicha clase Scanner, que se encuentra en el paquete java.util. Lo haremos de la siguiente manera:

```
//Importación de la clase "Scanner" del paquete "java.util"
import java.util.Scanner;
```

Quedando la parte superior de la clase de la siguiente manera:

```
import java.util.Scanner;

public class miclase {
        …
```

Ejemplos:

```
// Declaración de variables para permitir recoger diversos datos
int mientero;
float mifloat;
double midouble;
String mistring = "";

// Creación de un objeto de la clase Scanner llamado miescaner,
// que permitirá recoger los datos en tiempo de ejecución del programa.
Scanner miescaner = new Scanner(System.in);

// Leer un String
// Con next leemos una palabra (hasta que haya un espacio).
// Con nextLine una línea entera (hasta que haya un Intro).
System.out.print("Ingrese un String: ");
mistring = miescaner.next();

// Leer un int
System.out.print("Ingrese un entero : ");
mientero = miescaner.nextInt();

//Leer un float
System.out.print("Ingrese un float : ");
mifloat = miescaner.nextFloat();

// Leer un double
System.out.print("Ingrese un double : ");
midouble = miescaner.nextDouble();

// Se muestran los datos
System.out.println("String: " + mistring);
System.out.println("int: " + mientero);
System.out.println("float: " + mifloat);
System.out.println("double: " + midouble);
System.out.println("String: " + mistring);
```

EJERCICIO: Trabajo con la clase Scanner para recoger datos en tiempo de ejecución.

1. Crear un proyecto llamado **Miproyecto**.
2. Crear una clase llamada **Datos**.
3. Crear el código correspondiente para que en su ejecución se pidan los siguientes datos al usuario:
   - Nombre.
   - Apellidos.
   - Dirección.
   - Población.
   - Edad.
   - Sueldo.
4. Generar una expresión de salida con el siguiente resultado de ejemplo:
   - Bienvenido/a Juan; te recordamos que según tus apellidos "González López" y que Dirección --- \Calle Delicias, 38\ está en la población de Madrid, no te corresponde generar el documento. Debido a tu edad de 54 años y tu sueldo de ***1200 €*** tampoco te corresponde prestación adjunta.
5. Enviar al tutor el archivo **.java** correspondiente a la clase **Datos**.

CONVERSIONES DE TIPO

Descripción

La conversión de tipos (type casting) consiste en la transformación de un tipo de dato en otro. Esto se hace para tomar las ventajas que pueda ofrecer el tipo a que se va a convertir.

Por ejemplo, los valores de un conjunto más limitado, como números enteros (por ejemplo **int**), se pueden almacenar en un formato más compacto y más tarde convertidos a un formato diferente que permita las operaciones que anteriormente no eran posibles, tales como trabajo con decimales, (por ejemplo **float**).

Tipos de conversiones

Hay dos tipos de conversión:

- Implícita: realizada automáticamente por Java.
- Explícita: realizada explícitamente por el programador.

Conversión implícita

La conversión implícita la realiza automáticamente el compilador. En la conversión implícita se convierte un tipo de dato de menor rango a un supertipo (tipo de dato de mayor rango); este tipo de conversión lo realiza el compilador, ya que no hay pérdida de datos si, por ejemplo, se pasa dato entero **int** a dato **long**.

Si al hacer la conversión de un tipo a otro se da la circunstancia de que los dos tipos son compatibles, y el tipo de variable destino es de un rango mayor al tipo de la variable que se va a convertir (promoción), entonces la conversión entre tipos es automática (implícita), es decir, la máquina virtual de Java (JVM) tiene toda la información necesaria para realizar la conversión sin necesidad de "ayuda" externa (del programador). A esto se le conoce como conversión implícita (o conversión automática).

Ejemplo:

```
//Variable que almacena el mayor rango de datos de los dos
short midatomayor;

//Variable que almacena el menor rango de datos de los dos
byte midatomenor = 40;

//Se la asigna a la variable que tiene mas capacidad de las dos
//el dato de la que menos capacidad tiene de las dos.
//(sin problema, ya que si ponemos algo pequeño, dentro de algo mas grande,
//sobra espacio).
midatomayor = midatomenor;
```

Se está asignando un valor byte (que puede tener valores entre -128 a 127) a una variable tipo short (que tiene valores entre -32768 a 32767), sin necesidad de una conversión explicita por parte del programador.

A continuación se presenta un resumen de algunas de las asignaciones que no requieren de una conversión explicita, es decir, que tienen una conversión implícita (automática) en Java.

Cada tipo de los mostrados a continuación, puede ser asignado de forma implícita a cualquiera de los tipos que están a su izquierda (de derecha a izquierda).

double <= float <= long <= int <= short <= byte

Conversión explícita mediante clases

En la conversión explícita, el compilador no es capaz de realizarla por sí solo y por ello debe definirla (explícitamente) el programador en el código del programa.

Se utilizan clases para realizar la conversión entre los datos.

Ejemplos de conversión explícita mediante clases:

String a float:

```
//Declaración e inicialización de cadena
String miscaracteres = "10";

//Variable para almacenar el dato float al que se convertirá la cadena
float midato = 0;

//Con la clase "Float" y su método "parsefloat"
//pasando como argumento la variable de la cadena
//y se transforma dicha cadena en tipo "float"
midato = Float.parseFloat(miscaracteres);

//Se realiza una operación con el dato ya convertido
System.out.println(midato - 3);
```

String a int (el mismo ejemplo anterior, pero utilizando lo siguiente):

```
//Declaración e inicialización de cadena
String miscaracteres = "10";

//Variable para almacenar el dato int al que se convertirá la cadena
int midato = 0;

//Con la clase "Integer" y su método " parseInt"
//pasando como argumento la variable de la cadena
//y se transforma dicha cadena en tipo "int"
midato = Integer.parseInt(miscaracteres);

//Se realiza una operación con el dato ya convertido
System.out.println(midato - 3);
```

OJO: hay que tener cuidado al convertir tipos, ya que en este ejemplo se puede convertir a tipos numéricos porque la cadena tiene las características de un número, si tuviésemos una palabra "lentejas" como cadena en lugar de "10" surgiría un error ya que no se puede convertir a número "lentejas", ya que lentejas no es un número.

int a String:

```
//Número que se desea convertir a cadena
int mivalor = 15;

//Cadena que inicializamos vacía y que almacenará
//el número después de su conversión a cadena
String micadena = "";
//Con la clase "String" y su método "valueOf"
//pasando como argumento la variable del número
//y se transforma dicho número en tipo "String" (cadena)
micadena = String.valueOf(mivalor);

//Se muestra el número, ya convertido en cadena
System.out.println(micadena);
```

Conversión explícita mediante el operador "cast"

Es otro tipo de conversión explícita, pero ahora usando el operador "cast", llamado "casting". El operador "cast" realiza este proceso, es decir convierte datos, variables o expresiones a un nuevo tipo de dato.

Su sintaxis es:

variable_que_almacena_el_dato_final = (nuevo_tipo) dato_a_convertir_al_tipo_del_dato_final;

int a float:

```
//Entero que se desea convertir a float
int alfa = 50;

//Variable para almacenar el dato float.
//Se le asigna el resultado de la conversión
float dato = (float) alfa;

//Se muestra el dato
System.out.println(dato);
```

Como se aprecia, solo hay que anteponer al dato deseado, el tipo de dato al que se desea convertir, entre paréntesis.

float a int:

El resultado será el valor de float, pero sin la parte decimal.

Ejemplo con un método:

```
mipotencia = Math.pow ((double)5, (double)3);
```

Como se observa, por ejemplo, se puede usar pow(), directamente con los argumentos requeridos (si estos son numéricos), pero transformándolos con el operador "cast".

Ejemplo con expresión matemática:

y = 3 * Math.pow(x, (double)3) - Math.pow(x, (1/3.0)) + 4 * Math.pow(x, (double)2));

Más ejemplos de conversiones

double a int, (usando el operador "cast"):

```
int alfa1 = 10;
double alfa2 = 3.1416;
alfa1= (int) alfa2;
```

int a String, usando clases:

```
int zeta = 50;
String alfa = String.valueOf(zeta);
```

String a double, (usando clases):

```
String alfa = "3.5";
double beta= 0;
beta = Double.parseDouble(alfa);
```

OPERADORES

Descripción

Los operadores son signos que permiten realizar operaciones con los datos. Son parte fundamental de las expresiones. Los operadores son partes indispensables en la construcción de expresiones.

El valor y tipo que devuelve, depende de cada operador y del tipo de sus operandos. Por ejemplo, los operadores aritméticos devuelven un número como resultado de su operación.

Expresión

Existen muchas definiciones técnicas para el término expresión. Puede decirse que una expresión es una combinación de operandos ligados mediante operadores.

Los operandos pueden ser variables, constantes, funciones, literales, etc. y los operadores cualquiera de los mostrados en este tema.

Notación para los operadores

Hay 3 tipos de notaciones para los operadores:

- Prefija: el operador aparece antes que su operando. Ejemplo: **++**dato.
- Posfija: el operador aparece después que su operando. Ejemplo: dato**++**.
- Infija: el operador aparece entre dos operandos. Ejemplo: 5 **+** 3.

Operadores según su número de operandos

Los operadores, según el número de operadores con los que realicen su función, se clasifican de la siguiente forma:

- Unarios: realizan su función sobre un operando.
- Binarios: realizan su función sobre dos operandos.
- Ternarios: realizan su función sobre tres operandos.

Operadores unarios

Los operadores que requieren un operando son llamados **operadores unarios**. Por ejemplo, el operador **++** es un operador unario que incrementa el valor de su operando en una unidad (ejemplo: mivalor++), así como el operador **--** es un operador unario que decrementa el valor de su operando en una unidad (ejemplo: mivalor--).

Los operadores unarios en Java pueden utilizar tanto la notación prefija (ejemplo: ++mivalor), como la posfija (ejemplo: mivalor++). Estos operadores ("++" y "--") entran en la categoría de Operadores aritméticos:

Operador	Ejemplos	Descripción
++	mivalor++	Incrementa mivalor en 1. Se evalúa al valor anterior al incremento
++	++mivalor	Incrementa mivalor en 1. Se evalúa al valor posterior al incremento
--	mivalor--	Decrementa mivalor en 1. Se evalúa al valor anterior al incremento
--	--mivalor	Decrementa mivalor en 1. Se evalúa al valor posterior al incremento

Operadores binarios

Los operadores que requieren dos operandos se llaman operadores binarios. Por ejemplo el operador **=** es un operador binario, que asigna el valor del operando del lado derecho al operando del lado izquierdo.

Todos los operadores binarios en Java utilizan notación infija, lo cual indica que el operador aparece entre sus operandos:

operando1 **operador** operando2

Operadores ternarios

Los operadores ternarios requieren tres operandos. El lenguaje Java tiene el operador ternario, "?", que es una sentencia similar a la "if-else".

Este operador ternario usa notación infija; y cada parte del operador aparece entre operandos:

expresión1 ? operación2 : operación3

Ejemplo:

```
//Variables
byte miedad = 15;
String mimenor = "No puedo votar todavía, soy menor de edad.";
String mimayor = "Sí puedo votar, soy mayor de edad.";

//Sentencia condicional utilizando el operador ternario
String mirespuesta = miedad < 18 ? mimenor : mimayor;

//Se muestra el resultado
System.out.println(mirespuesta);
```

Tipos de operadores

Los operadores de Java se pueden dividir en las siguientes cuatro categorías:

- Operadores aritméticos.
- Operadores de comparación y condicionales.
- Operadores de asignación.
- Operadores a nivel de bits y lógicos.
- Operadores de concatenación.

Operadores aritméticos

Los operadores aritméticos cubren las operaciones aritméticas (suma, resta, multiplicación, división), además del operador módulo (división entera).

Operador	Ejemplo	Descripción
+	op1 + op2	Suma op1 y op2
-	op1 - op2	Resta op2 de op1
*****	op1 * op2	Multiplica op1 por op2
/	op1 / op2	Divide op1 entre op2
%	op1 % op2	Calcula el resto de dividir op1 entre op2

El tipo de los datos devueltos por una operación aritmética depende del tipo de sus operandos; por ejemplo, si se suman dos enteros, se obtiene un entero, con el valor de la suma de los dos enteros.

Los operadores **+** y **-** tienen versiones unarias que realizan las siguientes operaciones:

Operador	Ejemplo	Descripción
+	+op	Convierte op a entero si es un byte, short o char
-	-op	Niega aritméticamente op

Operadores de asignación

El operador de asignación es **=,** que se utiliza para asignar un valor a otro. Por ejemplo:

```
int contador = 0;
```

Inicia la variable contador con un valor 0.

Java además proporciona varios operadores de asignación, que en combinación con los operadores aritméticos, permiten realizar atajos en la escritura de código. Permiten realizar operaciones aritméticas, lógicas y de asignación con un único operador.

Supongamos que necesitamos sumar un número a una variable y almacenar el resultado en la misma variable:

```
i = i + 2;
```

Se puede abreviar esta sentencia con el operador de atajo "+=":

```
i += 2;
```

Operadores combinados de asignación, así como sus equivalentes largos:

Operador	Ejemplo	Equivalente a
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
***=**	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
^=	op1 ^= op2	op1 = op1 ^ op2

Operadores condicionales (o de comparación)

Un operador de comparación compara dos valores y determina la relación existente entre ambos. Por ejemplo, el operador "!=" devuelve verdadero (true) si los dos operandos son distintos. La siguiente tabla resume los operadores condicionales (de comparación) de Java:

Operador	Ejemplo	Devuelve verdadero si
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 son distintos
?		USADO EN EL OPERADOR TERNARIO

Además Java soporta un operador ternario (o condicional): **?**. Se comporta como una versión reducida de la sentencia **if-else**:

expresion ? operacion1 : operacion2

El operador **?** evalúa la expresion y devuelve **operación1** si es cierta, o devuelve **operación2** si expresion es falsa.

Operadores lógicos

Operan con valores de tipo booleano (boolean), siendo el resultado también de tipo **boolean**.

Los operadores lógicos en Java son los siguientes:

Operador	Ejemplo	Devuelve verdadero si...
&& (Es el operador Y [AND])	op1 && op2	op1 y op2 son ambos verdaderos
|| (Es el operador O [OR])	op1 || op2	op1 o op2 es verdadero
! (Es el operador NO [NOT])	! op1	niega op1

Ejemplos de operaciones lógicas (tabla de verdad):

Tabla de verdad para el operador "&&": (Y)	
EXPRESIÓN	RESULTADO
false && false	false
false && true	false
true && false	false
true && true	true
Tabla de verdad para el operador "||": (O)	
false && false	false
false && true	true
true && false	true
true && true	true

Operadores de concatenación

El lenguaje Java "sobrecarga" la definición del operador **+** para permitir la concatenación de cadenas.

En una expresión, el valor que esté a la derecha del operador "+" se convierte automáticamente en una cadena de caracteres.

Ejemplo:

```
//Variable
byte miedad = 26;

//Se muestra el resultado
System.out.println("Tengo una edad de: " + miedad);
```

Prioridad de operadores

Cuando en una sentencia aparecen varios operadores el compilador deberá de elegir en qué orden aplica los operadores. A esto se le llama prioridad o precedencia de operadores.

Los operadores con mayor precedencia son evaluados antes que los operadores con una precedencia menor.

Cuando en una sentencia aparecen operadores con la misma precedencia, dichos operadores son evaluados de izquierda a derecha.

Se puede indicar explícitamente al compilador de Java cómo se desea que se evalúe la expresión, mediante los paréntesis "( )". Para hacer que el código sea más fácil de leer y mantener, es preferible ser explícito e indicar con paréntesis "( )" los operadores que deben ser evaluados primero.

Los operadores se organizan de acuerdo al nivel del precedencia de cada uno:

1. Paréntesis.
2. Unarios.
3. Aritméticos.
4. Condicionales.
5. Relacionales.
6. Booleanos.
7. Asignación.

La siguiente tabla muestra la precedencia asignada a los operadores de Java (de mayor a menor precedencia). Los operadores en la misma línea tienen la misma precedencia:

Descripción	Operadores
Paréntesis	( )
operadores posfijos	op++ op--
operadores unarios	++op --op +op -op !
multiplicación y división	* / %
suma y resta	+ -
operadores condicionales	< > <= => instanceof
igualdad	== !=
AND (booleano)	&&
OR (booleano)	||
condicional (ternario)	? :
asignación	= += -= *= /= %= ^=

Se puede observar que la última categoría es la de asignación. Esto es lógico, ya que se debe calcular primeramente la expresión antes de asignar un resultado a una variable.

Si todos los operadores tienen un nivel idéntico de precedencia se evalua la expresión de izquierda a derecha.

Uso de paréntesis

Se utilizan para aislar una porción de la expresión de forma que el cálculo se ejecute de forma independiente. Anula cualquier regla de precedencia.

Como en matemáticas, podemos anidar los paréntesis. Se comenzara a evaluar los internos hasta llegar a los externos.

Los paréntesis no disminuyen el rendimiento de los programas. Por lo tanto, agregar paréntesis no afecta negativamente al programa, y lo hace mucho mas claro y legible.

Los paréntesis se utilizan para anular la predencia de operadores y/o para aclarar y hacer mas legible el código de programación.

ESTRUCTURAS DE CONTROL

Descripción

Las estructuras de control son construcciones hechas a partir de palabras reservadas del lenguaje, que permiten modificar el flujo de ejecución de un programa.

Bloques con varias sentencias y llaves

Un bloque es un conjunto de instrucciones encerradas entre llaves "{ }".

Las llaves son obligatorias si hay mas de una instrucción.

Si hay una sola instrucción, se pueden omitir las llaves, aunque es recomendable ponerlas en todos los casos (bloques con una sola instrucción o bloques con varias instrucciones) por claridad en el código de programación.

ESTRUCTURAS DE CONTROL: SENTENCIAS CONDICIONALES

Descripción

Las sentencias condicionales permiten ejecutar instrucciones, dependiendo si se cumplen ciertas condiciones.

Estructura if simple

Permite ejecutar un bloque de instrucciones si se cumple una determinada condición.

Sintaxis:

if (expresión) {
Bloque instrucciones
}

Ejemplo:

```
//Variable
byte miedad = 26;

//Se evalúa una condición
if (miedad >= 18)
        System.out.println("Soy mayor de edad.");
```

Como solo hay una instrucción dentro del bloque **if**, no es obligatorio el uso de llaves.

Pero se recomienda usar siempre las llaves, por claridad en el código:

```
//Variable
byte miedad = 26;

//Se evalúa una condición
if (miedad >= 18) {
        System.out.println("Soy mayor de edad.");
}
```

Estructura if compuesta: if-else

Permite ejecutar un bloque de instrucciones si se cumple una determinada condición, y si no se cumple dicha condición, se ejecutará otro bloque de instrucciones.

Sintaxis:

if (expresión) {
Bloque instrucciones
}
else {
Bloque instrucciones
}

Ejemplo:

```
// Se evalúa una condición
if (miedad >= 18) {
        System.out.println("Soy mayor de edad.");
}
else {
```

```
        System.out.println("No soy mayor de edad.");
}
```

Estructuras if anidadas

En determinadas ocasiones, se pueden anidar estructuras **if-else**, de forma que se evalúe una condición posteriormente, si anteriormente no se ha cumplido otra.

Ejemplo:

```
//Variable
float nota = 6F;

//Se evalúa la condición
if (nota >= 0 && nota < 5) {
        System.out.println("Suspenso");
}
else {
        if (nota >= 5 && nota < 6) {
                System.out.println("Suficiente");
        }
        else {
                System.out.println("Has sacado muy buena nota");
        } //FIN "if" anidado
} //FIN if
```

Estructura if múltiple: else-if

Permite estructuras **if-else** más complejas.

Esta estructura está indicada para cuando en cada **else if** se desea evaluar cosas diferentes.

Por ejemplo, para validar un formulario, ya que en cada **else if** se hace alusión a cada control del formulario.

```
if (condición1) {
        Acciones1;
}
else if (condición2) {
        Acciones2;
}
else if (condición3) {
        Acciones3;
}
else {
        Acciones si no se cumple ninguna condición anterior;
}
```

Estructura switch

Permite evaluar la condición de cada uno de los bloques de instrucciones existentes en la estructura. Si se cumple la condición del primer bloque, se ejecutan sus instrucciones, y si después existe una instrucción **break**, finaliza la estructura.

En primer lugar se evalúa la primera expresión. Si dicha expresión coincide con el valor establecido para dicha primera condición, se ejecutarán sus instrucciones asociadas y terminará la estructura si existe una instrucción **break**, si no, también ejecutaría el resto de instrucciones.

Sintaxis:

```
switch (expresión) {
        case valor1: instrucciones1;
        break;
        case valor2: instrucciones2;
        break;
        …
        default: instrucciones si no se cumple ninguna de las anteriores.
}
```

Ejemplo:

```
//Variables
int micodigo = 7;
int resultado = 0;

//Se evalúa la variable "micodigo"
switch (micodigo) {
        case 3:
                resultado = micodigo * 2;
                break;
        case 7:
                resultado = micodigo + 5;
                break;
        case 11:
        case 12:
        case 16:
                resultado = 2;
                break;
        case 20:
                resultado = micodigo * 9;
                break;
        default:
                resultado = micodigo;
                break;
}

//Se muestra el resultado
System.out.println("El resultado es: " + resultado);
```

EJERCICIOS: Trabajo con estructuras de control condicionales.

1. Crear un programa que recoja del usuario su nombre y su edad, y que dependiendo si la edad es 18 o mayor, o menor que 18, muestre por pantalla el texto:
   - **Hola ---, bienvenido a la web**.
2. Crear un programa que recoja del usuario su nombre y su edad, y que dependiendo si la edad es 18 o mayor, o menor que 18, muestre por pantalla el texto:
   - **Hola ---, puedes entrar a la web** (en caso de se mayor de edad), o que muestre el texto: **Lo siento ---, no puedes entrar al ser menor de edad** (en caso de ser menor de edad).
3. Utilizando una estructura condicional **if** anidada, crear un programa que recoja un número del usuario entre 0 y 10, ambos inclusive, y que dependiendo de que número sea, se muestre el siguiente mensaje de texto por pantalla:
   - 0, 1, 2: **La nota es MUY DEFICIENTE**.
   - 3: **La nota es DEFICIENTE**.
   - 4: **La nota es INSUFICIENTE**.
   - 5: **La nota es SUFICIENTE**.
   - 6: **La nota es BIEN**.
   - 7, 8: **La nota es NOTABLE**.
   - 9, 10: **La nota es SOBRESALIENTE**.

4. Crear un programa igual que el anterior, pero que valide la nota introducida por el usuario, de tal manera que si dicha nota es inferior a 0 o superior a 10, aparezca por pantalla el mensaje: **NOTA NO VÁLIDA**.
5. Crear un programa igual que el anterior, pero utilizando una estructura de control condicional **switch**, para que de esta forma quede el código más breve y compacto.

ESTRUCTURAS DE CONTROL: BUCLES

Descripción

Los bucles son estructuras de control que permiten repetir bloques de instrucciones, de forma repetitiva e iterativa, tanto un número determinado de veces, así como también dependiendo de que se cumpla cierta condición.

Estructura for

El bucle **for** se ejecuta un número definido de veces.

Se utilizará el bucle **for** cuando se conozca el número exacto de veces que ha de repetirse un determinado bloque de instrucciones.

Sintaxis:

for (inicialización; condición; incremento) {
bloque de instrucciones para ejecutar un número determinado de veces
}

- inicialización: es una instrucción que se ejecuta una sola vez al inicio del bucle, es una variable que inicializa un dicha variable que hará de contador de vueltas, es decir, de contador de veces.
- condición: es una expresión lógica, que se evalúa al inicio de cada nueva iteración (vuelta) del bucle. En el momento en que dicha expresión se evalúe a **false** (no se cumpla), se dejará de ejecutar el bucle y el control del programa pasará a la siguiente instrucción (a continuación del bucle for).
- incremento: es una instrucción que se ejecuta en cada iteración del bucle como si fuera la última instrucción dentro del bloque de instrucciones. Generalmente se trata de una instrucción de incremento o decremento de alguna variable. Lo más habitual es incrementar la variable contador.

NOTA:

- Cualquiera de estas tres cláusulas de la estructura **for** puede estar vacía, aunque siempre hay que poner los puntos y coma.

Ejemplo:

```
//Variable contador para el bucle
int i = 0;

//Variable para controlar el número de veces
//que se ejecuta el bucle
int misveces = 9;

//Bucle que muestra una línea por cada
//vuelta que da el bucle
for (i = 0; i < misveces; i++) {
        System.out.println("Vuelta del bucle: " + i);
}
```

Estructura while

Este tipo de bucle se ejecuta mientras se cumpla una condición.

Se utilizará cuando no se conoce exactamente el número de veces que se ejecutará el bucle. En este bucle, si no se cumple la condición, no se ejecutará ni una sola vez.

Sintaxis:

while (condición) {

bloque instrucciones que se ejecutan mientras se cumpla la "condición"
}

Ejemplo:

```
//Variable
int mivalor = 7;
int milimite = 10;

//Bucle que ejecuta tantas líneas como el resultado
//de la diferencia entre "milimite" y "mivalor"
while (mivalor < milimite) {
        mivalor++;
        System.out.println("mivalor = " + mivalor);
}
```

OJO:

- En este tipo bucle, si la condición nunca se cumple, será un **bucle infinito** (bucle sin fin).

Estructura do while

Este tipo de bucle se ejecuta mientras se cumpla una condición.

Se utilizará cuando no se conoce exactamente el número de veces que se ejecutará el bucle. En este bucle, si no se cumple la condición, se ejecutará, por lo menos, una sola vez.

Sintaxis:

do {
bloque instrucciones que se ejecutan mientras se cumpla la "condición"
} while (condición)

Ejemplo:

```
//Variable
int mivalor = 12;
int milimite = 10;

//Bucle que ejecuta tantas líneas como el resultado
//de la diferencia entre "milimite" y "mivalor".
//Se ejecuta como mínimo una vez, aunque no se
//cumpla la condición.
do {
        System.out.println("mivalor = " + mivalor);
} while (mivalor < milimite);
```

OJO:

- En este tipo de bucle, si la condición nunca se cumple, será un bucle "infinito" ("bucle sin fin").

Estructura for each

La estructura **for each** es la adecuada para el recorrido de los elementos de una matriz.

Sintaxis:

for (tipoMatriz variableElemento : matriz)

- tipoMatriz: tipo de dato de la matriz.
- variableElemento: es la variable que representa a cada elemento de la matriz, y mediante el cual se podrá acceder a cada uno de dichos elementos de la matriz.

- matriz: nombre de la matriz.

Ejemplo:

```
//Declaración y asignación de la matriz
String[] ciudades = {"Madrid", "Londres", "Estocolmo"};

//Recorrido de los elementos de la matriz
for (String mielemento : ciudades) {
        System.out.println(mielemento);
}
```

ESTRUCTURAS DE CONTROL: SALIDA FORZADA

Descripción

La salida forzada permite salir de forma controlada de las estructuras de control.

En Java existen dos formas de realizar un salto incondicional en el flujo normal de un programa. Es con las instrucciones: **break** y **continue**.

Salida forzada de una estructura de control

La instrucción **break** sirve para abandonar una estructura de control, tanto de las alternativas (**if-else** y **switch**) como de las repetitivas o bucles (**for**, **while**, **do while**).

En el momento que se ejecuta la instrucción **break**, el control del programa sale de la estructura en la que se encuentra y continua con la parte siguiente del código.

Ejemplo:

```
//Variable contador para el bucle
int i = 0;

//Variable para controlar el número de veces
//que se ejecuta el bucle
int misveces = 9;

//Bucle que muestra una línea por cada vuelta que da el bucle.
//Cuando i valga 4, sale del bucle.
for (i = 0; i < misveces; i++) {
        if (i == 4) {
                break;
        }
        System.out.println("Vuelta del bucle: " + i);
}
```

Salir de la iteración actual: continue

La instrucción **continue** sirve para transferir el control del programa desde la instrucción **continue** directamente a la cabecera (principio) del bucle.

Ejemplo:

```
//Variable contador para el bucle
int i = 0;

//Variable para controlar el número de veces
//que se ejecuta el bucle
int misveces = 9;

//Bucle que muestra una línea por cada vuelta que da el bucle.
//Cuando i valga 4, se redirige al principio del bucle, por lo que
//en este ejemplo, la línea 4 no se imprime.
for (i = 0; i < misveces; i++) {
        if (i == 4) {
                continue;
        }
        System.out.println("Vuelta del bucle: " + i);
}
```

MATRICES

Descripción

Una **matriz** es un conjunto de datos del mismo tipo, estructurados en una única variable. En Java, las matrices son en realidad objetos y por lo tanto se puede llamar a sus métodos.

Las matrices también se llaman **arrays**, **vectores** o **tablas**.

Las matrices son objetos en **Java**.

Los **Strings** (o cadenas de caracteres) en **Java** son objetos pertenecientes a las clase **String**. (NO son matrices de tipo **char** como en otros lenguajes).

Declaración de matrices

Existen dos formas equivalentes de declarar matrices en Java:

- tipoVector[] nombreMatriz; (Estilo de lenguaje **Java**).
- tipoVector nombreMatriz[]; (Estilo de lenguaje **C**).

Ejemplo de declaración de una matriz de tipo **int** sin inicializar:

- int matriz1[];

Ejemplo de declaración de dos matrices de tipo **int** sin inicializar:

int matriz1[], matriz2[];

NOTA:

- La declaración de una matriz no reserva espacio en memoria, Lo que realmente reserva espacio en memoria es la asignación de elementos a dicha matriz.

Declaración de matrices reservando memoria

Para declarar una matriz sin asignar elementos, pero reservando la memoria, hay que crear un objeto mediante el operador **new**.

Ejemplo de matriz unidimensional:

```
//Declaración de matriz tipo int
int mimatriz[];

//Reserva de memoria para la matriz, de 5 elementos
mimatriz = new int[5];
```

Ejemplo de matriz bidimensional:

```
//Declaración de matriz tipo int bidimensional
int mimatriz[][];

//Reserva de memoria para la matriz, de 3 filas de elementos y 5 columnas
mimatriz = new int[3][5];
```

Ejemplo que declara una matriz y reserva espacio en memoria para 5 elementos:

```
int mimatriz[5] = new int[5];
```

Declaración e inicialización de matrices

Las matrices, al igual que las demás variables, pueden ser inicializadas en el momento de su declaración.

En este caso, no es necesario especificar el número de elementos máximo reservado, ya que Java "cuenta" los elementos especificados y así sabe dicho número de elementos. Se reserva así el espacio justo para almacenar los elementos especificados en dicha declaración.

Ejemplo que declara una matriz y le asigna 4 elementos:

```
String personas[] = {"Mario", "Eva", "Sonia", "Luis"};
```

NOTA:

- La declaración de una matriz no reserva espacio en memoria, Lo que realmente reserva espacio en memoria es la asignación de elementos a dicha matriz.
- Cuando los elementos se asignan a la matriz es cuando dichos elementos de la matriz obtienen la memoria necesaria para ser almacenados.

Dimensionado de una matriz

Consiste en darle dimensión al array, es decir el número de elementos que va a contener.

El dimensionado se puede hacer en la misma declaración, así como posteriormente.

Índice de la matriz

Es un número que identifica el orden de cada elemento de la matriz.

En Java, el primer elemento de la matriz tiene índice 0.

Número de elementos de la matriz

Es el número de elementos de la matriz.

El índice del último elemento de la matriz sera: **número de elementos - 1**.

Acceso de los elementos de la matriz

Para hacer referencia a los elementos de la matriz, se utiliza el identificador de la matriz (su nombre), junto con el índice del elemento de la matriz entre corchetes:

nombreMatriz[índiceMatriz]

Ejemplo que lee el valor de un elemento de la matriz:

```
//Declaración y asignación de elementos de la matriz
String personas[] = {"Mario", "Eva", "Sonia", "Luis"};

//Se muestra el valor del segundo elemento de la matriz (de índice "1")
System.out.println("Segundo elemento de la matriz: " + personas[1]);
```

El intento de acceder a un elemento fuera del rango de la matriz, provoca una excepción (error) que, deberá ser controlado.

Ejemplo que escribe un nuevo dato (reemplazando el anterior) en un elemento de la matriz:

```
//Declaración y asignación de elementos de la matriz
String personas[] = {"Mario", "Eva", "Sonia", "Luis"};

//Se muestra el tercer elemento
```

```
System.out.println("Tercer elemento de la matriz: " + personas[2]);

//Reemplazamos el dato del tercer elemento de la matriz
personas[2] = "Mariano";

//Se muestra el tercer elemento
System.out.println("Tercer elemento de la matriz: " + personas[2]);
```

Longitud de una matriz

Para obtener el número de elementos de una matriz en tiempo de ejecución se accede al atributo **length** de la clase de la matriz.

Dicho atributo **length** almacena un número entero correspondiente al número de elementos de la matriz.

Hay que recordar que las matrices en Java, son objetos.

Ejemplo:

```
//Declaración y asignación de elementos de la matriz
String personas[] = {"Mario", "Eva", "Sonia", "Luis"};

//Se muestra el número de elementos de la matriz
System.out.println("Número de elementos de la matriz: " + personas.length);
```

Recorrido de una matriz

Para recorrer una matriz se puede utilizar la estructura de control **for**, así como la estructura **for each**.

Ejemplo que utiliza la estructura for para recorrer los elementos de una matriz:

```
//Declaración y asignación de elementos de la matriz
String personas[] = {"Mario", "Eva", "Sonia", "Luis"};

//Bucle que muestra los elementos de la matriz
for (int i = 0; i < personas.length; i++) {
        System.out.println(personas[i]);
}
```

Ejemplo que utiliza la estructura **for each** para recorrer los elementos de la matriz:

```
//Declaración y asignación de la matriz
String[] ciudades = {"Madrid", "Londres", "Estocolmo"};

//Recorrido de los elementos de la matriz
for (String mielemento : ciudades) {
        System.out.println(mielemento);
}
```

Matrices multidimensionales

También pueden utilizarse matrices de más de una dimensión:

Ejemplo de declaración con reserva de memoria:

```
//Declaración de matriz tipo int bidimensional (dos dimensiones)
int mimatriz[][];

//Reserva de memoria para la matriz, de 3 filas de elementos y 5 columnas
```

```
mimatriz = new int[3][5];
```

Ejemplo de declaración y asignación de elementos a la matriz bidimensional:

```
//Declaración de una matriz bidimensional de enteros, de 3 filas y 4 columnas.
//Cada sección de llaves, corresponde a una fila de la matriz multidimensional.
int[][] mimatriz = {{10, 12, 14, 16}, {18, 20, 22, 24}, {26, 28, 30, 32}};
```

Acceso a elementos de una matriz multidimensional

Para acceder a un elemento en una matriz multidimensional, (a diferencia de otros lenguajes), en Java, los índices van entre corchetes por separado.

Ejemplo que accede a la matriz llamada mimatriz, al elemento de la tercera fila (índice 2), cuarta columna (índice 3):

mimatriz[2][3]

NO se accede como en algún otro lenguaje de la forma: ~~mimatriz[2,3]~~

Ejemplo de matriz bidimensional:

```
//Declaración de una matriz bidimensional de enteros, de 3 filas y 4 columnas.
//Cada sección de llaves, corresponde a una fila.
int[][] mimatriz = {{10, 12, 14, 16}, {18, 20, 22, 24}, {26, 28, 30, 32}};

//Se muestra un dato de la matriz bidimensional.
//De la segunda fila (índice 1), su tercer elemento (índice 2).
System.out.println("De la segunda fila, su tercer elemento: " + mimatriz[1][2]);
```

Recorrido de una matriz multidimensional

Ejemplo:

```
//Declaración de una matriz bidimensional de enteros, de 3 filas y 4 columnas.
//Cada sección de llaves, corresponde a una fila.
int[][] mimatriz = {{10, 12, 14, 16}, {18, 20, 22, 24}, {26, 28, 30, 32}};

//Bucle que recorre la matriz bidimensional.
//El bucle externo con variable "i" recorre las filas.
//El bucle interno con variable "j" recorre las columnas de cada fila.
for(int i = 0; i < mimatriz.length; i++) {
        for(int j = 0; j < mimatriz[i].length; j++) {
                System.out.println(mimatriz[i][j]);
        }
}
```

Recorrido de matrices con "for each"

La estructura **for each** es muy adecuada para el recorrido de los elementos de una matriz.

Sintaxis:

for (tipoMatriz variableElemento : matriz)

- tipoMatriz: tipo de dato de la matriz.
- variableElemento: es la variable que representa a cada elemento de la matriz, y mediante la cual se podrá acceder a cada uno de dichos elementos de la matriz.
- matriz: nombre de la matriz.

Ejemplo:

```
//Declaración y asignación de la matriz
String[] ciudades = {"Madrid", "Londres", "Estocolmo"};

//Recorrido de los elementos de la matriz
for (String mielemento : ciudades) {
        System.out.println(mielemento);
}
```

EJERCICIO: Trabajo con matriz de una dimensión.

1. Crear un programa que recoja del usuario 6 números enteros que su valor sea del 1 al 100, ambos inclusive. Para ello, se trabará con una matriz de una dimensión que almacenará dichos 6 números.
    - Esto es importante, pensando que en el futuro este programa podría recoger muchos más números, y al estar en la matriz, todo se automatiza mejor y el programa se hace eficiente.
2. Una vez recogidos, se desplegará lo siguiente, y en este orden (esto es un ejemplo de cómo debe quedar, respetando espacios, puntos, guiones, tabulaciones, saltos de línea, etc.):

---DATOS DE SALIDA

Los números recogidos son los siguientes:
Número 1: 45.
Número 2: 34.
Número 3: 12.
Número 4: 93.
Número 5: 59.
Número 6: 70.

Se han recogido un total de 6 números.

Primer número recogido: 45.
Último número recogido: 70.

Serie completa: / 45 - 34 -12 - 93 - 59 - 77 /

La media aritmética de la serie numérica es: 313.
El valor máximo es: 93.
El valor mínimo es: 12.

La suma total de la serie es: 320.

FIN DE INFORME

3. Enviar al profesor el archivo de clase **.java**.

NOTAS:
- El programa se realizará utilizando una matriz de una dimensión que almacena los números.
- La escritura será elegante con tabulaciones, comentarios, etc.

POO: INTRODUCCIÓN

Descripción

En **Java**, todo son clases (trabajo con objetos).

El lenguaje obliga a la programación orientada a objetos y no permite la posibilidad de programar mediante ninguna otra técnica que no sea ésta.

Por esta razón, un programa **Java** estará formado por uno o varios ficheros fuente (java) y en cada uno de ellos habrá definida una clase.

NOTA:

- Aunque dentro de un fichero **.java** es posible tener varias clases, es recomendable que haya una sola clase.

Clases

Desde un punto de vista general, una clase es un conjunto de valores (atributos) junto con las funciones y procedimientos que operan sobre los mismos (métodos), todo ello tratado como una entidad.

Estas clases constituyen los bloques principales en los cuales se encuentra contenido el código.

Archivo fuente java

En un archivo fuente puede declararse una o más clases. Lo mas habitual y recomendable es tener una sola clase por cada archivo **java**.

El nombre del archivo fuente **java** también hace distinción entre mayúsculas y minúsculas a la hora de compilarlo, aunque el sistema operativo empleado no haga esta distinción.

La extensión de dicho archivo fuente debe ser **java**.

Ejemplo de estructura de archivo **java** con una sola clase:

```
public class Clase1 {
...
}
```

Archivo fuente java con varias clases

Ejemplo de estructura de archivo **java** con mas de una clase:

```
class Clase1 {
...
}

class Clase2 {
...
}

...
```

Si hay varias clases dentro de un archivo fuente **java**, solo una puede ser **public**.

POO: CLASE: ESTRUCTURA

Descripción

La clase tiene diferentes partes y elementos que conforman su estructura.

Declaración de la clase

La declaración de la clase es lo que abre el código de la clase (la cabecera de dicho código).

La declaración de la clase contendrá, como mínimo, la palabra reservada **class** y el nombre de la clase.

Si no se especifica el modificador de acceso antes del nombre de la clase, dicha clase tendrá el modificador de acceso por defecto, que es **default** (en caso de desear el módificador de clase **default**, no hace falta escribirlo).

Sintaxis:

modificador-Acceso **class** **nombre-Clase**

Ejemplo de declaración de clase llamada **Miclase**, con modificador de acceso **default** (no hace falta escribir **default**):

```
class Miclase
```

Ejemplo de declaración de clase llamada Miclase, con modificador de acceso public (que es lo habitual):

```
public class Miclase
```

Sintaxis general para la declaración de una clase

La sintaxis general para declarar una clase es la siguiente (solo la palabra **class** y el nombre de la la clase [**NombreClase**] son obligatorios):

modificador class NombreClase extends NombreSuperclase implements listaInterfaces

Explicación de cada parte:

- modificador: indica el modificador para la clase (el mas común es **public**).
- class: indica que es una clase.
- NombreClase: indica el nombre de la clase. Por convención, la primera letra en mayúscula.
- extends: palabra clave que indica que esta clase hereda de la especificada a continuación de dicha palabra **extends**.
- NombreSuperclase: nombre de la superclase de la que hereda esta clase.
- Implements: palabra clave que indica que esta clase implementa la/s interface/s especificada/s a continuación de dicha palabra **implements**.

Ejemplo:

```
public class MiClase extends MiClasePadre implements MiInterface1, MiInterface2
```

Clase llamada **MiClase** con modificador de acceso público que hereda de la superclase **MiClasePadre** y que implementa dos interfaces: **MiInterface1** y **MiInterface2**.

Cuerpo de la clase

El cuerpo de las clases comienza con una llave abierta { y termina con una llave cerrada }.

Dentro del cuerpo de la clase se encuentra el contenido de la clase: la declaración de los atributos y los métodos de dicha clase.

Ejemplo:

```
public class Miclase {
        // CUERPO DE LA CLASE
        (Declaración de atributos)
        (Declaración de métodos)
}
```

NOTA:

- Para que un programa **Java** se pueda ejecutar, debe contener, al menos, una clase que tenga un método **main** con la siguiente declaración:

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

Estructura completa de una clase

```
/*
Declaramos que esta clase (Miclase) pertenece al paquete (package) "paquete".
La clausula package debe ser la primera sentencia del archivo java
*/
package paquete;

/*
Importación de paquetes.
Todos los paquetes con sus clases, así como solo clases sueltas
que se vayan a utilizar en esta clase.
*/

// Importación de todas las clases del paquete "java.math"
import java.math.*;

// Importación de la clase "Scanner" del paquete "java.util"
import java.util.Scanner;

// Declaración de la clase
public class Clase
{
//CUERPO DE LA CLASE

	// Declaración de atributos de la clase.
	// ...

	/* Método constructor
	Es el primer método que se ejecuta cuando se ejecuta la clase.
	Se llama igual que el fichero de la clase y también igual que la clase.
	El método constructor puede recibir parámetros.
	El método constructor es opcional.
	public Clase()
	{
		// Aquí se suelen inicializar los atributos de la clase.
		// ...
	}

	/*
	Éste es el método "main".
	La clase que tiene este método main es la clase que arranca la aplicación.
	Éste método se ejecuta al lanzarse la aplicación, es decir, es el primer
```

```
        método que se ejecuta cuando la aplicación arranca.
        (Hay que tener en cuenta que una aplicación está compuesta normalmente
        por varias clases).
        El resto de clases de la aplicación, no deberían tener este método "main".
        */
        public static void main(String[] args)
        {
                /*
                 * Aquí se pone todo lo que se desea que se ejecute
                 * cuando arranca la aplicación.
                 * Normalmente, aquí se ponen las llamadas
                 * a otras clases de la aplicación.
                 */

                /*
                 * En este ejemplo, se muestra el texto "Hola amigos".
                 * Para ello, se hace referencia a la clase "System",
                 * contenida en el paquete "java.lang".
                 * Dicha clase "System" tiene un método "println"
                 * que permite mostrar un texto por la consola.
                 */
                System.out.println("Hola amigos");
        }

        // Declaración de métodos de la clase.
        // ...

} //FIN DEL CUERPO DE LA CLASE
```

Miembros de una clase

Los miembros de una clase son sus atributos y sus métodos.

Esquema general de la estructura de una clase

El esquema general de la estructura de una clase es el siguiente:

- DECLARACIÓN DEL PAQUETE AL QUE PERTENECE LA CLASE.
- IMPORTACIÓN DE CLASES Y/O PAQUETES.
- CABECERA DE LA CLASE.
- ATRIBUTOS DE LA CLASE.
- MÉTODO CONSTRUCTOR DE LA CLASE.
- MÉTODOS DE LA CLASE.

Importación de paquetes o clases

Todas las clases disponibles en **Java** se encuentran almacenadas en paquetes, para su correcta organización.

En la parte superior del archivo **java** correspondiente a la clase, se realizarán todas las importaciones correspondientes a las clases o paquetes de clases que se deseen usar en esta clase.

Ejemplo para importar un paquete de clases entero (todas las clases de dicho paquete). Se importan todas las clases del paquete **math** (ruta de paquetes **java.math**):

```
import java.math.*;
```

Ejemplo para importar una sola clase de un paquete. Se importa la clase **Scanner** del paquete **util** (ruta de paquetes **java.util.Scanner** [**Scanner** es la clase]):

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

NOTA:

- Es recomendable importar solo lo necesario, es decir, solo lo que vayamos a utilizar, para ahorrar recursos.

Método "main"

Consideraciones generales sobre el método **main**:

- El método **main** está escrito en la clase que arranca la aplicación, es decir, la clase que contenga el método **main** es la clase que arrancará la aplicación.
- En el método **main** se pone todo lo que se desea que suceda cuando arranque la aplicación.
- En el método **main** se suelen poner las llamadas a otras clases de la aplicación.
- Todo programa independiente escrito en **Java** empieza a ejecutarse (como en lenguaje C) a partir del método main().
- Se pueden compilar clases que no posean método **main()**, pero el intérprete **Java** no podrá ejecutarlas.
- Para poder ejecutar el método **main()** no se crea ningún objeto de la clase que contenga dicho método **main**.

Ejemplo de método **main** que muestra un mensaje:

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

Explicación del anterior método main:

- public: indica que el método **main()** es público y, por tanto, puede ser llamado desde otras clases.
  - Todo método **main()** debe ser público para poder ejecutarse desde el intérprete **Java** (**JVM**), ya que es el primer método que se ejecuta de la aplicación.
- static: indica que la clase no necesita ser instanciada para poder utilizar este método **main**.
  - No es necesario crear un objeto de esta clase donde se encuentra el método **main** para poder ejecutar dicho método **main**.
  - También indica que, en caso de que se instanciara la clase donde se encuentra dicho método **main** (creando así uno o varios objetos de dicha clase donde está dicho método **main**), dicho método **main** sería compartido (global) por todos los objetos pertecientes a dicha clase donde está dicho método **main**; es decir, dicho método **main** es el mismo para todas las instancias que pudieran crearse de la clase donde está dicho método **main**.
- void: indica que este método **main** es un método que no devuelve ningún valor.
- main: indica el nombre del método. Es un nombre reservado que **Java** sabe que tiene que buscar para arrancar la aplicación.
- (String[] args): indica que este método **main** está capacitado para recibir un conjunto de argumentos, y que los almacenará en una matriz de tipo **String**, llamada **args**.
  - El método **main** debe aceptar siempre, como parámetro, una matriz de strings (cadenas), que contendrá los posibles argumentos que se le pasen al programa.
- System.out.println("Hola amigos"): indica que se va a ejecutar el método println(), encargado de mostrar un valor a través de la salida estándar (en el ejemplo, un String) y después realiza un retorno de carro automático y nueva línea.
  - Este método **println** pertenece al atributo **out**. Este atributo se encuentra incluido dentro de la clase **System**.

POO: CLASE: MODIFICADORES DE CLASE

Descripción

Los modificadores de clase son palabras reservadas que se anteponen a la declaración de clase. Los modificadores de clase son los siguientes:

- public.
- abstract.
- final.

Modificador de clase public

El ámbito para una clase con modificador **public** es el paquete donde se encuentra dicha clase y cualquier otro paquete existente.

Cuando se crean varias clases, agrupadas dentro de un paquete (package), únicamente las clases declaradas **public** pueden ser accedidas desde otro paquete.

Toda clase **public** debe ser declarada en un archivo fuente **java**.

En un archivo fuente **java** puede haber más de una clase, pero sólo puede haber una con el modificador **public**.

Ejemplo de clase con modificador **public**:

```
public class Miclase {
	...
}
```

Modificador de clase abstract

Las clases abstractas (con el modificador de clase **abstract**) no pueden ser instanciadas directamente.

La clase abstracta es una clase que sirve como clase base COMÚN para que otras clases hijas hereden de ella sus miembros que se deseen "comunes" (la implementación de dichos miembros [atributos y métodos] se deja a las subclases).

Ejemplo de clase con modificadores **public** y **abstract**:

```
public abstract class Miclase {
```

Modificador de clase final

Una clase con el modificador de clase **final** impide que pueda ser superclase de otras clases. Es decir, ninguna clase puede heredar de una clase **final**.

Esto es importante, por ejemplo, cuando se crean clases que acceden a recursos del sistema o realizan operaciones críticas o de seguridad. Si estas clases no se declaran como **final**, cualquiera podría redefinirlas y aprovecharse para realizar operaciones sólo permitidas a dichas clases pero con nuevas intenciones, posiblemente oscuras.

Ejemplo de clase con modificadores **public** y **final**:

```
public final class Miclase {
```

Omisión de modificador de clase

Si no se especifica el modificador de clase en la declaración de una clase, dicha clase tomará el modificador de clase por defecto, que es **default** (también llamado **package**), aunque no es necesario reflejar dicha palabra **default** ni **package** (si se desea aplicar a la clase dicho modificador por defecto).

Una clase con modificador de clase **default** (o **package**) será visible en todas las clases declaradas en el mismo paquete, pero no será visible en las clases de otros paquetes diferentes (externos al paquete actual).

Ejemplo de declaración de clase que no especifica de forma explícita el modificador de clase, por lo que toma por defecto el modificador de clase **default** (o **package**):

```
class Miclase {
	...
}
```

POO: CLASE: INSTANCIACIÓN

Descripción

Para utilizar una clase, desde otra clase, primero hay que instanciarla.

Mediante la instanciación, se crea un objeto de dicha clase, que contiene una copia de todos los miembros de la clase.

Instanciación de la clase

Para crear un objeto a partir de su clase correspondiente, se utiliza la palabra reservada **new**.

A partir de ese momento, se hará referencia a dicho objeto para acceder a sus datos y para poder ejecutar sus métodos.

En el momento de la instanciación de la clase (y creación del objeto), también se podrán pasar argumentos a dicha clase, que servirán para inicializar atributos del objeto, dándole de esta manera un "estado" inicial.

Ejemplo de instanciación de la clase **Miclase**, creando un objeto llamado **miobjeto**, correspondienta a dicha clase **Miclase**:

```
Miclase miobjeto = new Miclase();
```

Como no se pasan parámetros de inicialización del objeto, los paréntesis están vacíos.

A partir de ese momento, se hará referencia a dicho objeto llamado **miobjeto** para acceder a sus datos y para poder ejecutar sus métodos.

POO: CLASE: ATRIBUTOS

Descripción

Un atributo es miembro de una clase que hace de variable.

Tipos generales de atributos

Existen dos tipos generales de atributos:

- Atributos de objeto: son variables que almacenan valores correspondientes al objeto de su clase.
  - Cada objeto correspondiente a su clase, almacena sus propios atributos.
  - Un atributo de objeto tendrá tantas copias de ese atributo como objetos se instancien, y cada una ellas será independiente y solo visible en su objeto correspondiente.
  - Es la opción por defecto, es decir, por defecto, todos los atributos son de objeto.
- Atributos de clase: son variables que almacenan el valor y dicho valor está disponible para todos los objetos instanciados a partir de esa clase.
  - Es decir, es como una variable global (compartida) a todos los objetos creados (instanciados) a partir de la misma clase.
  - Un atributo de clase tendrá una sola copia de dicho atributo, que será compartido por todos los objetos que se instancien de la misma clase.
  - Se consigue mediante el modificador **static**.

Modificadores de ámbito de atributos

Los modificadores de ámbito de atributos especifican la forma en que puede accederse a dichos atributos desde otras clases.

Estos modificadores de ámbito son:

- private.
- public.
- protected.
- Ámbito por defecto.

private

El modificador de ámbito **private** (privado) es el más restrictivo de todos.

Todo atributo **private** es visible únicamente dentro de la clase en la que se declara. No existe ninguna forma de acceder al mismo, si no es desde dentro de su clase, a través de un método (que no sea **private**) que permite leer o modificar el valor de dicho atributo **private**.

Una buena metodología de diseño de clases es declarar los atributos **private** siempre que sea posible, ya que esto evita que algún objeto pueda modificar su valor si no es a través de alguno de sus métodos diseñados para ello.

Por lo que lo adecuado sería crear una pareja de métodos (get y set) para cada atributo **private**, por ejemplo: **getValor** (para obtener el valor del atributo **private**) y **setValor** (para modificar el valor del atributo **private**).

Si se omite la creación del método **setValor**, lo que se tendría es un atributo de solo lectura.

Ejemplo de declaración de un atributo **private** dentro de una clase:

```
public class Miclase {

      private int mivalor;
```

NOTA: si no se asigna ningún valor a un atributo int (de clase), se le asigna por defecto **0**.

Ejemplo de declaración y asignación de un atributo private dentro de una clase:

```
public class Miclase {

      private int mivalor = 6;
```

public

El modificador de ámbito **public** (público) es el menos restrictivo de todos. Un atributo **public** será visible en cualquier clase de cualquier paquete.

Ejemplo de declaración de un atributo **public**:

```
public int mivalor = 44;
```

Si se desea acceder a un atributo **public**, simplemente se deberá anteponer a dicho atributo public el nombre de la clase:

```
miobjeto.mivalor
```

Ejemplo que muestra el dato de un atributo **public** de la clase **Miclase**:

```
// Se instancia la clase, creando el objeto
// correspondiente a dicha clase
Miclase miobjeto = new Miclase();

// Se muestra el atributo mivalor, de tipo public
System.out.println(miobjeto.mivalor);
```

Ejemplo que modifica el dato del atributo **public** de la clase **Miclase**:

```
// Se muestra el atributo mivalor, de tipo public
System.out.println(miobjeto.mivalor);

// Se modifica el atributo de tipo public
miobjeto.mivalor = 15;

// Se muestra el atributo mivalor, de tipo public,
// que ahora muestra su dato ya modificado
System.out.println(miobjeto.mivalor);
```

Las aplicaciones bien diseñadas minimizan el uso de los atributos **public** y maximizan el uso de atributos **private**.

La forma apropiada de acceder y modificar atributos de objetos es a través de métodos **public** que accedan a dichos atributos **private**.

El único motivo para crear atributos **public** es acelerar el proceso de programación, permitiendo así modificar los valores de dichos atributos públicos directamente desde otras clases, pero esto es un "apaño provisional".

protected

Los atributos **protected** (protegido) pueden ser accedidos por las clases del mismo paquete (**default** o **package**) y también por las subclases (clases hijas) del mismo paquete. Es decir, que los atributos **protected** están disbonibles para las clases del mismo paquete y para las clases hijas que hereden de la clase donde esté dicho atributo **protected**.

Un atributo **protected** se puede decir que sería como un **private** hacia el exterior (no se puede acceder directamente a un atributo **protected** desde otra clase) y como un **public** para las subclases (una subclase que herede de la clase donde está dicho atributo **protected**, tendrá acceso a dicho atributo **protected**).

Ejemplo de declaración de un atributo **protected**:

```
protected int midato = 3;
```

Ámbito por defecto de los atributos

Los atributos que no llevan ningún modificador de ámbito pueden ser accedidos desde las clases del mismo paquete, pero no desde otros paquetes.

Ejemplo de declaración de un atributo con modificador por defecto:

```
int midato;
```

Acceso a atributos

Un atributo **public** será visible en cualquier clase que desee acceder a él, simplemente, desde la clase deseada, anteponiendo a dicho atributo **public** el nombre de la clase.

Ejemplo:

```
miobjeto.mivalor
```

Ejemplo que muestra el dato de un atributo **public** de la clase **Miclase**:

```
// Se instancia la clase, creando el objeto
// correspondiente a dicha clase.
Miclase miobjeto = new Miclase();

// Se muestra el atributo public, llamado mivalor
System.out.println(miobjeto.mivalor);
```

Ejemplo que modifica el dato del atributo **public** de la clase **Miclase**:

```
// Se muestra el atributo public, llamado mivalor
System.out.println(miobjeto.mivalor);

// Se modifica el atributo
miobjeto.mivalor = 15;

// Se muestra de nuevo el atributo,
// que ahora muestra su dato ya modificado
System.out.println(miobjeto.mivalor);
```

Las aplicaciones bien diseñadas evitan el uso de atributos **public** y procuran usar atributos **private**.

La forma apropiada de acceder y modificar atributos de objetos es a través de métodos **public** que accedan a dichos atributos **private**.

Modificadores especiales de atributos

Los modificadores especiales de atributos permiten modificar la forma en la que se comportan dichos atributos.

Estos modificadores de ámbito son:

- static.
- final.
- transient.
- volatile.

static

Mediante la palabra reservada **static** se declaran atributos de clase.

Un atributo de clase es compartido por todos los objetos que se crean a partir de la clase. Todos los objetos de esta clase pueden acceder al mismo atributo y manipularlo.

Para acceder al atributo **static** desde cualquier objeto creado a partir de la clase donde se declaró dicho atributo **static**, se hará anteponiendo a dicho atributo **static** el nombre de la clase donde se declaró.

Ejemplo que declara e inicializa un atributo **static**:

```
public static int mivalor = 0;
```

Ejemplo que accede al atributo **static** llamado **mivalor** desde cualquier objeto correspondiente a la clase **Caja** (en dicha clase **Caja** es donde se declaró dicho atributo **static**):

```
Caja.mivalor
```

Ahora, se incrementa su valor, incrementándolo:

```
Caja.mivalor++
```

final

El modificador de atributo **final** crea una constante.

La palabra reservada **final** sirve para declarar constantes, por lo que no se permite la modificación de su valor posteriormente.

El valor de un atributo **final** debe ser asignado obligatoriamente en la declaración del mismo (declaración y asignación obligatoria en la misma línea).

Cualquier intento de modificar su valor posteriormente, generará un error por parte del compilador.

Ejemplo que declara una constante (atributo **final**) privada (atributo **private**) y le asigna un valor en la propia declaración (obligatorio):

```
private final int mivalor = 44;
```

transient

Los atributos de un objeto se consideran, por defecto, persistentes (que se controla mediante la serialización). Esto significa que a la hora de, por ejemplo, almacenar objetos en un fichero, los valores de dichos atributos deben también almacenarse.

Los atributos **transient** son los atributos que no forman parte del estado persistente del objeto porque almacenan estados transitorios o puntuales del objeto, y se denominan transitorios. De esta forma, un atributo de tipo **transient** no sería persistente, por lo que no se almacenaría con los datos del objeto.

Ejemplo que declara e inicializa un atributo **transient**:

```
transient int numVecesAlterado = 0;
```

En este caso, la variable **numVecesAlterado** almacena el número de veces que se altera el objeto.

A la hora de almacenar el objeto en un fichero para su posterior recuperación (serialización) puede que no interese el número de veces que ha sido modificado. Declarando dicho atributo como **transient** este valor no se almacenará.

volatile

Si una clase contiene atributos de objeto que son modificados asíncronamente por **threads** que se ejecutan concurrentemente, se pueden utilizar atributos **volatile** que permitan a **Java** cargar dicho atributo **volatile** desde memoria antes de utilizarlo y volver a almacenarlo en memoria después, con el objetivo de que cada thread pueda "verlo" en un estado coherente.

Esto nos ayudará a mantener la coherencia de las variables que puedan ser utilizadas en threads (concurrentemente).

Ejemplo que declara e inicializa un atributo **volatile**:

```
volatile int contador = 0;
```

POO: CLASE: MÉTODOS

Descripción

Un método es una parte de código que resuelve un problema en concreto, ejecutando el conjunto de instrucciones que tenga dicho métod.

Tipos de métodos

Existen dos tipos de métodos:

- Métodos que devuelven un valor.
  - Hay que especificar en la declaración del método el tipo de dato que devolverá dicho método.
- Métodos que no devuelven valor (**void**).
  - Hay que especificar en la declaración del método que sea de tipo **void**.

Sintaxis general

Declaración de método {
    Cuerpo del método
}

Declaración de métodos

Declaración mínima

La declaración mínima (sin modificadores) de un método es:

tipoDevuelto/void nombreMétodo (listaParámetros)

Donde:

- tipoDevuelto/void: es el tipo de dato devuelto por el método (ejemplo: **int**). Si el método no devuelve ningún valor, en su lugar se indica la palabra reservada **void**.
- nombreMétodo: es un identificador válido en **Java**.
- listaParámetros: si tiene parámetros, es una sucesión de pares "tipo nombrevariable", separados por comas. Ejemplo:
  - int mayor(**int x , int y**).

Declaración de método que devuelve un valor

Ejemplo de declaración de un método que devuelve un valor **int**:

```
int calcular() {
```

Declaración de método que no devuelve valor (void)

Ejemplo de declaración de un método que no devuelve valor (**void**):

```
void mostrar() {
```

Declaración completa de un método

Sintaxis general:

modificadorÁmbito modificadorEspecial tipoDevuelto/void nombreMétodo (listaParámetros)

Donde:

- modificadorÁmbito: es el modificador de ámbito (private, public, etc.).
- modificadorEspecial: es cualquiera de los modificadores: static, abstract, final, etc.
- tipoDevuelto/void: es el tipo de dato devuelto por el método (ejemplo: "int"). Si el método no devuelve ningún valor, en su lugar se indica la palabra reservada "void".
- nombreMétodo: es un identificador válido en Java.
- listaParámetros: si tiene parámetros, es una sucesión de pares "tipo nombrevariable", separados por comas. Ejemplo: int mayor(**int x , int y**).

Ejemplo de declaración completa de método:

```
public static int mimetodo(int dato1, int dato2) {
```

Modificadores de ámbito de métodos

Son los ya vistos para los atributos:

- private.
- public.
- protected.
- Ámbito por defecto de los métodos.

Modificadores especiales de métodos

Algunos, son los ya vistos para los atributos:

- static.
- abstract.
- final.
- native.
- synchronized.

Métodos **abstract**:

Los métodos abstract se declaran en las clases **abstract**. Es decir, si se declara algún método de tipo **abstract**, entonces, la clase debe declararse obligatoriamente como **abstract**.

Cuando se declara un método **abstract** (en una clase **abstract**), no se implementa el cuerpo del método, sólo su signatura.

Las clases que se declaran como subclases de una clase **abstract** deben implementar los métodos **abstract**. Una clase **abstract** no puede ser instanciada directamente, únicamente sirve para ser utilizada como superclase de otras clases (subclases).

Ejemplo de declaración de método **abstract** dentro de una clase **abstract**:

```
abstract void dibujar();
```

Métodos **final**:

Los métodos de una clase que se declaran de tipo **final** no pueden ser redefinidos por las subclases.

Esta opción puede adoptarse por razones de seguridad, para que ciertos miembros de las clases no puedan ser extendidas por otros.

Un método **final** no puede redefinirse en la subclases. Cualquier intento de redefinición de un método **final** en una subclase generaría un error del compilador.

Ejemplo de declaración de método **final** dentro de una clase:

```
public final int resumen() {
```

Argumentos (parámetros) en los métodos

Los argumentos (o parámetros) en un método de una clase es una sucesión de pares **tipo-valor**, separados por comas, que sirven para recoger datos y trabajar con dichos datos en el propio método.

Los parámetros pueden ser también objetos.

Tipos de argumentos

- Variables de tipo primitivo (simple): en este caso, el paso de parámetros se realiza SIEMPRE POR VALOR.
  - Lo que se le pasa al método es una copia del valor. Es decir, el valor del parámetro recogido en el método, al ser una copia, no puede modificar el método el valor original.
  - El método trabaja con una copia del valor pasado en la llamada.
- Variables de tipo referencia (objeto, matrices y Strings):
  - En este caso, lo que realmente se pasa al método es un objeto (o matriz o String) y, por lo tanto, el valor del parámetro de llamada sí que puede ser modificado dentro del método (El método trabaja directamente con el valor utilizado en la llamada, es decir, el objeto [o matriz o String]).
  - Los datos de tipo referencia pueden ser: objeto, matriz y Strings.

Escritura de los argumentos en el método

Ejemplo de método que declara dos argumentos (uno "String" y otro "int"):

```
void mostrar(String nombre, int edad) {
```

Formas de pasar los argumentos al método: valor, referencia

Los tipos primitivos (simples) de datos SIEMPRE SE PASAN POR VALOR y los objetos, matrices y Strings, por referencia.

Pasar argumentos al método

Ejemplo de método que realiza una llamada a un método de un objeto, pasándole dos argumentos:

```
// Se instancia la clase, creando el objeto "miobjeto"
Miclase miobjeto = new Miclase();

// Mediante el objeto, se ejecuta su método mostrar
// y se le pasan dos argumentos.
miobjeto.mostrar("Juan", 28);
```

Recoger los datos de un método

El dato que devuelve un método se deberá procesar adecuadamente.

Lo más habitual es recogerlo en una variable para utilizarlo posteriormente.

Cuerpo del método

Sintaxis general de los métodos:

```
Declaración de método {
        Cuerpo del método
}
```

El cuerpo del método contiene la implementación del mismo y se codifica entre las llaves del método.

Dentro de dicho método pueden declararse variables locales al mismo.

El único caso en el que no se implementa el cuerpo del método es en la declaración de clases abstractas.

Devolución de un dato en un método mediante return

En los métodos que devuelven un valor (los que no son **void**), se utiliza la palabra reservada **return** para devolver un dato de un método.

Dicha palabra reservada **return** también declara la salida del método. Es decir, que, por ejemplo, mediante **return midato** se sale del método, retornando dicho dato llamado por ejemplo **midato**.

Si en la declaración de la cabecera del método se ha declarado un tipo de dato devuelto por el mismo, entonces se devolverá el valor correspondiente al método mediante **return** seguido de una expresión cuyo resultado es del mismo tipo que el declarado en la cabecera de dicho método.

Ejemplo de método que devuelve una cadena (String):

```
String mostrar(String nombre, int edad) {
	// Se devuelve la cadena (String)
	return "Hola " + nombre + ", tienes " + edad + " años.";
}
```

Si el método es tipo **void**, para terminar dicho método, se especificará la palabra reservada **return** en una sola línea, normalmente al final.

Si el tipo de dato devuelto del método ha sido declarado **void**, entonces únicamente se sale del cuerpo del método mediante la instrucción **return**.

Ejemplo de método **void**, con **return** para finalizar dicho método:

```
void mostrar(String nombre, int edad) {
	// Se muestra la cadena
	System.out.println("Hola " + nombre + ", tienes " + edad + " años.");
	// Finaliza el método
	return;
}
```

Llamada a métodos

Para llamar a un método de un objeto, una vez creado el objeto correspondiente, se aplicará la siguiente sintaxis:

nombreObjeto.nombreMétodo(listaParámetros);

Ejemplo de llamada a método de tipo **String**, pasándole dos argumentos:

```
// Se instancia la clase, creando el objeto "miobjeto"
Miclase miobjeto = new Miclase();

// Se ejecuta el método "mostrar" (de tipo "String")
// perteneciente al objeto "miobjeto"
// y se le pasan dos argumentos
System.out.println(miobjeto.mostrar("Juan", 28));
```

NOTA:

- Se puede observar que en este caso, como se llama a un método tipo **String** que devuelve un valor (también de tipo **String**), hace falta ejecutar directamente dicho valor que se recoge, o también se podría almacenar en una variable dicho valor que se recoge.

- En este caso, en lugar de almacenar en una variable el dato que devuelve el método, se muestra directamente con **System.out.println()**.

Ejemplo de llamada a método de tipo **void**, pasándole dos argumentos:

```
// Se instancia la clase, creando el objeto "miobjeto"
Miclase miobjeto = new Miclase();

// Se ejecuta el método "mostrar" (de tipo "String")
// perteneciente al objeto "miobjeto"
// y se le pasan dos argumentos
miobjeto.mostrar("Juan", 28);
```

NOTA:

- Se puede observar que en este caso, como se llama a un método tipo **void** que no devuelve valor, no hace falta recoger dicho valor en una variable, sino simplemente ejecutar dicho método.

Ejemplo completo de implementación y llamada a método que devuelve un tipo de dato (en este caso **String**):

IMPLEMENTACIÓN:

```
String mostrar(String nombre, int edad) {
	// Se devuelve la cadena (String)
	return "Hola " + nombre + ", tienes " + edad + " años.";
}
```

LLAMADA:

```
// Se instancia la clase, creando el objeto "miobjeto"
Miclase miobjeto = new Miclase();

// Se ejecuta el método "mostrar" (de tipo "String")
// perteneciente al objeto "miobjeto"
// y se le pasan dos argumentos
System.out.println(miobjeto.mostrar("Juan", 28));
```

Ejemplo completo de implementación y llamada a método de tipo **void**:

IMPLEMENTACIÓN:

```
void mostrar(String nombre, int edad) {
	// Se muestra la cadena
	System.out.println("Hola " + nombre + ", tienes " + edad + " años.");
	// Finaliza el método
	return;
}
```

LLAMADA:

```
// Se instancia la clase, creando el objeto "miobjeto"
Miclase miobjeto = new Miclase();

// Se ejecuta el método "mostrar" (de tipo "String")
// perteneciente al objeto "miobjeto"
// y se le pasan dos argumentos
miobjeto.mostrar("Juan", 28);
```

Método constructor

El constructor es un método especial de las clases que tiene como característica principal ser el primer método que se ejecuta de forma automática en el momento de crearse el objeto.

Es decir, después de crearse el objeto, automáticamente se ejecuta su método constructor.

Las características del método constructor son las siguientes:

- Es el primer método que se ejecuta de forma automática cuando se instancia la clase (creando así el objeto).
- El nombre del método constructor debe ser obligatoriamente el mismo que el nombre de la clase (así como del archivo fuente **java**).
- El método constructor no tiene tipo, ni **void**.
- Su cometido principal es inicializar los atributos de los objetos, en el momento de crearse dichos objetos. De esta forma, el objeto ya se crearía con un "estado inicial".
- El método constructor siempre es obligatorio, de hecho, todas las clases tienen uno por defecto, ya que si el programador no lo declara de forma explícita, lo hará **Java** de forma implicita.
- El método constructor puede estar declarado, pero estar vacío.
- Pueden existir varios constructores, que se llamarán todos igual, pero que se distinguirán unos de otros por el número y/o tipo de datos de sus parámetros.
    - En este caso se denominará **sobrecarga de constructores**.
    - En caso de **sobrecarga de constructores**, **Java** ejecutará el que se corresponda con el número o tipo de parámetros enviados.
- El constructor o constructores pueden recibir parámetros. Estos parámetros serán enviados en el momento de la creación del objeto.

Ejemplo de declaración de método constructor dentro de su clase, que recibe dos parámetros:

```
// Declaración de atributos
private int valor1;
private int valor2;

// Método constructor que recibe dos parámetros
// de la instanciación del objeto.
// Dichos parámetros se asignan a los atributos
// declarados al principio de la clase, con lo que
// los valores de dichos atributos quedan disponibles
// para toda la clase.
Miclase(int v1, int v2) {
	valor1 = v1;
	valor2 = v2;
}
```

Ejemplo de creación del objeto (llamada al método constructor). Se le envían dos parámetros:

```
// Declaración y asignación de variables que se
// pasarán al objeto (a su método constructor).
int dato1 = 5;
int dato2 = 7;

// Se instancia la clase, creando el objeto "miobjeto" y
// enviando 2 parámetros al método constructor.
Miclase miobjeto = new Miclase(dato1, dato2);
```

Uso de this como referencia a la clase actual

La palabra reservada **this** hace referencia a la clase actual.

Si por ejemplo, utilizando un IDE como Eclise, se escribe **this** y despues se escribe un punto, aparecerá una lista con todos los miembros de la clase actual.

Si por ejemplo, se hiciese referencia a los atributos o métodos (miembros) de la clase actual, se podría anteponer al nombre de dichos miembros, la palabra reservada **this** seguida de un punto y después el nombre del miembro (atributo o método), de la siguiente forma:

```
private int valor1;
private int valor2;

Miclase(int v1, int v2) {
	this.valor1 = v1;
	this.valor2 = v2;
}
```

Aunque se aconseja en uso de **this**, si se omite dicha palabra reservada **this**, se entiende por defecto como si la llevase, por lo que se puede omitir:

```
private int valor1;
private int valor2;

Miclase(int v1, int v2) {
	valor1 = v1;
	valor2 = v2;
}
```

Es aconsejable utilizar la palabra clave **this** para hacer referencia a los miembros de la clase actual y así hacer más legible el código:

```
// Declaración de atributos privados ("private").
private int valor1;
private int valor2;

/*
Método constructor que recibe dos parámetros de la
instanciación del objeto.
Dichos parámetros se asignan a los atributos privados
declarados al principio de la clase, con lo que
los valores de dichos atributos quedan disponibles
para toda la clase.
Con la palabra clave "this" hacemos referencia a que dichos
atributos ("valor1" y "valor2") son de esta clase y además
se diferencian a simple vista de los parámetros que se
han recogido del constructor ("v1" y "v2" [que no son datos
pertenecientes a esta clase, sino que se han enviado desde el
exterior, aunque se utilicen en el método constructor]).
*/

Miclase(int v1, int v2) {
	this.valor1 = v1;
	this.valor2 = v2;
}
```

Referencia a atributos de la clase con el mismo nombre: ocultación

Si existen en un método variables locales o parámetros de dicho método con el mismo nombre que atributos de la clase, y hacemos referencia a dicha variable local o parámetro desde dentro de dicho método, tendrá preferencia dicha variable o parámetro de dicho método, por lo que el atributo de la clase quedará "ocultado".

Si desde dentro de dicho método, se desea hacer referencia al atributo de la clase, en lugar de a la variable o parámetro de dicho método, se antepone la palabra reservada **this** seguida de un punto y a continuación el nombre del atributo de clase.

Ejemplo el que se produce ocultación de un atributo de la clase:

```
// Atributos de la clase
private int valor1 = 5; //QUEDA OCULTADO (OCULTACIÓN)
private int valor2 = 6;

// Método de la clase
void mostrar() {
	// Atributo local a este método, que tiene el mismo nombre
	// que un atributo de la clase.
	int valor1 = 10;
	// Se accede al atributo local de este método,
	// quedando "ocultado" (ocultación) el atributo
	// de la clase que tiene el mismo nombre.
	System.out.println(valor1);
}
```

NOTA:

- En el ejemplo anterior, el resultado de la ejecución del programa es: 10.

Ejemplo el que no se produce "ocultación" de un atributo de la clase, ya que se hace referencia directamente a dicho atributo de la clase, mediante la palabra reservada **this**:

```
// Atributos de la clase
private int valor1 = 5;
private int valor2 = 6;

// Método de la clase
void calcular() {
	// Atributo local a este método que tiene el mismo nombre
	// que el atributo de la clase
	int valor1 = 10;

	// Se accede al atributo local de este método,
	// quedando "ocultado" (ocultación) el atributo
	// de clase que tiene el mismo nombre.
	System.out.println(this.valor1);
}
```

NOTA:

- En el ejemplo anterior, el resultado de la ejecución del programa es: 5.

Pasar una matriz como argumento de llamada a un método

Para pasar una matriz en la llamada a un método, como un argumento, se hará como si fuese un dato normal:

```
// Declaración e inicialización de la matriz
String[] mimatriz = {"Luis", "Eva", "Ana", "Javier"};

// Se instancia la clase, creando el objeto
Miclase miobjeto = new Miclase();

// Se ejecuta el método del objeto y se le pasa a dicho
// método la matriz, como argumento.
// Se envía como si fuese un dato normal.
miobjeto.mimetodo(mimatriz);
```

NOTA:

- A partir de este momento dentro del método mimetodo se podrá utilizar la matriz pasada como argumento, de la misma manera que si hubiésemos declarado la matriz dentro del método mimetodo.

Recoger una matriz como argumento de un método

Para recoger una matriz como argumento de un método, se hará como si fuese un dato normal, pero en la declaración, se pondrán los corchetes correspondientes, ya que se trata de una matriz:

```
// Se recoge la matriz en el método, que viene como argumento.
// Se recoge la matriz como si fuese un dato normal, solo que
// se añaden los corchetes (ya que es una matriz).
// A partir de que recojamos la matriz en este método del ejemplo,
// la matriz se llamará "mispersonas".
public void mimetodo(String[] mispersonas)
{
	// Se recorre la matriz y se muestran sus elementos
	for(int i=0; i < mispersonas.length; i++)
	{
		System.out.println(mispersonas[i]);
	}
}
```

Devolver una matriz desde un método, con return

Desde un método, se podrá devolver una matriz, con la palabra reservada **return**, como cualquier tipo de dato normal.

Clase que retorna una matriz:

```
// Declaración e inicialización de la matriz
String[] mimatriz = {"Luis", "Eva", "Ana", "Javier"};

// Se recoge la matriz en el método, que viene como argumento.
// Se recoge la matriz como si fuese un dato normal, solo que
// se añaden los corchetes en el tipo de la matriz (ya que es una matriz).
// A partir de recoger el método dicha matriz, en este ejemplo,
// la matriz se llamará "mispersonas".
public String[] mimetodo() {
	return mimatriz;
}
```

Clase que ejecuta el método de la clase que retorna una matriz:

```
// Declaración de la matriz que almacenará posteriormente
// la matriz que devolverá un método.
String[] lamatriz;

// Se instancia la clase, creando el objeto
Miclase miobjeto = new Miclase();

// Se ejecuta el método del objeto mimetodo.
// Dicho método mimetodo devuelve una matriz.
// Dicha matriz se almacena en la variable "mimatriz".
lamatriz = miobjeto.mimetodo();

// Se recorre la matriz y se muestran sus elementos
for(int i=0; i < lamatriz.length; i++)
{
	System.out.println(lamatriz[i]);
}
```

POO: EJEMPLO COMPLETO DE CLASE

Descripción

Crearemos una clase completa.

Ejemplo completo de clase

Clase Caja:

```
// DECLARAMOS QUE ESTA CLASE PERTENECE AL PAQUETE (PACKAGE) miproyecto
package miproyecto;

// CLASE QUE IMPLEMENTA LA LÓGICA DE NEGOCIO DE UNA CAJA
public class Caja {

	// ATRIBUTOS DE LA CLASE
	// =====================

	// Dimensiones de la caja
	private double largo;
	private double ancho;
	private double alto;

	// Número de elementos que contiene la caja
	// La caja la inicializamos vacía (sin elementos)
	private int elementos = 0;

	//MÉTODO CONSTRUCTOR
	//RECOGE 3 PARÁMETROS PARA CONSTRUIR LA CAJA
	Caja(double miancho, double milargo, double mialto) {
		//INICIALIZAMOS LOS ATRIBUTOS DE LA CLASE
		//CON LOS VALORES RECOGIDOS EN EL CONSTRUCTOR
		this.largo = milargo;
		this.ancho = miancho;
		this.alto = mialto;
		}

	// MÉTODOS DE LA CLASE
	// ===================

	// MÉTODO QUE DEVUELVE EL LARGO DE LA CAJA
	public double getLargo() {
		return this.largo;
	}

	// MÉTODO QUE DEVUELVE EL ANCHO DE LA CAJA
	public double getAncho() {
		return this.ancho;
	}

	// MÉTODO QUE DEVUELVE EL ALTO DE LA CAJA
	public double getAlto() {
		return this.alto;
	}

	// MÉTODO QUE DEVUELVE LA SUPERFICIE DE LA CAJA
	public double getSuperficie() {
		return this.ancho * this.largo;
	}
```

```
	// MÉTODO QUE DEVUELVE EL VOLUMEN DE LA CAJA
	public double getVolumen() {
		return this.ancho * this.largo * this.alto;
	}

	// MÉTODO QUE INSERTA ELEMENTOS EN LA CAJA
	// DEVOLVEMOS UN STRING INFORMATIVO
	public String insertar(int miselementos) {
		this.elementos += miselementos;
		return "SE HAN INSERTADO " + miselementos + " ELEMENTOS";
	}

	// MÉTODO QUE EXTRAE ELEMENTOS DE LA CAJA
	// DEVOLVEMOS UN STRING INFORMATIVO
	public String extraer(int miselementos) {
		// COMPROBAMOS SI HAY ELEMENTOS SUFICIENTES ANTES DE EXTRAER
		if(elementos >= miselementos) {
			this.elementos -= miselementos;
			return "SE EXTRAJERON " + miselementos + " ELEMENTOS... HABÍA
ELEMENTOS DISPONIBLES";
		} else {
			return "ELEMENTOS INSUFICIENTES... NO SE EXTRAJO NINGÚN ELEMENTO";
		}
	}

	// MÉTODO QUE MUESTRA EL NÚMERO DE ELEMENTOS QUE HAY EN LA CAJA
	public int getElementos() {
		return this.elementos;
	}

	// MÉTODO QUE VACÍA LA CAJA DE ELEMENTOS
	public void vaciar() {
		this.elementos = 0;
	}

}
```

Clase Accesocaja:

```
// Declaramos que esta clase pertenece al paquete (package) "miproyecto"
package miproyecto;

// IMPORTAMOS LA CLASE Scanner DEL PAQUETE java.util
import java.util.Scanner;

// CLASE QUE UTILIZA LA CLASE Caja
public class Accesocaja {
	public static void main(String args[]) {

		// DECLARAMOS VARIABLES PARA RECOGER LOS DATOS QUE CONSTRUIRÁ LA CAJA
		double milargo, miancho, mialto = 0;

		// CREACIÓN DE UN OBJETO DE LA CLASE Scanner
		// QUE PERMITIRÁ RECOGER DEL USUARIO LOS DATOS PARA CONSTRUIR LA CAJA
		Scanner miescaner = new Scanner(System.in);

		// RECOGEMOS DEL USUARIO LOS DATOS PARA CREAR LA CAJA
		System.out.println("Introduca el largo: ");
		milargo = miescaner.nextDouble();
```

```
            System.out.println("Introduca el ancho: ");
            miancho = miescaner.nextDouble();
            System.out.println("Introduca el alto: ");
            mialto = miescaner.nextDouble();

            // INFORMAMOS DE LOS DATOS RECOGIDOS
            System.out.println("Datos recogidos: ");
            System.out.println("Largo: " + milargo);
            System.out.println("Ancho: " + miancho);
            System.out.println("Alto: " + mialto);

            // INSTANCIAMOS LA CLASE Caja
            // ENVIAMOS LOS 3 PARÁMETROS NECESARIOS PARA CONSTRUIR LA CAJA
            Caja micaja = new Caja(milargo, miancho, mialto);

            System.out.println("CAJA CONSTRUÍDA CORRECTAMENTE");

            // UNA VEZ CREADA LA CAJA...
            // OBTENEMOS INFORMACIÓN DE LA CAJA

            // OBTENEMOS EL LARGO
            System.out.println("Largo: " + micaja.getLargo());

            // OBTENEMOS EL ANCHO
            System.out.println("Ancho: " + micaja.getAncho());

            // OBTENEMOS EL ALTO
            System.out.println("Alto: " + micaja.getAlto());

            // OBTENEMOS LA SUPERFICIE
            System.out.println("Superficie: " + micaja.getSuperficie());

            // OBTENEMOS EL VOLUMEN
            System.out.println("Volumen: " + micaja.getVolumen());

            // INSERTAMOS ELEMENTOS
            // Y RECOGEMOS Y MOSTRAMOS EL MENSAJE CORRESPONDIENTE
            String mensajeInsertar = micaja.insertar(7);
            System.out.println(mensajeInsertar);

            // OBTENEMOS EL NÚMERO ACTUAL DE ELEMENTOS
            System.out.println("Elementos actualmente: " + micaja.getElementos());

            // EXTRAEMOS ELEMENTOS
            // Y RECOGEMOS Y MOSTRAMOS EL MENSAJE CORRESPONDIENTE
            String mensajeExtraer = micaja.extraer(3);
            System.out.println(mensajeExtraer);

            // OBTENEMOS EL NÚMERO ACTUAL DE ELEMENTOS
            System.out.println("Elementos actualmente: " + micaja.getElementos());

            // VACIAMOS LA CAJA DE ELEMENTOS
            micaja.vaciar();
            System.out.println("Se ha vaciado la caja");

            // OBTENEMOS EL NÚMERO ACTUAL DE ELEMENTOS
            System.out.println("Elementos actualmente: " + micaja.getElementos());
        }
}
```

POO: ENCAPSULACIÓN

Descripción

La encapsulación es la característica de la orientación a objetos que consisten en que los objetos ocultan sus atributos (e incluso los métodos), a otros objetos.

Protección de datos

Los datos se protegen, al no hacerlos accesibles directamente desde el exterior, sino accesibles de una forma controlada por el programador, mediante sus métodos accesibles correspondientes.

La forma natural de construir una clase es la de definir una serie de atributos que, no son accesibles fuera del mismo objeto (atributos con modificador de ámbito "private"), sino que únicamente pueden modificarse a través de los métodos que son definidos como accesibles desde el exterior de esa clase.

Ejemplo de una encapsulación correcta dentro de una clase:

```
//Declaración de atributos privados ("private").
//No son accesibles directamente desde el exterior.
private int valor1;
private int valor2;

//Método constructor que recibe dos parámetros
//de la instanciación del objeto.
//Dichos parámetros se asignan a los atributos privados
//declarados al principio de la clase, con lo que
//los valores de dichos atributos quedan disponibles
//para toda la clase.
Miclase(int v1, int v2) {
	valor1 = v1;
	valor2 = v2;
}

//Método que al no tener especificado el modificador de ámbito,
//tendría el modificador por defecto ("default" o "package"),
//con lo que dicho método es accesible desde una clase exterior
//del mismo paquete.
//Por lo que con este método se accede directamente al
//atributo privado "valor1".
getValor1() {
	return valor1;
}
```

Facilidad en el mantenimiento de la clase

Con la encapsulación tambien se facilita el mantenimiento de las clases, ya que la encapsulación permite de una manera organizada, controlar lo accesible y lo no accesible en las clases.

POO: SOBRECARGA DE MÉTODOS

Descripción

La sobrecarga de métodos consisten en que en una clase existan dos o más métodos que tengan el mismo nombre, pero que cada uno de ellos tenga una signatura (cabecerá de declaración del método con los argumentos que recibe) diferente (distinto número de argumentos, distintos tipos de argumento, distinto orden de recepción de argumentos).

El objetivo de la sobrecarga de métodos es tener diferentes implementaciones para un método, ya que si no, habría que hacer métodos con diferente nombre, para realizar prácticamente la misma acción.

Sobrecarga de un método

Por ejemplo, supongamos que se dispone de un método "producto", que multiplica dos números. Si se desea que el método pueda recibir 3 números o 4 números, se sobrecargaría el método "producto" de la siguiente manera:

```
//Método con implementación para recibir 2 argumentos
void producto(int val1, int val2)
{
	System.out.println("Resultado: " + (val1 * val2));
}

//Método con implementación para recibir 3 argumentos
void producto(int val1, int val2, int val3)
{
	System.out.println("Resultado: " + (val1 * val2 * val3));
}

//Método con implementación para recibir 4 argumentos
void producto(int val1, int val2, int val3, int val4)
{
	System.out.println("Resultado: " + (val1 * val2 * val3 * val4));
}
```

Ahora se podría llamar al método sobrecargado, de cualquiera las siguientes maneras:

```
//Se instancia la clase, creando el objeto
Miclase miobjeto = new Miclase();

//Se ejecuta el método pasando 2 argumentos
miobjeto.producto(2, 4);

//Se ejecuta el método pasando 3 argumentos
miobjeto.producto(2, 4, 5);

//Se ejecuta el método pasando 4 argumentos
miobjeto.producto(2, 4, 5, 7);
```

Sobrecarga del método constructor

También se puede sobrecargar el método constructor, con el objetivo de que en la creación del objeto, se pueda inicializar dicho objeto con estados diferentes, es decir, con diferentes valores en sus atributos.

Por ejemplo, se podría crear un objeto coche, inicializando su atributo gasolina, con el valor 0, con el valor 10 ó con el valor 30, lo que daría la posiblidad de inicializar cada coche con los litros de gasolina deseados.

Ejemplo de sobrecarga de método constructor, implementando 3 formas diferentes de crear el objeto:

```
//Implementación del constructor que no recibe parámetros.
//Se establecen valores por defecto en todos los atributos.
Miclase() {
	this.nombre = "Pendiente";
	this.apellidos = "Pendiente";
	this.edad = 18;
}

//Implementación del constructor que recibe 1 parámetro.
//Se establecen valores por defecto en el resto de atributos.
Miclase(String elnombre) {
	this.nombre = elnombre;
	this.apellidos = "Pendiente";
	this.edad = 18;
}

//Implementación del constructor que recibe 2 parámetros.
//Se establecen valores por defecto en el resto de atributos.
Miclase(String elnombre, String losapellidos) {
	this.nombre = elnombre;
	this.apellidos = losapellidos;
	this.edad = 18;
}

//Implementación del constructor que recibe los 3 parámetros.
//No se establece ningún parámetro por defecto.
Miclase(String elnombre, String losapellidos, byte laedad) {
	this.nombre = elnombre;
	this.apellidos = losapellidos;
	this.edad = laedad;
}

//Método que muestra el resultado
void mostrar()
{
	System.out.println("Nombre: " + this.nombre);
	System.out.println("Apellidos: " + this.apellidos);
	System.out.println("Edad: " + this.edad);
}
```

Ahora se podría llamar al método construtor sobrecargado, de cualquiera las siguientes maneras:

```
//Se instancia la clase, creando el objeto.
//No se le pasan parámetros al método constructor.
Miclase miobjeto1 = new Miclase();

//Se instancia la clase, creando el objeto.
//Se le pasa 1 parámetro al método constructor.
Miclase miobjeto2 = new Miclase("Juan");

//Se instancia la clase, creando el objeto.
//Se le pasan 2 parámetros al método constructor.
Miclase miobjeto3 = new Miclase("Juan", "Sanz Rojas");

//Se instancia la clase, creando el objeto.
//Se le pasan 2 parámetros al método constructor.
//Se hace casting (conversión) con el número 45, ya que
//al escribir directamente el número 45, por defecto, Java
//lo toma como un int.
```

```
//Lo mas adecuado sería declarar previamente el atributo de
//la edad y después, escribir aquí dicho atributo, en lugar
//de escribir directamente el número.
Miclase miobjeto4 = new Miclase("Juan", "Sanz Rojas", (byte) 45);

//Se ejecuta el método del objeto1 que muestra el resultado
miobjeto1.mostrar();

//Se ejecuta el método del objeto2 que muestra el resultado
miobjeto2.mostrar();

//Se ejecuta el método del objeto3 que muestra el resultado
miobjeto3.mostrar();

//Se ejecuta el método del objeto4 que muestra el resultado
miobjeto4.mostrar();
}
```

POO: HERENCIA

Descripción

Es una de las principales ventajas de la POO. La herencia permite definir clases descendientes de otras, de forma que la nueva clase (la clase descendiente) hereda de la clase antecesora todos sus miembros (atributos y métodos) que tengan el modificador adecuado para ello (modificador por defecto, "public" o "protected").

En Orientación a objetos, la herencia se denomina "generalización".

Ventajas de la herencia

La herencia es un mecanismo natural de reutilizar los objetos. Por ejemplo, se podría tener una clase "Casa" que tuviera unos miembros generales, para que después crear subclases (por ejemplo: apartamento, chalé, etc [es decir, tipos de casas]) que hereden de dicha clase "Casa" dichos miembros "generales" de dicha clase "Casa".

Por ejemplo, un chalé es un tipo de casa, que además tiene un jardín. Tiene las mismas características que una casa, pero además, tiene jardín.

Por ejemplo, un pato es un ave que nada. Mantiene las mismas características que las aves y únicamente habría que declarar un método (el método nadar).

La superclase es la clase de la cual heredan sus miembros (atributos y métodos) otras subclases.

Denominación de las clases en la herencia

Existen varias denominaciones que se pueden dar a las clases en el mecanismo de la herecia. A continuación se muestran las diferentes denominaciones en un orden de más usada a menos usada:

CLASE QUE CONTIENE LOS MIEMBROS	CLASE QUE HEREDA LOS MIEMBROS
Superclase	Subclase
Padre, Madre	Hija
Principal	Secundaria
Antecesora	Sucesora

Reutilización de código

Mediante la herencia, se consigue la reutilización de código, que permite aprovechar el código de clases ya existentes, modificándo si se desea la subclase, para adaptarla a las nuevas necesidades.

Herencia de la superclase "Object"

Todas las clases, de forma implícita, heredan los miembros de la superclase "Object". La clase "Object" se encuentra en el paquete "java.lang" (java.lang.Object).

Extender (heredar) de una superclase

Para especificar que una clase herede de otra clase, se debe utilizar la palabra reservada "extends" al final de la declaración de la clase hija, seguida de la superclase de la que se desea heredar.

La sintaxis para realizar la herencia es la siguiente:

classs NombreSubclase extends NombreSuperclase.

Ejemplo de declaración de subclase "Perro" que hereda de la superclase "Animal":

```
class Perro extends Animal {
	...
}
```

Creación de herencia en Java

Ejemplo de implementación de una superclase:

```
class MiSuperclase {

	//DECLARACIÓN DE ATRIBUTOS DE LA SUPERCLASE
	//----------------------------------------
	//Atributo privado.
	//NO accesible desde el exterior.
	//NO lo heredan las clases hijas.
	private int superclase_atributo_private = 1;

	//Atributo protegido.
	//NO accesible desde el exterior para clases de otros paquetes.
	//SI accesible desde el exterior para clases del mismo paquete.
	//SI lo heredan las clases hijas.
	protected int superclase_atributo_protected = 2;

	//Atributo publico.
	//SI accesible desde el exterior.
	//SI lo heredan las clases hijas.
	public int superclase_atributo_public = 3;


	//DECLARACIÓN DE MÉTODOS DE LA SUPERCLASE
	//---------------------------------------
	//Método privado.
	//NO accesible desde el exterior.
	//NO lo heredan las clases hijas.
	private void superclase_metodo_private() {
		System.out.println("superclase_metodo_private");
	}

	//Método protegido.
	//NO accesible desde el exterior para clases de otros paquetes.
	//SI accesible desde el exterior para clases del mismo paquete.
	//SÍ lo heredan las clases hijas.
	protected void superclase_metodo_protected() {
		System.out.println("superclase_metodo_protected, de la SUPERCLASE");
	}

	//Método público.
	//SI accesible desde el exterior.
	//SI lo heredan las clases hijas.
	public void superclase_metodo_public() {
		System.out.println("superclase_metodo_public");
	}
```

Ejemplo de implementación de subclase que heredará los miembros (los no privados) de la superclase:

```
class Subclase extends MiSuperclase {

```

```
	...

}
```

NOTAS:

- Con este sencillo mecanismo de herencia mediante la palabra reservada "extends", sin tener que escribir nada de código en la subclase, se heredan automáticamente los miembros (no privados) de la superclase, es decir, los miembros "public" y "protected".
- Si ahora en "eclipse" se crea un objeto de la subclase y se escribe el nombre de dicho objeto seguido de un punto, aparecerá una lista con todos los miembros a los que puede acceder dicha subclase, que serán los miembros "public" y "protected" de su superclase.

Cuadro de modificadores de acceso

OJO: fijarse que protected es accesible desde clases del mismo paquete, pero no desde clases de otro paquete.

MODIFICADOR	CLASE	PACKAGE	SUBCLASE	TODOS
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
No especificado	Sí	Sí	No	No
private	Sí	No	No	No

Definición de nuevos miembros en la subclase

La subclase que hereda los miembros (no privados) de la superclase, también puede definir nuevos miembros (atributos y clases) para dicha subclase. Con este mecanismo, la subclase hereda de la superclase y además implementa sus propios miembros que sean necesarios para el funcionamiento de dicha subclase.

Ejemplo de implementación de subclase que heredará los miembros (los no privados) de la superclase, y además implementa un atributo y un método nuevos, para así, completar o ampliar el funcionamiento específico de esta subclase:

```
class Subclase extends Superclase {

	//Atributo perteneciente a la subclase
	private int subclase_atributo_private = 5;

	//Método perteneciente a la subclase
	private void subclase_metodo_private() {
		System.out.println("subclase_metodo_private");
	}

}
```

NOTAS:

- Si ahora en "eclipse", dentro del método de esta subclase se escribe la palabra "this" seguida de un punto, aparecerá una lista con todos los miembros a los que puede acceder dicha subclase, que serán los miembros "public" y "protected" heredados de la superclase, y además el atributo y el método específicos de esta subclase.

Redefinición (sobreescritura) de miembros (atributos y métodos)

La subclase puede redefinir (sobreescribir) los miembros heredados de su superclase. Este mecanismo sirve para crear miembros en la subclase con el mismo nombre que miembros de la superclase, con el objetivo de redefinir algún miembro para cambiarle su implementación en la subclase, y con ello, cambiar, por ejemplo, las acciones que realiza un método heredado de la superclase, para que tenga otro funcionamiento en la subclase. O también, redefinir un atributo de la superclase. La redefinición se produce, porque coincide el nombre del miembro que se implementa en la subclase con el nombre de otro miembro heredado de la superclase.

Ejemplo de implementación de un método en la subclase, que redefine un método heredado de la superclase. Al ejecutar el método, se ejecutará dicho método heredado de la subclase y el método heredado de la clase superclase queda "anulado":

```
class Subclase extends Superclase {

	//Atributo perteneciente a la subclase
	private int subclase_atributo_private = 5;

	//Método perteneciente a la subclase
	private void subclase_metodo_private() {
		System.out.println("subclase_metodo_private");
	}

	//Método que se hereda de la superclase, pero que se
	//redefine (sobreescribe) en esta subclase.
	//(ya que tiene el mismo nombre que dicho método heredado
	//de la superclase).
	protected void superclase_metodo_protected() {
		System.out.println("superclase_metodo_protected, ");
		System.out.println("REDEFINIDO EN LA SUBCLASE");
	}

}
```

NOTA: si se eliminara este método de la subclase que ha redefinido el método de la superclase, o se impidiera su ejecución mediante comentarios, no se redefiniría el método de la superclase, y el método que se ejecutaría sería dicho método de la superclase.

Referencia a miembros de la subclase ("this") y de la superclase ("super")

Con la palabra reservada "this" se hace referencia a la clase actual.

Mediante la palabra reservada "super" se hace referencia, desde una subclase, a cualquier miembro de la superclase de los que ha heredado dicha subclase.

Ejemplo que desde una subclase, donde se ha redefinido un método heredado de la superclase, se hace referencia al método redefinido de la propia subclase, y también se hace referencia al método heredado de la superclase:

```
//SUPERCLASE
class Superclase {

	...

	//MÉTODO DE LA SUPERCLASE
	protected void superclase_metodo_protected() {
		System.out.println("MÉTODO DE LA SUPERCLASE");
	}

	...
```

```
}
```

```
//SUBCLASE
class Subclase extends Superclase {

	...

	//Método que se hereda de la superclase, pero que se
	//redefine (sobreescribe) en esta subclase.
	//(ya que tiene el mismo nombre que dicho método heredado
	//de la superclase).
	protected void superclase_metodo_protected() {
		System.out.println("superclase_metodo_protected, ");
		System.out.println("REDEFINIDO EN LA SUBCLASE");
	}

	//Método que permite referirse al método de la superclase y
	//al método de esta subclase.
	protected void ejecutar_padre_hija() {
		//En este ejemplo, mediante "this", se hace referencia
		//al método de esta subclase.
		//Con ello, se ejecuta el método que ha redefinido
		//al método de la superclase.
		this.superclase_metodo_protected();
		//En este ejemplo, mediante "super",
		//se hace referencia al método de la superclase.
		//Con ello, aunque el método de la superclase
		//se ha redefinido en esta subclase,
		//mediante "super" se hace referencia al método
		//redefinido de la superclase.
		super.superclase_metodo_protected();
	}

}
```

Métodos finales "final"

Los métodos "final" en una clase hace que dichos métodos "final" no puedan ser redefinidos en las subclases.

En una clase pueden existir métodos "final", sin que la clase que contiene dichos métodos "final", tenga que ser "final".

Los métodos "final" es un mecanismo que sirve para proteger algunos métodos de la clase (los que sean "final") para que no puedan ser redefinidos por las subclases. Dichos métodos "final" no podrán ser redefinidos en las subclases.

Ejemplo de método "final" que no podrá ser redefinido en las subclases. Este método está implementado en una superclase:

```
final protected void superclase_metodo_protected() {
	System.out.println("superclase_metodo_protected, de la SUPERCLASE");
}
```

Al intentar redefinir el método "final" del ejemplo anterior, en una subclase de dicha clase del ejemplo anterior, de esta forma...:

```
protected void superclase_metodo_protected() {
	System.out.println("superclase_metodo_protected, ");
```

```
      System.out.println("REDEFINIDO EN LA SUBCLASE");
}
```

El editor o compilador, mostrará el siguiente error:

"Cannot override the final method from Superclase"

(No se puede redefinir [sobreescribir] el método final "superclase").

Herencia simple

Herencia simple significa que una clase solo puede heredar de solo una superclase, mediante el mecanismo de herencia con la palabra reservada "extends". Si se desea que una clase implemente los miembros de varias clases (miembros sin implementación) entonces se necesitará que la clase que desee dicho funcionamiento "implemente" una o varias interfaces.

CLASE SUPERCLASE: OBJECT

Descripción

Todas clases heredan automáticamente de la superclase "Object", superclase de todas las clases. Por lo que todas las clases en Java heredan automáticamente todos los miembros correspondientes a la superclase "Object".

La superclase "Object" es la raíz en la jerarquía de todas las clases de Java.

La superclase "Object" se encuentra en el paquete "java.lang" (java.lang.**Object**).

POO: CLASES FINALES

Descripción

Una clase "final" es una clase que no permite heredar de ella, es decir, una clase "final" no permite crear subclases a partir de dicha clase "final".

Se puede declarar una clase como "final", cuando no nos interesa crear clases derivadas (subclases) de dicha clase.

Seguridad con clases finales

Uno de los mecanismos para dañar o para obtener información privada en los sistemas, es la de crear una subclase a partir de una clase original, y sustituir dicha clase original por la subclase. Dicha subclase tiene el mismo comportamiento que la clase original, pero también se puede implementar en dicha subclase cualquier código dañino que se desee.

Para prevenir los posibles daños, se declaran dichas clases críticas del sistema como "final", impidiendo a cualquier programador la creación de clases derivadas de ésta.

Por ejemplo, la clase "String" (que es una de las más importantes en la programación Java), está declarada como "final". De esta forma, Java garatiza que siempre que se utilice un string, dicho string sea un objeto de la clase "String" (que se encuentra en el paquete "java.lang.**String**").

Declaración de clases finales

Ejemplo que declara una superclase "final":

```
final class Superclase {
```

Si ahora, se intentara crear una subclase que heredara de la superclase "final", de la siguiente manera...:

```
class Subclase extends Superclase {
```

El editor o compilador, mostrará el siguiente error:

"The type Subclase cannot subclass the final class Superclase"

(La subclase "Subclase" no puede ser subclase de la clase final "Superclase").

Otro ejemplo de declaración de clase "final":

```
final class Cuadrado extends Rectangulo {
```

La subclase "Cuadrado" (que hereda de la superclase "Rectangulo"), se puede declarar como "final", ya que, se puede dar el caso de que ningún programador necesite crear clases derivadas de esta subclase "Cuadrado".

POO: CLASES ABSTRACTAS

Descripción

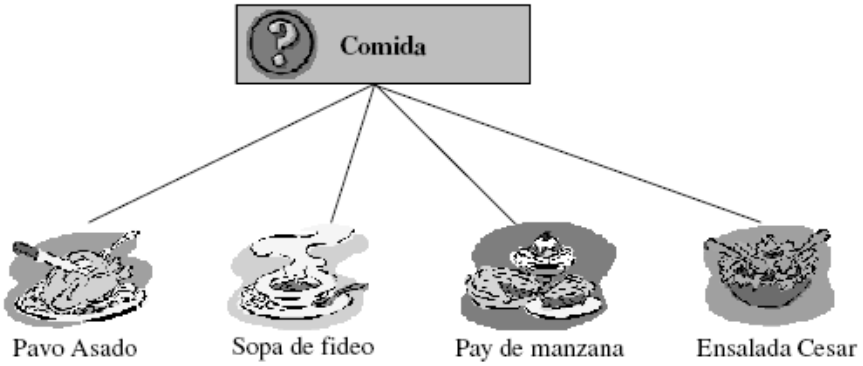
Una clase abstracta sirve de base a subclases que hereden de dicha clase abstracta, teniendo en cuenta que dichas subclases deben redefinir los métodos de la clase abstracta que han sido declarados "abstract".

Como una clase abstracta está pensada para servir de "base" o "plantilla" para subclases que hereden de dicha clase abstracta, dicha clase abstracta no se puede instanciar directamente.

La clase abstracta NO puede instanciarse (es decir no se pueden crear objetos de una clase abstracta). Esto es coherente, dado que una clase abstracta no tiene completa su implementación (solo tiene declarados los miembros, no implementados) y encaja bien con la idea de que algo abstracto no puede materializarse.

Una clase abstracta es útil para crear una jerarquía de clases y tener en cuenta que la clase abstracta "padre" será la BASE COMÚN para todas sus clases "hijas". Una clase abstracta sólo puede servir como clase base (sólo se puede heredar de ella). Sus clases hijas SÍ pueden instanciarse.

La clase abstracta define las definiciones "por defecto" esperadas en cada subclase que quiera heredar de dicha clase abstracta. Las operaciones (métodos) en una clase abstracta es como dejar "moldes" hechos, para que puedan ser utilizados (implementados) en las subclases que hereden dicha clase abstracta.



La "Comida" como tal, es solo un concepto abstracto que NO puede instanciarse.
Existen muchos alimentos que heredan sus características y
ellos SI pueden existir por si mismos.
"Comida" es una clase Abstracta.

Características generales de las clases abstractas

- La clase abstracta es una clase que sirve como clase base COMÚN para que otras clases hijas hereden de ella sus miembros que se deseen "comunes". La implementación de dichos miembros (atributos y métodos) se realiza en las subclases.
- Una clase abstracta solo tiene declarados sus métodos, pero no contiene su implementación (es decir, el código de los métodos), es decir, solo se declara su signatura (su cabecera y sus parámetros si los lleva), no se deberá declarar todo el código encerrado entre llaves { }.
    - La clase abstracta también puede incluir la implementación de sus métodos (el código de dichos métodos), pero no es lo habitual.
- Los miembros de una clase abstracta puede tener cualquier tipo (final, static, etc.).
- Las clases abstractas (con el modificador "abstract") no pueden ser instanciadas directamente.
- La clase abstracta define las definiciones "por defecto" esperadas en cada subclase que quiera heredar de dicha clase abstracta. Las operaciones (métodos) en una clase abstracta es como dejar "moldes" hechos, para que puedan ser utilizados (implementados) en las subclases que hereden dicha clase abstracta.
- Un método abstracto (de una clase abstracta) únicamente declara su signatura o cabecera (su nombre y parámetros) y no el cuerpo de dicho método.

- No es obligatorio que todos los métodos de una clase abstracta sean abstractos, incluso es posible que ninguno de ellos lo sea. En el caso de que ningún método de una clase abstracta, sea abstracto, la clase será considerada como abstracta y no podrán declararse objetos de esta clase, es decir, no se podrá instanciar directamente.
- Cuando se declara un método "abstract", no se implementa el cuerpo del método, sólo su signatura.
- Las clases que se declaran como subclases de una clase "abstract" deben implementar obligatoriamente los métodos abstractos que estén declarados en la clase "abstract".
- Una clase abstract no puede ser instanciada, únicamente sirve para ser utilizada como superclase de otras clases.
- Cuando alguno de los métodos de una clase es declarado abstracto, la clase debe ser obligatoriamente abstracta, de lo contrario, el compilador genera un mensaje de error.
- Las clases abstractas se crean para ser superclases de otras clases. En el ejemplo de una clase abstracta "Animal" se puede declarar el método "habla()" como abstracto, porque se desea que todos los animales puedan hablar, pero no sabemos qué es lo que van a decir (qué acciones se van a realizar), por lo que es declarada de tipo abstract. Las clases que heredan de la clase abstracta "Animal" deben implementar obligatoriamente un método "habla()" para poder heredar las características de Animal.

Declaración de clases abstractas

Ejemplo que declara una clase abstracta:

```
abstract class Figura {
```

Declaración de los miembros en las clases abstractas

Ejemplo que declara los miembros dentro de la clase abstracta:

```
abstract class Figura {

	//DECLARACIÓN DE LA SIGNATURA DE LOS MÉTODOS DE LA CLASE ABSTRACTA
	//---------------------------------------------------------------

	//Solo se declara la signatura de los métodos.
	//Su implementación se declara en las subclases
	//que hereden de esta clase abstracta.

	//Método que permite dibujar la figura.
	abstract void dibujar();

	//Método que permite borrar la figura.
	abstract void borrar();

	//Método que permite editar la figura.
	abstract void editar();

}
```

Declaración de la subclase que hereda de la clase abstracta

La clase abstracta se declara simplemente con el modificador "abstract" en su declaración.

Ejemplo de subclase "Circulo" que hereda de la superclase abstracta "Figura":

```
class Circulo extends Figura {
```

NOTA: si en la subclase no se implementan todos los métodos abstractos de la superclase abstracta de la que se hereda, el editor o compilador lanzará el siguiente error:

"The type Circulo must implement the inherited abstract method Figura.dibujar()"

(La clase Círculo debe implementar el método abstracto heredado Figura.dibujar())

Implementación en la subclase de los métodos de la clase abstracta

Ejemplo que implementa en la subclase los métodos de la clase abstracta:

```
class Circulo extends Figura {

	//Método que permite dibujar la figura.
	void dibujar() {
		System.out.println("Dibujar el círculo");
	}

	//Método que permite borrar la figura.
	void borrar() {
		System.out.println("Borrar el círculo");
	}

	//Método que permite editar la figura.
	void editar() {
		System.out.println("Editar el círculo");
	}

}
```

NOTA: si en la subclase no se implementan todos los métodos abstractos de la superclase abstracta de la que se hereda, el editor o compilador lanzará el siguiente error:

"The type Circulo must implement the inherited abstract method Figura.dibujar()"

(La clase Círculo debe implementar el método abstracto heredado Figura.dibujar())

El editor o compilador ofrecerá la solución de que, o bien, se implementan dichos métodos, o bien se declara la clase abstracta.

Uso de la clase abstracta

La clase abstracta no se puede instanciar directamente. Lo que se debe instanciar es una subclase que herede de la superclase abstracta.

Si se intenta instanciar directamente la clase abstracta, el editor o compilador mostrará el siguiente error:

"Cannot instantiate the type Figura"

(No puede instanciar la clase Figura)

Lo que se debe hacer es instanciar una subclase de la superclase. Una vez instanciada dicha subclase y creado el objeto, se podrán utilizar todos los métodos de dicha subclase, es decir, como cualquier subclase normal y corriente.

Incompatibilidad entre abstracta y final

Una clase no puede ser a la vez "abstract" y "final" ya que no tiene sentido, debido a que la clase "abstract" está pensada para que se herede de ella, y "final" impide que se pueda heredar de ella.

Pero una clase sí puede ser "public abstract" o "public final".

POO: INTERFACES

Descripción

Una interfaz es un conjunto de constantes y métodos, pero de los métodos solamente su signatura, no su implementación.

Una interfaz es una plantilla que define el contenido que debe aparecer en la clase o clases que implementen dicha interfaz.

Una interfaz es una especie de "contrato" mediante el cual el programador tiene la obligación de implementar todos los métodos de una interfaz (signatura + código) o interfaces, que haya implementado en la clase deseada.

Una interfaz contiene una colección de métodos que se implementan en otro lugar. Siendo estos métodos implícitamente "**abstract**" y "**public**".

El concepto de Interface lleva un paso más adelante la idea de las clases abstractas. En Java una interface es una clase abstracta pura, es dcir una clase donde todos los métodos son abstractos (no se implementa ninguno). Permite al diseñador de clases establecer la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno, pero no bloques de código). Una interface puede también contener datos miembro, pero estos son siempre static y final. Una interface sirve para establecer un "protocolo" entre clases, es decir, obliga a que las clases implementen ciertas acciones (métodos) para funcionar, es decir, para cumplir con dicho "protocolo".

Los métodos abstractos son útiles cuando se quiere que cada implementación de la clase parezca y funcione igual, pero se necesita crear una nueva clase para utilizar los métodos abstractos. Las interfaces proporcionan un mecanismo para abstraer los métodos a un nivel superior, lo que permite simular la herencia múltiple de otros lenguajes.

Una interfaz sublima el concepto de clase abstracta hasta su grado más alto. Una interfaz podrá verse simplemente como una forma, parecido a un molde, solamente permite declarar nombres de métodos, listas de argumentos, tipos de retorno y adicionalmente miembros datos (los cuales podrán ser únicamente tipos básicos y serán tomados como constantes en tiempo de compilación).

Las interfaces Java son expresiones puras de diseño. Se trata de auténticas conceptualizaciones no implementadas que sirven de guía para definir un determinado concepto (clase) y lo que debe hacer, pero sin desarrollar un mecanismo de solución.

Se trata de declarar métodos abstractos y constantes que posteriormente puedan ser implementados de diferentes maneras según las necesidades de un programa.

Por ejemplo una misma interfaz podría ser implementada en una versión de prueba de manera poco óptima, y ser acelerada convenientemente en la versión definitiva tras conocer más a fondo el problema.

Características generales de las interfaces

- Una interfaz NO es una clase.
- Una interfaz NO puede tener la implementación de sus métodos (el código implementado en sus métodos). Una interfaz solo debe tener declarada la signatura de sus métodos.
- Los modificadores de acceso de los miembros de una interfaz son todos "public".
- En una interfaz, todos sus atributos (si los tiene) deben ser constantes "final" y de clase "static".
- Una clase solo puede heredar de una sola clase (tener un solo padre), pero puede implementar varias interfaces, aunque dichas clases no estén relacionadas con herencia.
    - Este mecanismo es similar a la herencia múltiple (de otros lenguajes).
- Una interfaz puede heredar de varias interfaces.

Consejos para utilizar interfaces

- Se deben crear interfaces que contengan declaraciones de métodos (solo su signatura) que después implementen una o varias clases. Dichas clases que implementen dicha interfaz, implementarán todos los métodos (con su código correspondiente) que contenga dicha interfaz.
- Simular la herencia múltiple.
- Las interfaces son útiles para recoger las similitudes entre clases no relacionadas, forzando una relación entre ellas.
- Declarar métodos que forzosamente una o más clases han de implementar.
- Tener acceso a un objeto permitiendo el uso de dicho objeto sin revelar su clase, son los llamados "objetos anónimos" (creados a partir de una clase anónima), que son muy útiles cuando se vende un paquete de clases a otros desarrolladores.

Diferencias: clase abstracta - interfaz

- Una clase abstracta puede tener, o no, la implementación de sus métodos (el código implementado en sus métodos), sin embargo, las interfaces no pueden implementar código en ninguno de sus métodos.
- En una clase abstracta, sus miembros pueden tener diferentes modificadores de acceso (público, privado, etc.), sin embargo, los miembros de una interfaz son todos públicos.
- En una clase abstracta, sus miembros pueden tener cualquier modificador, sin embargo, en una interfaz todos los miembros deben ser constantes "final" y de clase "static".
- Las interfaces proporcionan una forma de herencia múltiple, ya que pueden implementar múltiples interfaces. Una clase sólo puede extender a otra clase, incluso aunque esa clase sólo tenga métodos "abstract".
- Una clase "abstract" puede tener una implementación parcial, partes protegidas, métodos estáticos, etc., mientras que las interfaces están limitadas a constantes públicas y métodos públicos sin implementación.
- La principal diferencia entre una clase abstracta y una interfaz es que una interfaz proporciona un mecanismo de encapsulación de los protocolos de los métodos de dicha interfaz, y todo ello sin la necesidad de utilizar herencia.

Declaración de interfaces

Para crear una interfaz, se utiliza la palabra clave "interface" en lugar de "class". La interfaz puede definirse "public" o sin modificador de acceso (igual que para las clases).

Todos los métodos que declara una interfaz son siempre "public". Una interface puede también contener datos miembro (atributos), pero estos son siempre "static" y "final".

Sintaxis para declarar una interfaz:

```
interface nombre_interface {
        tipo_retorno nombre_metodo (lista_argumentos);
...
}
```

Ejemplo que declara una interfaz:

```
interface Video {
```

Declaración de los métodos de la interfaz

Dentro de la interfaz habrá que declarar los métodos (solo la signatura, es decir, la cabecera y parámetros, pero no la implementación de su código).

Por defecto, es decir, de forma implícita, los métodos declarados dentro de la interfaz serán automáticamente "abstract" y "public".

```
interface Video {

      //DECLARACIÓN DE LA SIGNATURA DE LOS MÉTODOS DE LA INTERFAZ
```

```
	//-------------------------------------------------------
	//Solo se declara la signatura de los métodos.
	//Su implementación se declara en las clases
	//que implementen esta interfaz.
	//----------------------------
	//De forma implícita, los métodos de la interfaz
	//son "abstract" y "public".

	//Método que reproduce el video
	void play();

	//Método que hace pausa en el video
	void pause();

	//Método que detiene el video
	void stop();

}
```

Si se intenta implementar el código en un método de la interfaz, por ejemplo, de esta manera...:

```
void pause() {
	System.out.println("Pausa en video");
}
```

el editor o compilador mostrará el siguiente error:

"Abstract methods do not specify a body"

(Los métodos abstractos no implementan un cuerpo).

Recordar que por defecto (de forma implícita) los métodos de las interfaces son "abstract" y "public".

NOTA: en las clases abstractas no genera error implementar el código de un método (aunque no es recomendable hacerlo), pero si se implementa el código de un método en una interfaz, generará error.

Declaración de implementación de una interfaz en una clase

Para indicar que una clase implementa una interfaz (y que por supuesto, implementa los métodos de dicha interfaz) se utiliza la palabra clave "implements".

El compilador se encargará de verificar que la clase efectivamente declare e implemente todos los métodos de la interfaz.

Ejemplo de clase "Miclase" que implementa una interfaz "Video":

```
class Miclase implements Video {
```

Implementación de los métodos de la interfaz en la clase

La clase que implementa una o varias interfaces tiene la obligación de implementar el código de TODOS los métodos declarados en dicha interfaz que implementa.

```
class Miclase implements Video {

	//La implementación de los métodos debe hacerse con el
	//modificador "public", ya que de forma implícita, los
	//métodos de la interfaz son "abstract" y "public".

	//Método que permite reproducir el video
```

```
	public void play() {
		System.out.println("Reproducir video");
	}

	//Método que permite hacer una pausa en el video
	public void pause() {
		System.out.println("Pausa en video");
	}

	//Método que permite detener el video
	public void stop() {
		System.out.println("Detener video");
	}

}
```

Si no se implenta el código de algún método de la interfaz, en la clase, el editor o compilador mostrará el siguiente error:

"This method requires a body instead of a semicolon"

(Este método requiere un cuerpo en lugar de un punto y coma).

NOTA: ya que con el punto y coma se termina la declaración de cada método en la interfaz, y dicho punto y coma significa que termina la declaración de cada método en la interfaz y en dicha interfaz no se implementa el código. Pero aquí en la clase hay que implementar el código de cada método.

Uso de la interfaz

Para usar la interfaz, se instanciará la clase que implementa la interfaz:

```
//Se instancia la clase que implementa la interfaz
Miclase miobjeto = new Miclase();

//Se ejecutan los métodos de dicha clase, que son los métodos de la
//interfaz, que la clase implementó con el código de dichos métodos.
miobjeto.play();
miobjeto.pause();
miobjeto.stop();
```

La interfaz no se puede instanciar directamente, ya que está diseñada para que otras clases implementen dicha interfaz.

Si se intenta instanciar directamente una interfaz, el editor o compilador mostrará el siguiente error:

"Cannot instantiate the type Video"

(No se puede instanciar el tipo (la intefaz) "Video").

Implementación de varias interfaces en una clase

Una clase puede implementar más de una interface, con lo que tendrá que la obligación de implementar el código correspondiente a todos los métodos que contengan dichas interfaces que implemente.

Ejemplo de clase "Miclase" que implementa dos interfaces: "Video" y "Sonido":

```
class Miclase implements Video, Sonido {
```

Ejemplo que declara la interfaz "Sonido" y también declara sus métodos:

```
interface Sonido {

	//Método que reproduce el sonido
	void playsonido();

	//Método que hace pausa en el sonido
	void pausesonido();

	//Método que detiene el sonido
	void stopsonido();

}
```

Ejemplo de clase "Miclase" que implementa las interfaces "Video" y "Sonido" y que obligatoriamente implementa el código correspondiente a todos los métodos existentes en dichas interfaces "Video" y "Sonido":

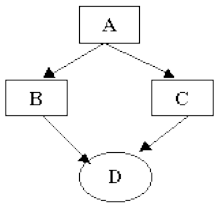
```
class Miclase implements Video, Sonido {

	//IMPLEMENTACIÓN DE MÉTODOS DE LA INTERFACE "Video"
	//------------------------------------------------

	//Método que permite reproducir el video
	public void play() {
		System.out.println("Reproducir video");
	}

	//Método que permite hacer una pausa en el video
	public void pause() {
		System.out.println("Pausa en video");
	}

	//Método que permite detener el video
	public void stop() {
		System.out.println("Detener video");
	}

	//IMPLEMENTACIÓN DE MÉTODOS DE LA INTERFACE "Sonido"
	//-------------------------------------------------

	//Método que permite reproducir el sonido
	public void playsonido() {
		System.out.println("Reproduce sonido");
	}

	//Método que realiza una pausa en el sonido
	public void pausesonido() {
		System.out.println("Pausar sonido");
	}

	//Método que permite detener el sonido
	public void stopsonido() {
		System.out.println("Detiene sonido");
	}

}
```

Herencia múltiple

Herencia simple significa que una clase solo puede heredar de solo una superclase, mediante el mecanismo de herencia con la palabra reservada "extends". Si se desea que una clase implemente los miembros de varias clases (miembros sin implementación) entonces se necesitará que la clase que desee dicho funcionamiento "implemente" una o varias interfaces.

Mediante las interfaces se puede "simular" la herencia múltiple de otros lenguajes. Aunque también es importante saber que el lenguaje Java no implementa la herencia múltiple "pura" ya que considera que heredar miembros de múltiples clases no es muy recomendable con respecto a la organización y el diseño de clases.

Ejemplo que muestra el mecanismo de hacer la herencia múltiple en Java, que se representa con la herencia en forma de diamante:



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 { }
```

NOTA: la clase D tendría que implementar el código de los métodos pertenecientes a las interfaces: A, B y C. (También los de la A).

POO: POLIMORFISMO

Descripción

Poli = muchos/as
morfismo = formas

"Muchas formas", lo que significa, que por ejemplo, con un solo método, dependiendo del objeto que reciba, se podrá ejecutar el método perteneciente a dicho objeto recibido.

El polimorfismo se junto con la herencia.

El polimorfismo es un concepto de la programación orientada a objetos que nos permite programar en forma general, en lugar de hacerlo en forma específica.

Sirve para programar objetos con características comunes (por ejemplo, que comparten un método, como "mover") y que dichos objetos compartan la misma superclase en una jerarquía de clases (es decir, que dichos objetos hereden de dicha superclase), con lo que en estos casos, se simplifica la programación.

Por dicho motivo, al mecanismo del polimorfismo se le denomina también "enlace dinámico" o "ligadura dinámica", ya que el polimorfismo se produce en "tiempo de ejecución", escogiendo el método adecuado, dependiendo del objeto que se reciba.

Ejemplo de polimorfismo

Por ejemplo, cinco subclases correspondientes a cinco animales, heredan de una superclase "Animal". Dicha clase "Animal" implementa un método "mover" que describe como se mueve un animal. Pero como cada uno de los cinco animales de las subclases se mueve de forma diferente, cada una de dichas subclases redefine el método "mover" de la subclase, implementando cada forma particular de moverse cada uno de dichos animales.

Bien, pues mediante el polimorfismo, se podría tener un único método que recibiese como parámetro el objeto correspondiente al tipo de animal (por ejemplo, gato), pero como tipo de dato "Animal", con lo que se ejecutaría el método correspondiente al animal que se ha recibido como parámetro. Es decir, por ejemplo, se envía el objeto correspondiente a un animal (un objeto correspondiente a la clase "Perro") y se le especifica que es de la clase "Animal" (**Animal miperro**), y como la clase "Animal" "conoce" que perro es una subclase de "Animal", se ejecutaría el método correspondiente a la subclase "Perro". Es decir, que especificando el objeto de la subclase, se ejecutará el método correspondiente a dicha subclase.

Creación de polimorfismo

Ejemplo de una superclase:

```
//SUPERCLASE

class Animal {

	//Método "mover" de la clase Animal.
	//En este ejemplo, este método no se ejecuta directamente,
	//pero tiene que existir, ya que el método "muevete" de la clase
	//"Polimorfismo" recibe un parámetro de clase "Animal"
	//(es decir, un objeto de tipo "Animal"), y después dicho método ejecuta
	//el método "mover", haciendo referencia inicial al método "mover" de la
	//clase "Animal", pero como en la clase "Inicio" pasa al método "muevete"
	//el objeto correspondiente a las subclases, por dicho motivo se ejecuta
	//el método "mover" correspondiente a cada una de dichas subclases que
	//corresponden a los 3 animales que heredan de la clase "Animal".
	void mover() {
		System.out.println("SE MUEVE UN ANIMAL");
	}
```

```
}
```

Ejemplo de tres subclases que heredan de la superclase:

```
class Perro extends Animal {

	//Método para que se mueva el perro
	public void mover() {
	    System.out.println("Se mueve el PERRO");
	}

}
```

```
class Gato extends Animal {

	//Método para que se mueva el gato
	public void mover() {
	    System.out.println("Se mueve el GATO");
	}

}
```

```
class Caballo extends Animal {

	//Método para que se mueva el caballo
	public void mover() {
	    System.out.println("Se mueve el CABALLO");
	}

}
```

Ejemplo de clase que realiza el polimorfismo:

```
class Polimorfismo {

	//Éste método recibe un parámetro de clase "Animal" (es decir,
	//un objeto de tipo "Animal"), que se le envía desde la clase "Inicio".
	//Después, este método, ejecuta el método "mover", haciendo referencia
	//inicial al método "mover" de la clase "Animal", pero como "mianimal"
	//representa al objeto que le viene de la clase "Inicio" (es decir,
	//los objetos de las subclases, es decir, de cada uno de los animales
	//[perro, gato, caballo]), se ejecuta el método "mover" correspondiente a
	//cada una de dichas subclases que corresponden a los 3 animales que heredan
	//de la clase "Animal".
	//ES DECIR:
	//ÉSTE MÉTODO RECIBE UN DATO DE TIPO "ANIMAL", PERO COMO DICHO DATO ES UN
	//ANIMAL EN CONCRETO, EJECUTARÁ EL MÉTODO CORRESPONDIENTE A DICHO ANIMAL,
	//ES DECIR, QUE ESTE MÉTODO COMO RECIBE EL OBJETO CORRESPONDIENTE AL ANIMAL
	//QUE SE LE ENVÍA, SABE QUE TIENE QUE EJECUTAR EL MÉTODO CORRESPONDIENTE
	//A DICHO ANIMAL.
	void muevete(Animal mianimal) {
		mianimal.mover();
	}

}
```

Ejemplo de ejecución del polimorfismo (este código, por ejemplo se ha puesto en la clase que incluye el método "main"):

```
//Se crean los objetos de las subclases
Gato migato = new Gato();
Perro miperro = new Perro();
Caballo micaballo = new Caballo();

//Se crea el objeto para al clase que realiza el polimorfismo
Polimorfismo accion = new Polimorfismo();

//Con el objeto correspondiente a la clase que realiza el polimorfismo
//se llama a su método "muevete".
accion.muevete(miperro);
accion.muevete(migato);
accion.muevete(micaballo);
```

NOTA: observar que con un único objeto, y pasándole como parámetro el objeto correspondiente a cada subclase, se ejecuta automáticamente el método asociado a cada uno de dichas subclases.

Ventajas de la utilización de polimorfismo

El polimorfismo es un concepto de la programación orientada a objetos que nos permite programar en forma general, en lugar de hacerlo en forma específica.

Sirve para programar objetos con características comunes (por ejemplo, que comparten un método, como "mover", por medio de herencia) y que dichos objetos compartan la misma superclase en una jerarquía de clases (es decir, que dichos objetos hereden de dicha superclase), con lo que en estos casos, se simplifica la programación.

Polimorfismo y sobrecarga

La sobrecarga es un tipo especial de polimorfismo, ya que la sobrecarga permite "muchas formas" de implementar un mismo método.

La gran diferencia es que el polimorfismo se produce en "tiempo de ejecución" (enlace dinámico) y la sobrecarga se produce en "tiempo de compilación".

POO: RECURSOS ÚTILES PARA EL TRABAJO CON OBJETOS

Descripción

Java dispone de otros recursos útiles para el trabajo con objetos.

Operador "instanceof"

El operador "instanceof" sirve para conocer si un objeto pertenece a una clase determinada, o a su superclase. El operador "instanceof" sólo puede usarse con variables que contengan la referencia a un objeto.

Ejemplo de superclase:

```
//SUPERCLASE
public class Vehiculo {

	private String modelo;
	private String color;

}
```

Ejemplo que comprueba si un objeto pertenece a una clase:

```
public class Coche extends Vehiculo {

	public static void main (String[] args) {

		//Se crea el objeto de tipo Coche (es esta misma clase)
		Coche micoche = new Coche();

		//Se comprueba que el objeto "micoche" pertenece
		if(micoche instanceof Coche) {
			System.out.println("micoche es un coche y a su vez, un vehículo.");
		}
	}

}
```

Ejemplo que comprueba si un objeto pertenece a una superclase:

```
public class Coche extends Vehiculo {

	public static void main (String[] args) {

		//Se crea el objeto de tipo Coche (es esta misma clase)
		Coche micoche = new Coche();

		//Se comprueba que el objeto "micoche" pertenece
		if(micoche instanceof Vehiculo) {
			System.out.println("micoche es un coche y a su vez, un vehículo.");
		}
	}

}
```

Métodos útiles de la clase Objetc: equals

El método "equals", perteneciente a la superclase "Object" sirve para comparar dos objetos, retornando "true" si dichos objetos son exactamente iguales. Por lo que el método "equals" es un método de tipo "boolean".

Exactamente iguales significa que dichos dos objetos tienen "el mismo estado", es decir, tienen los mismos datos en sus atributos en un momento dado.

Ejemplo que comprueba que dos objetos son iguales (que tienen el mismo estado, es decir, que sus atributos contienen los mismos valores):

Ejemplo de una clase:

```
public class Pieza {

	private int longitud;
	private int anchura;

	//Método constructor
	Pieza (int lo, int an) {
		this.longitud = lo;
		this.anchura = an;
	}

	//Método que devuelve la longitud
	int getLongitud() {
		return longitud;
	}

	//Método que devuelve la anchura
	int getAnchura() {
		return anchura;
	}

}
```

Ejemplo de comprobación de la igualdad de dos objetos de la clase "Pieza":

```
public class Inicio {

	public static void main(String[] args) {

		Pieza pieza1 = new Pieza(2, 5);
		Pieza pieza2 = new Pieza(3, 5);

		//SE MUESTRAN EL ESTADO (VALOR DE LOS ATRIBUTOS) DE LOS OBJETOS
		if(pieza1.equals(pieza2)) {
			System.out.println("LOS OBJETOS SON IGUALES");
			System.out.println("Longitud mipieza1: " + pieza1.getLongitud());
			System.out.println("Anchura mipieza1: " + pieza1.getAnchura());
			System.out.println("Longitud mipieza2: " + pieza2.getLongitud());
			System.out.println("Anchura mipieza2: " + pieza2.getAnchura());
		}
		else {
			System.out.println("LOS OBJETOS SON DIFERENTES.");
			System.out.println("SU ESTADO ES DIFERENTE.");
			System.out.println("Longitud mipieza1: " + pieza1.getLongitud());
			System.out.println("Anchura mipieza1: " + pieza1.getAnchura());
			System.out.println("Longitud mipieza2: " + pieza2.getLongitud());
			System.out.println("Anchura mipieza2: " + pieza2.getAnchura());
		}
```

```
        }

}
```

PAQUETES

Descripción

Los paquetes son grupos relacionados de clases e interfaces. Proporcionan un mecanismo organizado para el manejo de un gran conjunto de clases e interfaces. También, evitan los conflictos de nombres.

Ventajas de la utilización de paquetes

Si se desea que un conjunto de clases estén disponibles para otros programadores, lo adecuado es empaquetarlas en paquetes. De esta forma, otros programadores pueden determinar fácilmente para qué sirven esos paquetes de clases, cómo utilizarlos, etc.

Paquetes y conflictos de nombres

Los paquetes resuelven el problema del conflicto entre los nombres de las clases. Al crecer el número de clases, crece la probabilidad de designar con el mismo nombre a dos clases diferentes.

Creación de paquetes personalizados

Además de los paquetes existentes en Java, el programador puede crear sus propios paquetes y poner en ellos las clases e interfaces deseadas para organizarlas de forma adecuada.

Creación de paquetes

Para crear un paquete se utiliza la sentencia "package". Se escribirá en la parte superior del fichero .java.

Ejemplo que crea un paquete llamado "variedades":

```
package variedades;
```

La clase o clases definidas en el fichero que contiene dicha sentencia de creación del paquete, serán miembros de dicho paquete.

Ejemplo de fichero con varias clases, pertenecientes al paquete "variedades":

```
package variedades;

class Miclase1 {

}

class Miclase2 {

}

class Miclase3 {

}
```

Ejemplo de fichero con una sola clase, perteneciente al paquete "variedades" (es lo más habitual, ya que se recomienda que haya una sola clase por fichero .java):

```
package variedades;

public class Miclase1 {

}
```

Paquetes y carpetas

Cada paquete creado con todas sus clases, creará una carpeta en el disco que contendrá todos los archivos .java correspondientes a todas las clases del paquete. Dichas carpetas estarán en el disco, dentro de la carpeta del proyecto.

Importar paquetes de clases

Para importar otros paquetes se utiliza la sentencia import, seguida del paquete que se desea importar. Habrá que especificar el asterisco, lo que significa que se importarán todas las clases de dicho paquete.

Estando situados en la clase "Pieza", perteneciente al paquete "paquete", se importa todo el paquete de clases "variedades", es decir, todas las clases del paquete "variedades":

```
package paquete;

import variedades.*;

public class Pieza {

}
```

NOTA: en la clase "Pieza" ya se podrían utilizar todas las clases del paquete "variedades".

Importar clases

A veces solo es necesario una o algunas clases, por lo que no será necesario importar todo el paquete de clases. Java permite importar solo las clases deseadas. Esto es útil para una mejor organización, eficiencia, así como para no recargar la aplicación con clases cargadas que no se van a usar.

Estando situados en la clase "Pieza", perteneciente al paquete "paquete", se importa solo la clase "Miclase" del paquete "variedades":

```
package paquete;

import variedades.Miclase;

public class Pieza {

}
```

NOTA: en la clase "Pieza" ya se podría utilizar la clase "Miclase" (perteneciente al paquete "variedades").

Estando situados en la clase "Pieza", perteneciente al paquete "paquete", se importan las clases "Miclase1" y "Miclase2" del paquete "variedades":

```
package paquete;

import variedades.Miclase1;
import variedades.Miclase2;

public class Pieza {

}
```

NOTA: en la clase "Pieza" ya se podrían utilizar las clases "Miclase1" y "Miclase2" (pertenecientes al paquete "variedades").

Paquetes de Java de uso general mas comunes

Son los paquetes mas comunes que contiene Java, y que son de propósito general.

- java.awt: contiene las clases para crear interfaces de usuario, independientes de la plataforma.
- java.io: entrada/salida. Contiene las clases para controlar los flujos de datos.
- java.util: clases útiles para uso del programador.
- etc.

CLASES DE USO GENERAL: STRING

Descripción

En Java, una cadena es siempre un objeto de tipo "String", es decir, un objeto perteneciente a la clase "String". Dentro de una cadena (objeto "String") Java crea un array de caracteres (como en "C" o "C++").

Creación de cadenas de forma implícita

Crear una cadena de forma implícita e inicializarla:

```
String micadena = "Hola amigos";
```

Creación de cadenas de forma explícita

Crear una cadena de forma explícita e inicializarla:

```
String micadena = new String("Hola amigos");
```

Creación de cadenas nulas

Una cadena nula es la cadena que no tiene caracteres, pero es un objeto de la clase "String".

Declarar una cadena, pero no crea el objeto:

```
String micadena;
```

Después, se podría crear el objeto:

```
micadena = new String();
```

O crear el objeto e inicializarlo:

```
micadena = new String("Hola amigos");
```

Crear una cadena nula de forma implícita:

```
String micadena = "";
```

Crear una cadena nula de forma explícita:

```
String micadena = new String();
```

Longitud de una cadena

Para obtener la longitud de una cadena (número de caracteres) que almacena dicha cadena, se usa el método "length":

```
//Cadena
String micadena = "Hola amigos";

//Obtención del número de caracteres
int milongitud = micadena.length();
```

Comienzo y final de la cadena

Podemos conocer si una cadena comienza con un determinado prefijo, llamando al método "startsWith", que devuelve "true" o "false", según que el string comience o no por dicho prefijo:

```
String str = "La primera entrevista";
boolean resultado = str.startsWith("La");
```

En este ejemplo la variable resultado tomará el valor "true".

De modo similar, podemos saber si un string finaliza con un conjunto dado de caracteres, mediante la función miembro "endsWith":

```
String str = "El primer programa";
boolean resultado = str.endsWith("programa");
```

Posición de caracteres en la cadena

Si se quiere obtener la posición de la primera ocurrencia de la letra "p", se usa la función "indexOf":

```
String str = "El primer programa";
int pos = str.indexOf('p');
```

Para obtener las sucesivas posiciones de la letra "p", se llama a otra versión de la misma función:

```
pos = str.indexOf('p', pos+1);
```

El segundo argumento le dice a la función "indexOf" que empiece a buscar la primera ocurrencia de la letra "p" a partir de la posición "pos+1", es decir, a partir del siguiente carácter al encontrado.

Otra versión de "indexOf" busca la primera ocurrencia de un substring dentro del string.

```
String str = "El primer programa";
int pos = str.indexOf("pro");
```

Comparación de cadenas

La comparación de strings nos da la oportunidad de distinguir entre el operador de comparación "==" y el método "equals" de la clase String:

```
String str1 = "El lenguaje Java";
String str2 = new String("El lenguaje Java");

//Se evalua si son los mismos objetos
if(str1 == str2) {
	System.out.println("Los mismos objetos");
	}
else {
	System.out.println("Distintos objetos");
}

//Se evalua si los dos objetos tienen el mismo contenido
//(la misma secuencia de caracteres)
if(str1.equals(str2)) {
	System.out.println("El mismo contenido");
	}
else {
	System.out.println("Distinto contenido");
}
```

La salida será:

```
Distintos objetos
```

```
El mismo contenido
```

... ya que son distintos objetos (cada cadena es un objeto independiente ["str1" y "str2"], y tienen el mismo contenido (los dos objetos contienen los mismos caracteres ("El lenguaje Java").

Los dos objetos ocupan posiciones distintas en memoria, pero guardan los mismos datos.

Si se ahora se tiene lo siguiente:

```
String str1="El lenguaje Java";
String str2 = str1;
System.out.println("Son el mismo objeto: " + (str1==str2));
```

Devuelve:

```
Son el mimso objeto: true
```

Los objetos str1 y str2 guardan la misma referencia al objeto de la clase String creado. La expresión (str1 == str2) devolverá "true".

Por lo que el método "equals" compara dos objetos, en este caso Strings, y devuelve true cuando dos objetos (en este caso strings) son iguales, o false si son distintos.

```
String str = "El lenguaje Java";
boolean resultado = str.equals("El lenguaje Java");
```

La variable "resultado" tomará el valor "true".

Extracción de subcadenas de cadenas

En muchas ocasiones es necesario extraer una porción o substring de un string dado. Para este propósito hay una función miembro de la clase "String" denominada "substring".

Para extraer un substring desde una posición determinada hasta el final del string escribimos:

```
String str = "El lenguaje Java";
String subStr = str.substring(12);
```

Se obtendrá el substring: "Java".

Una segunda versión de la función miembro substring, nos permite extraer un substring especificando la posición de comienzo y la el final:

```
String str = "El lenguaje Java";
String subStr = str.substring(3, 11);
```

Se obtendrá el substring "lenguaje". Recuérdese, que las posiciones se empiezan a contar desde cero.

Conversión de cadenas

Para convertir un número en string:

```
int valor = 10;
String str = String.valueOf(valor);
```

La clase String proporciona versiones de "valueOf" para convertir los datos primitivos: int, long, float, double.

Para convertir un string en número entero (y se aprovecha para recortar espacios que pueda haber a ambos lados de la cadena:

```
String str = "  12 ";
int numero = Integer.parseInt(str.trim());
```

Otra forma de convertir string en número entero:

```
String str = "12";
int numero = Integer.valueOf(str).intValue();
```

CLASES DE USO GENERAL: MATH

Descripción

La clase "Math" contiene los métodos necesarios para realizar operaciones matemáticas.

Métodos de uso general de la clase Math

Una tabla resumen de algunas funciones de la clase "Math":

Funciones Matemáticas	Significado	Ejemplo de uso	Resultado
abs	Valor absoluto	int x = Math.abs(2.3);	x = 2;
atan	Arcotangente	double x = Math.atan(1);	x = 0.78539816339744;
sin	Seno	double x = Math.sin(0.5);	x = 0.4794255386042;
cos	Coseno	double x = Math.cos(0.5);	x = 0.87758256189037;
tan	Tangente	double x = Math.tan(0.5);	x = 0.54630248984379;
exp	Exponenciación neperiana	double x = Math.exp(1);	x = 2.71828182845904;
log	Logaritmo neperiano	double x = Math.log(2.7172);	x = 0.99960193833500;
pow	Potencia	double x = Math.pow(2,3);	x = 8.0;
round	Redondeo	double x = Math.round(2.5);	x = 3;
random	Número aleatorio	double x = Math.ramdom();	x = 0.20614522323378;

CLASES DE USO GENERAL: CALENDAR

Descripción

La clase "Calendar" contiene los métodos necesarios para realizar trabajo con fecha y hora.

Uso de la clase "Calendar"

```
//CREACIÓN DE UN OBJETO DE TIPO CALENDARIO.
//No se puede instanciar directamente la clase "Calendar".
Calendar micalendario = Calendar.getInstance();

//Se almacenan los datos en variables
//-----------------------------------

//Día del mes (1 - 31)
String dia = Integer.toString(micalendario.get(Calendar.DATE));

//Mes (0 - 11)
String mes = Integer.toString(micalendario.get(Calendar.MONTH));

//Año
String año = Integer.toString(micalendario.get(Calendar.YEAR));

//VECTOR PARA LOS MESES
String[] mismeses =
             {
             "Enero",
             "Febrero",
             "Marzo",
             "Abril",
             "Mayo",
             "Junio",
             "Julio",
             "Agosto",
             "Septiembre",
             "Octubre",
             "Noviembre",
             "Diciembre"
             };

//DECLARACIÓN DE VARIABLE PARA ALMACENAR TODA LA CADENA FINAL QUE SE MOSTRARÁ.
//"micalendario.get(Calendar.MONTH)" DEVUELVE EL NÚMERO DEL MES, QUE HARÁ DE
//ÍNDICE DE LA MATRIZ.
//NO SE UTILIZA LA VARIABLE mes PORQUE YA ESTÁ CONVERTIDA A String Y LA MATRIZ
//NOS PIDE UN int DE ÍNDICE.
String todo =
             "Hoy es " +
             dia +
             " de " +
             mismeses[micalendario.get(Calendar.MONTH)] +
             " de " +
             año;

//SE MUESTRAN LOS DATOS
System.out.print(todo);
```

COLECCIONES

Descripción

Una colección es una matriz dinámica que permite almacenar datos de mismo tipo o de diferente tipo.

En Java las colecciones se trabajan con la clase "ArrayList". Dicha clase permite método para crear y mantener colecciones.

Clase "ArrayList"

Para trabajar con colecciones será necesario importar la clase "ArrayList", que está en el paquete "java.util":

```
import java.util.ArrayList;
```

Crear un objeto "ArrayList"

Crear el objeto de tipo "ArrayList" vacío:

```
//CREACIÓN DEL OBJETO DE TIPO ARRAYLIST
ArrayList micoleccion = new ArrayList();
```

Añadir elementos a la colección

Se realizará mediante el método "add":

```
//AÑADIR ELEMENTOS A LA COLECCIÓN.
//PUEDEN SER DE CUALQUIER TIPO.
micoleccion.add("Luis");
micoleccion.add(46);
micoleccion.add(167);
micoleccion.add(34.67);
```

El método "add" añade el elemento al final.

El único objeto de origen es la cadena (String), los demás son tipos primitivos, pero la clase ArrayList convierte a todos los elementos en objetos.

Especificando la posición del elemento insertado:

```
//AÑADE UN ELEMENTO A LA COLECCIÓN Y LO COLOCA
//EN LA SEGUNDA POSICIÓN (0 ES LA PRIMERA POSICIÓN.
micoleccion.add(1, "Pablo");
```

Mostrar los elementos de la colección

Mostrar el segundo elemento del ArrayList:

```
System.out.println(micoleccion.get(1));
```

Mostrar todos los elementos del ArrayList en una línea:

```
//MOSTRAR TODOS LOS ELEMENTOS DEL ARRAYLIST EN UNA LÍNEA
System.out.println(micoleccion);
```

La salida será:

```
[Luis, 46, 167, 34.67]
```

Mostrar todos los elementos del ArrayList con un bucle for:

```
//MOSTRAR LOS ELEMENTOS DE LA COLECCIÓN
for(int i = 0; i<micoleccion.size(); i++){
    System.out.println(micoleccion.get(i));
}
```

Métodos útiles de la clase "ArrayList"

Método "size" para mostrar el número de elementos del ArrayList:

```
//MOSTRAR EL NÚMERO DE ELEMENTOS DEL ARRAYLIST
System.out.println(micoleccion.size());
```

Método "remove" para eliminar un elemento del ArrayList:

```
//BORRA EL PRIMER ELEMENTO DE LA COLECCIÓN
micoleccion.remove(0);
```

Método "set" para reemplazar un elemento del ArrayList:

```
//REEMPLAZA EL PRIMER ELEMENTO Y COLOCA EN
//SU LUGAR "Ana".
micoleccion.set(0, "Ana");
```

Método "clear" para eliminar todos los elementos del ArrayList (dejándolo vacío):

```
//ELIMINAR TODOS LOS ELEMENTOS DEL ARRAYLIST
micoleccion.clear();
```

Resumen de métodos de la clase "ArrayList"

Algunos métodos que proporciona ArrayList son:

MÉTODO	DESCRIPCIÓN
size()	Devuelve el número de elementos (int)
add(X)	Añade el objeto X al final. Devuelve true.
add(posición, X)	Inserta el objeto X en la posición indicada.
get(posicion)	Devuelve el elemento que está en la posición indicada.
remove(posicion)	Elimina el elemento que se encuentra en la posición indicada. Devuelve el elemento eliminado.
remove(X)	Elimina la primera ocurrencia del objeto X. Devuelve true si el elemento está en la lista.
clear()	Elimina todos los elementos.
set(posición, X)	Sustituye el elemento que se encuentra en la posición indicada por el objeto X. Devuelve el elemento sustituido.
contains(X)	Comprueba si la colección contiene al objeto X. Devuelve true o false.
indexOf(X)	Devuelve la posición del objeto X. Si no existe devuelve -1

EXCEPCIONES

Descripción

Una excepción es un objeto, que hereda de su superclase "Exception", que lanza Java cuando ocurre un problema no controlado en la ejecución de una aplicación.

Cuando se produce un error en tiempo de ejecución que no se controla, Java lanza una excepción.

Si dicha excepción no se controla adecuadamente, se detiene la ejecución normal del programa.

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.

Las excepciones son clases, que heredan de la clase genérica "Exception". Dicha clase "Exception" representa a una excepción genérica, es decir, que engloba a todos las demás tipos de excepciones.

Control de excepciones

Para controlar las posibles excepciones (errores en tiempo de ejecución de la aplicación) que se pueden producir, se utiliza un controlador de excepciones.

Controlador de excepciones

El controlador de excepciones se implementa mediante una estructura de control (try - catch - finally) que permite capturar una posible excepción que se pueda producir, y en caso de que se produzca, controlarla, ejecutando el código que se especifique.

La estructura de control de excepciones es: "try - catch - finally".

Por ejemplo, si se va a ejecutar un código que posiblemente produzca un error, ya que depende de algún dato que introduzca el usuario o de alguna acción inesperada, se controlará dicha posible excepción que se pueda producir y en su lugar, mostrar por ejemplo un mensaje al usuario de que los datos que ha introducido no son los adecuados.

Realmente, el controlador de excepciones funciona de forma parecida a una sentencia condicional, de tal manera que:

si se produce una excepción... se captura (controla) y... se realizan las acciones correspondientes.

Estructura de control de excepciones "try - catch - finally"

La estructura de control de excepciones "try - catch - finally" es la siguiente:

```
try {
        ...
        CÓDIGO SENSIBLE A PROVOCAR UNA EXCEPCIÓN
        ...
} catch (Excepcion miexception) {
        . . .
        CÓDIGO QUE SE EJECUTA EN CASO DE QUE SE PRODUZCA UNA EXCEPCIÓN.
        EN CASO DE QUE SE PRODUZCA DICHA EXCEPCIÓN, SE PUEDE ACCEDER A
        INFORMACIÓN RELATIVA A DICHA EXCEPCIÓN, MEDIANTE EL OBJETO DE
        TIPO Exception QUE RECIBE ESTA CLAÚSULA catch Y QUE LE ENVÍA
        LA PROPIA EXCEPCIÓN.
        SI NO SE PRODUCE ALGUNA EXCEPCIÓN, ESTE BLOQUE catch NO SE EJECUTA.
        ...
} finally {
```

```
        ...
        CÓDIGO QUE SE EJECUTA SIEMPRE, SE PRODUZCA O NO UNA EXCEPCIÓN
        ...
}
```

Bloque "try"

En el bloque "try" se implementa el código sensible a provocar alguna excepción. Por ejemplo, conexión con una base de datos, apertura de un fichero, etc.

Bloque "catch"

El bloque "catch", en caso de que se produzca una excepción, captura dicha excepción.

En el bloque "catch" se implementa el código que se ejecutará en el caso de que se produzca alguna excepción. Normalmente es una posible solución alternativa o un mensaje de aviso al usuario.

Si no se produce una excepción, este bloque "catch" no se ejecuta.

Ejemplo de uso de los bloques "try - catch" para controlar la posible excepción que se producirá al insertar un elemento en una colección, si se especifica que la posición donde se inserte dicho elemento esté fuera de una posición posible en dicha colección:

```
//CREACIÓN DEL OBJETO DE TIPO ARRAYLIST
ArrayList micoleccion = new ArrayList();

//SE AÑADEN 3 ELEMENTOS A LA COLECCIÓN
micoleccion.add("Luis");
micoleccion.add(46);
micoleccion.add(34.67);

//MEDIANTE UN CONTROLADOR DE EXCEPCIONES SE CONTROLA
//QUE EN CASO DE QUE SE INTENTE INSERTAR UN ELEMENTO
//EN LA MATRIZ FUERA DE LOS LÍMITES.
//(EN ESTE CASO, LA MATRIZ TIENE 3 ELEMENTOS [0,1,2]
//Y SE INTENTA INSERTAR UN ELEMENTO NUEVO EN UNA
//POSICIÓN QUE NO EXISTE).
//DE ESTA FORMA, NO SE DETIENE LA APLICACIÓN Y ADEMÁS
//REALIZAMOS ALGUNA ACCIÓN DE CONTROL, COMO EN ESTE
//CASO, QUE SE MUESTRA UN MENSAJE AL USUARIO.
try {
      micoleccion.add(6, "hola");
} catch (Exception miexception) {
      System.out.println("No se ha podido insertar el elemento.");
}
```

Bloque "finally"

El bloque "finally" es opcional. Implementa el código que se ejecutará siempre, en cualquier caso de que se produzca una excepción o en caso de que no se produzca.

Normalmente se utiliza para cerrar archivos, cerrar bases de datos, inicializaciones de atributos, etc.

Como este bloque "finally" es opcional, si no es necesario, no se pone.

Ejemplo que implementa el bloque "finally" para mostrar la colección de elementos, si se produce la excepción o si no se produce dicha excepción:

```
try {
      micoleccion.add(1, "hola");
```

```
} catch (Exception miexception) {
	System.out.println("NO SE HA PODIDO INSERTAR EL ELEMENTO.");
} finally {
	System.out.println("La colección actualmente, es la siguiente: ");
	System.out.println(micoleccion);
}
```

Obtener información de la excepción

El objeto de tipo exception que es la lanzado cuando se produce algún error en el programa, dispone de unos métodos que se pueden consultar, para obtener información acerca de la propia excepción producida.

Ejemplo que utiliza algunos métodos útiles de la clase "Exception" para obtener información acerca de la excepción:

```
//Mediante los métodos de la excepción, se muestra información
//general acerca de la excepción producida.
try {
	micoleccion.add(7, "hola");
} catch (Exception miexcepcion) {
	System.out.println("NO SE HA PODIDO INSERTAR EL ELEMENTO.");
	System.out.println("-------------------------------------");
	System.out.println("getMessage: " + miexcepcion.getMessage());
	System.out.println("toString: " + miexcepcion.toString());
	System.out.println("getClass: " + miexcepcion.getClass());
}
```

La salida es la siguiente:

```
NO SE HA PODIDO INSERTAR EL ELEMENTO.
-------------------------------------
getMessage: Index: 7, Size: 3
toString: java.lang.IndexOutOfBoundsException: Index: 7, Size: 3
getClass: class java.lang.IndexOutOfBoundsException
```

ARCHIVOS

Descripción

Java permite el trabajo con los archivos, tanto para leerlos como para escribir en ellos.

Lectura de un archivo

Ejemplo que accede a un archivo, lee su contenido y lo muestra:

```
import java.io.*;

class Archivos {

	//CARGA Y LECTURA DEL ARCHIVO
	//---------------------------

	//Variable que acumulará el texto que extraigamos del archivo
	String mitexto = "";

	// variable que guardará la línea actual leída,
	// ya que el archivo se lee línea a línea
	String milinea = "";


	void leerArchivo() {

	try {
		// FileReader es la clase que permite trabajar con un archivo.
		// Especificamos la ruta y nombre del archivo
		// si lo tengo en otra ruta, lo especificaré:
		// ("c:\\bbb\\miarchivo.txt")
		// lo haré con doble barra para que java no lo
		// confunda con una secuencia de escape.
		// con la barra que le añadimos, le decimos
		// que lo siguiente lo vea como un carácter literal
		// y no como una secuencia de escape.
		FileReader milector = new FileReader("c:\\aaa\\miarchivo.txt");

		// BufferedReader es la clase que se encargará de la lectura
		// del archivo. Se le pasa el objeto de tipo FileReader.
		BufferedReader entrada = new BufferedReader(milector);

		// El método readLine() lee línea a línea mientras
		// haya líneas que leer en el archivo, por medio de
		// entrada (que es el BufferedReader).
		// Cada línea se acumula en la variable milinea,
		// y se acumulará en la variable mitexto en cada vuelta.
		// De esta forma al ir acumulándose línea a línea
		// en la variable mitexto, se irá completando
		// todo el contenido del archivo, es decir, todas sus líneas.
		// (Es decir, al final, en la variable mitexto
		// se almacenará todo el contenido del archivo).
		// En cada vuelta, entrada.readline() lee una línea
		// y la guarda en la variable milinea.
		// (mientras suceda esto, significará que habrá líneas para leer
		// en el archivo, es decir, mientras haya líneas que leer en el
		// archivo, la variable milinea será: != null (es decir, no vacía,
		// hay líneas).
		// Cuando ya no haya líneas que leer en el archivo, porque ya se
```

```
            // hayan leído todas, la variable milinea será: = null, y el bucle
            // while dejará terminará la lectura del archivo.
            // Resumiendo, el while lee el archivo línea a línea hasta el final,
            // es decir, mientras tenga líneas que leer.
            while ( (milinea = entrada.readLine()) != null ) {
                  // en la variable mitexto se va acumulando todo el
                  // contenido del archivo. Esto se realiza porque
                  // la variable milinea, que lee línea a línea el
                  // archivo, en cada vuelta, le pasa la línea a
                  // la variable texto. En dicha variable mitexto,
                  // se va formando línea a línea toda la cadena
                  // que corresponde al archivo completo.
                  mitexto += milinea;
                  }

                  // Se cierra el archivo.
                  // Se cierra el BufferedReader entrada,
                  // una vez que ya hemos leído todo el archivo.
                  entrada.close();
      }
      catch(Exception miexcepcion) {
            System.out.println("Error en la carga del archivo");
            System.out.println(miexcepcion.getMessage());
      }

      // Se muestra el contenido del archivo
      System.out.println(mitexto);
      }
}
```

Escritura en un archivo

Ejemplo de método que accede y escribe en un archivo de texto:

```
void escribirArchivo() {

      try
      {
      // BufferedWriter se encargará de la escritura del archivo.
      // Le pasamos el archivo (su ruta) creando un FileWriter
      //(que será el escritor).
      //true -> añadimos texto al final del ya existente.
      //false -> reemplazamos el texto existente por el nuevo que grabemos.
      //(EN CUALQUIER CASO, SIEMPRE CREA EL ARCHIVO, PERO LANZA UNA EXCEPCIÓN).
BufferedWriter miescribir = new BufferedWriter(new FileWriter("c:\\aaa\\miarchivo.txt",
true));

      // Creamos un PrintWriter que se encargará de escribir (grabar) el archivo.
      // Le pasamos el BufferedWriter (el BufferedWriter miescribir ya apunta
      //al archivo especificado).
      PrintWriter salida = new PrintWriter(miescribir);

      // Con el objeto PrintWriter (con su método print) escribimos en el archivo.
      // A dicho método Print, le pasamos como parámetro lo que queremos escribir
      // en dicho archivo.
      salida.print("NUEVO TEXTO.");

      // Se cierra el PrintWriter entrada, una vez que ya hemos grabado (escrito)
      // en el archivo.
      salida.close();
```

```
        }
catch (Exception miexcepcion)
        {
        System.out.println("Error en la escritura del archivo");
        System.out.println(miexcepcion.getMessage());
        }

}
```

BASES DE DATOS

Descripción

Java permite el trabajo con bases de datos: Oracle, Access, etc.

Paquete necesario para el trabajo con bases de datos

Para trabajar con bases de datos y con el lenguaje de consultas SQL, será necesario importar el paquete "java.sql", mediante la siguiente sentencia "import":

```
import java.sql.*;
```

Declaración de variables generales

Las siguiente variables serán necesarias para el trabajo con la base de datos y serán declaradas en la parte superior de la clase:

```
// DECLARACIÓN DE LA CONEXIÓN CON LA BASE DE DATOS.
// Se utiliza un objeto de tipo "Connection".
Connection miconexion;

// DECLARACIÓN DEL RESULSET.
// Un ResulSet almacena el conjunto de registros que devuelve una consulta SQL.
ResultSet miresulset;

// Este ResulSet se utilizará para obtener el número
// de registros que devuelve la consulta
ResultSet miresulsetnumeroregistros;

// DECLARACIÓN DE STATEMENT.
// Statement es el objeto que se encarga de ejecutar una consulta SQL.
// Aquí por comodidad y por diferenciarlos en su objetivo, se crean
// dos objetos Statement. uno para las consultas de tipo "SELECT"
// y otro para las consultas de modificación (INSERT, DELETE, UPDATE).

// Declaración de Statement para consultas "SELECT".
Statement mistatementseleccion;

// Declaración de Statement para consultas de modificación.
// (INSERT, DELETE, UPDATE).
Statement mistatementmodificacion;

// Variable que almacena la cadena de consulta SQL
String miconsulta;
```

Carga del driver de la base de datos

Ejemplo que carga el driver para acceder a una base de datos de Access:

```
try {
	// CARGA DEL DRIVER DE LA BASE DE DATOS DE ACCESS
	Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
	System.out.println("Driver cargado correctamente.");
}
catch(Exception ex) {
	System.out.println("No se pudo cargar el Driver");
}
```

Creación del DSN para conectar con la base de datos.

El DSN es el mecanismo para poder conectar con una base de datos de tipo ODBC, por ejemplo de Access.

Para crear un DSN con una base de datos de Access, se seguirá el siguiente proceso:

1. Inicio - Panel de control - Sistema y seguridad - Herramientas administrativas - Orígenes de datos ODBC.
2. Pulsar en la pestaña "DSN de sistema".
3. Pulsar el botón "Agregar".
4. Seleccionar la opción "Microsoft AccessDriver (*.mdb)".
5. Pulsar el botón "Finalizar".
6. En la opción "Nombre del origen de datos", escribir el nombre deseado.
    - Por ejemplo: miempresa.
7. Pulsar el botón "Seleccionar".
8. Seleccionar la base de datos de Access deseada.
9. Pulsar el botón "Aceptar".
10. Pulsar el botón "Aceptar".

Conexión con la base de datos

Ejemplo que realiza la conexión con la base de datos:

```
try {
	// Variable que almacena la cadena DSN que permitirá la conexión
	// con la base de datos.
	// En esta cadena DSN, ponemos el nombre del DSN "miempresa"
	// creado anteriormente.
	String midsn = "jdbc:odbc:miempresa";

	// Cadena de conexión: le pasamos el DSN
	miconexion = DriverManager.getConnection(midsn);

	// Informamos de que la conexión se ha abierto correctamente
	System.out.println("Conexión creada correctamente.");

	// miconexion es la conexion definida con la base de datos, que nos permite
	// conectarnos con la base de datos, para poder acceder a sus datos.
	// Por tanto, a partir de esa conexión (miconexion) crearemos
	// los objetos Statement.
	// El primer Statement será para las consultas SELECT que no
	// modifican datos. El segundo Statement será para las consultas
	// que modifican datos (INSERT, UPDATE, DELETE).
	// El Statement es el objeto que nos permite ejecutar consultas.
	// Aquí hacemos la asignación a los dos objetos Statement
	// que ya declaramos en la zona superior.
	// Utilizamos el método createStatement del objeto miconexion.
	mistatementseleccion =
miconexion.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
	mistatementmodificacion =
miconexion.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
}
catch (Exception ex) {
	System.out.println("No se pudo establecer la conexión");
}
```

BASES DE DATOS: LENGUAJE SQL

Descripción

El lenguaje SQL es un lenguaje que permite realizar consultas en bases de datos.

SQL incrustado en el código

El lenguaje SQL se puede utilizar directamente en consolas de bases de datos, pero lo más común es incrustarlo en códigos de programación, con el fin de realizar consultas en bases de datos.

Ejemplos de sentencias SQL

Ejemplo que muestra todos los campos y todos los registros de la tabla clientes:

```
SELECT *
FROM clientes;
```

Ejemplo que muestra los campos nombre y edad y todos los registros de la tabla clientes:

```
SELECT nombre, edad
FROM clientes;
```

Ejemplo que inserta un registro nuevo, especificando los datos correspondientes a los campos de la tabla clientes: NOMBRE, APELLIDO y EDAD:

```
INSERT INTO clientes(NOMBRE, APELLIDO, EDAD) VALUES('Julio', 'Ros', 33);
```

Ejemplo que elimina el registro con ID igual a 51:

```
DELETE FROM clientes
WHERE ID = 51;
```

Ejemplo que actualiza la edad del registro con ID igual a 32 y establece dicha edad en 31::

```
UPDATE clientes
SET edad = 31
WHERE ID = 32;
```

BASES DE DATOS: ACCESO A DATOS

Descripción

Una vez conectados a la base de datos, se puede acceder a sus datos. Para acceder a los datos se utiliza la claúsula SELECT del lenguaje SQL.

Cargar los registros de la consulta

Ejemplo que carga los registros la consulta:

```
try {
	// Se almacena en "miresulset" (que es el objeto Resulset) el conjunto
	// de registros que devuelve la sentencia SQL resultantes de ejecutar
	// el método executeQuery del objeto mistatementseleccion de tipo Statement,
	// pasándole como parámetro la consulta que tiene que ejecutar
	// OJO: el método executeQuery se utiliza para consultas de selección.
	// Dicho método executeQuery pertenece al Statement
	// (en este caso "mistatementseleccion").
	miresulset = mistatementseleccion.executeQuery("SELECT * FROM clientes");

	// Nos colocamos en el primer registro de la tabla
	miresulset.next();
} catch(Exception ex) {
	System.out.println("Error al cargar los registros en la tabla");
}
```

NOTA: ahora los registros están cargados en el Resulset, pero todavía no se han mostrado.

Mostrar los datos de un registro de la consulta

Ejemplo que muestra los datos del primer registro de la consulta:

```
try {
	// Nos asegurarnos que nos encontramos en un registro válido.
	// Si mostramos el anterior del primero o mostramos el posterior del último,
	// es decir, un registro que no existe, nos podría dar error.
	if(miresulset.isBeforeFirst() || miresulset.isAfterLast()) {
		System.out.println("REGISTRO NO VÁLIDO");
	}

	// Si mostramos un registro existente (es decir, válido)
	// (desde el primero hasta el último).
	else {
		// Mostramos en las cajas los datos de cada campo correspondiente.
		// mirecord acumula el conjunto de registros.
		// Con getString() recuperamos el dato del campo indicado.
		// Le podemos poner en getString el nombre del campo entre comillas
		// o pasarle el número de orden del campo (El primer campo es el 1)
		System.out.println("" + miresulset.getString("NOMBRE"));
		System.out.println("" + miresulset.getString("APELLIDO"));
		System.out.println("" + miresulset.getString("EDAD"));
	}
}
catch(Exception ex) {
	System.out.println("Error al mostrar el registro.");
}
```

Mostrar los datos de todos los registros de la consulta

Ejemplo que muestra los datos del todos los registros de la consulta:

```
try {
	//Variables para recoger los datos
	String minombre = "";
	String miapellido = "";
	String miedad = "";

	//Imprimir las cabeceras de los campos.
	System.out.println("NOMBRE" + "\t" + "APELLIDOS" + "\t" + "EDAD");

	//Bucle que recorre todos los registros.
	//Se muestran los datos de cada registro.
	while(miresulset.next()) {
		minombre = miresulset.getString("NOMBRE");
		miapellido = miresulset.getString("APELLIDO");
		miedad = miresulset.getString("EDAD");
		//Se muestran los datos del registro actual
		System.out.println(minombre + "\t" + miapellido + "\t" + miedad);
	}
}
catch(Exception ex) {
	System.out.println("Error al mostrar los registros");
}
```

BASES DE DATOS: MODIFICACIÓN DE DATOS

Descripción

Una vez conectados a la base de datos, se pueden modificar los datos de la base de datos. Mediante las sentencias de SQL: INSERT, DELETE y UPDATE se pueden insertar, eliminar y actualizar los datos.

Inserción de registros

Ejemplo que inserta un registro en la tabla clientes:

```
try {
	//Creación de la sentencia SQL de inserción
	String laconsulta = "INSERT INTO clientes(NOMBRE, APELLIDO, EDAD) ";
	laconsulta += "VALUES('Manuel', 'Mir', 28)";

	// Ejecutamos la sql insert anterior.
	// OJO: para consultas de modificación, utilizaremos
	// el método executeUpdate, que es un método del objeto
	// de tipo Statement (que ejecuta la consulta).
	mistatementmodificacion.executeUpdate(laconsulta);
```

Después de realizar la inserción del registro, mostramos todos los datos de la tabla clientes:

```
	// Una vez insertado el registro, Ahora vamos a mostrar los registros.
	// Para ello...
	// Creamos la sql que muestre todos los datos de la tabla clientes
	miconsulta = "SELECT * FROM CLIENTES";

	// Ejecutamos la consulta.
	// OJO: para consultas de selección, utilizaremos el método
	// executeQuery (este método pertenece al objeto Statement).
	miresulset = mistatementseleccion.executeQuery(miconsulta);

	// Nos colocamos en el primer registro
	miresulset.next();

	//Variables para recoger los datos
	String minombre = "";
	String miapellido = "";
	String miedad = "";

	// Mostramos los registros.
	//Imprimir las cabeceras de los campos.
	System.out.println("NOMBRE" + "\t" + "APELLIDOS" + "\t" + "EDAD");

	//Bucle que recorre todos los registros.
	//Se muestran los datos de cada registro.
	while(miresulset.next()) {
		minombre = miresulset.getString("NOMBRE");
		miapellido = miresulset.getString("APELLIDO");
		miedad = miresulset.getString("EDAD");
		//Se muestran los datos del registro actual
		System.out.println(minombre + "\t" + miapellido + "\t" + miedad);
	}
}
catch(SQLException ex) {
	System.out.println("Error al insertar el registro.");
}
```

Actualización de registros

Ejemplo que actualiza un registro en la tabla clientes:

```
try {
	//Creación de la sentencia SQL de actualización
	String laconsulta = "UPDATE clientes SET edad = 99 WHERE ID = 32";

	// Ejecutamos la sql.
	// OJO: para consultas de modificación, utilizaremos
	// el método executeUpdate, que es un método del objeto
	// de tipo Statement (que ejecuta la consulta).
	mistatementmodificacion.executeUpdate(laconsulta);
```

Ahora, mostramos los registros:

```
	// Una vez actualizado el registro, Ahora vamos a mostrar los registros.
	// Para ello...
	// Creamos la sql que muestre todos los datos de la tabla clientes
	miconsulta = "SELECT * FROM CLIENTES";

	// Ejecutamos la consulta.
	// OJO: para consultas de selección, utilizaremos el método
	// executeQuery (este método pertenece al objeto Statement).
	miresulset = mistatementseleccion.executeQuery(miconsulta);

	// Nos colocamos en el primer registro
	miresulset.next();

	//Variables para recoger los datos
	String minombre = "";
	String miapellido = "";
	String miedad = "";

	// Mostramos los registros.
	//Imprimir las cabeceras de los campos.
	System.out.println("NOMBRE" + "\t" + "APELLIDOS" + "\t" + "EDAD");

	//Bucle que recorre todos los registros.
	//Se muestran los datos de cada registro.
	while(miresulset.next()) {
		minombre = miresulset.getString("NOMBRE");
		miapellido = miresulset.getString("APELLIDO");
		miedad = miresulset.getString("EDAD");
		//Se muestran los datos del registro actual
		System.out.println(minombre + "\t" + miapellido + "\t" + miedad);
	}
	System.out.println("Registro actualizado correctamente.");
}
catch(SQLException ex) {
	System.out.println("Error al actualizar el registro.");
}
```

Eliminación de registros

Ejemplo que elimina un registro en la tabla clientes:

```
try {
	//Creación de la sentencia SQL de inserción
```

```
      String laconsulta = "DELETE FROM clientes WHERE ID = 30";

      // Ejecutamos la sql.
      // OJO: para consultas de modificación, utilizaremos
      // el método executeUpdate, que es un método del objeto
      // de tipo Statement (que ejecuta la consulta).
      mistatementmodificacion.executeUpdate(laconsulta);
```

Ahora, mostramos los registros:

```
      // Una vez eliminado el registro, Ahora vamos a mostrar los registros.
      // Para ello...
      // Creamos la sql que muestre todos los datos de la tabla clientes
      miconsulta = "SELECT * FROM CLIENTES";

      // Ejecutamos la consulta.
      // OJO: para consultas de selección, utilizaremos el método
      // executeQuery (este método pertenece al objeto Statement).
      miresulset = mistatementseleccion.executeQuery(miconsulta);

      // Nos colocamos en el primer registro
      miresulset.next();

      //Variables para recoger los datos
      String minombre = "";
      String miapellido = "";
      String miedad = "";

      // Mostramos los registros.
      //Imprimir las cabeceras de los campos.
      System.out.println("NOMBRE" + "\t" + "APELLIDOS" + "\t" + "EDAD");

      //Bucle que recorre todos los registros.
      //Se muestran los datos de cada registro.
      while(miresulset.next()) {
            minombre = miresulset.getString("NOMBRE");
            miapellido = miresulset.getString("APELLIDO");
            miedad = miresulset.getString("EDAD");
            //Se muestran los datos del registro actual
            System.out.println(minombre + "\t" + miapellido + "\t" + miedad);
      }
      System.out.println("Registro eliminado correctamente.");
}
catch(SQLException ex) {
      System.out.println("Error al eliminar el registro.");
}
```

MULTITAREA

Descripción

La Máquina Virtual Java (JVM) es un sistema multi-thread (multi-hilo), es decir, es capaz de ejecutar varias secuencias de ejecución (programas) simultáneamente.

La JVM gestiona todos los detalles, asignación de tiempos de ejecución, prioridades, etc, de forma similar a como gestiona un Sistema Operativo múltiples procesos.

La diferencia básica entre un proceso de Sistema Operativo y un Thread Java, es que los Threads Java corren dentro de la JVM, (que la JVM es un proceso del Sistema Operativo) y por tanto comparten todos los recursos, incluida la memoria y las variables y objetos allí definidos. A este tipo de procesos donde se comparte los recursos se les llama a veces "procesos ligeros" (lightweight process).

Por lo que:

- Aplicación Java no multi-thread: proceso del sistema operativo, que internamente ejecuta un solo thread (hilo).
- Aplicación Java multi-thread: proceso del sistema operativo, que internamente ejecuta mas de un thread, que cada uno de ellos utiliza una tarea.

Concurrencia

La computación concurrente es la simultaneidad en la ejecución de múltiples tareas. Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa.

En caso de una aplicación concurrente en Java, sería una aplicación que contiene un conjunto de procesos (hilos de ejecución). Cada hilo de ejecución ejecuta alguna o algunas acciones en concreto.

La máquina virtual asigna tiempos a cada thread.

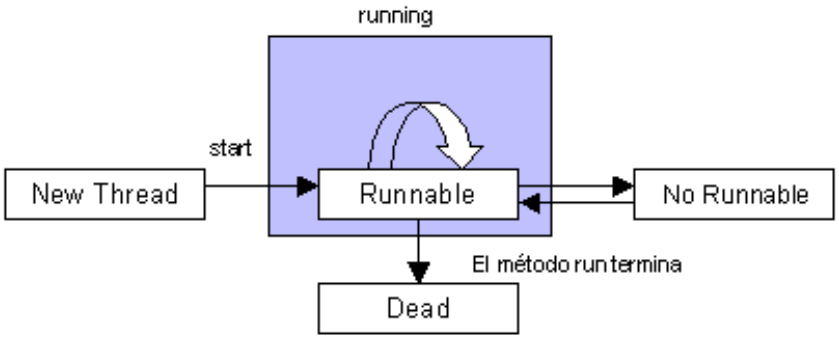
Threads (Hilos de ejecución)

Java da soporte al concepto de Thread desde el mismo lenguaje. Desde el punto de vista de las aplicaciones, los threads son útiles porque permiten que el flujo del programa sea divido en dos o más partes, cada una (cada thread) ocupándose de alguna tarea.

Por ejemplo un Thread puede encargarse de la comunicación con el usuario, mientras otros actuan en segundo plano, realizando la transmisión de un fichero, accediendo a recursos del sistema (cargar sonidos, leer ficheros ...), etc.

De hecho todos los programas con interface gráfico (AWT o Swing) son multithread de forma implícita, porque los eventos y las rutinas de dibujado de las ventanas corren en un thread distinto al principal.

Ciclo de vida de un thread



1. NEW THREAD: cuando la subclase hereda de la clase Thread se crea un nuevo Thread que está en en su estado inicial.
    - En este estado es simplemente un objeto más. No existe todavía el thread en ejecución. El único método que puede invocarse sobre él es el método "start".
2. START: cuando se invoca el método start sobre el thread, el sistema crea los recursos necesarios, planifica al thread, asignándole prioridad e invoca al método "run".
    - En este momento el thread está corriendo (estado "runnable").
3. RUNNABLE: estando el thread en plena ejecución, es decir, en estado "runnable", si dicho thread ejecuta internamente el método "sleep" o el método "wait", o dicho thread tiene que esperar por alguna operación de entrada/salida, entonces, dicho thread pasa al estado "no runnable" (no corriendo), hasta que finalice dicha condición o estado de espera.
    - Durante el estado "no runnable" del thread, el sistema puede ceder el control a otros thread activos.
4. DEAD: por último, cuando el método "run" finaliza el thread termina y pasa a la situación 'Dead' (muerto, finalizado).

Paquetes y clases para el trabajo con threads

Java da soporte al concepto de Thread desde el mismo lenguaje, con algunas clases e interfaces definidas en el package "java.lang" y con métodos específicos para la manipulación de Threads en la clase Object.

La clase "Thread" está en el package "java.lang", por lo que no es necesaria ninguna importación, ya que dicho paquete "java.lang" se importa de forma implícita.

La clase "Thread"

La forma más directa para hacer un programa multi-thread es extender la clase "Thread", es decir, heredar de la clase "Thread", y redefinir el método run() de dicha clase "Thread".

La subclase que herede de la superclase "Thread" será un thread, es decir, un hilo de ejecución.

El método "run"

La clase "Thread" implementa un método llamado "run". Después de heredar de la clase "Thread", se debe redefinir dicho método "run".

El método "run" contendrá todo lo que se desea que realice el thread (hilo de ejecución). Es decir, el método "run" contendrá todo el código del thread (hilo de ejecución).

El método "start"

Para ejecutar un thread (hilo de ejecución) es necesario invocar el método "run", y para invocar dicho método "run" hay que ejecutar el método "start" de la clase "Thread". Una vez hecho esto, el thread empezará a correr, ejecutándose todo el código que esté dentro del método "run".

Es decir, cuando se ejecuta el método "start" de la clase "Thread", se invoca automáticamente el método "run" del thread, y se ejecutará. inicia el thread (mediante una llamada al método start() de la clase thread). El thread se inicia con la llamada al método run y termina cuando termina éste.

El método "start()" inicia el nuevo thread y llama al método "run()" de dicho thread.

Asincronía

La ejecución de los threads es asíncrona, es decir, se realiza la llamada al método "start" (por ejemplo, desde el método "main"), comenzando la ejecución del thread, y devolviendo la ejecución a dicho método "main". De esta manera cada thread que se ejecuta funciona de forma independiente.

- Síncrono: envío de un flujo continuo de datos. El emisor no espera a que el receptor reciba dichos datos, es decir, los manda continuamente.

- Asíncrono: el emisor envía unos datos y espera a que el receptor los reciba, antes de enviar los siguientes o antes de que dicho emisor continue con la ejecución de sus tareas.

Ejemplo completo

Ejemplo de clase que es un thread:

```
public class Hilo1 extends Thread {
	public void run() {
		//BUCLE QUE MUESTRA 10 VECES UN MENSAJE
		for(int i=1; i<=10; i++) {
			System.out.println("HILO 1. Vez: " + i);

			//CONTROLADOR DE EXCEPCIONES.
			//(sleep OBLIGA A PONERLO).
			try {
				//CON sleep HACEMOS UNA PAUSA EN EL HILO.
				//(1000 milisegundos -> 1 SEGUNDO)
				sleep(500);
			}
			catch (Exception e) {
			}
		}
	}
}
```

Ejemplo de otra clase que es otro thread:

```
public class Hilo2 extends Thread {
	public void run() {
		//BUCLE QUE MUESTRA 10 VECES UN MENSAJE
		for(int i=1; i<=10; i++) {
			System.out.println("HILO 2. Vez: " + i);

			//CONTROLADOR DE EXCEPCIONES.
			//(sleep OBLIGA A PONERLO).
			try {
				//CON sleep HACEMOS UNA PAUSA EN EL HILO.
				//(1000 milisegundos -> 1 SEGUNDO)
				sleep(500);
			}
			catch (Exception e) {
			}
		}
	}
}
```

Ejemplo de clase main, que ejecuta los dos threads anteriores:

```
public class Inicio {
	public static void main(String args[]) {
		// DECLARACIÓN DE LOS OBJETOS PARA LAS DOS CLASES HILO.
		// (CADA UNA DE DICHAS CLASES ES UN HILO INDEPENDIENTE).
		Hilo1 mihilo1;
		Hilo2 mihilo2;

		// CREACIÓN DE LOS DOS OBJETOS DE TIPO THREAD (HILO)
		mihilo1 = new Hilo1();
		mihilo2 = new Hilo2();
```

```
            // INICIO DE LOS HILOS, LLAMANDO AL MÉTODO start() DE CADA UNO DE ELLOS.
            // ÉSTO HARÁ QUE SE EJECUTE EL MÉTODO run() DE CADA HILO,
            // EJECUTÁNDOSE EN CADA HILO, EL CÓDIGO QUE HAYA DENTRO DE DICHO
            // MÉTODO run() EN CADA UNO DE DICHOS HILOS.
            mihilo1.start();
            mihilo2.start();
            }
}
```

INTERFAZ GRÁFICA: INTRODUCCIÓN

Descripción

El paquete Swing contiene las clases necesarias para permitir la creación de interfaces gráficas de usuario (Graphics User Interfaces [GUIs]).

Java Foundation Classes (JFC)

Las JFC "Java Foundation Classes" (en castellano "Clases Base Java") es un framework (marco de trabajo) para construir interfaces gráficas de usuario portables, basadas en Java.

JFC se compone de lo siguiente:

- AWT (Abstract Window Toolkit).
- Swing.
- Java 2D.

Juntas, suministran una interfaz de usuario consistente para programas Java, tanto si el sistema de interfaz gráfica final es para Windows, Mac OS X o Linux.

AWT

Abstract Windows Toolkit. AWT es la más antigua de las APIs de interfaz gráfica, y fue criticada duramente por ser poco más que una envoltura alrededor de las capacidades gráficas nativas de la plataforma anfitrión. Esto significa que los elementos gráficos de AWT están bastante unidos a la plataforma en la que se desarrollen, por lo que su portabilidad es mas inflexible.

Por ejemplo, si se desarrolla en el sistema operativo Windows, una aplicación gráfica Java con AWT y posteriormente se ejecuta dicha aplicación en otra máquina con Windows, perfecto, ya que se verá exactamente igual y correrá más rápido. Pero si se ejecuta en una máquina con Linux, la aplicación no tendrá la misma apariencia y no correrá tan rápido, ya que dicha aplicación no está diseñada 100% para cualquier tipo de sistema operativo, es decir, no es totalmente portable.

AWT también es muy limitada en tipos de elementos gráficos (en comparación con Swing).

Swing

Swing es la más moderna de las APIs de interfaz gráfica. Es la más utilizada actualmente, por su flexibilidad, portabilidad, riqueza de elementos y posibilidades inmensas para el desarrollador.

Por ejemplo, si se desarrolla en el sistema operativo Windows, una aplicación gráfica Java con Swing y posteriormente se ejecuta dicha aplicación en otra máquina con Windows, dicha aplicación se verá exactamente igual. Si se ejecuta en una máquina con Linux, dicha aplicación Swing adaptará la visualización de los controles gráficos a dicho entorno de Linux, por ejemplo, la forma de los botones, la apariencia de las listas desplegables, etc., ya que Swing trata de emular la apariencia del sistema operativo en el que se ejecuta.

Por lo que una aplicación desarrollada con Swing, es totalmente portable.

INTERFAZ GRÁFICA: SWING

Descripción

Swing permite crear aplicaciones Java con entorno gráfico totalmente portables.

Estructura básica de una aplicación Swing

Para crear una aplicación básica en Swing, debe existir, al menos, un contenedor de alto nivel (Top-Level Container), que provee el soporte que las componentes Swing necesitan para el pintado y el manejo de eventos. Es contenedor es por lo general, una ventana.

Paquetes necesarios para trabajar con Swing

Los componentes Swing pertenecen al paquete "javax.swing".

Por lo que en la parte superior de la clase que desee utilizar Swing, se deberá realizar la importación de dicho paquete.

Ejemplo que importa todas las clases del paquete "javax.swing":

```
import javax.swing.*;
```

Pero como Swing se apoya en AWT, principalmente en el manejo de los eventos, los paquetes que habitualmente se importan en una aplicación Swing, son los siguientes:

```
import java.awt.*; //Paquete de AWT
import java.awt.event.*; //Paquete para el trabajo con eventos
import javax.swing.*; //Paquete de Swing
```

Nombres de las clases correspondientes a los controles de Swing

El nombre de las clases de los controles de Swing, comienza con "J", seguido del nombre de dicho control. Por ejemplo, el nombre de la clase correspondiente a una etiqueta en Swing, es "JLabel".

INTERFAZ GRÁFICA: SWING: PROCESO DE CREACIÓN DE CONTROL

Descripción

Para realizar el despliegue completo de un control en Swing, será necesario seguir un proceso ordenado, desde su creación hasta su ajuste de las propiedades.

Proceso de creación de un control en Swing

Para crear un control en Swing, se recomienda el siguiente proceso y en este orden:

1. Declaración y creación del control.
2. Establecer ubicación y tamaño del control.
3. Si se desea, establecer otras propiedades del control.
4. Añadir el control a la ventana "JFrame".

Declaración y creación del control

Para declarar el control, se creará el objeto correspondiente, utilizando la clase de dicho control:

```
ClaseControl miobjeto = new ClaseControl();
```

También se podría declarar e inicializar posteriormente:

```
// Declaración
ClaseControl miobjeto;

// Asignación
miobjeto = new ClaseControl();
```

Ubicación y tamaño del control

Mediante el método "setBounds" del control, se especificarán los parámetros correspondientes a la ubicación y al tamaño del control:

```
miobjeto.setBounds(X, Y, Anchura, Altura);
```

Los parámetros son los siguientes:

- X: coordenada X de la esquina superior izquierda del control, con respecto a su contenedor, es decir, a su ventana.
- Y: coordenada Y de la esquina superior izquierda del control, con respecto a su contenedor, es decir, a su ventana.
- Anchura: anchura del control en puntos de pantalla.
- Altura: altura del control en puntos de pantalla.

Añadir el control a la ventana

Una vez creado y configurado el control, el último paso será añadirlo a la ventana. En este paso es cuando realmente se muestra el control en dicha ventana.

Ejemplo que añade el control "mietiqueta" a la ventana "miventana":

```
miventana.getContentPane().add(mietiqueta);
```

INTERFAZ GRÁFICA: SWING: VENTANAS

Descripción

Las ventanas son el control contenedor principal en una aplicación Swing. Las ventanas alojan a otros controles: etiquetas, cajas de texto, botones, menús desplegables, etc.

Para crear una ventana se utiliza la clase "JFrame".

Ventana: JFrame

Ejemplo que crea una ventana llamada "miventana":

```
// VENTANA
JFrame miventana = new JFrame();
```

Establecer uso de coordenadas absolutas para la ubicación de controles

De esta manera se podrán ubicar libremente los controles en la ventana, mediante coordenadas:

```
// ESTABLECER EL USO DE COORDENADAS ABSOLUTAS, PARA
// LA LIBRE UBICACIÓN DE LOS CONTROLES EN LA VENTANA.
miventana.getContentPane().setLayout(null);
```

Ubicación y tamaño de la ventana

Ejemplo que ubica y da tamaño a la ventana:

```
// UBICACIÓN Y TAMAÑO DE LA VENTANA.
miventana.setBounds(50, 50, 600, 400);
```

Propiedades generales de la ventana

```
// TÍTULO DE LA VENTANA
miventana.setTitle("Título de la ventana");

// QUE LA VENTANA NO SEA REDIMENSIONABLE
miventana.setResizable(false);
```

Mostrar la ventana

Una vez que están configuradas todas las carácterísticas de la ventana, es el momento de mostrarla en la aplicación:

```
// SE MUESTRA DICHA VENTANA.
miventana.setVisible(true);
```

INTERFAZ GRÁFICA: SWING: CONTROLES

Descripción

Los controles van alojados dentro de una ventana "JFrame". Los controles permiten interactuar con los datos de muy diversas formas, ya que hay multitud de controles: etiquetas, cajas de texto, botones, listas desplegables, etc.

El nombre de todas las clases, correspondientes a todos los controles de Swing, comienzan con "J".

Etiqueta: JLabel

Permite crear etiquetas de texto fijo. Este texto se puede modificar posteriormente en tiempo de ejecución. Para añadir el texto se utiliza el método "setText".

Ejemplo que crea una etiqueta de texto:

```
// CREACIÓN DE LA ETIQUETA
JLabel mietiqueta = new JLabel();

// UBICACIÓN Y TAMAÑO DE LA ETIQUETA.
mietiqueta.setBounds(20, 20, 200, 25);

// TEXTO DE LA ETIQUETA
mietiqueta.setText("Mi texto de la etiqueta");

// AÑADIR LA ETIQUETA A LA VENTANA
miventana.getContentPane().add(mietiqueta);
```

Cuadro de texto: JTextField

Permite crear cuadros de texto donde el usuario puede escribir. Este texto se puede modificar posteriormente en tiempo de ejecución, mediante. Para añadir el texto se utiliza el método "setText".

Ejemplo que crea un cuadro de texto:

```
// CREACIÓN DEL CUADRO DE TEXTO
JTextField micuadrotexto = new JTextField();

// UBICACIÓN Y TAMAÑO
micuadrotexto.setBounds(20, 50, 200, 25);

// AÑADIR CONTROL A LA VENTANA
miventana.getContentPane().add(micuadrotexto);
```

Botón: JButton

Permite crear botones. Mediante el método setText se puede modificar el texto del botón. Posteriormente, habrá que asignar acciones para el evento de pulsar el botón.

Ejemplo que crea un botón:

```
// CREAR EL BOTÓN
JButton miboton = new JButton();

// UBICACIÓN Y TAMAÑO
miboton.setBounds(20, 100, 150, 25);

// TEXTO DEL BOTÓN
```

```
miboton.setText("Texto del botón");

// AÑADIR CONTROL A LA VENTANA
miventana.getContentPane().add(miboton);
```

Lista desplegable: JComboBox

Permite crear listas desplegables.

Ejemplo que crea una lista desplegable:

```
// CREA LA LISTA DESPLEGABLE (COMBOBOX)
JComboBox micombobox = new JComboBox();

// UBICACIÓN Y TAMAÑO
micombobox.setBounds(20, 200, 150, 25);

// AÑADIR ELEMENTOS
micombobox.addItem("Uno");
micombobox.addItem("Dos");
micombobox.addItem("Tres");

// AÑADIR CONTROL A LA VENTANA
miventana.getContentPane().add(micombobox);
```

Área de texto con scroll: JTextArea / JScrollPane

Permite crear áreas de texto, para escritir textos extensos, cargar contenidos de archivos, etc..

Ejemplo que crea un área de texto:

```
// CREAR EL ÁREA DE TEXTO
JTextArea miareatexto= new JTextArea();

// CARACTERÍSTICAS DE LA FUENTE
miareatexto.setFont(new Font("Arial",Font.PLAIN,16));

// COLOR DE FONDO
miareatexto.setBackground(new Color(255,255,140));

// COLOR DE TEXTO
miareatexto.setForeground(Color.BLACK);

// REBOTE DE LÍNEAS EN MARGEN DERECHO
miareatexto.setLineWrap(true);

// QUE EN EL REBOTE DEL MARGEN DERECHO, NO SE PARTAN LAS PALABRAS
miareatexto.setWrapStyleWord(true);

// EL TEXTO SE PUEDE EDITAR
miareatexto.setEditable(true);

// CREACIÓN DEL SCROLL.
// SE LE PASA EL TEXTAREA (areatexto).
// SE ESPECIFICA QUE SIEMPRE MUESTRA SOLO LA BARRA VERTICAL.
miscroll = new JScrollPane(miareatexto,
             JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
             JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

// SE UBICA Y DA TAMAÑO AL SCROLL (QUE CONTIENE EL TEXTAREA)
```

```
miscroll.setBounds(20, 300, 200, 70);

// ESCRIBIR EN EL TEXTAREA
miareatexto.setText("Estoy en el JTextArea");

// SE AÑADE EL CONTROL A LA VENTANA
miventana.getContentPane().add(miscroll);
```

Botones de radio y su grupo: JRadioButton

Permite crear botones de radio y su grupo lógico, para que solo pueda haber una opción seleccionada.

Ejemplo que crea un grupo de botones de radio:

```
// DECLARAR LOS BOTONES DE RADIO
JRadioButton miradio1, miradio2, miradio3;

//CREACIÓN DE LOS BOTONES DE RADIO
miradio1 = new JRadioButton("Botón radio 1");
miradio2 = new JRadioButton("Botón radio 2");
miradio3 = new JRadioButton("Botón radio 3");

// GRUPO DE LOS BOTONES DE RADIO.
// ESTE GRUPO NO ES UN GRUPO FÍSICO QUE SE PUEDA VISUALZAR.
// (ES UN GRUPO LÓGICO).
// SE AÑADE CADA BOTÓN DE RADIO AL GRUPO DE BOTONES DE RADIO.
// DE ESTA FORMA, FUNCIONAN COMO UN GRUPO AUTOEXCLUYENTE.
//ESTE GRUPO ES UN GRUPO "LÓGICO" PARA QUE AL
//PULSAR EN UNO, SE EXCLUYA EL RESTO.
ButtonGroup migrupo = new ButtonGroup();
migrupo.add(miradio1);
migrupo.add(miradio2);
migrupo.add(miradio3);

// UBICACIÓN Y TAMAÑO
miradio1.setBounds(300, 40, 150, 25);
miradio2.setBounds(300, 70, 150, 25);
miradio3.setBounds(300, 100, 150, 25);

// SE AÑADEN LOS BOTONES DE RADIO A LA VENTANA.
// OJO: NO SE AÑADE EL GRUPO, SINO TODOS LOS BOTONES DE RADIO.
miventana.getContentPane().add(miradio1);
miventana.getContentPane().add(miradio2);
miventana.getContentPane().add(miradio3);
```

Casilla de verificación: JCheckBox

Permite crear casillas de verificación, que tienen estado de activado (true) y desactivado (false). Puede haber varias seleccionadas, ya que no tienen grupo.

Ejemplo que crea un conjunto de casillas de verificación:

```
// DECLARACIÓN DE VARIOS CHECKBOX
JCheckBox micheckbox1, micheckbox2, micheckbox3;

//CREACIÓN DE LAS CASILLAS DE VERIFICACIÓN
micheckbox1 = new JCheckBox("Inglés");
micheckbox2 = new JCheckBox("Francés");
micheckbox3 = new JCheckBox("Alemán");
```

```
// UBICACIÓN Y TAMAÑO
micheckbox1.setBounds(300, 150, 150, 25);
micheckbox2.setBounds(300, 180, 150, 25);
micheckbox3.setBounds(300, 210, 150, 25);

// SE AÑADEN LOS CONTROLES A LA VENTANA.
miventana.getContentPane().add(micheckbox1);
miventana.getContentPane().add(micheckbox2);
miventana.getContentPane().add(micheckbox3);
```

Tabla: JTable

Ejemplo que crea una tabla con JTable:

```
import javax.swing.table.DefaultTableModel; //PAQUETE PARA EL TABLEMODEL

// DECLARACIÓN DEL CONTROL JTABLE
JTable mijtable;
// DECLARACIÓN DEL SCROLL PARA EL CONTROL JTABLE
JScrollPane miscroll;

//--------------------
// DATOS PARA LA TABLA
//--------------------
// NOMBRES DE LAS COLUMNAS
String[] miscampos = {"Nombre", "Edad", "Sueldo"};
// DATOS DE LA TABLA
String[][] misdatos =
{
            {"Juan", "26", "1100"},
            {"Eva", "31", "1300"},
            {"Luis", "24", "1000"},
            {"Ana", "48", "1200"},
            {"Mario", "33", "1000"},
            {"Sara", "22", "1000"},
            {"Pedro", "54", "1400"},
            {"Mar", "44", "1300"}
      };

//---------------------
// CREACIÓN DE LA TABLA
//---------------------
// CREACIÓN DEL TABLEMODEL, QUE PERMITIRÁ LA GESTIÓN DEL JTABLE
DefaultTableModel mitablemodel = new DefaultTableModel(misdatos, miscampos);

// CREACIÓN DE LA TABLA
mijtable = new JTable(mitablemodel);

// CREACIÓN DEL SCROLL. SE LE PASA EL JTABLE.
// SE ESPECIFICA QUE SIEMPRE MUESTRA SOLO LA BARRA VERTICAL.
miscroll = new JScrollPane(mijtable,
            JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

// SE UBICA Y DA TAMAÑO AL SCROLL (QUE CONTIENE EL JTABLE)
miscroll.setBounds(20, 20, 250, 100);

// AÑADIR CONTROL A LA VENTANA
miventana.getContentPane().add(miscroll);
```

```
//====================
// EDICIÓN DE LA TABLA
//====================
// AGREGAR FILA NUEVA
Object[] newRow = {"Jose", "51", "2000"};
mitablemodel.addRow(newRow);

// MODIFICAR CELDA DE LA SEGUNDA FILA, TERCERA COLUMNA.
// (LA PRIMERA COLUMNA Y LA PRIMERA COLUMNA SON LA 0).
mitablemodel.setValueAt("ZZZ", 1, 2);
```

Menú: JMenuBar, JMenu, JmenuItem

Permite crear una barra de menú, menús y las opciones de los menús.

Ejemplo que crea un menú:

```
// CREAR LA BARRA DE MENÚ
mibarramenu = new JMenuBar();

// COLOCAR LA BARRA DE MENÚ EN LA VENTANA
miventana.setJMenuBar(mibarramenu);

// CREAR EL MENÚ
mimenu = new JMenu("Texto menú");

// AÑADIR EL MENÚ A LA BARRA DE MENÚ
mibarramenu.add(mimenu);

// OPCIONES DEL MENÚ
miopcion1 = new JMenuItem("Texto de opción 1"); //CREAR LA OPCIÓN
mimenu.add(miopcion1); // AÑADIR LA OPCIÓN AL MENÚ

miopcion2 = new JMenuItem("Texto de opción 2");
mimenu.add(miopcion2);

mimenu.addSeparator(); // SEPARADOR

miopcion3 = new JMenuItem("Texto de opción 3");
mimenu.add(miopcion3);
```

INTERFAZ GRÁFICA: SWING: EVENTOS

Descripción

Los eventos son las diferentes situaciones, provocadas por lo general por el usuario, en las que se puede ejecutar una acción.

Ejemplo de eventos: pulsar un botón, situar el ratón dentro de algún control, pulsar una tecla, etc.

Cada vez que el usuario interactúa con la aplicación, por ejemplo, pulsando un botón, se dispara (se produce) un evento.

Para que un control determinado reaccione a dicho evento, dicho control debe poseer un escuchador (listener), con al menos, un método determinado que se ejecutará al escuchar dicho evento en particular.

Paquete necesario para trabajar con eventos

Swing hereda todo el manejo de eventos de AWT, con lo que el trabajo con eventos en Swing requiere importar el paquete de eventos perteneciente a AWT.

Por lo que si en una aplicación realizada en Swing se desea funcionar con eventos (que es lo habitual), se deberá realizar la importación de dicho paquete correspondiente a AWT.

Las clases para el trabajo con eventos pertenecen al paquete "java.awt.event".

Ejemplo que importa todas las clases del paquete " java.awt.event":

```
import java.awt.event.*;
```

Tipos de eventos en Swing

Swing puede generar un variado conjunto de eventos. En la siguiente tabla se resumen los más comunes con sus respectivos "escuchadores" ("listeners").

EJEMPLOS DE EVENTOS Y SUS ESCUCHADORES	
Acción que genera un evento	**Tipo de escuchador**
El usuario hace un click en un control, presiona Return en un área de texto, o selecciona un menú	ActionListener (es el más utilizado)
Todos los eventos de ratón	MouseListener
Eventos del teclado	KeyListener
Eventos del foco	FocusListener

Configurar el evento "clic" de un control

El evento click, sirve para todos los controles.

Ejemplo de programación de un escuchador (listener) para controlar eventos de tipo "ActinoListener" para el botón "miboton1". Éste escuchador posee un método que implementa el código de lo que sucederá cuando se pulse en dicho botón:

```
// ESCUCHADOR PARA EL EVENTO CLICK DEL BOTÓN miboton1.
// SE AÑADE UN ESCUCHADOR (LISTENER) DE TIPO DE EVENTOS ActionListener
// AL CONTROL miboton1.
// CON EL MÉTODO SE ESPECIFICA LO QUE SUCEDE CUANDO
// SE HAGA CLIC EN EL BOTÓN.
miboton1.addActionListener(new ActionListener()
      {
```

```
	public void actionPerformed(ActionEvent e) {
	micuadrotexto.setText("PULSADO miboton1");
	}
});
```

Eventos de ratón

Ejemplo que configura los eventos de ratón:

```
// ESCUCHADOR PARA LOS EVENTOS DEL RATÓN DE miboton1.
// IMPLEMENTA LA INTERFAZ MouseListener, CON LO QUE OBLIGA
// A IMPLEMENTAR TODOS LOS MÉTODOS.
// SI NO SE DESEA ALGUNO, SE PUEDE COMENTAR SU CÓDIGO.
miboton1.addMouseListener(new MouseListener() {
	public void mousePressed(MouseEvent e) {
		micuadrotexto.setText("Mientras presionas el botón");
	}
	public void mouseReleased(MouseEvent e) {
	micuadrotexto.setText("Sueltas el botón");
	}
	public void mouseEntered(MouseEvent e) {
		micuadrotexto.setText("Te metes en el área del botón");
	}
	public void mouseExited(MouseEvent e) {
		micuadrotexto.setText("Te sales del área del botón");
	}
	public void mouseClicked(MouseEvent e) {
		micuadrotexto.setText("Haces click en el botón");
	}
});
```

Eventos de teclado

Ejemplo que configura los eventos de teclado:

```
// ESCUCHADOR PARA LOS EVENTOS DEL TECLADO DE miboton1.
// SE PRODUCIRÁN CUANDO EL FOCO ESTÉ SITUADO EN miboton1
// Y SE PULSE LA TECLA ESPECIFICADA EN EL if.
// IMPLEMENTA LA INTERFAZ KeyListener, CON LO QUE OBLIGA
// A IMPLEMENTAR TODOS LOS MÉTODOS.
// SI NO SE DESEA ALGUNO, SE PUEDE COMENTAR SU CÓDIGO.
miboton1.addKeyListener(new KeyListener() {
	public void keyPressed(KeyEvent e) { // PULSAR LA TECLA
		char mikeyChar = e.getKeyChar();
		if (mikeyChar == 'a') { // SI EL CARÁCTER ES LA a...
			micuadrotexto.setText("HAS PULSADO LA TECLA a");
		}
	}
	public void keyReleased(KeyEvent e) { // SOLTAR LA TECLA
		int mikeyCode = e.getKeyCode();
		if (mikeyCode == 65) { // SI EL CÓDIGO ES 66 (tecla "b")...
			micuadrotexto.setText("HAS SOLTADO LA TECLA a");
		}
	}
	public void keyTyped(KeyEvent e) { // MANTENER LA TECLA PRESIONADA
		char mikeyChar = e.getKeyChar();
		if (mikeyChar == 'a') {
			micuadrotexto.setText("ESTÁS PULSANDO LA TECLA a");
		}
	}
```

```
});
```

Eventos del foco

Ejemplo que configura los eventos de foco:

```
// ESCUCHADOR PARA LOS EVENTOS DEL DEL FOCO DEL CONTROL miboton1.
// IMPLEMENTA LA INTERFAZ FocusListener, CON LO QUE OBLIGA
// A IMPLEMENTAR TODOS LOS MÉTODOS.
// SI NO SE DESEA ALGUNO, SE PUEDE COMENTAR SU CÓDIGO.
miboton1.addFocusListener(new FocusListener() {
	public void focusGained(FocusEvent e) {
		miventana.setTitle("Cuando se pone el foco en el boton");
	}
	public void focusLost(FocusEvent e) {
		miventana.setTitle("Cuando se quita el foco en el botón");
	}
});
```

INTERFAZ GRÁFICA: SWING: VENTANAS DE DIÁLOGO

Descripción

Las ventanas de diálogo son ventanas que se lanzan cuando se produzca la acción deseada, y que informan al usuario o permiten que tome alguna decisión.

Ventana de mensaje: showMessageDialog

Permite mostrar una ventana, con un mensaje en su interior y el botón "Aceptar". Sirve para informar al usuario. Recibe dos parámetros: el nombre del objeto que representa a la ventana contenedora (por ejemplo, el JFrame) y el mensaje que mostrará la ventana.

Ejemplo que muestra una ventana de tipo "showMessageDialog":

```
JOptionPane.showMessageDialog(miventana, "PROCESO REALIZADO CORRECTAMENTE.");
```

Ventana de confirmación: showConfirmDialog

Permite mostrar una ventana, con un mensaje en su interior y los botones: "Sí", "No" y "Cancelar". Se puede controlar qué botón ha pulsado el usuario para realizar la acción o acciones desadas.

Ejemplo que muestra una ventana de tipo "showConfirmDialog", con el código en su escuchador, que controla la pulsación de los tres botones:

```
// SE RECOGE EN LA VARIABLE EL BOTÓN PULSADO POR EL USUARIO
int miconfirmacion = JOptionPane.showConfirmDialog(miventana, "¿Deseas salir?");

// SE EJECUTAN ACCIONES, DEPENDIENDO DE DICHO BOTÓN PULSADO
if (JOptionPane.OK_OPTION == miconfirmacion) {
System.exit(0);
}
if (JOptionPane.NO_OPTION == miconfirmacion) {
JOptionPane.showMessageDialog(miventana, "PUES VALE.");
      }
if (JOptionPane.CANCEL_OPTION == miconfirmacion) {
JOptionPane.showMessageDialog(miventana, "SIGUES EN LA APLICACIÓN.");
}
```

Ventana de requerimiento: showInputDialog

Permite mostrar una ventana, con un mensaje en su interior y una caja de texto que permite recoger un dato y almacenarlo en una variable, para después utilizarlo.

Ejemplo que muestra una ventana de tipo "showInputDialog", que pide el nombre al usuario:

```
// VENTANA QUE RECOGE EL NOMBRE AL USUARIO Y LO ALMACENA EN LA VARIABLE
String minombre = JOptionPane.showInputDialog(miventana, "ESCRIBE TU NOMBRE:");

// SE MUESTRA EL DATO RECOGIDO EN UN CUADRO DE TEXTO
micuadrotexto.setText(minombre);
```