



UNIDAD 2

PRIMEROS PASOS EN JAVA.

1. VERSIONES DE JAVA.
2. INSTALAR EL ENTORNO DE DESARROLLO.
3. PRIMEROS PROGRAMAS EN JAVA.
 - 3.1. Cosas A Tener En Cuenta.
 - 3.2. Analizar un Programa Sencillo.
 - 3.2. Compilar y Ejecutar Desde la Línea de Comandos.
 - 3.3. Herramienta JavaDoc.
 - 3.4. Paquetes y Sentencia import.
4. ELEMENTOS DE UN PROGRAMA.
 - 4.1. Identificadores.
 - 4.2. Datos: Variables y constantes.
 - 4.3. Tipos de Datos: primitivos y objetos.
 - 4.4. Conversión de Tipos (casting).
 - 4.5. Operadores y precedencia.
5. MÁS ELEMENTOS: CLASES Y OBJETOS DE FÁBRICA.
 - 5.1. Algunas clases y objetos de uso Habitual.
 - 5.2. Introducción a las enumeraciones.
6. ENTRADA Y SALIDA DE DATOS.
7. SENTENCIAS DE DECISIÓN.
 - 7.1. Decisión Simple (if-else).
 - 7.2. Decisión Múltiple (switch).
 - 7.3. La Sentencia Vacía.
8. EJERCICIOS.

BIBLIOGRAFÍA:

- Java, Cómo Programar (10ª Ed.). Pearson. Paul Deitel (2016).
- Introduction to Programming Using Java (7ªEd). David J. Eck (2014)
- Documentación oficial de Java, Oracle.





2.1. VERSIONES DE JAVA.

Recuerda que para "crear" programas en Java necesitamos los programas que realizan el precompilado y para ejecutarlos necesitamos los programas que realizan la interpretación de ese código.

Hay entornos (**frameworks**) que permiten la creación de los bytecodes y que incluyen herramientas con capacidad de ejecutar aplicaciones de todo tipo. El más popular (además es gratuito y de libre uso hasta la versión 11) es el **Java Developer Kit (JDK)** (antes de **Sun**, ahora de **Oracle**), que se encuentra disponible en la dirección (<https://www.oracle.com/technetwork/java/javase/overview/index.html>).

Se le llama **SDK** (Software Developer Kit) y al descargarlo de Internet hay que elegir la plataforma deseada (SE, EE o ME, ...).

Para poder crear los bytecodes de un programa Java, hace falta el SDK. **Sin embargo este kit va evolucionando: se actualiza el lenguaje Java, el compilador y las herramientas de apoyo como las máquinas virtuales, etc.** De ahí que se hable de Java 1.1, Java 1.2, etc. Los nombres de los distintos SDK y del lenguaje correspondiente.

Las primeras versiones de Java han cumplido más de 20 años. Hemos pasado de la versión 1.0 a la versión 1.16. **¿Qué es lo que se ha ido añadiendo al lenguaje?** Vamos a echar un vistazo a algunas de las versiones y sus novedades (no todas).

- **Versiones de Java (1.0 - 1.2):** Java aparece en 1995 como un nuevo lenguaje de programación con soporte multiplataforma desarrollado por James Gosling y Sun Microsystems.
 - **Versión JDK 1.0:** La primera versión del lenguaje contiene las clases principales, la maquina virtual y el API gráfico AWT.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- **Versión JDK 1.1:** Aparece en 1997 e incorpora al lenguaje varias clases que faltaban como **Readers/Writers**, **Calendars** y **Bundles**. Pero sin lugar a dudas su mayor aportación es la inclusión del estándar **JavaBeans** (componentes independientes reutilizables) y el **API JDBC** para conexión a bases de datos. Además podemos mencionar **RMI** (llamadas a métodos remotos, una técnica de comunicación de procesos a través de la red), **internacionalización** (para crear programas adaptables a todos los idiomas).
- **Versión 1.2:** En 1998 aparece otra evolución importante con la llegada del framework **Collections** y el API **Swing** que permite desarrollar interfaces de ventanas más complejas. **JFC** (Java Foundation classes) es el conjunto de clases de todo tipo para crear programas más atractivos. Dentro de este conjunto están: El paquete **Swing** mencionando arriba, **Enterprise Java beans** (componentes para aplicaciones distribuidas del lado del servidor), **Java 2D** (gráficos de alta calidad), **Java 3D** (Gráficos tridimensionales), **Java Media** (conjunto de paquetes para trabajar con datos multimedia) que incluye **Java Speech** (reconocimiento de voz), **Java Sound** (audio de alta calidad), **Java TV** (Televisión interactiva). Otras novedades son **JNDI** (Java Naming and Directory Interface, un servicio general de búsqueda de recursos. Integra los servicios de búsqueda más populares como LDAP por ejemplo), **Servlets** (para aplicaciones web), **Java Cryptography** (Algoritmos para encriptar, desencriptar, etc.), **Java Help** (creación de sistemas de ayuda, **Jini** (programación de electrodomésticos), **Java card** (programar pequeños



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

dispositivos electrónicos), **Java IDL** (Lenguaje de definición de interfaz, para crear aplicaciones tipo **CORBA** para sistemas distribuidos).

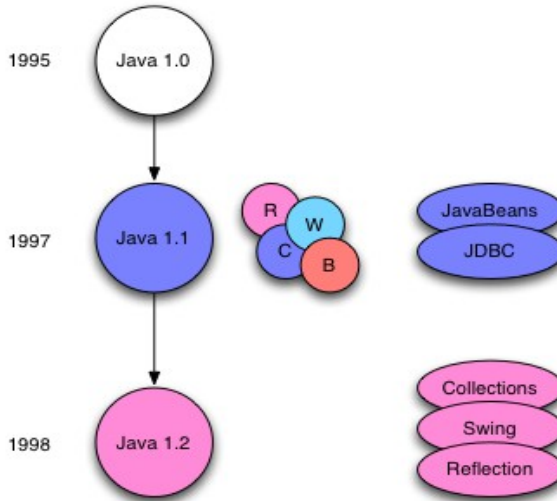


Figura 2.1: Aportaciones de JDK 1.0 a 1.2.

- **Versiones de Java (1.3 - 1.5):** Java en estos momentos ya es una plataforma madura a la cual el fabricante va añadiendo nuevas características.
 - **Versión JDK 1.3:** Avances pequeños en cuanto a APIs y el lenguaje, se añade soporte JNDI a RMI (para que trabaje con CORBA). Sin embargo el avance en cuanto a la arquitectura de la maquina virtual es importante ya que aparece la máquina HotSpot (más rápida y segura) con compilación **JIT (Just-in Time)**. En cuanto a herramientas, aparece **JPDA** (Java Platform Debugger Architecture).



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- **Versión JDK 1.4:** Se produce un salto importante en cuanto a nuevas APIs. Se incorpora **JAXP** (soporte de XML), **Expresiones Regulares**, **Criptografía**, aserciones (**assert**) y **NIO** (nuevo interfaz de entrada y salida de datos).
- **Versión JDK 1.5:** Java 5 incorpora dos saltos importantes a nivel del core del lenguaje. Por una parte la inclusión de **tipos Genéricos** que se echaban en falta en el mundo de las colecciones. Por el otro lado la inclusión del **concepto de metadatos con el uso de anotaciones**. Además se amplía el soporte de APIs orientadas a programación concurrente (threading), **Autoboxing** (conversión automática de tipos a tipos envoltura), **Enumeraciones**, argumentos variables (varargs) y mejora del bucle **for**.

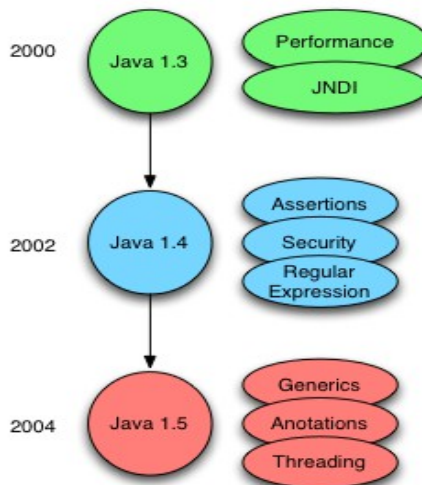


Figura 2.2: Aportaciones de JDK 1.3 a 1.5.

- **Versiones de Java (1.6 - 1.8):** Java ha madurado mucho y es



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

una de las plataformas de referencia para desarrollo.

- **Versión JDK 1.6:** pocos cambios, como un API de compilación "on-the-fly" (gestionar servicios web), combinación con otros lenguajes (PHP, Ruby, Perl,...) y nuevas especificaciones **JAX-WS 2.0**, **JAXB 2.0**, **STAX** y **JAXP** para servicios web.
- **Versión JDK 1.7:** cambios mínimos en el lenguaje y mejora de la máquina virtual como nuevos recolectores de basura.
- **Versión JDK 1.8:** Java 8 es un gran salto en el lenguaje. Se abren las puertas a la **programación funcional** con el uso de **expresiones Lambda** y **Streams**. Se realiza una revisión de APIS y se actualiza la gestión de fechas.

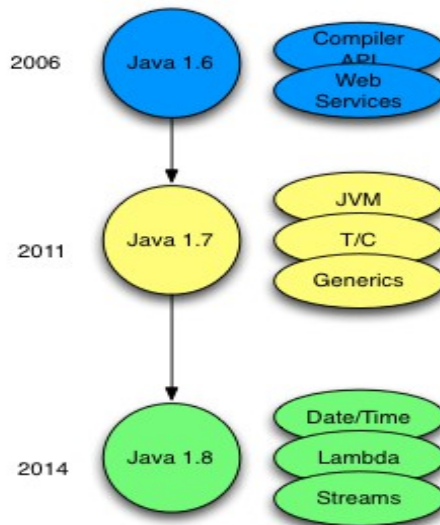


Figura 2.3: Aportaciones de JDK 1.6 a 1.8.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

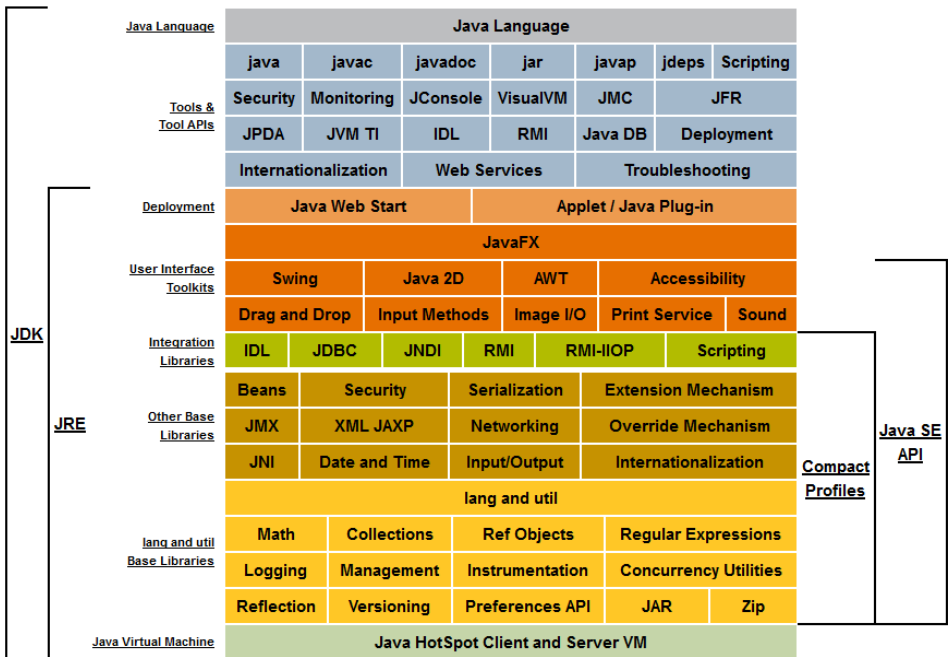


Figura 2.4: Componentes de JDK 1.8 (Java 8).

- **Versiones de Java (1.9 – 1.12).** Las versiones más recientes. En el momento de redactar estos apuntes, la última versión es la 16 (https://en.wikipedia.org/wiki/Java_version_history).
 - **Versión Java 9:** añade mayor modularidad y permite la creación de nuevas arquitecturas. Define un nuevo componente llamado **module** (una colección de datos y código con nombre), una nueva fase opcional en el proceso de compilación (el linkado), un nuevo tipo de fichero jar modular, formato JMOD que es un JAR que puede incluir código nativo y ficheros de configuración.
 - **Versión JDK 10.** Sin cambios importantes en el lenguaje.
 - **Versión SDK 11:** El JDK 11 inicia una **nueva era en la**



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

licencia de uso. Hasta ahora podías descargar y programar con el JDK de Java oficial de Oracle y luego poner tu aplicación en producción o distribuirla [sin tener que pagar nada](#). Sin embargo, a partir de Java 11, aunque puedes seguir desarrollando con él, **tendrás que [pagar una licencia a Oracle](#) si quieres utilizarlo para poner las aplicaciones en producción**. El coste es de 2,5 dólares al mes por cada usuario de escritorio, y de 25 dólares por procesador en el caso de aplicaciones de servidor. Aquí tienes [la lista de precios oficial](#). Esto no afecta a versiones anteriores del JDK (Java 8, 9 o 10).

Desde Java 9, Oracle se ha comprometido a sacar una nueva versión "mayor" de Java (y del JDK) cada 6 meses, con dos actualizaciones garantizadas entre ellas: una al mes del lanzamiento (la x.0.1) y otra al cabo de tres meses (la x.0.2). Esto es un problema para muchas organizaciones puesto que un cambio de versión supone problemas e inestabilidad. Bien, pues **Java 11 es la primera versión de Java con un JDK denominado LTS o Long Term Support**. Esto significa que Oracle garantiza que te dará **soporte** y actualizaciones para la versión **durante 3 años**, en lugar de tan solo 6 meses. Eso sí, a cambio de un "módico" precio (¿sabes cómo se lee "Oracle" del revés? Pues eso...).

¡Ah!, y por cierto, si te preguntas qué novedades hay de verdad en Java 11, quitando [la palabra clave var](#) (copiada de C#), el nuevo [recolector de basura](#) y [el "Flight Recorder"](#) del framework para recopilar datos para diagnosticar aplicaciones, poco hay que contar.

- **Versión SDK 12:** Hay mejoras de la versión 11, no grandes novedades del lenguaje. Si quieres más información haz clic



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

en la imagen de la izquierda.

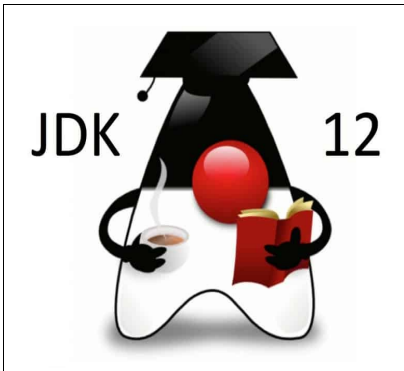


Figura 2.5: Dukes con enlaces.

(izq.) Blog de Versiones.

(der.) Página de Open JDK.

OpenJDK es una versión de Java gratuita y distribuible sin licencia, mantenida por la comunidad, con dos sabores: Licencia GNU y licencia GAL (Oracle) (<https://jdk.java.net/>).

2.2. INSTALAR EL ENTORNO DE DESARROLLO.

El JDK (SDK) puede descargarse e instalarse en muchos sistemas operativos de ordenadores y dispositivos. Veremos como hacerlo en los dos sistemas operativos (SO) más usados para ordenadores personales.

Puedes descargar los instalables del JDK SE desde la página <https://www.oracle.com/technetwork/java/javase/downloads/index.html>



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

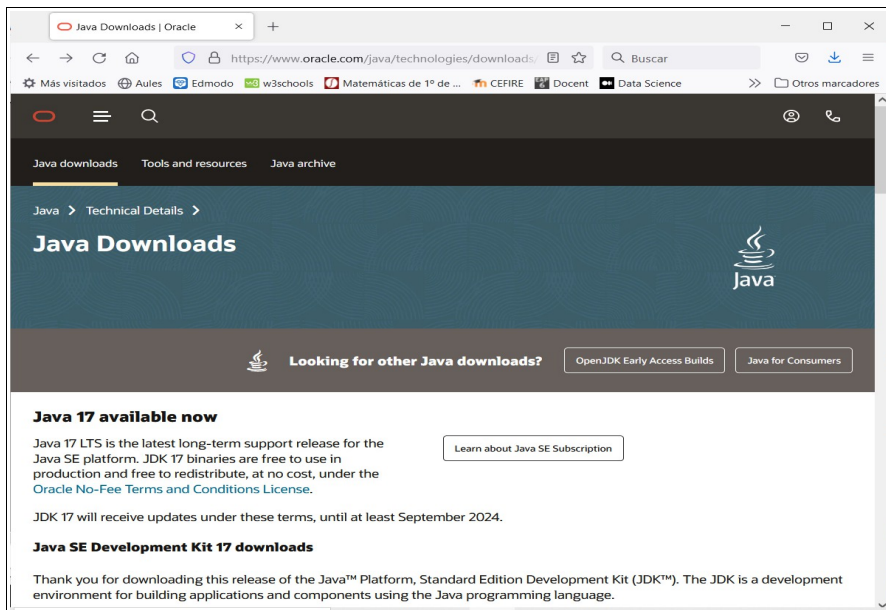


Figura 2.6: Página de descarga del SDK de Oracle.

EN MICROSOFT WINDOWS

Desde la página de descarga, se elige la versión deseada del entorno de desarrollo. Una vez descargado el programa de instalación del SDK, basta con ejecutarlo. En algunas versiones hay que prestar atención al directorio en el que se ha instalado el SDK. La razón es que debemos modificar 3 variables del sistema (variables que utiliza Windows para la configuración correcta de comandos). Son:

PATH

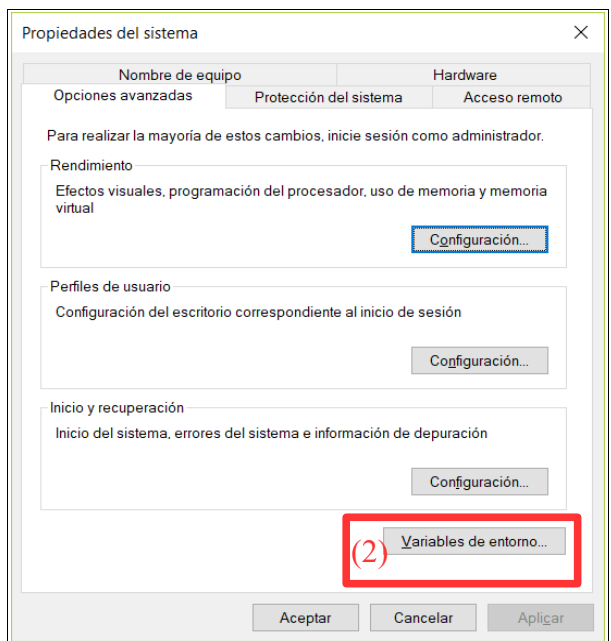
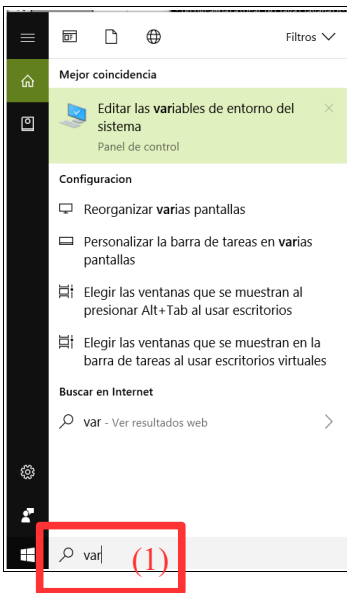
Variable que contiene rutas por defecto a los programas que indiquemos. La razón es que por ejemplo el comando **javac** debe de estar disponible estemos en la carpeta que estemos. Ese comando y el resto de comandos del SDK, están en la carpeta **bin** dentro de la carpeta en la que hemos instalado el SDK.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

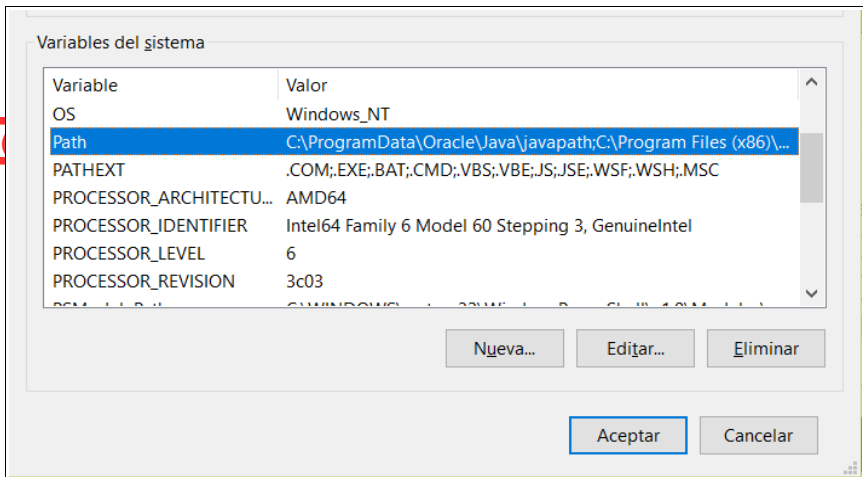
Si por ejemplo, al instalar el SDK SE 12, decido instalarlo en `d:\java\jdk12`, debo añadir a lo que tenía antes la variable PATH de mi sistema, la ruta: `d:\java\jdk12\bin`. Ejemplo de contenido de la variable **path**:

- (1) escribir en la cja de búsqueda variables de entorno, aparece una opción en el menú que es Editar las variables de entorno del sistema. Al hacer clic aparece la ventana de al lado.
- (2) Pulsar el botón Variables de entorno.
- (3) En el diálogo de las variables aparecen dos listas, la primera es para cambiar solamente para el usuario actual. La segunda es para cambiar a todos los usuarios (a todo el sistema). En la segunda lista (la del sistema), marcas la variable a cambiar (Path en este caso) y se pulsa el botón Editar.

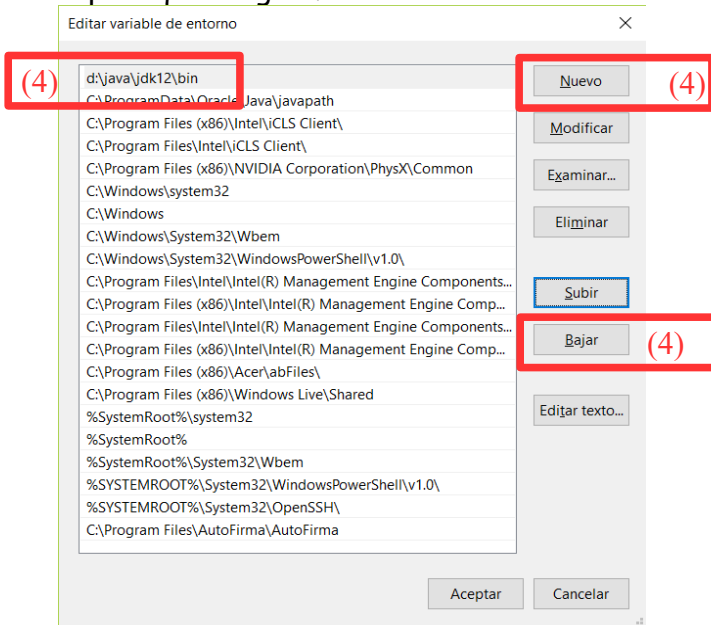




1DAM PROG. UNIDAD 2. Primeros Pasos en Java.



(4) Ahora creas/modificas la entrada y si hay una anterior la subes o borras para que tenga efecto la última.





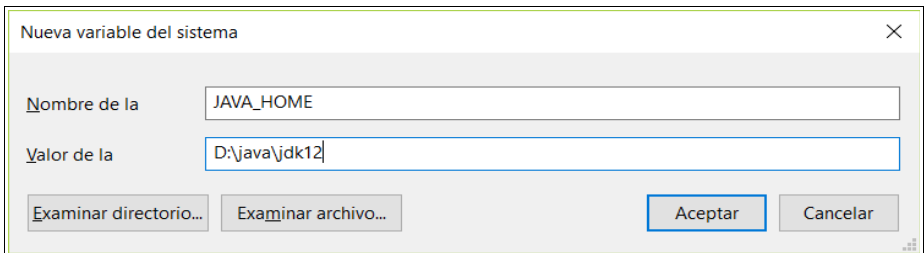
1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Nota: *Hasta que no reinicies, los cambios no tienen efecto.*

Nota: *Puedes comprobar el valor ejecutando en una ventana de comandos `echo %PATH%`*

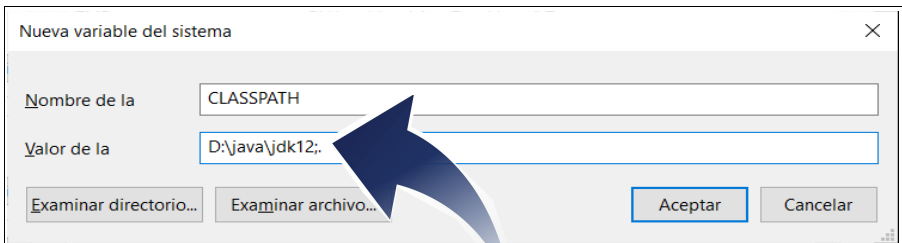
JAVA_HOME.

Variable utilizada por la mayoría de aplicaciones basadas en Java que contiene la ruta a la carpeta en la que se instaló el SDK. Por ejemplo, debería contener el valor `d:\java\jdk12\` en mi ejemplo.



CLASSPATH.

Es una variable similar al PATH que sirve para indicar rutas a las carpetas en las que se almacenan los archivos de clases de las aplicaciones Java. Debería contener la carpeta raíz del SDK y la carpeta actual que se expresa mediante el carácter (.).



D:\java\jdk12;



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

EN GNU/LINUX

Para conocer si hay una versión instalada habría que ejecutar el comando **java -version**. Si deseamos instalar Java o actualizarlo, hay varias opciones.

- La primera es utilizar el gestor **Synaptic** para descargar la última versión del paquete de desarrollo en Java.
- También podemos instalar desde la línea de comandos, sería algo como:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install default-jdk
```

- Finalmente, siempre podemos acudir a la página de descargas y descargarnos el instalable del SDK que queramos.

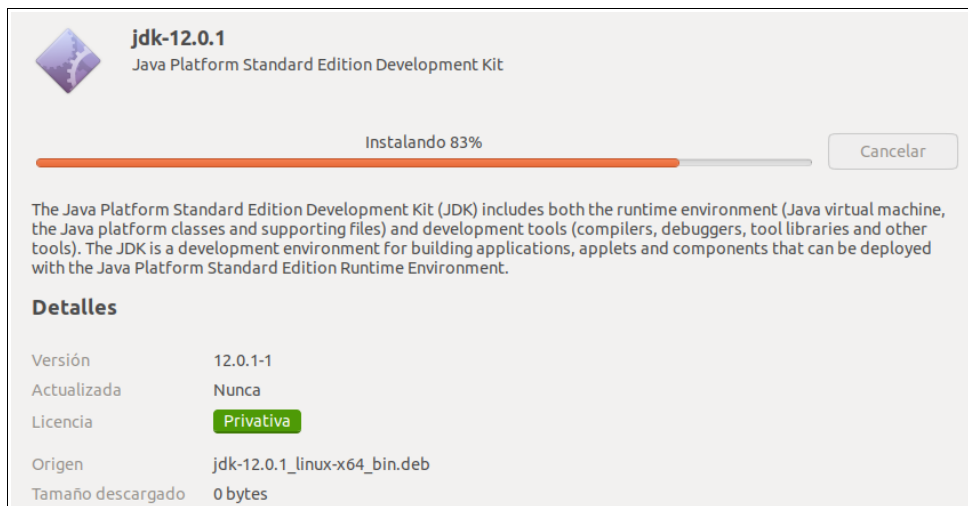


Figura 2.7: Instalando SDK 12 de un .deb descargado de Oracle.

El archivo hay que descomprimirlo (si es un .tar.gz) o ejecutarlo (si es un paquete rpm o .deb). Se crea un directorio con todo el SDK de



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

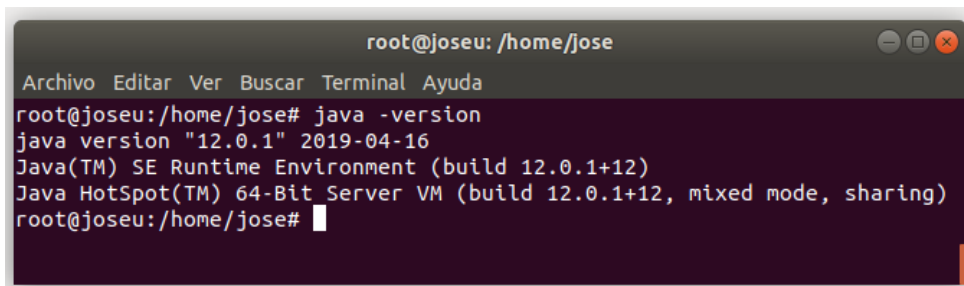
Java.

Nota: suele quedarse dentro del directorio `/usr/lib/jvm/nombre_del_SDK`.

En todo caso, sea cual sea la forma de instalar el SDK, si vas a usar los programas desde la línea de comandos habrá que modificar al menos 3 variables de entorno. Para conseguirlo puedes modificar el fichero `/etc/bash.bashrc` y al final añadir las siguientes entradas:

- `export JAVA_HOME="ruta al SDK de Java"` La ruta podría ser algo como `/usr/lib/jvm/jdk-12.0.1` en el ejemplo.
- `export PATH="$PATH:$JAVA_HOME/bin"` para definir la variable que indica al sistema donde debe buscar ejecutables.
- `export CLASSPATH="rutas a los directorios en los que se almacenarán programas en Java"`. Es conveniente incluir el directorio raíz del SDK que quieras usar y el directorio actual. Así que por ejemplo, una posible opción sería usar este comando: `export CLASSPATH="$JAVA_HOME:."`

Para probar que todo ha ido bien, tanto en Windows (ejecutar cmd) como en GNU/Linux (ejecutar terminal) puedes abrir una consola (ventana de comandos) y usar los comandos `java -version` y `javac`



```
root@joseu: /home/jose
Archivo Editar Ver Buscar Terminal Ayuda
root@joseu:/home/jose# java -version
java version "12.0.1" 2019-04-16
Java(TM) SE Runtime Environment (build 12.0.1+12)
Java HotSpot(TM) 64-Bit Server VM (build 12.0.1+12, mixed mode, sharing)
root@joseu:/home/jose#
```

Figura 2.8: Comprobando la versión de java.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

DOCUMENTACIÓN

En la página de Oracle puedes acceder a la documentación oficial de varios SDK. La más interesante es la documentación del API de Java (clases estándar de Java). Aunque inicialmente es conveniente pasar por **Get Started**. La única pega es que no hay documentación en castellano. Otra posibilidad (más recomendable incluso), es descargarse la documentación en un archivo **zip**. De esta forma no dependes de la conexión a Internet para consultarla.

2.3. PRIMEROS PROGRAMAS EN JAVA.

Con el SDK ya instalando y funcionando, ya tenemos todo lo necesario para hacer programas usando el lenguaje Java.

2.3.1. COSAS A TENER EN CUENTA.

El código fuente de un programa Java se escribe en documentos de texto con extensión **.java**. Para escribirlo basta cualquier editor de texto como el bloc de notas de Windows o **vi**, **nano** o **gedit** de Linux.

Nota: si usas el bloc de notas, recuerda que al guardar debes quitar la opción Documentos de texto (*.txt) o si no, añade **.txt** al final del fichero. En la figura el fichero se llamará **ejemplo.java.txt** y eso te puede llevar a errores.

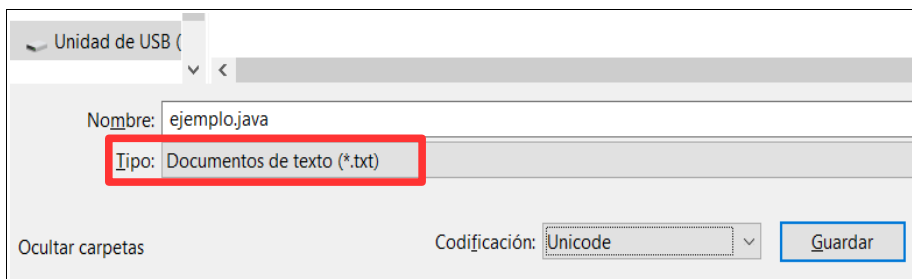


Figura 2.9: Error al dar nombre en el bloc de notas.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Nota: En el caso de Windows es preferible que uses otros editores más adecuados para programar como por ejemplo: notepad++, sublime, visual studio code, bracket, etc.

Al ser Java un lenguaje multiplataforma y pensado para su integración en redes, **la codificación de texto utiliza el estándar Unicode**, lo que implica que los programadores de lenguas romances (castellano, valenciano, francés, etc.) podemos utilizar símbolos de nuestra lengua como la ñe o las vocales con tildes o diéresis a la hora de dar nombre a los identificadores de un programa. **Pero no es aconsejable hacerlo.**

Algunos detalles importantes que debes recordar:

- **Java es case sensitive:** al igual que otros lenguajes como C, C++ y C#, en Java se diferencia entre mayúsculas y minúsculas y por tanto las palabras `if` o `porCcentaje` no son las mismas que `If` y `porcentaje`.
- **Sentencias acaban en punto y coma.** Cada sentencia del lenguaje termina con un carácter punto y coma (;).
- **Escritura libre:** Una instrucción puede extenderse más de una línea. Además se pueden dejar espacios y tabuladores a la izquierda e incluso en el interior de la instrucción para separar los elementos de la misma. Pese a que la escritura es libre, siempre **hay unos hábitos o convenciones que se han demostrado ventajosos** y que los programadores de cada lenguaje se esfuerzan en respetar. Con independencia del lenguaje, estos hábitos son muy recomendables: **indentar código, comentarlo y no apelotonar elementos de cada sentencia.** El motivo es que se escriben programas fuente más fáciles de leer, entender y cambiar si es necesario.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- **Los comentarios en Java:** en Java hay comentarios de una línea y de varias líneas. Si son de una línea deben comenzar con doble barra (//) y todo lo que sigue hasta el final de la línea es ignorado por el traductor. Si ocupan más de una línea deben comenzar con /* y terminar con */. Ejemplo de comentarios:

```
/* Comentario  
de varias líneas */  
  
// Comentario de una línea
```

- **Java usa bloques de sentencias:** En Java hay **sentencias simples** y **sentencias compuestas** (formadas por más de una sentencia simple). La forma de indicar donde comienza y acaba un grupo de sentencias es mediante las llaves quebradas:
 - Comienza un grupo: {
 - Acaba un grupo: }

Ejemplo:

```
sentencias antes del bloque  
{  
    sentencias dentro del bloque  
}  
sentencias después del bloque
```

2.3.2. ANALIZAR UN PRIMER PROGRAMA.

Cuando un usuario pide a la máquina virtual de Java que ejecute un programa, le indica el nombre de la clase que quiere ejecutar (desde línea de comandos por ejemplo: java PrimerPrograma.java) y la JVM busca en la clase un método (subprograma) exactamente igual a este:

```
public static void main(String[] args) {  
    // Aquí van las sentencias que compila y ejecuta...  
}
```

La JVM ejecutará las sentencias que tenga este método en el orden en



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

que estén escritas. Una clase para poder ejecutarse (ser una aplicación) debe ser pública y contener este método **main()** que sirve para saber por donde comenzar a ejecutar.

ANATOMÍA DE UNA CLASE EN JAVA

Vamos a comentar brevemente qué cosas podemos encontrar dentro de una clase (no forzosamente, pero podrían aparecer):

- Comentarios.
- Declaración de datos globales: de la clase o de cada objeto.
- Código de inicialización: sentencias encerradas en llaves {}
- Declaración de métodos: de la clase o de sus objetos. Como por ejemplo el método `...main(){...}`
- Declaración de otras clases: solo existen dentro de la clase padre que las contiene.

Este código escrito en Java, sirve para escribir el texto ¡Hola Mundo! en la pantalla. No te preocupes si no entiendes la mayoría de las cosas, precisamente lo vemos para empezar a entenderlo:

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

EJERCICIO 1: escribe el código en un editor de texto. Lo guardas con el nombre **PrimerPrograma.java** y lo ejecutas de dos formas:

- Primera forma (llamamos al intérprete, la JVM, que lo compila al vuelo -no guarda los bytecodes generados- y lo ejecuta).
- Segunda forma (primero compilamos y generamos los bytecodes que se guardan en el `.class` y luego pedimos al intérprete que ejecute los bytecodes del `.class`).



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Vamos a explicar sus elementos para ir reconociéndolos:

- **Java es totalmente orientado a objetos.** Eso significa que hasta un programa funcionando es un objeto. Un programa fuente debe crear la clase de ese objeto. Así que la primera línea (**public class PrimerPrograma**) declara el nombre de la clase del código.

Cuando declaramos una clase, lo primero que ponemos es la cabecera y luego indicamos donde comienza y acaba con un bloque, por ejemplo:

```
public class Ejemplo { /* cuerpo de la clase... */ }
```

La primera palabra nos proporciona la posibilidad de dar permisos de acceso a la clase. Los permisos a nivel de clase son los siguientes:

- **"public"**: Una clase "public" es accesible (visible o utilizable) desde cualquier otra clase, no obstante, para que esto suceda, debe ser primero accesible el **"package"** donde se almacena. Para que un "package" sea accesible, debe de estar en el directorio que señala la variable "CLASSPATH" que definimos al instalar nuestro entorno Java y claro está, tener permiso de lectura en ese directorio. También podemos aportar indicaciones en la línea de comandos.
- **"package"**: Usar este identificador en la cabecera de la clase es opcional, pues es la opción por defecto, es decir, si no indicamos nada, es como indicar acceso de tipo package. Ejemplo:

```
class Ejemplo { /* cuerpo de la clase... */ }
```

Es lo mismo que escribir:

```
package class Ejemplo { /* cuerpo de la clase... */ }
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Las clases "package" (las que no son "public") son accesibles solo al usarlas desde su propio package.

También hay otras implicaciones: si es public, el fichero debe llamarse como ella (por tanto, como consecuencia, solo puede haber una clase pública en cada fichero .java).

Nota: el nombre de la clase pública del programa y el nombre del fichero donde se almacena deben coincidir, de lo contrario, dará error al compilar cuando tengamos aplicaciones formadas por más de una clase. Resumiendo: **haz coincidir el nombre del archivo .java y el nombre de la clase pública que contiene** (como un programa), tanto en mayúsculas como en minúsculas.



Figura 2.10: Nombre de clase y del fichero .java deben coincidir.

Nota: por convención, a las clases se les da nombres que comienzan en mayúscula. Si el nombre lo forman varias palabras, la inicial de cada palabra comienza también en mayúscula.

- La línea `public static void main(String args[])`, sirve para indicar donde debe comenzar a ejecutar sentencias el programa, lo



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

indica el inicio del método `main()`. Este método contiene las instrucciones que se ejecutarán cuando el programa arranque. Es decir, lo que está dentro de las llaves del `main()`, es lo que comienza a ejecutar el programa.

- La instrucción `System.out.println()` es una llamada al método `println()` que sirve para escribir texto en pantalla y luego saltar de línea. Como le pasamos un texto, se encierra entre comillas dobles.

ERRORES EN LOS PROGRAMAS.

En las distintas fases de creación de un programa, durante su codificación, su prueba o su explotación, pueden detectarse errores. Los errores pueden ser de distinto tipo:

- **Errores de codificación (léxicos o sintácticos):** aparecen al utilizar mal las reglas del lenguaje: palabras reservadas mal escritas, instrucciones incompletas, mal formadas, etc. Estos errores se detectan durante la fase de compilación, cuando el programa se traduce del lenguaje de programación de alto nivel a código máquina o bytecodes en el caso de Java. La herramienta de traducción nos avisa del error, por tanto **son fáciles de detectar y corregir.**
- **Errores de ejecución:** Se producen durante la ejecución del programa porque se realiza una operación no permitida o porque se da una situación imprevista de la que el programa no se puede recuperar. Ejemplos: se agota la memoria, se intenta acceder a un fichero de disco que no existe o no se tienen permisos, no queda espacio de disco para escribir, cierto dato produce una situación no contemplada por el programa, etc. El compilador no detecta este tipo de errores y en general, son



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

más difíciles de detectar (cuando aparecen) y de corregir.

- **Errores lógicos:** El programa no produce error, ni durante la compilación ni durante su ejecución. Simplemente, bajo ciertas condiciones no produce el resultado correcto que se esperaba. Son los más complicados de solucionar.

2.3.2. COMPILAR Y EJECUTAR EN LÍNEA DE COMANDOS.

La compilación del código java se realiza mediante el programa **javac** incluido en el SDK. La forma básica de compilar es (desde la línea de comandos):

```
javac Archivo.java
```

El resultado de esto es un archivo con el mismo nombre que el archivo java pero con la extensión **.class**. Esto ya es el archivo con el código en forma de bytecode. Es decir con el código precompilado.

El programa es ejecutable si contiene el método main y el código se puede interpretar usando el programa **java** del kit de desarrollo (que se encuentra en el mismo sitio que javac). Sintaxis:

```
java Archivo.class // El intérprete usa el .class
java Archivo.java  // El intérprete llama a javac
java Archivo       // Si el .class ya está generado
```

Estos comandos hay que escribirlos desde la línea de comandos en la carpeta en la que se encuentre el programa. Pero antes hay que asegurarse de que los programas del JDK son accesibles desde cualquier carpeta del sistema (configurar las variables del sistema).

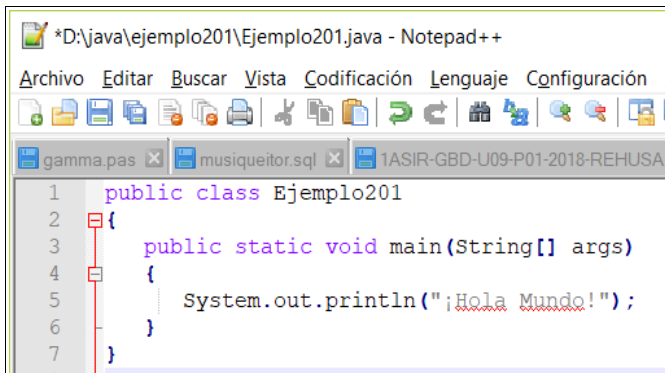
En el caso de GNU/Linux hay que ser especialmente cuidadoso con la variable de sistema CLASSPATH y asegurarnos de que contiene en sus rutas raíces de Java el símbolo "." (punto) que representa a la carpeta actual. De otro modo la ejecución del comando java podría fallar.



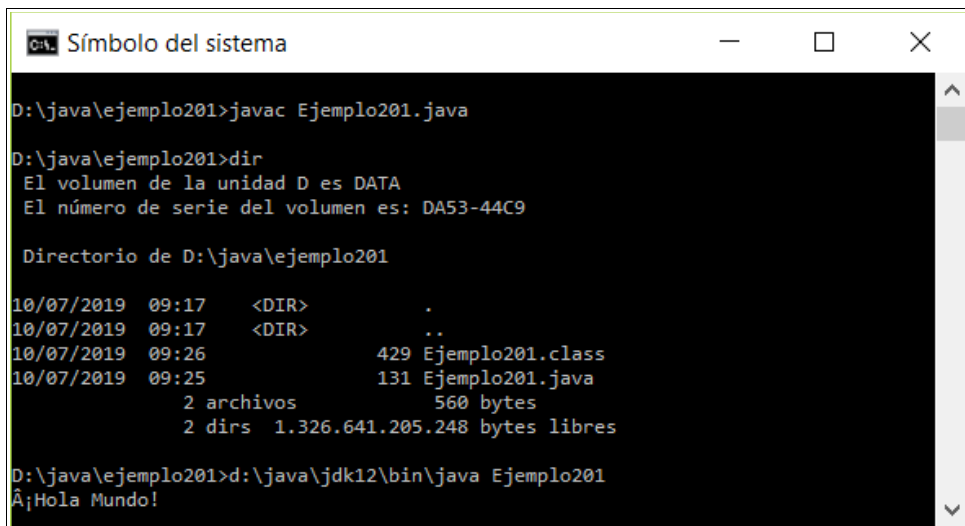
1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Nota: Para cada nuevo programa que hagamos, es conveniente crear una carpeta nueva y dejar sus ficheros dentro. Esta carpeta hará de package de las clases.

EJEMPLO 1: En Windows, crea el siguiente programa en un editor y lo guardas en la carpeta ejemplo201. Luego lo compilas, miras los ficheros generados y lo ejecutas.



```
*D:\java\ejemplo201\Ejemplo201.java - Notepad++
Archivo  Editar  Buscar  Vista  Codificación  Lenguaje  Configuración  !
gamma.pas  x  musiqueitor.sql  x  1ASIR-GBD-U09-P01-2018-REHUSA
1  public class Ejemplo201
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("¡Hola Mundo!");
6      }
7  }
```

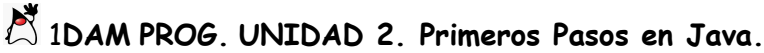


```
Símbolo del sistema
D:\java\ejemplo201>javac Ejemplo201.java
D:\java\ejemplo201>dir
El volumen de la unidad D es DATA
El número de serie del volumen es: DA53-44C9

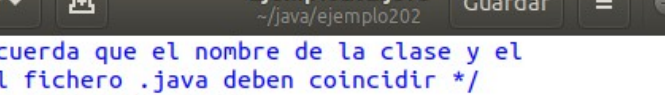
Directorio de D:\java\ejemplo201
10/07/2019  09:17    <DIR>          .
10/07/2019  09:17    <DIR>          ..
10/07/2019  09:26                429 Ejemplo201.class
10/07/2019  09:25                131 Ejemplo201.java
                2 archivos             560 bytes
                2 dirs 1.326.641.205.248 bytes libres

D:\java\ejemplo201>d:\java\jdk12\bin\java Ejemplo201
¡Hola Mundo!
```

EJEMPLO 2: Haz lo mismo en GNU/Linux pero crea la carpeta ~/java/ejemplo202.



```
jose@joseu: ~/java/ejemplo202
Archivo Editar Ver Buscar Terminal Ayuda
jose@joseu:~/java/ejemplo202$ ls
Ejemplo202.java
jose@joseu:~/java/ejemplo202$ javac Ejemplo202.java
jose@joseu:~/java/ejemplo202$ ls
Ejemplo202.class  Ejemplo202.java
jose@joseu:~/java/ejemplo202$ java Ejemplo202
¡Hola mundo!
jose@joseu:~/java/ejemplo202$
```



```
/* Recuerda que el nombre de la clase y el
   del fichero .java deben coincidir */
public class Ejemplo202
{
    public static void main(String[] args)
    {
        System.out.println("¡Hola mundo!");
    }
}
```

2.3.3. HERRAMIENTA JAVADOC.

Javadoc es una herramienta muy interesante del SDK para generar automáticamente documentación de los programas Java. Genera documentación para paquetes completos o para archivos java. Su sintaxis básica es:

```
javadoc Archivo.java
javadoc paquete
```

El funcionamiento es el siguiente: Los comentarios que comienzan con los códigos **/**** se llaman **comentarios de documento** y serán utilizados



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

por los programas de generación de documentación javadoc. Los comentarios javadoc comienzan con el símbolo `/**` y terminan con `*/`

Cada línea javadoc se suele iniciar con el símbolo de asterisco para mejorar su legibilidad. Dentro se puede incluir cualquier texto; incluso se pueden utilizar códigos HTML para que al generar la documentación en este formato. En el código javadoc se pueden usar etiquetas especiales, las cuales comienzan con el símbolo `@`. Pueden ser:

- **@author**. Tras esa palabra se indica el autor del código. Para que aparezca hay que indicarlo en la llamada a la herramienta, igual que la versión: `javadoc -author -version fichero.java`
- **@version**. Le sigue el número de versión de la aplicación
- **@see**. Tras esta palabra se indica una referencia a otro código Java relacionado con éste.
- **@since**. Indica desde cuándo esta disponible este código
- **@deprecated**. Palabra a la que no sigue ningún otro texto en la línea y que indica que esta clase/método es obsoleta/obsoleto.
- **@throws**. Indica las excepciones o problemas que pueden lanzarse en ese código.
- **@param**. Palabra a la que le sigue texto que describe a los parámetros que requiere el código para su utilización (el código en este caso es un método de clase). Cada parámetro se coloca en una etiqueta `@param` distinta, por lo que puede haber varios `@param` para el mismo método.
- **@return**. Tras esta palabra se describe los valores que devuelve el código (el código en este caso es un método de clase)

Los comentarios javadoc hay que colocarlos en tres sitios distintos



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

dentro del código java de la aplicación:

- **Al principio del código de la clase (antes de cualquier código Java).** En esta zona se colocan comentarios generales sobre la clase o interfaz que se crea mediante el código Java. Dentro de estos comentarios se pueden utilizar las etiquetas: **@author**, **@version**, **@see**, **@since** y **@deprecated**
- **Delante de cada método.** Los métodos describen las cosas que puede realizar una clase. Delante de cada método los comentarios javadoc se usan para describir al método en concreto. Además de los comentarios, en esta zona se pueden incluir las etiquetas: **@see**, **@param**, **@exception**, **@return**, **@since** y **@deprecated**
- **Delante de cada atributo/dato global.** Se describe para qué sirve cada dato en cada clase. Pueden contener las etiquetas: **@since** y **@deprecated**

EJEMPLO 3: Añade al programa Ejemplo1 o Ejemplo2 los comentarios javadoc resaltados en amarillo y luego usa la herramienta para generar la documentación. Luego abre el documento .html generado.

```
/** Esto es un comentario para probar javadoc
 * este texto aparece en el archivo HTML generado.
 * <strong>Realizado en agosto 2019</strong>
 *
 * @author José RR
 * @version 1.0
 */
public class Ejemplo202 {
    //Este comentario no aparece en javadoc
    /** Este método contiene el código ejecutable
     *
     * @param args Lista de argumentos de la línea de comandos
     * @return void
     */
    public static void main(String[] args){
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

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

Aparece como resultado la página web de la página siguiente.

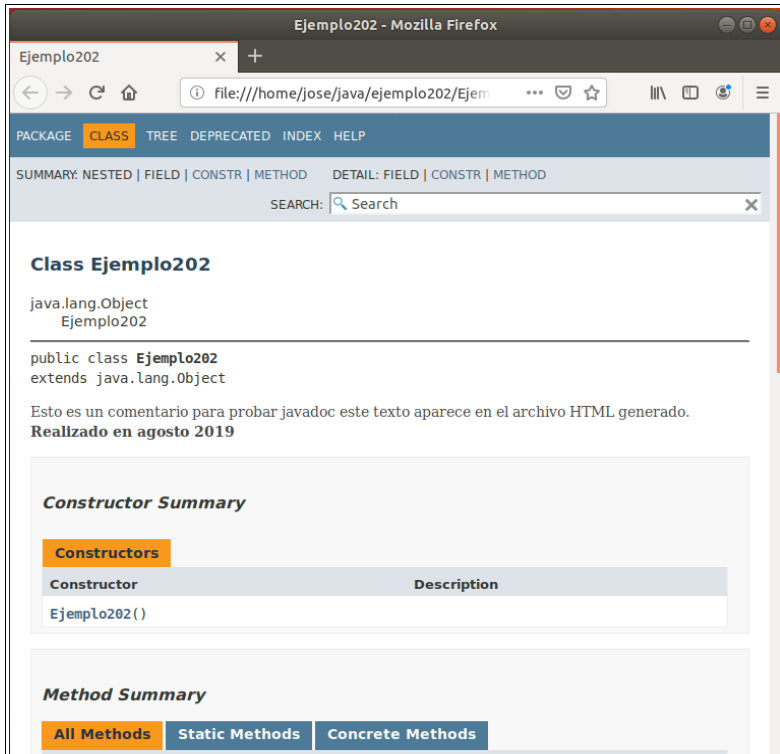


Figura 2.11: Página de documentación generada por javadoc.

2.3.4. PAQUETES Y SENTENCIA IMPORT.

En cualquier lenguaje de programación existen librerías que contienen código ya escrito con tareas comunes, que facilitan la creación de nuevos programas. En Java no se llaman librerías, sino paquetes.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Los paquetes son carpetas que contienen clases ya preparadas y más paquetes. Cuando se instala el JDK, además de los programas necesarios para compilar y ejecutar código Java, se incluyen miles de clases dentro de cientos de paquetes ya preparados, que facilitan la generación de programas.

Algunos paquetes sirven para utilizar funciones matemáticas, de lectura y escritura, comunicación en red, programación de gráficos y muchas otras cosas. Por ejemplo la clase `System` está dentro del paquete `java.lang` (paquete básico) y contiene entre otros, el método `out.println()` que necesitamos para escribir por pantalla.

Si quisiéramos utilizar la clase `Date` que sirve para trabajar con fechas, necesitamos incluir una instrucción que permita incorporar el código de la clase `Date` cuando compilemos nuestro programa. Para eso sirve la instrucción `import`. La sintaxis de esta instrucción es:

```
import paquete.subpaquete.subsubpaquete. ... .Clase;  
import paquete.*;
```

Esta instrucción se coloca arriba del todo en el código. Para la clase `Date` sería:

```
import java.util.Date;
```

Lo que significa, importar en el código la clase `Date` que se encuentra dentro del paquete `util` que, a su vez, está dentro del gran paquete llamado `java`.

También se puede utilizar el asterisco en esta forma:

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

Esto significa que se va a incluir en el código todas las clases que están dentro del paquete `util` de `java`. En realidad no es obligatorio incluir la



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

palabra **import** para utilizar una clase, pero sin ella debemos escribir el nombre completo de la clase. Es decir no podríamos utilizar el nombre `Date`, sino `java.util.Date` cada vez que queramos usar una fecha.

Nota: *ten en cuenta que si en `java.util`, que es donde está `Date` hubiese otros subpaquetes debajo, por ejemplo, `paquete1`, usar el asterisco solo importa todas las clases de `java.util`, no las clases de los subpaquetes que tenga.*

2.4. ELEMENTOS BÁSICOS DE UN PROGRAMA.

2.4.1. IDENTIFICADORES.

Los nombres son fundamentales a la hora de escribir programas. En un programa, cuando quieres usar una sentencia del lenguaje, debes saber el nombre que tiene. Además, cuando quieres dar nombre a un dato (una variable o una constante) o a un método (un trozo de código) o a una clase, debes usar nombres. Por tanto **debes conocer bien las reglas que se aceptan a la hora de dar nombre a los elementos que usas en un programa.**

A los nombres se les conoce como **Identificadores**. Un identificador en Java es una secuencia uno o más caracteres donde el primer carácter debe ser una letra o un carácter de subrayado (`_`) y el resto de caracteres pueden ser letras, dígitos o subrayados.

Algunos identificadores están reservados por el lenguaje y no los puedes usar para otra cosa, se conocen como **palabras reservadas**. En Java hay unas 57 y son las siguientes:

- | | | |
|-------------------------|----------------------|----------------------|
| • <code>abstract</code> | • <code>break</code> | • <code>catch</code> |
| • <code>assert</code> | • <code>byte</code> | • <code>char</code> |
| • <code>boolean</code> | • <code>case</code> | • <code>class</code> |



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

<ul style="list-style-type: none">• <code>const</code>• <code>continue</code>• <code>default</code>• <code>do</code>• <code>double</code>• <code>else</code>• <code>enum</code>• <code>extends</code>• <code>final</code>• <code>finally</code>• <code>float</code>• <code>for</code>• <code>goto</code>• <code>if</code>	<ul style="list-style-type: none">• <code>implements</code>• <code>import</code>• <code>instanceof</code>• <code>int</code>• <code>interface</code>• <code>long</code>• <code>native</code>• <code>new</code>• <code>package</code>• <code>private</code>• <code>protected</code>• <code>public</code>• <code>return</code>• <code>short</code>	<ul style="list-style-type: none">• <code>static</code>• <code>strictfp</code>• <code>super</code>• <code>switch</code>• <code>synchronized</code>• <code>this</code>• <code>throw</code>• <code>throws</code>• <code>transient</code>• <code>try</code>• <code>void</code>• <code>volatile</code>• <code>while</code>
--	--	--

El resto de identificadores puedes usarlos para dar nombre a tus clases, variables y subrutinas. Ejemplos de identificadores legales en Java:

`N` `n` `ratio` `x15` `poco` `a` `largo` `nombre` `nombreCompleto`

No se permiten espacios en blanco, por ejemplo `hoLaMundo` es un identificador legal, pero `hoLa Mundo` es ilegal.

Como Java es case sensitive, los identificadores `hoLamundo`, `hoLaMundo`, `hoLaMUNDO` y `HOLAmundo` se consideran diferentes.

Java es muy liberal en cuanto a las letras que quieras usar porque utiliza el conjunto de caracteres Unicode, lo que te permite usar letras propias de tu idioma (eñes, c trencadas, letras acentuadas, etc.) en tus identificadores. Sin embargo, hay que intentar evitarlos en las sentencias del programa (no en los mensajes que ven los usuarios) y ceñirnos a las letras del alfabeto inglés (sustituye eñes, acentos, c trencadas, etc.).



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Además hay reglillas que deberías seguir a la hora de dar nombres a lo que utilices en tus programas. Por ejemplo sería bueno:

- **Usar nombres representativos:** si tu programa usa veinte variables y las llamas v1, v2, v3, v4, v19 y v20, cuando pasen 3 días sin verlo, ni tu mismo, que has escrito el programa, lo comprenderás con facilidad. Si vas a dar un nombre a un dato, que el nombre de pistas del valor que contiene. Si das nombre a un método (representa una acción) normalmente el nombre será un verbo o al menos representa una acción, etc.
- **Primera letra de Clases e interfaces en mayúscula:** cuando des nombre a una clase, la primera letra debe estar en mayúsculas, mientras que los nombres de variables y métodos siempre comienzan con minúsculas.
- **Evitar los subrayados:** la mayoría de programadores de Java evitan usar subrayados en los nombres.
- **Nombres compuestos:** si un nombre tiene varias palabras, la primera letra de cada una se pone en mayúsculas (menos la primera si no es el nombre de una clase o una interface). Ej: una variable que tenga el nombre completo (nombre y apellidos de una persona) podemos llamarla *nombreCompleto*.
- **Nombres compuestos separados por puntos:** Finalmente ten en cuenta que cuando se usa el punto (.) como separador de nombres, se está indicando que lo de la derecha está contenido en lo de la izquierda. Por ejemplo ya hemos usado `System.out.println()`, esto indica que `System` contiene algo que se llama `out`, que a su vez contiene algo que se llama `println()`. Es decir, el punto es un operador y las palabras `izquierda.derecha` significan que `izquierda` contiene a `derecha`.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

2.4.2. DATOS: CONSTANTES Y VARIABLES.

Los datos con los que trabajan nuestros programas pueden ser constantes o variables. Los datos constantes no pueden cambiar de valor mientras que los datos variables en cierto momento pueden tener un valor y más tarde ha cambiado a otro valor.

Las constantes pueden ser **literales** si el valor es literalmente lo que se escribe en el programa, o **simbólicas** si le damos un nombre aunque su valor no pueda cambiar.

Las variables son contenedores que sirven para almacenar los datos que utiliza un programa. Dicho más sencillamente, son nombres que asociamos o damos a determinados datos. La realidad es que cada variable ocupa un espacio en la memoria RAM del ordenador para almacenar el dato al que se refiere. Es decir, cuando utilizamos el nombre de la variable en el programa, realmente estamos haciendo referencia a un dato que está en memoria RAM (nos da igual donde).

DECLARACIÓN DE VARIABLES

Antes de poder utilizar una variable, en Java se debe declarar mediante esta sintaxis:

```
[modificadores] tipoDeDato nombreDeVariable;
```

Donde los modificadores son opcionales (los comentamos más adelante), tipoDeDato es el tipo de valores que puede almacenar la variable (texto, números enteros, números con decimales, un valor de verdad-booleanos-, un carácter, un objeto, etc.) y nombreDeVariable es el nombre con el que se conoce al dato en el programa. Ejemplos:

```
int dias;           // días es un número entero, sin decimales
boolean decision;   // decisión sólo puede ser verdadera o falsa
```

También se puede hacer que la variable tome un valor inicial al



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

declararla. Juntamos en una misma sentencia declaración e inicialización (darle un primer valor). Ejemplo:

```
int diasEnAny= 365;
```

Y se puede declarar/inicializar más de una variable a la vez del mismo tipo en la misma línea si las separamos con comas. Ejemplos:

```
int dias=365, anyo=23, semanas;
```

Al declarar una variable se puede incluso utilizar una expresión:

```
int a= 13, b= 18;  
int c= a + b;           // es válido, c vale 31
```

Java es un lenguaje muy estricto al utilizar tipos de datos. Variables de datos distintos son incompatibles. Algunos autores hablan de **lenguaje fuertemente tipado o lenguaje muy tipificado**. El caso contrario sería el lenguaje C o C++, en los que jamás se comprueban de manera estricta los tipos de datos. Parte de la seguridad y robustez de Java se deben a esta característica.

ASIGNACIÓN (CAMBIAR EL VALOR)

En Java para asignar/cambiar valores a una variable, basta con utilizar el carácter igual (=) que representa la **operación de asignación**. La sintaxis básica es:

nombreVariable = expresión;

A la izquierda del operador va el nombre de una variable y a la derecha una expresión que genera un valor que se almacena en la variable. Ya se ha visto antes que al declarar una variable se puede asignar un valor:

```
int x= 7;
```

Pero la asignación se puede utilizar en cualquier momento (tras haber



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

declarado la variable). El siguiente código es equivalente al anterior:

```
int x;  
x = 7;  
x = x + 1;    // aquí deja de ser equivalente
```

Como se ve en la última línea anterior, la expresión para dar el valor a la variable puede ser tan compleja como queramos.

Nota: A diferencia de lenguajes como C, en Java se asigna un valor inicial a algunas variables según el lugar del programa donde se declaran. En el caso de los números es el cero.

```
int x; // x ya vale cero, si no es local
```

CONSTANTES SIMBÓLICAS

Una constante es una variable de sólo lectura. Dicho de otro modo más correcto, es un valor que no puede cambiar (por lo tanto no es una variable). La forma de declarar constantes es la misma que la de crear variables, sólo que hay que anteponer la palabra reservada **final** (un **modificador**) que es la que indica que estamos declarando una constante y por tanto no podremos cambiar su valor inicial:

```
final double PI= 3.141591;  
PI = 4; //Error, no podemos cambiar el valor de PI
```

Nota: Como medida aconsejable (aunque no es obligatoria), los nombres de las constantes acostumbran a ir en mayúsculas.

ÁMBITO DE CONSTANTES Y VARIABLES:

Toda variable tiene un ámbito que es **la parte del código en la que se puede utilizar**. De hecho las variables tienen un ciclo de vida:

(1) En la declaración se reserva el espacio necesario de memoria RAM para que se puedan utilizar (digamos que se avisa de su futura existencia).



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- (2) Se inicializa: Se la asigna su primer valor (la variable nace)
- (3) Se utiliza en diversas sentencias.
- (4) Cuando finaliza la ejecución del bloque en el que fue declarada, la variable muere. Es decir, se libera el espacio de memoria que ocupa. No se la podrá volver a utilizar.

Una vez que la variable ha sido eliminada, no se puede utilizar. Dicho de otro modo, no se puede utilizar una variable más allá del bloque en el que ha sido definida. Ejemplo:

```
{  
    int x= 9;  
}  
int y = x; // error, ya no existe x, estamos fuera de su ámbito
```

2.4.3. TIPOS DE DATOS.

Cualquier dato que se almacene en la RAM del ordenador debe representarse como un número binario (una cadena de ceros y unos). La cantidad mínima de memoria que un programa puede intercambiar con la memoria RAM suele ser un byte (8 bits).

Cada tipo de valor tiene unas necesidades de memoria, una representación interna y una serie de operaciones que se le pueden aplicar. Eso es lo que resume un tipo de dato. Cuando declaramos una variable tenemos que avisar de esas características (peso en bytes, representación y operaciones permitidas) al compilador. En Java hay unos tipos de datos denominados **primitivos** que son estos:

- **byte**: almacena 8 bits que representan un número entero en formato complemento a 2, entre -128 y 127 inclusive.
- **short**: pesa 2 bytes (16 bits). Pueden almacenar números enteros en complemento a 2 con valores entre -32768 y 32767.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- **int:** pesa 4 bytes (32 bits). Puede almacenar números enteros en complemento a 2 con valores desde -2147483648 hasta 2147483647.
- **long:** pesa 8 bytes (64 bits). Puede almacenar números enteros en complemento a 2 con valores desde -9223372036854775808 hasta 9223372036854775807.
- **float:** pesa 4 bytes de memoria y puede almacenar números con decimales representados en coma flotante. El máximo valor que puede almacenar es 10 elevado a 38. Tiene 7 dígitos significativos (por tanto los números 32.3989231134 y 32.3989234399 son redondeados a 32.398923).
- **double:** pesa 8 bytes, almacena números con decimales en coma flotante. El mayor valor es 10 elevado a 308, y tiene 15 dígitos significativos (el doble de precisión de un float).
- **char:** pesa 2 bytes y almacena un único carácter como 'A', '*' o un espacio en blanco: ' '. También tiene caracteres no imprimibles como tabulador o salto de línea.
- **Boolean:** pesa 1 byte y almacena los valores **true** y **false**.

En la siguiente figura aparecen en forma de esquema.

Nota: no intentes memorizar *ni el esquema ni los detalles de cada tipo de dato. Basta con que te quedes con los nombres y la idea de que según los datos con que quieras trabajar, debes usar unos u otros. Es algo que irás interiorizando poco a poco.*

Si eres observador, verás que en el esquema a parte de los tipos primitivos aparecen tipos objeto, y dentro de estos, abajo hay un apartado con **tipos envoltorio** o **wrapper** que contiene clases con nombres similares a los tipos primitivos (pero la primera letra en mayúsculas), vamos a aclarar distintas cuestiones sobre el esquema.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

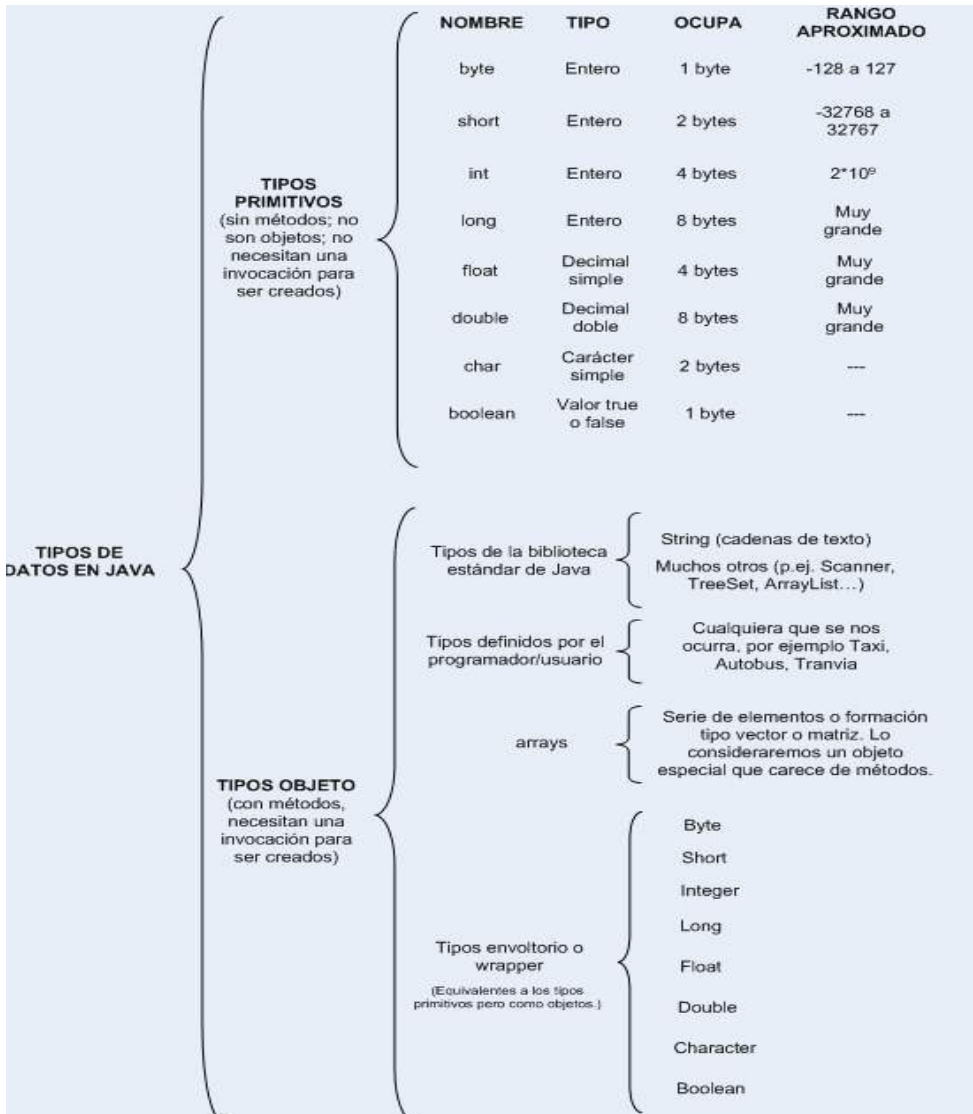


Figura 2.12: Esquema de tipos de datos de Java.

Un objeto es una cosa distinta a un tipo primitivo, aunque



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

almacenen la misma información. Hay que tener siempre presente que los objetos en Java tienen un tratamiento y los tipos primitivos otro. Que en un momento dado contengan la misma información no significa en ningún caso que sean lo mismo.

De momento, recuerda que el tipo primitivo es algo elemental y el objeto algo más complejo. Imagina una cesta de manzanas en la calle (algo elemental). Imagina las mismas manzanas dentro de una nave espacial (algo complejo). El contenido es el mismo (manzanas), pero no es lo mismo una cesta de mimbre que una nave espacial.

¿Para qué tener esa aparente duplicidad entre tipos primitivos y tipos envoltorio? Tened en cuenta una cosa: un tipo primitivo es un dato elemental, mientras que un objeto es una entidad compleja y dispone de métodos (código que puede realizar trabajos). Por otro lado, de acuerdo con la especificación de Java, es posible que necesitemos utilizar dentro de un programa un objeto que "almacene" como contenido un número entero. Desde el momento en que sea necesario un objeto, tendremos que pensar en un envoltorio, por ejemplo Integer. Inicialmente nos puede costar un poco distinguir cuándo usar un tipo primitivo y cuándo un envoltorio en situaciones en las que ambos sean correctos. **Seguiremos esta regla: usaremos por norma general tipos primitivos a no ser que sea necesario un objeto.**

Es decir, si somos caperucita roja y vamos a llevar manzanas de merienda a nuestra abuelita, podemos usar una nave espacial o una cesta de mimbre para trasportarlas, pero no tiene mucho sentido fardar de nave. Salvo que nuestra abuela viva en marte, usaremos la cesta.

Los nombres de tipos primitivos y envoltorio se parecen mucho. En realidad, excepto entre `int` e `Integer` y `char` y `Character`, la



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

diferencia se limita a que en un caso la inicial es minúscula (por ejemplo `double`) y en el otro es mayúscula (`Double`). Esa similitud puede confundirnos inicialmente, pero hemos de tener muy claro qué es cada tipo y cuándo utilizarlo.

Una cadena de caracteres es un objeto. El tipo `String` en Java nos permite crear objetos que contienen texto (palabras, frases, etc.). **El texto debe ir siempre entre comillas dobles.** Muchas veces se cree erróneamente que el tipo `String` es un tipo primitivo por analogía con otros lenguajes. En Java no es así, siempre es un objeto.

¿Cuántos tipos de la biblioteca estándar de Java hay? Cientos o miles. Es imposible conocerlos todos.

¿Un array es un objeto? Los arrays los consideraremos objetos especiales, los únicos objetos en Java que no tienen métodos.

Ahora veremos un poco en detalle los tipos de datos que más vamos a usar en nuestros programas.

NÚMEROS ENTEROS

Los tipos `byte`, `short`, `int` y `long` sirven para almacenar datos numéricos enteros (sin decimales) tanto positivos como negativos. Se pueden utilizar constantes literales enteras escritas en decimal, o bien en binario o en octal o en hexadecimal. **Los valores binarios se prefijan con un cero y una b (`0b` ó `0B`), los octales se indican anteponiendo un cero a la izquierda del número (`012`), los hexadecimales anteponiendo un cero a la izquierda y una letra equis (`0x`).** Ejemplos:

```
int numero = 16; // 16 decimal
numero = 0b1000; // 8 en binario
numero = 020; // 20 octal=16 decimal
numero = 0x10; // 10 hexadecimal=16 decimal
```




1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Normalmente un número entero escrito de forma literal se entiende que **es de tipo int salvo si al final se le coloca la letra L**, y en ese caso se entenderá que es de tipo **long**. Ejemplo: 7L (7 de tipo long)

Por último, decir que **se puede usar el carácter subrayado para separar grupos de dígitos en los valores numéricos**:

```
int numero= 0b1000_0011; // 130 en binario
numero= 700_000; // setecientos mil +fácil de leer que 700000
```

No se acepta, en general, asignar variables de distinto tipo. Sí se pueden asignar valores de variables enteras a variables enteras de un tipo superior (por ejemplo asignar un valor **int** a una variable **long**). Pero al revés no se puede:

```
int i = 12;
byte b = i; //Error de compilación, posible pérdida de información
```

La solución es hacer una operación de casting (conversión forzosa) o cast. Esta operación permite convertir valores de un tipo a otro. Se usa así:

```
int i = 12;
byte b = (byte) i; // El (cast) evita el error
```

Hay que tener en cuenta en estos castings que si el valor asignado sobrepasa el rango del elemento, el valor convertido no tendrá ningún sentido ya que no pueden almacenar todos los bits necesarios para representar ese número. Ejemplo: ¿Qué valor tendrá b?

```
int i = 1200;
byte b = (byte) i; // El valor de b no tiene sentido
```

NÚMEROS EN COMA FLOTANTE

Los valores numéricos con decimales se almacenan en los tipos primitivos **float** y **double**. Se les llama de coma flotante por como son



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

almacenados por el ordenador. Los decimales no son almacenados de forma exacta, por eso siempre hay un posible error. En los decimales de coma flotante se habla, por tanto de **precisión**. Es mucho más preciso el tipo double que el tipo float. Para escribir valores literales con decimales en el código del programa, hay que tener en cuenta que **el separador decimal es el punto y no la coma**. Es decir para asignar el valor 2,75 a la variable x se haría:

```
x = 2.75;
```

A un valor literal (como 1.5 por ejemplo), se le puede indicar con una f al final del número que es float (1.5f por ejemplo) o una d para indicar que es double. **Si no se indica nada, un número literal con decimales siempre se entiende que es double**, por lo que al usar tipos **float** hay que convertir los literales. Se pueden representar en **notación científica**:

1.345E+3 significaría $1,345 \cdot 10^3$ o lo que es lo mismo 1345. Lógicamente no podemos asignar valores decimales a tipos de datos enteros:

```
int x = 9.5;           // error. Sí podremos mediante un cast
int x = (int) 9.5;     // pero perdemos los decimales (x vale 9)
```

El caso contrario sin embargo sí se puede hacer:

```
int x = 9;
double y = x; //correcto
```

Nota: Al declarar variables globales de tipo primitivo numérico del tipo que sea, si no se indican valores iniciales, Java asigna el valor cero de manera automática. No a las variables locales.

BOOLEANOS

Los valores booleanos (o lógicos) sirven para indicar si algo es verdadero (true) o falso (false). Si al declarar una variable global



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

booleana no se le da un valor inicial, su valor inicial por defecto es false. Por otro lado, a diferencia del lenguaje C, no se pueden en Java asignar números a una variable booleana (en C, el valor false se asocia al número 0 y cualquier valor distinto de cero se asocia a true). Tampoco tiene sentido intentar asignar valores de otros tipos de datos a variables booleanas mediante casting. Ejemplos:

```
boolean soyDelMadrid; // Valor inicial false si es global
boolean b = true;
boolean b = (boolean) 9; // Error. No tiene sentido en Java
```

CARACTERES

Los valores de tipo carácter sirven para almacenar símbolos de escritura (en Java se puede almacenar cualquier código Unicode). Los valores Unicode son los que Java utiliza para los caracteres. Ejemplo:

```
char letra;
letra = 'C'; // Los valores char van entre comillas, pero son int
letra = 67; // El código Unicode de la C es el 67. Hace lo mismo
```

carácter	significado
<code>\b</code>	Retroceso
<code>\t</code>	Tabulador
<code>\n</code>	Nueva línea
<code>\f</code>	Alimentación de página
<code>\r</code>	Retorno de carro
<code>\"</code>	Dobles comillas
<code>\'</code>	Comillas simples
<code>\\</code>	Barra inclinada (<i>backslash</i>)
<code>\udddd</code>	Las cuatro letras d, son en realidad números en hexadecimal. Representa el carácter Unicode cuyo código es representado por las dddd

También hay una serie de caracteres especiales que van precedidos por el símbolo contrabarra o barra a la izquierda (\), que sirve para **escapar del significado normal de una letra**. Son los que aparecen en



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

la tabla anterior.

2.4.4. CONVERSIÓN DE TIPOS (CASTING).

Ya se ha comentado anteriormente la necesidad de uso del operador de casting para realizar asignaciones entre tipos distintos. Consiste en poner delante de una expresión un tipo encerrado entre paréntesis y eso significa que **quieres convertir o transformar el valor en ese tipo de dato**. Como resumen general del uso de casting, mira estos ejemplos:

```
int a;  
byte b = 12;  
a = b;
```

El código anterior es correcto porque un dato de tipo byte es más pequeño que uno int y Java lo convertirá de forma implícita. Lo mismo pasa de int a double por ejemplo. Sin embargo en:

```
int a = 1;  
byte b;  
b = a;
```

El compilador devolverá error aunque el valor 1 sea válido para un dato byte. Para evitarlo hay que hacer un casting. Eso significa poner el tipo deseado entre paréntesis delante de la expresión.

```
int a = 1;  
byte b;  
b = (byte) a; // correcto
```

En el siguiente ejemplo:

```
byte n1 = 100, n2 = 100, n3;  
n3 = n1 * n2 / 100;
```

Aunque el resultado es 100 y ese valor es compatible con un tipo byte,



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Lo que ocurrirá en realidad es un error. Se debe a que la multiplicación $100 * 100$ da como resultado 10000, es decir un número de tipo int. Aunque luego se divide entre 100, no se vuelve a convertir a byte, ya que en cualquier operación, el tipo resultante siempre se corresponde con el tipo más grande que interviene en la operación. Lo correcto sería por ejemplo:

```
n3 = (byte) (n1 * n2 / 100);
```

A modo de resumen, la siguiente figura muestra mediante flechas azules las conversiones automáticas entre tipos primitivos, que no dan problemas y con flechas rojas las que pueden significar pérdidas de precisión. Las que no están, o necesitan casting o pueden fallar.

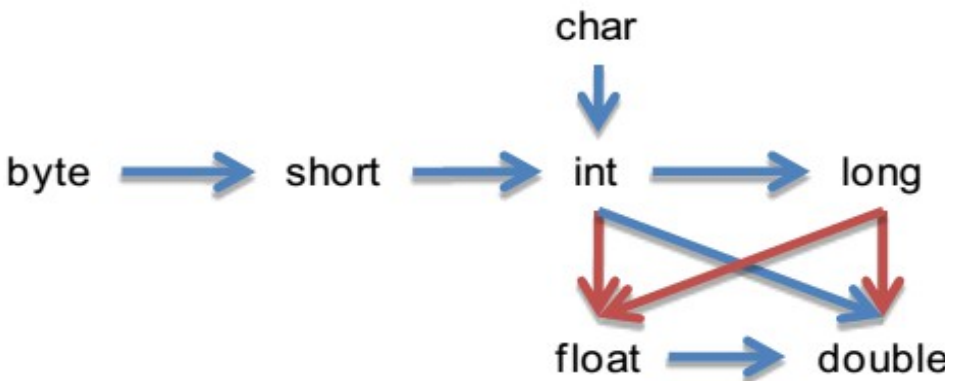


Figura 2.13: Resumen de Conversiones de Tipos primitivos.

2.4.5. OPERADORES Y PRECEDENCIA.

Los datos se manipulan utilizando operaciones con ellos. Los datos se suman, se restan, ... y a veces se realizan operaciones más complejas.

Las operaciones se pueden realizar mediante operadores o mediante funciones. En este apartado comentamos los operadores, que son



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

símbolos que representan una operación con datos.

operadores aritméticos

Son **operaciones matemáticas** que trabajan con datos numéricos:

operador	significado
+	Suma
-	Resta
*	Producto
/	División
%	Módulo (resto)

Hay que tener en cuenta que el resultado de estos operadores varía notablemente si usamos enteros o si usamos números de coma flotante. Por ejemplo:

```
double resultado1, d1= 14, d2= 5;
int resultado2, i1= 14, i2= 5;
resultado1= d1 / d2;
resultado2= i1 / i2;
```

resultado1 valdrá 2.8 mientras que resultado2 valdrá 2. Es más incluso:

```
double resultado;
int i1= 7, i2= 2;
resultado = i1 / i2;           // Resultado valdrá 3
resultado =(double) i1 / i2;  // Resultado valdrá 3.5
```

El operador módulo (%) sirve para calcular el resto de una división entera. Ejemplo:

```
int resultado, i1= 14, i2= 5;
resultado = i1 % i2;           // El resultado será 4
```

El módulo sólo se puede utilizar con tipos enteros (byte, short, int y long).

OPERADORES CONDICIONALES Y RELACIONALES



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Los condicionales sirven para comparar valores. Los relacionales sirven para poder hacer condiciones complejas uniendo varias condiciones simples. Siempre devuelven valores booleanos. Son:

operador	significado
<	Menor
>	Mayor
>=	Mayor o igual
<=	Menor o igual
==	Igual
!=	Distinto
!	No lógico (NOT)
&&	"Y" lógico (AND)
	"O" lógico (OR)

Los operadores lógicos (**AND**, **OR** y **NOT**), sirven para evaluar condiciones complejas. **NOT** sirve para negar una condición. Ejemplo:

```
boolean mayorDeEdad, menorDeEdad;  
int edad = 21;  
mayorDeEdad = edad >= 18;    // mayorDeEdad será true  
menorDeEdad = !mayorDeEdad;  // menorDeEdad será false
```

El operador **&&** (**AND**) sirve para evaluar dos expresiones de modo que si ambas son ciertas, el resultado será **true**, si no, el resultado será **false**. Ejemplo:

```
boolean tieneCarnetConducir = true;  
int edad = 20;  
boolean puedeConducir = (edad >= 18) && tieneCarnetConducir;  
// Si la edad es de al menos 18 años y tieneCarnetConducir es  
// true, puedeConducir es true (ambas cosas: una y la otra)
```

El operador **||** (**OR**) sirve también para evaluar dos expresiones. El resultado será **true** si al menos una de las expresiones es **true**. Ejemplo:

```
boolean nieva = true, llueve= false, graniza= false;
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
boolean malTiempo = nieva || llueve || graniza;
```

OPERADORES DE BITS

Manipulan los bits de los valores que participan. Son:

operador	significado
&	AND
	OR
~	NOT
^	XOR
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda
>>>	Desplazamiento derecha con relleno de ceros
<<<	Desplazamiento izquierda con relleno de ceros

MÁS OPERADORES DE ASIGNACIÓN

Permiten asignar (almacenar) valores a una variable. El operador fundamental es "=". Sin embargo se pueden usar expresiones más complejas como: `x += 3;`

En el ejemplo anterior lo que se hace es sumar 3 a la x (es lo mismo `x+= 3`, que `x= x + 3`). Eso se puede hacer también con todos estos operadores:

```
+=      -=      *=      /=      &=      |=      ^=      %=  
>>=    <<=
```

También se pueden concatenar asignaciones:

```
x1 = x2 = x3 = 5; // todas valen 5
```

Operadores de incremento/decremento

Los operadores "++" (incremento) y "--" (decremento) aumentan y disminuyen respectivamente el valor de una variable numérica la cantidad de 1. Ejemplos:



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
x++; // esto es x = x + 1;  
x--; // esto es x = x - 1;
```

Pero hay dos formas de utilizar el incremento y el decremento. Se puede usar a la izquierda de la variable (++x) o a la derecha (x++). Si la operación se realiza en una variable aislada, da igual.

La diferencia estriba en el modo en el que se comporta el cambio cuando la variable que cambia está participando en una expresión. Si el operador aparece a la izquierda (preincremento/predecremento), primero cambia la variable y luego se utiliza su valor en la expresión.

Si el operador aparece a la derecha, primero se usa en la expresión su valor actual, y luego, a posteriori se cambia el valor. Ejemplo:

```
int x= 5, y= 5, z;  
x++; // da igual ponerlo antes (++x) o después (x++)  
--x; // da igual ponerlo antes (--x) o después (x--)  
z= x++; // (es postincremento) z vale 5, luego x vale 6  
z= ++y + 4; // y vale 6 (preincremento), z vale 10 (6+4)
```

OPERADOR TERNARIO ?

Este operador (conocido como **if** de una línea) permite seleccionar una de dos expresiones según el valor de la primera expresión. Sintaxis:

(expresionLogica)? ExpresionSiVerdadero : expresionSiFalso;

Ejemplo:

```
paga = (edad > 18)? 6000:3000;
```

En este caso si la variable *edad* es mayor de 18, la *paga* será de 6000, si no, será de 3000. Se evalúa una condición y según es cierta o no se devuelve un valor u otro. Nótese que esta función ha de devolver un valor y no una expresión correcta. Es decir, no funcionaría:

```
(edad > 18)? Paga = 6000 : paga = 3000; // ERROR!!!!
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Lo destacado en amarillo es una sentencia, y lo que debe ir ahí es una expresión (una fórmula). Una sentencia puede contener expresiones (variable= expresión;) y una expresión puede contener operadores y datos (operandos), pero no otras sentencias.

PRECEDENCIA DE OPERADORES.

A veces hay expresiones con operadores que resultan confusas. Por ejemplo en:

```
resultado = 8 + 4 / 2;
```

Es difícil saber el resultado. ¿Cuál es? ¿seis o diez? La respuesta es 10 y la razón es que el operador de división siempre precede en el orden de ejecución (se ejecuta antes) que el de la suma. Es decir, siempre se ejecuta antes la división que la suma.

nivel	operador		
1	()	[]	.
2	++	--	~ !
3	*	/	%
4	+	-	
5	>>	>>>	<< <<<
6	>	>=	< <=
7	==	!=	
8	&		
9	^		
10			
11	&&		
12			
13	?:		
14	=	+=, -=, *=, ...	

En cada lenguaje se establece un orden de ejecución de sus operadores, a este orden se le llama precedencia. Si este orden prefijado en ciertas situaciones no te interesa, siempre puedes romperlo usando paréntesis. **Todo lo que haya dentro de un**



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

paréntesis se ejecuta antes de lo que hay fuera. Ejemplo:

```
resultado = (8 + 4) / 2; // Ahora no hay duda, el resultado es 6
```

En la tabla anterior se representa la precedencia usada en Java. Los operadores con mayor precedencia (se ejecutan antes) están en la parte superior, los de menor precedencia en la parte inferior.

De izquierda a derecha la precedencia es la misma. Es decir, tiene la misma precedencia el operador de suma que el de resta. Esto se presta a confusión, por ejemplo, en la expresión:

```
resultado = 9 / 3 * 3;
```

¿El resultado podría ser 1 ó 9? En este caso el resultado es 9, porque la división y el producto tienen la misma precedencia, por eso, **a igual precedencia, el compilador de Java realiza primero la operación que esté más a la izquierda**, que en este caso es la división. Una vez más los paréntesis podrían evitar este comportamiento si no es lo que necesitas:

```
resultado = 9 / (3 * 3);
```

2.5 CLASES Y OBJETOS DE FÁBRICA.

2.5.1. SUBROUTINAS Y FUNCIONES PRECONSTRUIDAS.

Si declaras variables dentro de un método, las variables son locales al método. Es una buena costumbre acompañar la declaración con un comentario que explique para que se utiliza la variable, o que valor almacena durante la ejecución del código. Ejemplo:

```
double principal; // Capital de dinero a invertir  
double interes; // ganancia debida al interés
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Puedes declarar variables en cualquier parte dentro del método, no solo al principio. Lo normal es declarar las variables importantes al principio y las auxiliares cuando se necesiten.

EJEMPLO 4: Programa que usa clases preconstruidas (de fábrica).

```
/**
 * Esta clase calcula el dinero obtenido como
 * intereses al invertir 17000 euros a un interés
 * del 0.027 durante un año. El interés y la
 * ganancia se imprimen por la salida estándar.
 */
public class Interes {
    public static void main( String[] args ) {
        /* Declarar las variables */
        double principal; // Capital de dinero a invertir
        double ratio;     // El ratio de interés anual
        double intereses; // ganancia al año debida al interés
        /* Hacer cálculos */
        principal = 17000;
        ratio = 0.027;
        interes = principal * ratio; // Calcular interes anual
        principal = principal + interes; // Dinero después de un año
        // (Nota: el nuevo valor reemplaza al anterior)
        /* Mostrar resultados */
        System.out.print("El interés ganado en €: ");
        System.out.println(interres);
        System.out.print("Capital después de un año en €: ");
        System.out.println(principal);
    } // final de main()
} // final de la clase Interes
```

Este programa usa varias llamadas a métodos. Recuerda que un método podía ser una función (si devolvía un valor) o procedimiento en caso contrario y que ambos son un conjunto de sentencias que se agrupan, a las que se da un nombre y opcionalmente se les pasan datos (parámetros) para que realicen una tarea. En el programa de ejemplo hemos usado dos: `System.out.print(dato)` y `System.out.println(dato)`. La diferencia entre ellas es que la segunda añade un salto de línea



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

después de imprimir la información, eso provoca el salto a una nueva línea de la pantalla. Pero `System.out.print()` no añade el salto de línea, así que cuando se imprima otra vez, se continúa en la misma línea.

Los valores que se pasan entre paréntesis es la información que se quiere imprimir. Es un dato que se pasa a la subrutina para que haga algo con él (en este caso mostrarlo por pantalla). A estos datos que se pasan entre paréntesis a un método se les llama **parámetros**. No a todos los métodos hay que pasarles parámetros. Cuando un método no tiene parámetros, se escribe su nombre con una pareja de paréntesis vacíos:

```
subrutina(p1, p2, ....); // llamada a método con parámetros
subrutina(); // llamada a método sin parámetros
```

En Java, no hay métodos definidos de forma aislada, porque es un lenguaje completamente orientado a objetos. Los métodos están contenidos dentro de clases o dentro objetos.

Algunas clases son parte del lenguaje Java y existen solamente para ofrecer a los programadores métodos de apoyo. También los objetos, como cualquier String, ofrecen muchos métodos prefabricados.

MÉTODOS DENTRO DE CLASES. EJEMPLO: Math y System

Hablemos primero de los métodos de las clases. Uno de los propósitos de las clases es agrupar juntos algunas variables y métodos. Una clase es un contenedor de variables (datos) y comportamiento (métodos). Hay clases que tienen tanto variables como métodos estáticos. Por ejemplo, un programa es una clase. La rutina `main()` es un miembro de la clase, por eso escribimos la palabra reservada `static` cuando escribimos un programa.

```
public static void main(...) // static significa de la clase
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Cuando una clase tiene variables o métodos estáticos y queramos usarlas desde el exterior de la clase, el nombre de la clase es parte del nombre completo de la variable o método. Por ejemplo, la clase **System** contiene un método estático llamado **exit()**. Para usarlo en tus programas, debes llamarlo escribiendo el nombre de la clase que lo contiene (no la contiene un objeto, la contiene una clase, al ser estático). Este método hace que un programa acabe su ejecución y devuelva al sistema operativo un valor numérico que puede servir de indicador o de "chivato" del motivo por el que ha finalizado. Además, la máquina virtual que lo ejecuta también acaba su ejecución:

```
System.exit(0); // El programa ha acabado de forma correcta
System.exit(1); // El programa acaba por la causa 1
```

System solo es una de las muchísimas clases que vienen preconstruidas en Java. Otra clase muy usada es **Math**. Esta clase tiene constantes estáticas como **Math.PI** y **Math.E** que son las aproximaciones a los números π y e . **Math** también contiene un gran número de operaciones matemáticas. De forma que cuando necesites realizar operaciones más complejas que la suma, resta, etc. que ofrecen los operadores aritméticos, esta clase te ofrece sus variables y métodos. Algunas funciones que la clase **Math** te ofrece son:

- **Valor absoluto:** **Math.abs(x)**
- **Funciones trigonométricas:** **Math.sin(x)**, **Math.cos(x)** y **Math.tan(x)** x representa un ángulo expresado en radianes, no grados. La equivalencia: $360 \text{ grados} = 2\pi \text{ radianes}$. Las funciones son el seno, coseno y tangente de un ángulo.
- **Funciones trigonométricas inversas:** **Math.asin(x)**, **Math.acos(x)** y **Math.atan(x)**. El valor devuelto se expresa en radianes, no en grados. Las funciones son el arcoseno, arcocoseno y arcotangente.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- **Función exponencial y logarítmicas:** `Math.exp(x)` calcula el número e elevado a x, y el logaritmo natural (en base 10) lo calcula `Math.log10(x)` y el logaritmo neperiano es `Math.log(x)`.
- **Función potencia:** `Math.pow(x,y)` calcula x elevado a y.
- **Función raíz cuadrada:** `Math.sqrt(x)`
- **Funciones de Redondeo:** `Math.floor(x)`, redondea x al entero menor o igual a x. Por ejemplo, `Math.floor(3.76)` es 3.0. `Math.round(x)` devuelve el entero más cercano a x. `Math.ceil(x)` redondea x al entero mayor que x.
- **Funciones para generar números aleatorios:** `Math.random()`, devuelve un n° double, aleatorio, $0.0 \leq \text{Math.random()} < 1.0$ que sigue una distribución uniforme (todos los valores del intervalo [0,1) tienen la misma probabilidad de salir).

Los parámetros que se le pasan a los métodos de Math pueden ser de cualquier tipo numérico (pueden ser expresiones en vez de valores literales) y los valores devueltos suelen ser de tipo double.

Observa que `Math.random()` no tiene parámetros, otro ejemplo de un método sin parámetros es `System.currentTimeMillis()`, que devuelve el número de milisegundos que han pasado desde el 1 de enero de 1970 a las 00:00 horas (devuelve un valor de tipo long).

EJEMPLO 5: Programa de ejemplo que usa los métodos matemáticos:

```
/**
 * Usar cálculos matemáticos, mostrar resultados e indicar
 * el tiempo que se ha tardado en hacerlo.
 */
public class UsarMath {

    public static void main(String[] args) {
        long tiempoInicio; // t cuando comienza, en milisegundos
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
long tiempoFinal; // t cuando acaba, en ms
double tiempo;    // Diferencia de tiempo en segundos

tiempoInicio = System.currentTimeMillis();
double ancho, alto, hipotenusa; // lados de triángulo
ancho = 42.0;
alto = 17.0;
hipotenusa = Math.sqrt( ancho * ancho + alto * alto );
System.out.print("Hipotenusa de triángulo de catetos de 42 y 17 ");
System.out.println(hipotenusa);
System.out.println("\nEn matemáticas, sin(x) * sin(x) + "
                  + "cos(x) * cos(x) - 1 debe ser 0.");
System.out.println("Lo comprobamos para x = 1:");
System.out.print(" sin(1) * sin(1) + cos(1) * cos(1) - 1 es ");
System.out.println( Math.sin(1) * Math.sin(1)
                  + Math.cos(1) * Math.cos(1) - 1 );
System.out.println("Hay errores de redondeo al hacer cálculos "
                  + " con números reales!");
System.out.print("\nAhora un número aleatorio: ");
System.out.println( Math.random() );
System.out.print("El valor de Math.PI es ");
System.out.println( Math.PI );
tiempoFinal = System.currentTimeMillis();
tiempo = (tiempoFinal - tiempoInicio) / 1000.0;
System.out.print("\nEl tiempo de ejecución en segundos ha sido: ");
System.out.println(tiempo);
} // final de main()
} // final de la clase UsarMath
```

MÉTODOS DENTRO DE OBJETOS.

Ahora vamos con operaciones que nos ofrecen los objetos.

STRING

Vamos a usar **String** que es la clase de las cadenas de texto. Cada cadena de texto de un programa Java es por tanto un objeto de la clase **String**. El dato que contiene el objeto es la secuencia de letras de la cadena de texto. También contiene métodos, que en este caso son funciones también. Por ejemplo tiene una que se llama **length()** y



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

que calcula y devuelve la longitud (la cantidad de letras) que tiene la cadena de texto. Ejemplo:

```
String aviso;  
aviso = "Aprovecha el día!";
```

La función **aviso.length()** devuelve 16, que es la cantidad de letras que tiene la cadena. También las podemos usar con literales. Ejemplo:

```
System.out.println( "Hola mundo".length() );
```

Ahora te indico algunas de las funciones que implementan los objetos, para describir las suponeremos que *s1* y *s2* son variables o valores String:

- **s1.equals(s2)** devuelve true si *s1* es la misma secuencia de caracteres que *s2* y false en otro caso.

IMPORTANTE:

Los objetos en Java (incluidas las cadenas de texto, porque son de tipo String) **no se comparan con el operador ==**.

Si lo haces, estás comprobando si son el mismo objeto, no si son equivalentes o tienen el mismo contenido. Ejemplo:

```
"Hola" == "Hola"           // Mal: podría devolver false  
"Hola".equals("Hola")      // Bien: devuelve true
```

- **s1.equalsIgnoreCase(s2)** como **equals()**, pero sin diferenciar entre mayúsculas y minúsculas.
- **s1.length()**, el nº de letras en la cadena *s1*.
- **s1.charAt(n)**, donde *n* es un entero, devuelve el carácter que ocupa la posición *n* comenzando por la izquierda, siendo 0 la posición del primer carácter de la cadena. Por tanto en cualquier cadena *s1*, el primer carácter es **s1.charAt(0)**, el segundo es **s1.charAt(1)** y el último es



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

`s1.charAt(s1.length() - 1)`. Si indicas un `n` fuera de las posiciones que tienen carácter (menor de 0 o mayor de `s1.length()-1`) se produce un error.

- `s1.substring(n, m)`, `n` y `m` son enteros, devuelve un `String` que tiene los caracteres de `s1` desde las posiciones `n`, `n+1`, ... `m-1`.
- `s1.substring(n)` devuelve la subcadena de `s1` desde el que ocupa la posición `n` hasta el final.
- `s1.indexOf(s2)` devuelve un número entero. Si la cadena o el carácter `s2` está dentro de `s1`, devuelve la posición donde aparece por primera vez, o devuelve -1 si no está dentro de `s1`.
- `s1.indexOf(x, N)`, devuelve la posición de `x`, si al buscarlo a partir de la posición `N` aparece en `s1`, o devuelve -1 si no se encuentra.
- `s1.lastIndexOf(x)`, devuelve la posición donde aparece `x` por última vez en `s1`, o devuelve -1 si no está.
- `s1.compareTo(s2)` devuelve un entero que indica el resultado de comparar `s1` con `s2`. Si el valor es 0, `s1` y `s2` son iguales. Si `s1` es menor que `s2`, devuelve un número negativo (menor que 0). Si `s1` es mayor que `s2`, devuelve un número mayor que cero.
- `s1.toUpperCase()` devuelve un `String` con los mismos caracteres que `s1`, pero convertidos a mayúscula.
- `s1.toLowerCase()` devuelve un `String` con los mismos caracteres que `s1`, pero convertidos a minúscula.
- `s1.trim()` devuelve un `String` donde los caracteres no imprimibles (espacios, tabuladores, saltos de línea) se han eliminado del principio y del final de `s1`.
- `s1.matches(s2)`: `s2` contiene una **expresión regular** que define un patrón. Si `s1` concuerda o casa con el patrón indicado en `s2`, devuelve `true`. En otro caso devuelve `false`. Por ejemplo: si queremos comprobar si un `string` contiene exactamente 8



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

dígitos numéricos, podemos usar la expresión regular "\\d{8}" o también "[0-9]{8}"

- **s1.split(String er)**: separa s1 en un array de Strings usando la expresión regular er como separador. Si quieres usar metacaracteres de las expresiones regulares como separador (caracteres \\^\$.|?+(){}[]) debes prefijar la cadena con "\\".

Nota: ten en cuenta que en las funciones `s1.toLowerCase()`, `s1.substring(3)`, `s1.trim()`, etc. la cadena `s1` no cambia, se devuelve un nuevo String. Si quieres cambiarlo debes usar la asignación: `s1 = s1.trim()`; por ejemplo.

Los objetos String pueden manipularse también con operadores. Puedes usar el operador suma (+) que sirve para concatenar (unir o pegar) cadenas de texto. Por ejemplo: "Hola" + "Mundo" se evalúa a "HolaMundo".

Observa que si quieres que quede un espacio entre las dos palabras, el espacio debe formar parte de una de las cadenas de texto. Por ejemplo: "Hola " + "Mundo" se evalúa a "Hola Mundo".

En la suma de Strings puedes usar Strings literales o variables. Por ejemplo:

```
String nombre = "Jose";
System.out.println("Hola, " + nombre + ". Bienvenido!");
```

Además puedes concatenar variables o valores de cualquier tipo con un String, el otro valor se convierte a String y se concatena. Por ejemplo, "Número" + 42 se evalúa a "Número42". Y todas estas sentencias:

```
System.out.print("Después de ");
System.out.print(anyos);
System.out.print(" años, el valor es ");
System.out.print(principal);
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Son equivalentes a esta otra:

```
System.out.print("Después de " +anyos + " años, el valor es " +principal);
```

CLASES PARA FECHAS

Inicialmente se utilizaban objetos de la clase `java.util.Date` para trabajar con fechas. Vemos un resumen de constructores y métodos (fuente API de Java 8):

Constructores:

- `Date()` crea una fecha que almacena el momento de tiempo en que se crea medido como los milisegundos pasados desde 1/1/1970.
- `Date(int year, int month, int dia)` *ahora se prefiere usar `Calendar.set(year + 1900, month, dia)` o `GregorianCalendar(year + 1900, month, date)`.*
- `Date(int year, int month, int dia, int hrs, int min)`
- `Date(int year, int month, int dia, int hrs, int min, int sec)`
- `Date(long date)` crea un objeto fecha a partir del long que contendrá el nº de milisegundos desde el 1/1/1970 00:00:00 GMT.
- `Date(String s)` *replaced by `DateFormat.parse(String s)`.*

Métodos

- boolean `after(Date when)` true si la actual es posterior a la pasada como parámetro.
- boolean `before(Date when)` Tests if this date is before the specified date.
- `Object clone()` Return a copy of this object.
- int `compareTo(Date anotherDate)` Compares two Dates for ordering.
- boolean `equals(Object obj)` Compares two dates for equality.
- int `getDate()` *replaced by `Calendar.get(Calendar.DAY_OF_MONTH)`.*
- int `getDay()` *replaced by `Calendar.get(Calendar.DAY_OF_WEEK)`.*
- int `getHours()` *replaced by `Calendar.get(Calendar.HOUR_OF_DAY)`.*
- int `getMinutes()` *replaced by `Calendar.get(Calendar.MINUTE)`.*
- int `getMonth()` *replaced by `Calendar.get(Calendar.MONTH)`.*



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- `int getSeconds()` replaced by `Calendar.get(Calendar.SECOND)`.
- `long getTime()` Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.
- `int getTimezoneOffset()` replaced by
`(Calendar.get(Calendar.ZONE_OFFSET) +
Calendar.get(Calendar.DST_OFFSET)) / (60 * 1000)`.
- `int getYear()` replaced by `Calendar.get(Calendar.YEAR) - 1900`.
- `int hashCode()` Returns a hash code value for this object.
- `static long parse(String s)` replaced by `DateFormat.parse(String s)`
- `void setDate(int date)` replaced by
`Calendar.set(Calendar.DAY_OF_MONTH, int date)`.
- `void setHours(int hours)` As of JDK version 1.1, replaced by
`Calendar.set(Calendar.HOUR_OF_DAY, int hours)`.
- `void setMinutes(int minutes)` replaced by
`Calendar.set(Calendar.MINUTE, int minutes)`
- `void setMonth(int month)` replaced by `Calendar.set(Calendar.MONTH, int month)`.
- `void setSeconds(int seconds)` replaced by
`Calendar.set(Calendar.SECOND, int seconds)`.
- `void setTime(long time)` sets this Date object to represent a point in time that is time milliseconds after January 1, 1970 00:00:00 GMT.
- `void setYear(int year)` replaced by `Calendar.set(Calendar.YEAR, year + 1900)`.
- `String toGMTString()` replaced by `DateFormat.format(Date date)`, using a GMT TimeZone.
- `String toLocaleString()` replaced by `DateFormat.format(Date date)`.
- `String toString()` Converts this Date object to a String of the form: `static long UTC(int year, int month, int date, int hrs, int min, int sec)` replaced by `Calendar.set(year + 1900, month, date, hrs, min, sec)` or `GregorianCalendar(year + 1900, month, date, hrs, min, sec)`, using a UTC TimeZone, followed by `Calendar.getTime().getTime()`.

Como la clase Date está "Deprecated" que vendría a indicar Obsoleta,



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

actualmente se prefiere utilizar en los nuevos programas la clase **Calendar** o **GregorianCalendar**. En los métodos de Date te he dejado el método equivalente. Algunas cosas que debes saber:

No se crean objetos **Calendar** con el operador **new**, sino con una llamada a un método suyo llamado **getInstance()**:

```
Calendar ahora = Calendar.getInstance();
```

Define una serie de constantes que debes utilizar para obtener y cambiar la información. Lo mejor es consultar la ayuda de la API de Java cuando quieras conseguir algo, pero aquí tienes algunas:

- static int AM. Value of the AM_PM field indicating the period of the day from midnight to just before noon.
- static int AM_PM. Field number for get and set indicating whether the HOUR is before or after noon.
- static int JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SETEMBER, OCTOBER, NOVEMBER, DECEMBER. Value of the MONTH field indicating the month of the year in the Gregorian and Julian calendars.
- protected boolean areFieldsSet. True if fields[] are in sync with the currently set time.
- static int DATE Field number for get and set indicating the day of the month.
- static int DAY_OF_MONTH Field number for get and set indicating the day of the month.
- static int DAY_OF_WEEK Field number for get and set indicating the day of the week.
- static int DAY_OF_WEEK_IN_MONTH Field number for get and set indicating the ordinal number of the day of the week within the current month.
- static int DAY_OF_YEAR Field number for get and set indicating the day number within the current year.
- static int DST_OFFSET Field number for get and set indicating the daylight saving offset in milliseconds.
- static int ERA Field number for get and set indicating the era,



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

e.g., AD or BC in the Julian calendar.

- static int [FIELD_COUNT](#) The number of distinct fields recognized by get and set.
- protected int[] [fields](#) The calendar field values for the currently set time for this calendar.
- static int [HOURL](#) Field number for get and set indicating the hour of the morning or afternoon.
- static int [HOURL_OF_DAY](#) Field number for get and set indicating the hour of the day.
- protected boolean[] [isSet](#) The flags which tell if a specified calendar field for the calendar is set.
- protected boolean [isTimeSet](#) True if then the value of time is valid.
- static int [LONG](#) A style specifier for [getDisplayName](#) and [getDisplayNames](#) indicating a long name, such as "January".
- static int [MILLISECOND](#) Field number for get and set indicating the millisecond within the second.
- static int [MINUTE](#) Field number for get and set indicating the minute within the hour.
- static int [MONDAY](#), [TUESDAY](#), [WEDNESDAY](#), [THURSDAY](#), [FRIDAY](#), [SATURDAY](#), [SUNDAY](#) Value of the [DAY_OF_WEEK](#) field.
- static int [MONTH](#) Field number for get and set indicating the month.
- static int [PM](#) Value of the [AM_PM](#) field indicating the period of the day from noon to just before midnight.
- static int [SECOND](#) Field number for get and set indicating the second within the minute.
- static int [SHORT](#) A style specifier for [getDisplayName](#) and [getDisplayNames](#) indicating a short name, such as "Jan".
- protected long [time](#) The currently set time for this calendar, expressed in milliseconds after January 1, 1970, 0:00:00 GMT.
- static int [UNDECIMBER](#) Value of the [MONTH](#) field indicating the thirteenth month of the year.
- static int [WEEK_OF_MONTH](#) Field number for get and set indicating the week number within the current month.
- static int [WEEK_OF_YEAR](#) Field number for get and set indicating



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

the week number within the current year.

- static int `YEAR` Field number for get and set indicating the year.
- static int `ZONE_OFFSET` Field number for get and set indicating the raw offset from GMT in milliseconds.

Métodos

- abstract void `add`(int field, int amount) Adds or subtracts the specified amount of time to the given calendar field, based on the calendar's rules.
- Boolean `after`(Object when) Returns whether this Calendar represents a time after the time represented by the specified Object.
- Boolean `before`(Object when) Returns whether this Calendar represents a time before the time represented by the specified Object.
- void `clear`() Sets all the calendar field values and the time value (millisecond offset from the `Epoch`) of this Calendar undefined.
- void `clear`(int field) Sets the given calendar field value and the time value (millisecond offset from the `Epoch`) of this Calendar undefined.
- Object `clone`() Creates and returns a copy of this object.
- int `compareTo`(Calendar anotherCalendar) Compares the time values (millisecond offsets from the `Epoch`) represented by two Calendar objects.
- protected void `complete`() Fills in any unset fields in the calendar fields.
- protected abstract void `computeFields`() Converts the current millisecond time value `time` to calendar field values in `fields[]`
- protected abstract void `computeTime`() Converts the current calendar field values in `fields[]` to the millisecond time value `time`.
- boolean `equals`(Object obj) Compares this Calendar to the specified Object.
- int `get`(int field) Returns the value of the given calendar



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

field.

- `int getActualMaximum(int field)` Returns the maximum value that the specified calendar field could have, given the time value of this Calendar.
- `int getActualMinimum(int field)` Returns the minimum value that the specified calendar field could have, given the time value of this Calendar.
- `Static Locale[] getAvailableLocales()` Returns an array of all locales for which the `getInstance` methods of this class can return localized instances.
- `String getDisplayName(int field, int style, Locale locale)` Returns the string representation of the calendar field value in the given style and locale.
- `Map<String,Integer> getDisplayNames(int field, int style, Locale locale)` Returns a Map containing all names of the calendar field in the given style and locale and their corresponding field values.
- `Int getFirstDayOfWeek()` Gets what the first day of the week is; e.g., SUNDAY in the U.S., MONDAY in France.
- `abstract int getGreatestMinimum(int field)` Returns the highest minimum value for the given calendar field of this Calendar instance.
- `Static Calendar getInstance()` Gets a calendar using the default time zone and locale.
- `Static Calendar getInstance(Locale alocale)` Gets a calendar using the default time zone and specified locale.
- `Date getTime()` Returns a Date object representing this Calendar's time value (millisecond offset from the `Epoch`).
- `Long getTimeInMillis()` Returns this Calendar's time value in milliseconds.
- `TimeZone getTimeZone()` Gets the time zone.
- `Int getWeeksInWeekYear()` Returns the number of weeks in the week year represented by this Calendar.
- `int getWeekYear()` Returns the week year represented by this Calendar.
- `void set(int field, int value)` Sets the given calendar field to the given value.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- `String toString()` Return a string representation of this calendar.

Otras clases para trabajar con fechas: `java.time.LocalDateTime`, `java.time.LocalDate`, `java.time.LocalTime`, `java.time.DateTimeFormatter` que tiene varios formatos predefinidos y permite crear personalizados.

CLASES PARA NÚMEROS ENTEROS GRANDES

Al hacer cálculos, a veces la capacidad de los tipos de datos predefinidos no es suficiente para almacenar los valores que necesitamos en algunas aplicaciones. Los tipos predefinidos suelen coincidir en capacidad y formato con los que manejan las CPU's de los dispositivos (el hardware). Si no es suficiente, el software debe resolver esta debilidad, aunque sea a costa de ralentizar las operaciones. En Java tenemos la clase **BigInteger** para utilizar números enteros de tamaño ilimitado.

EJERCICIO 2: Haz un programa que defina un número entero `v` de valor 9876543210 y calcule su cubo ($v * v * v$) usando tipos `int`, `long` y `double`. ¿Lo calcula bien en todos los tipos? ¿Y si lo elevas a 4?

La clase define algunas constantes y ofrece algunos métodos para crear objetos y realizar operaciones con ellos:

- static `BigInteger ONE` The BigInteger constant one.
- static `BigInteger TEN` The BigInteger constant ten.
- static `BigInteger ZERO` The BigInteger constant zero.
- `BigInteger abs()` Returns a BigInteger whose value is the absolute value of this BigInteger.
- `BigInteger add(BigInteger v)` Returns a BigInteger ($this + v$)
- `BigInteger and(BigInteger v)` Returns a BigInteger ($this \& v$).
- `BigInteger andNot(BigInteger val)` Returns a BigInteger ($this \& \sim v$).
- int `bitCount()` Returns the number of bits in the two's complement



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

representation of this `BigInteger` that differ from its sign bit.

- `int bitLength()` Returns the number of bits in the minimal two's-complement representation of this `BigInteger`, *excluding* a sign bit.
- `BigInteger clearBit(int n)` Returns a `BigInteger` whose value is equivalent to this `BigInteger` with the designated bit cleared.
- `Int compareTo(BigInteger v)` Compares this `BigInteger` with `v`.
- `BigInteger divide(BigInteger v)` Returns a `BigInteger` (`this / val`).
- `BigInteger[] divideAndRemainder(BigInteger v)` Returns an array of two `BigInteger`s containing (`this / v`) followed by (`this % v`).
- `double doubleValue()` Converts this `BigInteger` to a `double`.
- `boolean equals(Object x)` Compares this `BigInteger` with the specified `Object` for equality.
- `BigInteger flipBit(int n)` Returns a `BigInteger` whose value is equivalent to this `BigInteger` with the designated bit flipped.
- `Float floatValue()` Converts this `BigInteger` to a `float`.
- `BigInteger gcd(BigInteger val)` Returns a `BigInteger` whose value is the greatest common divisor of `abs(this)` and `abs(val)`.
- `int getLowestSetBit()` Returns the index of the rightmost (lowest-order) one bit in this `BigInteger` (the number of zero bits to the right of the rightmost one bit).
- `Int hashCode()` Returns the hash code for this `BigInteger`.
- `Int intValue()` Converts this `BigInteger` to an `int`.
- `boolean isProbablePrime(int certainty)` Returns `true` if this `BigInteger` is probably prime, `false` if it's definitely composite.
- `Long longValue()` Converts this `BigInteger` to a `long`.
- `BigInteger max(BigInteger v)` Returns the maximum of this `BigInteger` and `v`.
- `BigInteger min(BigInteger v)` Returns the minimum of this `BigInteger` and `v`.
- `BigInteger mod(BigInteger m)` Returns a `BigInteger` (`this mod m`).
- `BigInteger modInverse(BigInteger m)` Returns a `BigInteger` (`this-1 mod m`).
- `BigInteger modPow(BigInteger exponent, BigInteger m)` Returns a `BigInteger` whose value is (`thisexponent mod m`).



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- `BigInteger multiply(BigInteger v)` Returns a `BigInteger` (`this * val`).
- `BigInteger negate()` Returns a `BigInteger` whose value is `(-this)`.
- `BigInteger nextProbablePrime()` Returns the first integer greater than this `BigInteger` that is probably prime.
- `BigInteger not()` Returns a `BigInteger` whose value is `(~this)`.
- `BigInteger or(BigInteger v)` Returns a `BigInteger` (`this | val`).
- `BigInteger pow(int exponent)` Returns a `BigInteger` (`this^exponent`).
- static `BigInteger probablePrime(int bitLength, Random rnd)` Returns a positive `BigInteger` that is probably prime, with the specified `bitLength`.
- `BigInteger remainder(BigInteger v)` Returns a `BigInteger` (`this % v`).
- `BigInteger setBit(int n)` Returns a `BigInteger` whose value is equivalent to this `BigInteger` with the designated bit set.
- `BigInteger shiftLeft(int n)` Returns a `BigInteger` (`this << n`).
- `BigInteger shiftRight(int n)` Returns a `BigInteger` (`this >> n`).
- int `signum()` Returns the signum function of this `BigInteger`.
- `BigInteger subtract(BigInteger v)` Returns a `BigInteger` (`this - v`).
- boolean `testBit(int n)` Returns true if and only if the designated bit is set.
- byte[] `toByteArray()` Returns a byte array containing the two's-complement representation of this `BigInteger`.
- `String toString()` Returns the decimal `String` representation of this `BigInteger`.
- `String toString(int radix)` Returns the `String` representation of this `BigInteger` in the given radix.
- static `BigInteger valueOf(long v)` Returns a `BigInteger` whose value is equal to that of the specified long.
- `BigInteger xor(BigInteger v)` Returns a `BigInteger` (`this xor val`).

2.5.2. INTRODUCCIÓN A LAS ENUMERACIONES.

Java viene con 8 tipos de datos primitivos predefinidos y una gran cantidad de clases que definen otros muchos tipos de datos como los



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

String. Pero esto no es suficiente para cubrir todas las necesidades con las que un programador debe enfrentarse. Por eso Java (como otros lenguajes de alto nivel) tiene que dar al programador la posibilidad de crear sus propios tipos de datos.

En Java esto se hace principalmente creando nuevas clases. Pero ahora comentamos un caso particular de creación de nuevos tipos de datos, la posibilidad de definir enumeraciones (tipos enumerados).

Técnicamente una enumeración es un tipo de clase, pero como en muchos casos una enumeración solo necesita su formato simplificado, nos imaginaremos una enumeración como un tipo de dato que puede tomar una lista fija de posibles valores que se indican cuando se define el tipo. La sintaxis para definir de forma simplificada la enumeración:

```
enum nombre_del_tipo { lista_de_valores }
```

Esta definición no puede estar dentro de un método. Puedes escribirla fuera de la rutina main() de un programa (o dentro de una clase o fuera). La lista de valores es una lista de identificadores (nombres separados por comas). Por ejemplo, vamos a declarar un tipo que se llame Estaciones y represente las estaciones del año:

```
enum Estaciones { PRIMAVERA, VERANO, OTOÑO, INVIERNO };
```

Por convención, los identificadores se escriben en mayúsculas, el nombre del tipo enumerado con la primera letra en mayúscula (es una clase).

Las constantes simbólicas de la lista de identificadores se consideran números enteros simbólicos, que "están contenidos" en Estaciones, lo que significa que para referirse a ellos debemos usar la notación punteada: Estaciones.PRIMAVERA por ejemplo.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Una vez que el tipo de dato se ha declarado, puedes usarlo para crear variables de ese tipo, igual que usas los tipos `int`, `boolean` o `String`. Por ejemplo:

```
Estaciones vacaciones; // vacaciones es variable de tipo Estaciones
```

Después de declararla puedes asignarle cualquier valor siempre que sea uno de los indicados en la definición. Ejemplo:

```
vacaciones = Estaciones.VERANO;
```

Si en el programa imprimes un valor del tipo enumerado, te saldrá el nombre del identificador sin el tipo en el que está contenido. Ejemplo:

```
System.out.print( vacaciones ); // Se muestra VERANO
```

Como un tipo `enum` es técnicamente una clase, cada valor es técnicamente un objeto y tiene ciertos métodos. Uno de ellos es **`ordinal()`** y devuelve el orden que ocupa en la lista de valores. Ejemplo:

```
int z = vacaciones.ordinal();  
int x = Estaciones.PRIMAVERA.ordinal(); // x vale 0, el primer elemento  
int y = Estaciones.VERANO.ordinal();   // y vale 1, el segundo elemento
```

EJEMPLO 6: Vamos a hacer un pequeño programa de ejemplo que muestra la utilidad de tener la capacidad de crear nuevos tipos de datos. Tendríamos la opción de definirlos como constantes, sin embargo eso nos obliga a incrementar código en salidas y validaciones.

```
public class DemoEnum {  
    enum Dia { DOM, LUN, MAR, MIE, JUE, VIE, SAB }  
    enum Mes { ENE, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT,  
              NOV, DIC }  
    public static void main(String[] args) {  
        Dia tgif; // variable de tipo Dia  
        Mes libra; // variable de tipo Mes  
        tgif = Dia.VIE; // Asignar un dia de la semana  
        libra = Mes.OCT; // Asignar un mes a libra  
        System.out.print("Mi signo es libra, porque he nacido ");  
    }  
}
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
System.out.println(libra); // Muestra OCT
System.out.print("Es el " + libra.ordinal() +
    " mes del año" +
    " (Contando desde 0, claro!)");
System.out.print("Además, " + tgif + " es el " +
    tgif.ordinal() + " día de la semana");
} // final de main()
} // final de la clase DemoEnum
```

Otras operaciones que pueden aplicarse a los tipos enumerados porque las heredan de **java.lang.Enum**:

- **name()**: String con el nombre de la constante. Ej: libre = Dia.DOM; libre.name();
- **toString()**: String con el nombre de la constante. Ej: libre.toString();
- **ordinal()**: entero con la posición del enum según está declarada. Ej: libre.Ordinal(); // si es Dia.DOM vale 0
- **compareTo(Enum otro)**: compara el enum con el parámetro según el orden en el que están declarados. Ej: libre.compareTo(Dia.LUN); // menor que cero
- **values()**: devuelve un array que contiene todos los enum. Ej: Dia.values()[2]; // el valor es Dia.MAR

2.6. ENTRADA Y SALIDA DE DATOS.

La forma más básica de sacar datos desde Java ya la hemos usado, y es la función **System.out.print(x)**, donde x puede ser un valor o una expresión de cualquier tipo. Si el parámetro no es un String, se convierte a String y se imprime por lo que se conoce como salida estándar (normalmente la salida estándar de un programa está asociada a la pantalla de una consola (terminal o ventana de línea de comandos). También puede cambiarse esto desde el sistema operativo y asociarla a un fichero (c:\java EnumDemo > fichero.txt) o a otro



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

programa con una tubería de GNU/Linux (~\$ java EnumDemo | count). Aunque en los programas que usan interfaz gráfica (GUI) debe redirigirse a otro lugar, porque los usuarios no pueden ver esta información (no hay consola).

SALIDA BÁSICA DE DATOS (System.out)

System.out.print(x) y System.out.println(x)

Ambas muestran x por pantalla (por defecto), con la diferencia de quedarse en la misma línea o saltar una línea después de imprimir. De forma que **System.out.println(x)** equivale a:

```
System.out.print(x);  
System.out.println();  
System.out.print( x + "\n" );
```

System.out.printf("cadena de formato", lista_valores)

Si imprimes un número con decimales, como π , su salida será 3.141592653589793 y cuando imprimas cantidades de dinero en euros, obtendrás cifras como 1050.0 o 43.575, etc. Quizás prefieras o necesites controlar la cantidad de decimales que aparecen cuando muestras un número con decimales (por ejemplo, las cantidades de dinero en una factura deben tener siempre exactamente dos decimales). Controlar como se visualizan los valores se llama **formatear la salida**.

La función **System.out.printf()** puede mostrar datos formateados. **El primer parámetro es un String que indica el formato** (la apariencia) con la que escribir los datos a imprimir. **El resto de parámetros que siguen son los datos a imprimir**. Ejemplo de uso:

```
double importe = 321.783;  
System.out.printf( "%1.2f", importe );
```




1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

En la cadena de formato, cada vez que se quiere indicar el formato de un dato se escribe el carácter porcentaje (%). En el ejemplo de arriba, como solo queremos escribir un dato, el porcentaje solo aparece una vez. El especificador de formato del dato es: %1.2f.

Algunos formatos típicos son: %d, %12d, %10s, %1.2f, %15.8e y %1.8g.

Cada formato comienza con el signo de porcentaje (%) y acaba con una letra con indicadores del formato entre ellos. Por ejemplo %d y %12d, donde la letra "d" indica que se escribe un número entero y la cantidad "12" del formato %12d indica la cantidad mínima de espacios que ocupa la salida (ancho). Si el valor mostrado ocupa menos, se rellena con blancos los espacios que faltan hasta llegar a 12. Se dice que la salida está justificada a la derecha porque los espacios de relleno se ponen a la izquierda. Si el valor tiene más de 12 dígitos, se imprimen todos. Un formato %d equivale a %1d. (La "d" viene de decimal, números en base 10).

Puedes usar también una "x" para imprimir el número en hexadecimal.

La letra "s" al final de un formato indica cualquier tipo de valor. Un número como en %20s, indica la cantidad mínima de caracteres a usar. La "s" significa "string," y puede utilizarse para String o para cualquier tipo que se convierta a String.

El formato para valores numéricos con decimales es el más complejo. Una letra "f", como en %1.2f, se usa para indicar un valor en punto flotante. El 1.2 significa que como mínimo se usa un espacio para los dígitos enteros y el 2 es el número de dígitos decimales que aparecen en el valor mostrado. Por ejemplo, %12.3f indica que se muestra un valor numérico flotante con 3 decimales, con un total de 12 espacios, el valor justificado a la derecha.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Los valores flotantes muy grandes o muy pequeños deberían escribirse en notación científica (exponencial), como `6.00221415e23`, que representa "6.00221415 multiplicado por 10 elevado a 23". El especificador `%15.8e` indica un **formato exponencial "e"**, le letra e con 8 dígitos después del punto decimal.

Si usas la **"g"** en vez de la "e", la salida se muestra en **notación científica para valores muy pequeños o muy grandes y en punto flotante para el resto**. En `%1.8g`, el 8 indica el número total de dígitos (incluyendo los de antes del punto decimal y los de detrás).

En valores numéricos puedes usar la coma (,), que provoca que los dígitos se separen en grupos para hacer más sencilla la lectura. En USA, los grupos se separan con comas. Por ejemplo, si `x` es un billón, ejecutar `System.out.printf("%,d", x)` muestra la salida `1,000,000,000`. En otros países, el separador decimal es la coma y el separador de grupos es el punto. **La coma debe ser el primer carácter de formato que aparezca.** Ejemplo: `%,12.3f`.

Si quieres que la **salida se justifique a la izquierda** en vez de a la derecha, debes indicarlo con un **signo menos (-)** en el formato. Ejemplo: `%-20s`.

Además de los especificadores de formato, la cadena de formato puede contener otros caracteres que se imprimen en la salida tal y como aparecen en la cadena. Por ejemplo: si `x` e `y` son variables de tipo `int`, podrías ejecutar:

```
System.out.printf("El producto de %d por %d es %d", x, y, x * y);
```

Si quieres que en la salida aparezca un carácter porcentaje, puedes usar el especificador `%%`. Puedes usar `%n` para mostrar un salto de línea. También puedes usar el carácter de escape, la contrabarra (`\`),



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

para indicar dobles comillas, tabuladores, etc.

FORMATEAR TEXTO CON FORMAT()

Un String también tiene la posibilidad de aplicar formato a su contenido e incrustar campos de diferentes tipos de datos con el formato que quieras. Comparte con printf los indicadores. Por ejemplo:

```
String.format("Hoy es %1$te/%1$tm/%1$tY", new Date() )
```

Crearé una cadena con la fecha de hoy. Los especificadores de formato tienen esta sintaxis (ten en cuenta que los corchetes en [x] indican que x es opcional, no es obligatorio que aparezcan):

%[índice\$][flags][ancho][.precisión]conversión

El significado de cada elemento es:

- **%**: indica que aparece un campo de datos.
- **Índice\$**: es opcional, si no lo indicas los argumentos se usan en el orden en que aparecen `String.format("%d %d", 1, 2)` genera la cadena "1 2". Sin embargo `String.format("%2$d %1$d", 1, 2)` genera la cadena "2 1". Es decir, índice es un número entero que indica el número de argumento detrás de la cadena que quieres usar para mostrar el campo. Te permite usar el mismo varias veces como en el ejemplo, o saltar el orden en que aparecen los campos. El primer argumento es "1\$", el segundo es "2\$", etc.
- **flags**: es opcional, y lo forman una o varias letras que indican como quieres visualizar un argumento. Cada tipo de dato tiene los suyos propios.
- **Ancho**: es un número decimal positivo que indica el número mínimo de caracteres que se generan para el campo. Si al convertirlo no se alcanza, se rellena con espacios en blanco hasta que se alcance.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- **Precision:** decimal no negativo usado para restringir la cantidad de dígitos decimales que se visualizan.
- **Conversión:** es una letra obligatoria que indica cómo debe ser formateado el valor de la expresión. Según el tipo de la expresión podremos usar solo los que sean compatibles. Para el caso de fechas y horas, después de indicar la letra de conversión (t ó T) hay que indicar qué parte de la fecha/hora quieres usar.

La conversión puede agruparse en varias categorías:

- **General:** se aplica a varios tipos de datos.
- **Character:** aplicados a char, Character, byte y short.
- **Integral:** aplicado a tipos numéricos sin decimales como byte, Byte, short, Short, int, Integer, long, Long y BigInteger
- **Floating Point:** aplicada a valores numéricos con decimales como float, Float, double, Double y BigDecimal
- **Date/Time:** aplicada a tipos que pueden representar valores de tiempo como fechas y horas: long, Long, Calendar, Date y TemporalAccessor
- **porcentaje:** sirve para obtener el carácter '%'.
- **Separador:** obtiene separador de línea propio de la plataforma.

Nota: *si hay versiones en mayúscula de una conversión significa que la salida se muestra en mayúscula.*

La siguiente tabla recoge las conversiones disponibles (van detrás de un carácter '%':



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Conversión	Categoría	Descripción
'b', 'B'	general	Si argumento es null, "false". Si es un boolean o <code>Boolean</code> , <code>String.valueOf(arg)</code> . En otro caso "true"
'h', 'H'	general	Si argumento es null, "null". En otro caso devuelve <code>Integer.toHexString(arg.hashCode())</code>
's', 'S'	general	Si argumento es null, "null". En otro caso devuelve <code>argumento.toString()</code> .
'c', 'C'	character	Devuelve el carácter Unicode
'd'	integral	Devuelve un entero decimal
'o'	integral	Devuelve un entero octal
'x', 'X'	integral	Devuelve un entero hexadecimal
'e', 'E'	Flo. point	Notación científica: valor e+/-valor
'f'	Flo. point	Valor decimal con decimales
'g', 'G'	Flo. point	Según el tamaño del valor f ó g
'a', 'A'	Flo. point	Devuelve cadena exadecimal de mantis ay exponente (no para <code>BigDecimal</code>)
't', 'T'	date/time	Indica conversión de tiempo y necesita otra letra
'%'	porcentaje	Devuelve el literal '%' (<code>'\u0025'</code>)
'n'	separador	Devuelve el separador de línea de la plataforma

Las conversiones de tiempo (fecha y hora) necesitan un segundo carácter de conversión para indicar qué parte del dato es la que quieres usar. En esta tabla tienes los que seleccionan un solo campo de la fecha o la hora.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Letra	Descripción
'B'	Nombre completo del mes
'b', 'h'	Nombre del mes abreviado
'A'	Nombre completo del día de la semana
'a'	Nombre abreviado del día de la semana
'C'	Centuria con 2 dígitos. Devuelve de "00" a "99"
'Y'	Año con 4 dígitos
'y'	Últimos dos dígitos del año
'j'	Día del año. De 001 a 366
'm'	Número de mes con 2 dígitos. 01 - 12
'd'	Día del mes con 2 dígitos: 01 - 31
'e'	Día del mes: 1 - 31.
'H'	Horas
'M'	minutos
'S'	segundos
'p'	Antes o después del medio día
'I'	Hora en formato 12 horas
'z' y 'Z'	Timezone y timezone abreviada
'L'	Milisegundos
'N'	nanosegundos

Y esta tabla contiene los modificadores de tiempo que equivalen a varios de ellos combinados:

letra	Descripción
'R'	Reloj formato 24 horas. Como "%tH:%tM"



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

letra	Descripción
'T'	Reloj formato 24 horas. Como "%tH:%tM:%tS"
'r'	Reloj formato 12 horas. Como "%tI:%tM:%tS %Tp"
'D'	Fecha con formato "%tm/%td/%ty"
'F'	Fecha con formato ISO 8601. Como "%tY-%tm-%td"
'c'	Fecha y hora como "%ta %tb %td %tT %tZ %tY". Ejemplo: "Sun Jul 20 16:17:00 EDT 1969"

Flags

La siguiente tabla resume los flags disponibles y con qué categorías de conversiones se pueden utilizar.

Flag	General	carácter	Integral	Floating Point	tiempo	Descripción
'-'	y	y	y	y	y	Justificado a la izquierda
'#'	y	-	y	y	-	Conversión alternativa
'+'	-	-	y	y	-	Siempre incluye signo
' '	-	-	y	y	-	Incluye espacio de relleno en valores positivos
'0'	-	-	y	y	-	Relleno con ceros
','	-	-	y	y	-	Incluye separadores de miles
'('	-	-	y	y	-	Negativos encerrados en paréntesis

ENTRADA BÁSICA DE DATOS CON Scanner.

Por alguna razón, Java nunca se ha preocupado de hacer sencilla la lectura de valores de cierto tipo a los usuarios de un programa, mientras que se acertó al crear el objeto `System.out` para facilitar la



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

salida, el que se encarga de la entrada, el objeto `System.in` es muy primitivo y no muy sencillo de usar.

Hasta Java 5.0, no se simplifica la entrada de datos mediante la clase `java.util.Scanner`. Pero necesita algo de conocimiento de orientación a objetos (no es lo ideal para empezar). Java 6 introduce la clase `Console` para comunicar a los programas con los usuarios, pero tiene sus propios problemas. Así que vamos a comenzar a utilizar la clase `Scanner`.

Para poder crear objetos de la clase `Scanner`, debes importar la definición de la clase antes de que comiences a usarla en tu programa (public class...):

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

Dentro del programa, a nivel de clase (static) o dentro del método `main()`, debes crear un objeto de la clase. Ejemplo:

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

Esta declaración crea una variable llamada `sc` de tipo `Scanner`. A partir de este momento a través de la variable `sc` tienes acceso a una variedad de métodos que te permiten leer datos del usuario. Por ejemplo, la función `sc.nextInt()` lee un valor de tipo `int` y lo devuelve. Si el valor tecleado por el usuario no es un entero correcto, el programa se rompe. El valor tecleado debe estar seguido de un espacio en blanco o un final de línea.

Algunas funciones del objeto de tipo `Scanner`:

- `sc.nextInt()`, comentada antes.
- `sc.nextByte()`, lee un byte.
- `sc.nextShort()`, lee un entero short.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- `sc.nextLong()`, lee un entero long.
- `sc.nextBigInteger()`, lee un entero de precisión arbitraria.
- `sc.nextFloat()`, lee un float.
- `sc.nextDouble()`, lee un double.
- `sc.nextBoolean()`, lee true o false.
- `sc.nextLine()`, lee un String hasta el salto de línea.
- `sc.next()`, lee una palabra (hasta el primer espacio/salto).

Tiene muchos más, como versiones `hasNextInt()`, ... y cambiar los delimitadores, indicar expresiones regulares, etc.

Un ejemplo de uso: un programa que lee los datos de un fichero:

```
import java.util.Scanner;           // Poder usar la clase Scanner

public class InteresConScanner {

    public static void main(String[] args) {
        Scanner stdin = new Scanner( System.in ); // Crear objeto Scanner
        double principal; // valor de interes
        double ratio; // ratio anual
        double interes; // ganancia anual aportada por el interés
        System.out.print("Teclea el capital a invertir: ");
        principal = stdin.nextDouble();
        System.out.print("Teclea el interés anual (con decimales): ");
        ratio = stdin.nextDouble();
        interes = principal * ratio; // Calcular importe
        principal = principal + interes; // Sumar al capital
        System.out.printf("La ganancia es %1.2f€\n", interes);
        System.out.printf("Total después del año %1.2f€\n", principal);
    } // final de main()
} // final de la clase del programa
```

Con Scanner, aparte de que leer la información incorrecta (pides un entero y te escriben una palabra por ejemplo) puede hacer que el programa falle, aparecen ciertos fallos cuando se mezcla la lectura de valores numéricos con cadenas.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Cuando llamas a `nextInt()` y `nextDouble()` y el usuario teclea 5 y pulsa enter, por ejemplo, se deja el salto de línea pendiente de leer, y si a continuación pides un string con `next()` o con `nextLine()`, se encuentra ese salto pendiente de quitar y devuelve la cadena vacía de forma automática, como si el usuario hubiese pulsado enter cuando en realidad el usuario no ha tecleado nada.

EJEMPLO 7: fragmento de código con el problema de pérdida de entradas y su remedio.

```
System.out.print("Cuantos años tiene: ");
edad = sc.nextInt();
System.out.print("Como se llama: ");
nombre= sc.nextLine(); // No leerá el nombre, lo deja vacío
```

Solución: si es seguro que antes de llamar a `nextLine()` has dejado pendiente un salto de línea (por haber llamado a `nextInt()` por ejemplo) puedes eliminarlo haciendo dos llamadas, la primera para quitar el salto pendiente y la segunda para leer lo que teclee el usuario. El código anterior lo puedes cambiar por este otro:

```
System.out.print("Cuantos años tiene: ");
edad= sc.nextInt();
System.out.print("Como se llama: ");
sc.next(); // Este para quitar el salto
nombre= sc.nextLine(); // Ahora si lee lo que teclea el usuario
```

2.7 SENTENCIAS DE DECISIÓN.

Hasta ahora las instrucciones que hemos visto, son instrucciones que se ejecutan secuencialmente, es decir, podemos saber lo que hace el programa leyendo las líneas de izquierda a derecha y de arriba abajo.

Las instrucciones de control de flujo permiten alterar esta forma de ejecución. A partir de ahora habrá sentencias en el código que se



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

ejecutarán o no, dependiendo de una condición.

EXPRESIONES LÓGICAS

Esa condición se construye utilizando lo que se conoce como **expresión lógica**. Una expresión lógica es cualquier tipo de expresión Java que dé un resultado booleano (verdadero o falso). Las expresiones lógicas se construyen por medio de variables booleanas o bien a través de los operadores relacionales (`==`, `>`, `<`, ...) que comparan dos valores formando lo que se conoce como **condición simple**. Ejemplo:

```
int edad = 17;
boolean mayorDeEdad = edad >= 18;
```

Varias condiciones sencillas pueden unirse para formar **condiciones compuestas** mediante los operadores lógicos (`&&`, `||`, `!`). Ejemplo:

```
int edad = 17;
boolean mayorDeEdad = edad >= 18;
boolean tienePermisoConducir = false;
boolean puedeConducir = mayorEdad && tienePermisoConducir;
```

BLOQUES DE SENTENCIAS

El bloque de sentencias es la sentencia compuesta más simple. Su propósito simplemente es agrupar varias sentencias para que funcionen como una única sentencia. La sintaxis de un bloque es:

```
{
    sentencia_o_sentencias;
}
```

EL bloque comienza por una llave quebrada `{` y acaba en otra llave quebrada `}`. Dentro puede no haber nada (bloque vacío) o puede tener una o varias sentencias separadas por puntos y coma u otros bloques de sentencias dentro (anidados).



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Hay un lugar donde es obligatoria la existencia de un bloque: como puedes imaginar, el método **main()** debe tener obligatoriamente un bloque de sentencias. Dos ejemplos de bloques:

```
{
    System.out.print("La respuesta es ");
    System.out.println(respuesta);
}

{ // Este bloque intercambia los valores de x e y
  int temp; // una variable temporal del bloque
  temp = x; // guarda una copia de x en temporal
  x = y;    // Copia el valor de y a x
  y = temp; // Copia el valor de temp a y
}
```

En el segundo bloque se declara una variable. Su ámbito será el bloque, no puedes usarla fuera, es una **variable local** al bloque.

2.7.1 SENTENCIAS DE DECISIÓN ALTERNATIVA.

Sirven para ejecutar algunas sentencias de forma condicional, solo si se dan o se cumplen ciertas condiciones que el programador define.

DECISIÓN SIMPLE

Se trata de una sentencia que, tras evaluar una expresión lógica, ejecuta una serie de instrucciones en caso de que la expresión lógica sea verdadera. Si la expresión tiene un resultado falso, no se ejecutan las sentencias y el flujo continua . Su sintaxis es:

```
if (expresión lógica) {
    instrucciones ...
}
```

Las llaves son obligatorias sólo cuando hay varias instrucciones. En otro caso se puede escribir el **if** sin llaves o con llaves:

```
if (expresión lógica) sentencia;
```

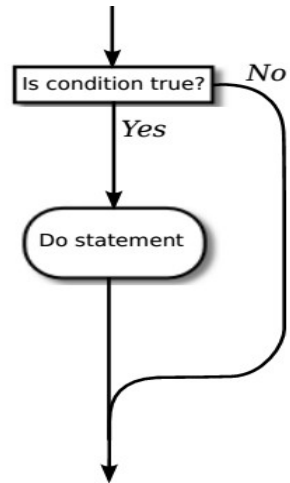


1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Ejemplo: de sentencia if

```
if( nota >= 5 ) {  
    System.out.println("Aprobado");  
    aprobados++;  
}
```

La idea gráfica del funcionamiento del código anterior sería:



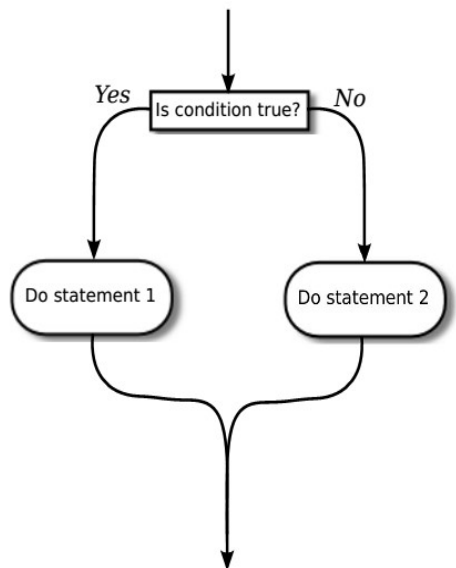
DECISIÓN SIMPLE DOS ALTERNATIVAS EXCLUYENTES.

Igual que la anterior, pero se añade una parte **else** que contiene instrucciones que se ejecutan si la expresión evaluada en el **if** es falsa.

Sintaxis:

```
if (expresión lógica) {  
    instrucciones ...  
}  
else {  
    instrucciones ...  
}
```

Como en el caso anterior, las llaves son necesarias sólo si se ejecuta más de una sentencia.



La representación gráfica de la lógica de la sentencia:



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

EJEMPLO 8: de sentencia if-else:

```
if( nota >= 5 ) {  
    System.out.println("Aprobado");  
    aprobados++;  
}  
else {  
    System.out.println("Suspenso");  
    suspensos++;  
}
```

ANIDACIÓN DE SENTENCIAS

Dentro de una sentencia **if** se puede escribir otra sentencia **if**. A esto se le llama **anidación** y permite crear programas donde se valoran expresiones complejas. La nueva sentencia puede ir tanto en la parte **if** como en la parte **else**. Las anidaciones se utilizan mucho. Sólo hay que tener en cuenta que siempre se debe cerrar primero el último if que se abrió. Es muy importante también tabular el código correctamente para que las anidaciones sean legibles. El código podría ser:

```
if(x == 1) {  
    instrucciones ...  
    if(y > 4) {  
        instrucciones ...  
    }  
}  
else {  
    if( x == 2) {  
        instrucciones ...  
    }  
    else {  
        if(x == 3) {  
            instrucciones ...  
        }  
    }  
}
```

EJERCICIO 3: Examina estos dos fragmentos de código que parecen



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

ser equivalentes. ¿Realmente lo son? Encuentra las diferencias.

```
int y;  
if (x < 0) {  
    y = 1;  
}  
else {  
    y = 2;  
}  
System.out.println(y);
```

```
int y;  
if (x < 0) {  
    y = 1;  
}  
if (x >= 0) {  
    y = 2;  
}  
System.out.println(y);
```

EJERCICIO 4: Vuelve a examinar ahora estos dos fragmentos de código que parece que hacen lo mismo. Averigua el valor de x.

```
int x;  
x = -1;  
if (x < 0)  
    x = 1;  
else  
    x = 2;
```

```
int x;  
x = -1;  
if (x < 0)  
    x = 1;  
if (x >= 0)  
    x = 2;
```

ENCADENAR SENTENCIAS IF

En un solo if puedes como máximo abarcar dos posibilidades (si la expresión es cierta, las sentencias del if y en caso contrario las del else). Pero si tienes un escenario en el que tu programa debe abarcar 3 posibilidades excluyentes, debes anidar sentencias if hasta abarcar todas las posibilidades.

Una forma de hacerlo es: en la parte if contemplas una y en el else el resto de las posibilidades que quedan pendientes:

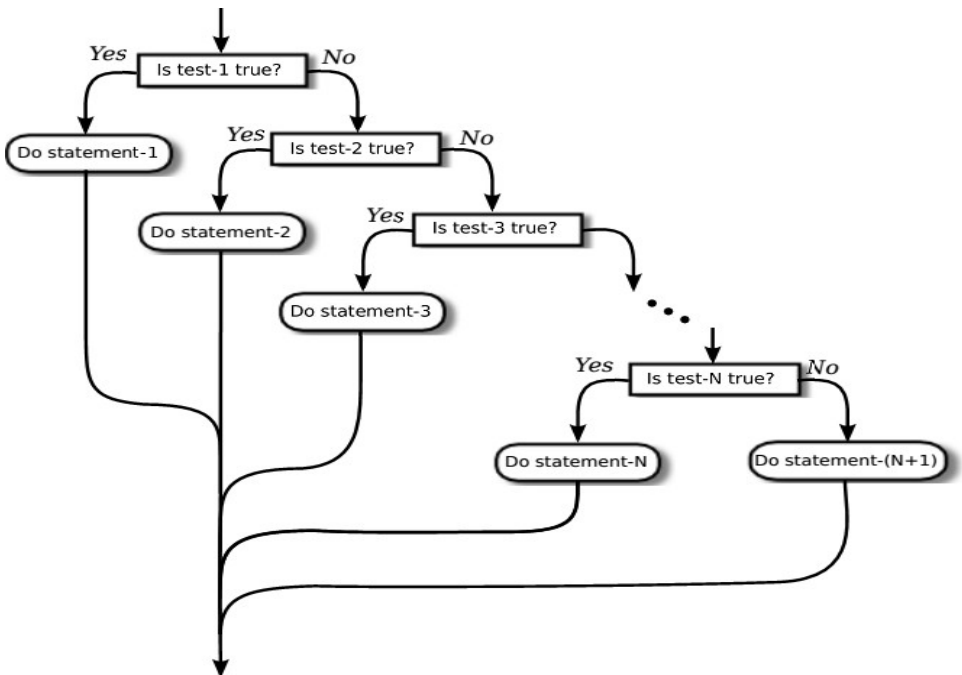
```
if (x == 1) {  
    instrucciones ...  
}  
else {  
    if( x == 2) {  
        instrucciones ...  
    }  
    else {  
        if(x == 3) {
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
        instrucciones ...  
    }  
}  
}
```

Una forma más legible de escribir ese mismo código sería no anidar las sentencias, si no encadenarlas (que la sentencia del último **else** sea otro **if**):



```
if (x==1) {  
    instrucciones...  
}  
else if (x==2) {  
    instrucciones...  
}  
else if (x==3) {  
    instrucciones...  
}
```




1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

Lo que consigues es tener sentencias que pueden realizar una ejecución condicional múltiple, es como tener otra sentencia **if-else-if**, tal y como se ve en el diagrama de la figura anterior.

EJEMPLO 9: Haz un programa en Java que lea 3 números enteros llamados n1, n2, n3 y los imprima en orden ascendente (de menor a mayor valor).

Si en cada **if** solo comprobamos una posibilidad, tenemos 9 posibilidades a contemplar y tendremos que anidar muchas sentencias para contemplar todas las posibilidades. Para abreviar el código, un buen enfoque para resolver este problema es fijarnos en la primera variable y decidir si es la menor, y si no comprobar si es la mayor, y si tampoco, es que está en el medio. En pseudocódigo:

```
SI (n1 < n2 Y n1 < n3) ENTONCES // n1 es el menor
    ESCRIBIR n1, (y las otras en el orden correcto)
SINO SI (n1 > n2 Y n1 > n3) ENTONCES // n1 es mayor
    ESCRIBIR (n2 y n3 en el orden correcto), n1
SINO
    ESCRIBIR n1 entre (n2 y n3 en el orden correcto)
FIN SI // n1 es intermedio
FIN SI
```

Averiguar el orden de n2 y n3 (lo que hay entre paréntesis en cada caso) necesita otra sentencia **if**, así que quedaría:

```
SI (n1 < n2 Y n1 < n3) ENTONCES // n1 es el menor
    SI n2 < n3 ENTONCES
        ESCRIBIR n1, n2, n3
    SINO
        ESCRIBIR n1, n3, n2
    FINSI
SINO SI (n1 > n2 Y n1 > n3) ENTONCES // n1 es mayor
    SI n2 < n3 ENTONCES
        ESCRIBIR n2, n3, n1
    SINO
        ESCRIBIR n3, n2, n1
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
        FINSI
    SINO
        SI n2 < n3 ENTONCES
            ESCRIBIR n2, n1, n3
        SINO
            ESCRIBIR n3, n1, n2
        FINSI
    FIN SI // n1 es intermedio
FIN SI
```

Este enfoque funciona bien incluso si algunos de los valores son iguales.

Nota: observa que en las expresiones lógicas no podemos escribir $n1 < n2$ and $n3$ ($n1$ menor que $n2$ y $n3$) porque el operador "menor que" necesita dos valores a comparar, y el operador "and" necesita dos valores true/false a mezclar. Pero $n3$ en este caso es un número entero, no un booleano. En Java esta expresión da un error sintáctico. En otros lenguajes como C++, no da error, pero se interpreta distinto a como se debe escribir de forma correcta: $n1 < n2$ and $n1 < n3$.

Hay más enfoques alternativos.

```
import java.util.Scanner;

public class Leer3 {

    static public void main( String args[] ) {
        Scanner teclado = new Scanner( System.in );
        int n1, n2, n3;

        System.out.print("Teclee un primer entero n1: ");
        n1= teclado.nextInt();
        System.out.print("Teclee un segundo entero n2: ");
        n2= teclado.nextInt();
        System.out.print("Teclee un tercer entero n3: ");
        n3= teclado.nextInt();
        if (n1 < n2 && n1 < n3) { // n1 es el menor
            if (n2 < n3)
                System.out.println(n1 + " " + n2 + " " + n3);
            else
                System.out.println(n1 + " " + n3 + " " + n2);
        }
        else if (n2 < n1 && n2 < n3) { // n2 es el menor
            if (n1 < n3)
                System.out.println(n2 + " " + n1 + " " + n3);
            else
                System.out.println(n2 + " " + n3 + " " + n1);
        }
        else { // n3 es el menor
            if (n1 < n2)
                System.out.println(n3 + " " + n1 + " " + n2);
            else
                System.out.println(n3 + " " + n2 + " " + n1);
        }
    }
}
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
        System.out.println(n1 + " " + n3 + " " + n2);
    }
    else if (n1 > n2 && n1 > n3) { // n1 es mayor
        if(n2 < n3)
            System.out.println(n2 + " " + n3 + " " + n1);
        else
            System.out.println(n3 + " " + n2 + " " + n1);
    }
    else {
        if(n2 < n3)
            System.out.println(n2 + " " + n1 + " " + n3);
        else
            System.out.println(n3 + " " + n1 + " " + n2);
    }
} // Fin de main
} // Fin de la clase Lee3
```

EJEMPLO 10: Hacer un programa que convierta medidas de longitud de unas unidades de medida (pulgadas, pies, yardas, millas y metros). El usuario tecleará por ejemplo: "17 pies" o "2.73 millas". El programa mostrará la equivalencia con el resto de unidades:

El algoritmo en pseudocódigo es sencillo:

```
Leer número
Leer unidad
SI unidad no es pulgadas ENTONCES
    Convertir número de unidad en pulgadas
    Mostrar pulgadas
FINSI
:
SI unidad no es metros ENTONCES
    Convertir unidad en metros
    Escribir los metros
FINSI
```

El programa, usa otro algoritmo aún más sencillo de implementar:

```
/**
 * Este programa convierte unidades de longitud
 * pulgadas, pies, yardas, millas, metros
 * El usuario teclea: número unidad
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
* El programa muestra la equivalencia con el resto de unidades.
*/
package ejer32;

import java.util.Scanner;

public class ConversorLongitud {

    public static void main(String[] args) {
        Scanner teclado = new Scanner( System.in );
        double valor; // cantidad numérica
        String unidad; // unidad de medida
        double pulgadas, pies, yardas, millas, metros; // valores
        boolean correcta= true; // indica si unidad es Ok
        System.out.println("Telee (valor unidad): ");
        valor = teclado.nextDouble();
        // Quizás debas hacer algo con el escaner?
        unidad = teclado.next();
        unidad = unidad.toLowerCase(); // convertir a minúscula
        if(unidad.equals("pulgadas") || unidad.equals("pulgada")) {
            pulgadas = valor;
        }
        else if ( unidad.equals("pies") || unidad.equals("pie") ) {
            pulgadas = valor * 12;
        }
        else if(unidad.equals("yardas") || unidad.equals("yarda")) {
            pulgadas = valor * 36;
        }
        else if(unidad.equals("millas") || unidad.equals("milla")) {
            pulgadas = valor * 12 * 5280;
        }
        else if(unidad.equals("metros") || unidad.equals("metro")) {
            pulgadas = valor * 39.37;
        }
        else {
            System.out.println("No comprendo \"" + unidad + "\".");
            correcta= false;
            pulgadas= -1; // para evitar un error de valor
        }
        /* Convierte desde pulgadas */
        if (correcta) {
            pies = pulgadas / 12;
            yardas = pulgadas / 36;
            millas = pulgadas / (12 * 5280);
        }
    }
}
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```

        metros = pulgadas / 39.37;
        System.out.printf("%12.5g pulgadas", pulgadas);
        System.out.printf("%12.5g pies", pies);
        System.out.printf("%12.5g yardas", yardas);
        System.out.printf("%12.5g millas", millas);
        System.out.printf("%12.5g metros", metros);
    }
    teclado.close(); // para evitar un warning de Eclipse
} // fin del main()
} // fin clase

```

Ejecución en Eclipse:

```

Telee (valor unidad):
127,40 centímetros
No comprendo "centímetros".

```

```

Telee (valor unidad):
1,5 metros
|      59,055 pulgadas      4,9213 pies      1,6404 yardas  0,00093205 millas      1,5000 metros

```

2.7.2. DECISIÓN MÚLTIPLE.

Se la llama **estructura condicional compleja** porque permite evaluar varias posibilidades a la vez. En realidad sirve como sustituta de expresiones de tipo **if-else-if** (if encadenados). Por tanto debes aplicarla cuando tengas que contemplar en tu programa muchas alternativas, porque el código queda más claro que usar muchos if encadenados. Sintaxis (los corchetes `[]` no se escriben, indican que lo de dentro es opcional):

```

switch (expresión) {
case valor1: instrucciones_del_valor_1;
               [break;]
[case valor2: instrucciones_del_valor_2;
               [break;]]

```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
[...]
default: instrucciones_si_expresión_no_toma_valores_anteriores;
}
```

Esta instrucción evalúa una expresión y según su valor ejecuta instrucciones. Cada **case** contiene un valor de la expresión. Si efectivamente la expresión equivale a ese valor, se ejecutan las instrucciones de ese **case** y de los siguientes.

La instrucción **break** se utiliza para salir del switch. De forma que si queremos que para un determinado valor se ejecuten las instrucciones de un apartado **case** y sólo las de ese apartado, entonces habrá que finalizar ese **case** con un **break**. Si en un **case** no escribes el **break**, y en una ejecución entra, también ejecutará las instrucciones del siguiente **case**, y de los sucesivos hasta que encuentre un **break**. Eso sirve para que varios valores puedan ejecutar las mismas sentencias.

El bloque **default** es opcional y sirve para ejecutar instrucciones para los casos en los que la expresión no se ajuste a ningún **case** de los escritos anteriormente.

EJEMPLO 11: expresar con switch.

```
switch (diaSemana) {
case 1: dia="Lunes";
        break;
case 2: dia="Martes";
        break;
case 3: dia="Miércoles";
        break;
case 4: dia="Jueves";
        break;
case 5: dia="Viernes";
        break;
case 6: dia="Sábado";
        break;
case 7: dia="Domingo";
        break;
}
```

```
// Equivale con if-else-if a:
if(diaSemana == 1)
    dia="Lunes";
else if(diaSemana == 2)
    dia="Martes";
else if(diaSemana == 3)
    dia="Miércoles";
else if(diaSemana == 4)
    dia="Jueves";
else if(diaSemana == 5)
    dia="Viernes";
else if(diaSemana == 6)
    dia="Sábado";
else if(diaSemana == 7)
    dia="Domingo";
else dia="?";
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
default: dia="?";  
}
```

EJEMPLO 12:

```
switch (diaSemana) {  
case 1:  
case 2:  
case 3:  
case 4:  
case 5: laborable=true;  
        break;  
case 6:  
case 7: laborable=false;  
}
```

// Equivale con if-else-if a:

```
if(diaSemana == 1 ||  
   diaSemana == 2 ||  
   diaSemana == 3 ||  
   diaSemana == 4 ||  
   diaSemana == 5)  
    laborable= true;  
else if(diaSemana == 6 ||  
        diaSemana == 7)  
    laborable= false;
```

EJEMPLO 13: Menús y Sentencia switch. Una de las muchas aplicaciones de la sentencia **switch** consiste en ofrecer menús de opciones al usuario de un programa para que elija una de las acciones que puede realizar el programa. Por ejemplo, vamos a hacer un programa que de al usuario la posibilidad de elegir la conversión que quiere realizar.

```
Scanner teclado= new Scanner( System.in );  
int opcion;      // Nº de opción del menú  
double valor;    // valor numérico tecleado por el usuario  
double pulgadas; // El valor convertido a pulgadas  
/* Muestra el menú */  
System.out.println("Escoja las unidades que va a teclear:");  
System.out.println();  
System.out.println(" 1. Pulgadas");  
System.out.println(" 2. Pies");  
System.out.println(" 3. Yardas");  
System.out.println(" 4. Millas");  
System.out.println(" 5. Metros");  
System.out.println();  
System.out.println("Escoja la opción (1-5): ");  
opcion = teclado.nextInt();  
/* Convertir a pulgadas */  
switch ( opcion ) {  
case 1: System.out.println("Teclee el número de pulgadas: ");
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
        valor = teclado.nextDouble();
        pulgadas = valor;
        break;
    case 2: System.out.println("Teclee el número de pies: ");
        valor = teclado.nextDouble();
        pulgadas = valor * 12;
        break;
    case 3: System.out.println("Teclee el número de yardas: ");
        valor = teclado.nextDouble();
        pulgadas = valor * 36;
        break;
    case 4: System.out.println("Teclee el número de millas: ");
        valor = teclado.nextDouble();
        pulgadas = valor * 12 * 5280;
        break;
    case 5: System.out.println("Teclee el número de metros: ");
        valor = teclado.nextDouble();
        pulgadas = valor * 39.37;
        break;
    default: System.out.println("Error! Número de opción incorrecta");
        System.exit(1);
} // fin switch
/* Ahora se convierte a pulgadas, pies, yardas, millas y metros */
```

Ejemplo similar, pero usando un String en la expresión del switch:

```
Scanner teclado= new Scanner( System.in );
int opcion;      // Nº de opción del menú
String unidad;  // Unidades tecleadas por el usuario
double valor;    // valor numérico tecleado por el usuario
double pulgadas; // El valor convertido a pulgadas
/* Leer las unidades del usuario */
System.out.println("Qué unidades de Longitud va a teclear?");
System.out.print("Legal: pulgadas,pies,yardas,millas o metros: ");
unidad = teclado.nextln().toLowerCase();
/* Leer valor y convertir */
System.out.print("Teclee el número de " + unidad + ": ");
valor = teclado.nextDouble();
switch ( unidad ) {
    case "pulgadas": pulgadas = valor;
                    break;
    case "pies":      pulgadas = valor * 12;
                    break;
```




1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
case "yardas":    pulgadas = valor * 36;
                  break;
case "millas":    pulgadas = valor * 12 * 5280;
                  break;
case "metros":    pulgadas = valor * 39.37;
                  break;
default: System.out.println("Unidad de medida incorrecta!");
          System.exit(1);
} // fin switch
```

EJEMPLO 14: Asignación Definida Mediante Switch. El siguiente fragmento de código hace una asignación aleatoria entre 3 posibilidades. Recuerda que usando la expresión `(int)(Math.random() * 3)` tendremos los enteros 0, 1 o 2 seleccionados de forma aleatoria con la misma probabilidad (1/3):

```
String computador;
switch ( (int)(3 * Math.random()) ) {
case 0: computador = "Piedra";
        break;
case 1: computador = "Papel";
        break;
case 2: computador = "Tijeras";
        break;
}
System.out.println("El ordenador saca " + computador); // ERROR!
```

El compilador de Java no se fiará de que la variable `computador` tenga asignado un valor antes de imprimirla. El compilador en muchas ocasiones no es lo suficientemente inteligente para darse cuenta que en este caso, al menos, no hay posibilidad de error. Pero bueno, para dejarlo contento, lo que haremos será añadir a la sentencia **switch** una parte **default**.

```
String computador;
switch ( (int)(3*Math.random()) ) {
case 0: computador = "Piedra";
        break;
case 1: computador = "Papel";
```



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

```
        break;
    case 2: computador = "Tijeras";
        break;
    default: computador= "Piedra";
}
System.out.println("El ordenador saca " + computador); // OK!
```



2.8. EJERCICIOS.

E1. TEST

1. Explica brevemente la diferencia entre la sintaxis y la semántica de un lenguaje de programación o piensa en un ejemplo que lo muestre.
2. ¿Qué efectos tiene ejecutar una sentencia de declaración de variables en el ordenador? Da un ejemplo.
3. ¿Qué es un tipo de dato en un lenguaje de programación?
4. Uno de los tipos primitivos de Java es boolean. ¿Qué es el tipo boolean? ¿Qué valores usa?
5. Indica para qué se utilizan estos operadores de Java:
a) ++ b) && c) != d) % e) ^ f) new g) >> h) | i) = j) ? k) .
6. Explica los efectos de una sentencia de asignación y pon un ejemplo. ¿Cuántas sentencias de asignación conoces?
7. ¿Qué indica la precedencia de operadores? ¿Cómo se calculan las expresiones? ¿Cómo se cambia este comportamiento?
8. ¿Qué es un literal en un programa?
9. En Java, las clases tienen dos propósitos fundamentales ¿Cuáles son?
10. ¿Por qué el resultado de evaluar la expresión `2 + 3 + "test"` es "5test" mientras el valor de `"test" + 2 + 3` es "test23". ¿Cuál es el valor de `"test" + 2 * 3` ?
11. Indica el tipo de dato de estos valores si aparecen en un programa Java.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- | | | | | |
|-----------|-------------|-----------|---------|-----------|
| a) 9.0 | e) 8 | h) 15d | k) 900F | n) 258234 |
| b) '8' | f) "888" | i)"16.0d" | l) 15L | o) "8" |
| c) 0x99 | g) (int)9.1 | j) 1e1 | m) 256f | p) 900L |
| d) 1.0e10 | | | | |

12. Indica el tipo de dato del valor obtenido al evaluar estas expresiones.

- | | | | |
|-------------|------------------|------------------|----------------|
| a) 1/2 | d) 0x9F + 0x3D | g)(int)(5 + 5.0) | j) 5>=2 1<3 |
| b) 6.0 % 9 | e) 1.0 * 1/1 | h) 9 + (double)4 | k) 5 >> 1 2 |
| c) 1.5f + 3 | f)(int)5.0 + 5.0 | i)(double)5+"6" | l) 9F+3D |

13. Explica el motivo de que este trozo de código genere un error de compilación en Java.

```
int x1, x2;  
double y = 1.0;  
x1 = (int)y;  
x2 = 1L;
```

14. Determina el resultado de la variable x al ejecutar este trozo de código Java.

```
double x;  
int y = 90;  
x = y / 100;  
System.out.println("x=" + x);
```

15. Escribe un programa en Java que realice los siguientes pasos.

- Declara una variable **float** llamada **f**.
- Declara una variable **int** llamada **k**.
- Asigna 22.5 a **f**.
- Asigna el valor de **f** a **k**.
- Convierte el valor actual de **f** a **short** y lo imprime por pantalla.

16. Aplica las convenciones de nombres y busca los tipos adecuados con menos peso posible al declarar:

- Para almacenar la edad de una persona.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- b. Para contar los kilómetros que ha recorrido de un coche (hasta 400 mil).
- c. Para almacenar el índice de masa corporal de una persona.
- d. Para saber si un año es o no bisiesto.
- e. El nombre completo (apellidos incluidos) de una persona.
- f. La fecha de contratación de un empleado.
- g. La constante de Fagenbaum que vale 4,66920.

17. Sean a , b y c tres variables enteras que representan las ventas de tres productos A , B y C , respectivamente. Utilizando dichas variables, escribe las expresiones que representen las siguientes afirmaciones:

- a) Las ventas del producto A son las más elevadas.
- b) Ningún producto tiene unas ventas inferiores a 200.
- c) Algún producto tiene unas ventas superiores a 400.
- d) La media de ventas es superior a 500.
- e) El producto B no es el más vendido.
- f) El total de ventas esta entre 500 y 1000.

18. Dada una variable c de tipo carácter, escribe las expresiones que representen las siguientes afirmaciones:

- a) c es una vocal.
- b) c es una letra minúscula.
- c) c es un símbolo del alfabeto.
- d) c es un carácter alfanumérico (una letra o un dígito).

19. Escribe una sentencia `if` para averiguar si un password es correcto:

- a) Si el valor leído en la variable entera password es 1234.
- b) Si el valor leído en la variable password de tipo String es "1234".



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

E2. Escribe un programa en Java que usando `println()` imprima tus iniciales en la salida estándar con letras que ocupen 9x9 puntos y usen un asterisco para iluminar uno de los puntos. Por ejemplo JRR podría ser:

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

E3. Escribe un programa que simule la tirada de un par de dados. Puedes simular la tirada de un dado escogiendo uno de los números enteros 1, 2, 3, 4, 5 y 6 al azar (Mira el método **`Math.random()`**). Por ejemplo:

```
El primer dado saca un 3
El segundo dado saca 5
Tu tirada suma 8
```

E4. Escribe un programa que pregunte al usuario su nombre y lo salude utilizando su nombre, pero en mayúsculas. Ejemplo:

```
Cómo se llama? Toni
Hola, TONI.
```

E5. Escribe un programa que ayude al usuario a contar el dinero de su bolsillo. El programa le preguntará cuantas monedas de 50 céntimos tiene. Luego le preguntará cuántas de 20 céntimos. Luego cuántas de 10 céntimos, cuántas de 5, cuántas de 2 céntimos y cuántas de 1 céntimo. Al final, el programa mostrará cuánto dinero tiene en euros. Ejemplo:

```
Cuántas monedas de 50 céntimos? 2
Cuántas monedas de 20 céntimos? 3
Cuántas monedas de 10 céntimos? 4
Cuántas monedas de 5 céntimos? 3
Cuántas monedas de 2 céntimos? 0
Cuántas monedas de 1 céntimo? 6

En total tiene 2.21 euros
```

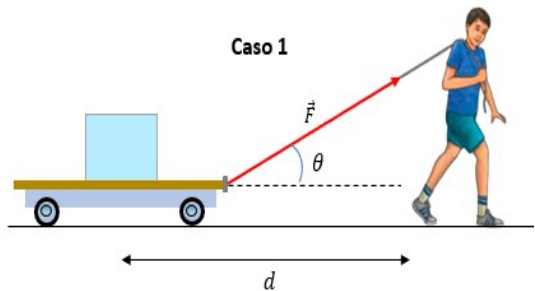


1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

E6. Escribe un programa que pregunte al dueño de una granja cuantos huevos han puesto hoy sus gallinas y calcule cuantos envases grandes necesita (un envase grande tiene 144 huevos), cuantos envases de una docena, cuantos de media docena y cuantos huevos sueltos le quedan sin envasar. (Pista: Si tienes N huevos, tienes $N/12$ docenas de huevos, y te quedan $N\%12$ huevos sueltos, este ejercicio va de las operaciones dividir y módulo). Ejemplo:

```
Cuántos huevos han puesto hoy? 308
Necesitas:
  2 envases grandes.
  1 envase de una docena.
  1 envase de media docena.
  Te quedan 2 huevos sin envasar.
```

E7. Escribe un programa que pregunte por la fuerza (F) ejercida por un operario que tira de una carga, el ángulo (en grados) y la distancia (d) recorrida por la carga en metros, y muestre el trabajo realizado (T) con 6 decimales:



$$T = F \times d \times \cos(\text{ángulo})$$

```
Teclee Fuerza (Newtons): 10.5
Teclee distancia (metros): 2.1
Teclee ángulo (grados): 45
Trabajo = 15,59170453 julios
```

E8. Haz un programa que pregunte al usuario por su nombre y dos apellidos sin partículas (Toni de la Dehesa López → Toni Dehesa López). Lee y almacena toda la respuesta en una variable String. Rompe



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

la cadena en 3 subcadenas, una que contenga el nombre, otra que contenga el primer apellido y otra que contenga el segundo apellido (**Pista:** para hacerlo puedes localizar donde comienzan los apellidos averiguando la posición de los espacios en blanco que separan unas palabras de otras). Luego el programa te muestra cada parte de tu nombre, con las latras que tiene cada una y cuales son tus iniciales. Ejemplo:

```
Teclee su nombre y dos apellidos (sin partículas): José Rosa Rodríguez
Su nombre es JOSÉ y tiene 4 letras.
Su primer apellido es ROSA y tiene 4 letras.
Su segundo apellido es RODRÍGUEZ y tiene 9 letras.
Sus iniciales son JRR.
```

E9. Escribe en un fichero de texto llamado "datos.txt" tu nombre en la primera línea y 3 notas en controles de una asignatura (pueden tener decimales). Haz un programa que lea la información por la entrada estándar y muestre el nombre, las tres notas y la media de las notas.

a) Debes ejecutarlo de forma normal (para que tengas que pasarle la información por teclado).

```
D:\java\e8\java E8
Antonio
4
5
6
Alumno: Antonio
Notas: 4, 5 y 6
Media: 5.00
```

b) Ahora lo ejecutas redirigiendo el fichero a su entrada estándar.

```
D:\java\e8\java E8 < datos.txt
Alumno: Antonio
Notas: 4, 5 y 6
Media: 5.00
```




1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

E10. Escribe un programa que pregunte por los caballos de vapor de un coche y muestre los Kilowatios de potencia que genera el motor, con 2 decimales. La conversión es: 1CV = 735.499 watios

```
Teclee los CV: 100
Equivalen a 73.55 Kw
```

E11. Escribe un programa que pregunte por un DNI (8 dígitos del 0-9) y saque el mensaje "Correcto" si es correcto o "Incorrecto" si no lo es.

- a) Leyendo un valor entero. Piensa en una desventaja.
- b) Leyendo una cadena de texto. Sin usar expresiones regulares en el método **matches()** de los objetos **String**, ni sentencias if-else, for, while, do-while..
- c) Leyendo una cadena de texto. Usando una expresión regular en el método **matches()** de los objetos **String**, pero no sentencias de control: if-else, while...

```
Teclee DNI (8 dígitos): 12345
Incorrecto
```

```
Teclee DNI (8 dígitos): 12345ABC
Incorrecto
```

E12. Lee dos números enteros e imprime el mayor sin usar sentencias de control.

E13. Lee dos números enteros e imprime si el mayor es el primero, si lo es el segundo o si son iguales sin usar sentencias de control.

E14. Lee un CIF (8 dígitos y una letra como 12345678A) e indica:

- Si tiene la longitud correcta (9): true/false.
- Si hay 8 dígitos y una letra: true/false.
- La letra es legal: true/false. Si la letra del dígito de control es



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

una de estas: "TRWAGMYFPDXBNJZSQVHLCKE" es legal, en otro caso no.

- CIF correcto: true/false. El resto de dividir las 8 primeras cifras entre 23 te indica la posición de la letra del dígito de control dentro de la cadena anterior.

No debe provocar errores de ejecución independientemente de la longitud o el formato de la cadena que se lea. Puedes usar if-else.

E15. Pregunta por una cantidad. Si es mayor o igual de 5 le cobras la unidad a 0.5. Si está entre 2 y 4 se la cobras a 0.6. Si es menor de 1 se la cobras a 0.7. Si es menor de 0, cambia por 0. Muestra la cuenta de su compra con 2 decimales de precisión.

E16. Utiliza el método **currentTimeMillis()** de la clase **System** para saber cuantos milisegundos han pasado desde el 1 de enero de 1970 a las 00:00:00 horas hasta el momento de ejecutarlo. ¿Serías capaz de hacer un programa que te diga la hora, los minutos y los segundos aprovechándote de ese valor? Pista: si al valor le quitas los días, te quedan horas, minutos y segundos...

E17. Usa la clase **Date** para crear e imprimir la fecha y hora actual, luego genera e imprime la fecha de noche buena (el 24 de diciembre del año actual) y por último usando su método **getTime()** indica cuantos días quedan por pasar hasta esa fecha.

E18. Usa la clase **java.util.Date** para crear un objeto con la fecha y hora actual. Luego crea un objeto de la clase **SimpleDateFormat** con este formato: "yyyy/MM/dd HH:mm:ss". Por último, imprime la fecha formateada con ese formato.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

E19. Usa ahora la clase `java.util.Calendar` para crear un objeto con la fecha y hora actual. Luego crea un objeto de la clase `SimpleDateFormat` con este formato: `"yyyy/MM/dd HH:mm:ss"`. Por último, imprime la fecha formateada con ese formato pero antes muestra el día de la semana usando del método `get()` de la clase `Calendar`:

E20. Usa la clase `java.time.LocalDateTime` para crear un objeto con su método `now()` y luego imprimir datos con `getYear()`, `getMonthValue()`, `getDayOfMonth()`, `getHour()`, `getMinute()` y `getSecond()`.

E21. Crea un programa que pida por teclado un número decimal llamado `x` e imprima por consola `|x|` (su valor absoluto, es decir si es menor que cero imprime `-x`, y en otro caso `x`).

- a) Usando una sentencia `if`.
- b) Usando el operador ternario.

E22. Crea un programa que pida por teclado el tamaño de un tornillo y muestre por pantalla el texto correspondiente al tamaño, según la siguiente tabla:

de 1 cm (incluido) hasta 3 cm (no incluido) → pequeño
de 3 cm (incluido) hasta 5 cm (no incluido) → mediano
de 5 cm (incluido) hasta 6.5 cm (no incluido) → grande
de 6.5 cm (incluido) hasta 8.5 cm (no incluido) → muy grande

E23. Crea un programa que pida una fecha formada por tres valores enteros (día, mes y año), y determine si la fecha corresponde a un valor válido. Pista: se debe tener presente el valor de los días en función de los meses y de los años. Es decir:

- Los meses 1, 3, 5, 7, 8, 10 y 12 tienen 31 días.
- Los meses 4, 6, 9 y 11 tienen 30 días.



1DAM PROG. UNIDAD 2. Primeros Pasos en Java.

- El mes 2 tiene 28 días, excepto cuando el año es bisiesto, que tiene 29 días.

E24. Crea un programa que pida los coeficientes a y b de una ecuación de primer grado y $(ax+b=0)$ y calcule la solución y la imprima con 4 decimales de precisión. Ten en cuenta que existen tres posibles soluciones:

- Cuando $a \neq 0$ existe la solución única $x = -b/a$.
- Cuando $a = 0$ y $b \neq 0$ no existe solución.
- Cuando $a = 0$ y $b = 0$ existen infinitas soluciones.

E25. Crea un programa que lea un valor entero que representa el día de la semana (1 = lunes, 2 = martes, ... 7 = domingo). Haz que si el valor está entre 1 y 7 escriba por pantalla su nombre o "día incorrecto" en caso de ser un valor fuera del intervalo.

- Usando sentencias if.
- Usando sentencia switch.

E26. Crea un programa que lea un valor entero que representa el día de la semana (1 = lunes, 2 = martes, ... 7 = domingo). Haz que si el valor está entre 1 y 5 escriba por pantalla "laborable" y si está entre 6 y 7 escriba "fin de semana" o en caso de ser un valor fuera del intervalo escriba "día incorrecto" con una sentencia switch.