

Programación en Java



Apuntes

1. Conceptos Básicos.....	4
Introducción.....	4
Un programa sencillo.....	8
Variables y Tipos de Datos	11
Operadores.....	18
Entrada y Salida por Consola	20
Primer Programa.....	22
2. Estructuras de Control: Condiciones.....	23
If...Then...Else	23
Switch	25
Operador ?	27
Operadores Relacionales y Lógicos.....	28
3. Estructuras de Control: Bucles	30
For	30
While.....	31
Do...While	33
Sentencias de Salto	34
Ejemplos de estructuras repetitivas.....	35
4. Cadenas.....	40
String	40
Format	44
StringBuilder y StringBuffer	45
5. Funciones	48
Paso de Parámetros por Valor.....	52
6. Clases y Objetos.....	53
Conceptos Básicos	53
Crear una Clase Paso a Paso.....	56
Modificadores de Acceso.....	61
Enumeraciones.....	63
Clases Interesantes	65
Métodos Comunes a Todas las Clases.....	70
7. Array y ArrayList	75
Clase Array	75
Clase ArrayList	84
Clase Collections.....	92
8. Clases y Herencia.....	93
Conceptos Básicos	93
Herencia	95
9. Polimorfismo.....	103
Tipos de Polimorfismo.....	103
Polimorfismo Puro	103
Sobrecarga de Métodos.....	106
10. Clases Abstractas e Interfaces	107
Clases Abstractas.....	107
Interfaces	110
11. Paquetes.....	114
12. Interfaz Gráfica de Usuario.....	117
Introducción.....	117
Programación Guiada por Eventos.....	117
Creando Una Aplicación Gráfica	118
Componentes Swing	123
Gestión de Eventos	135
Paneles y Layout Managers.....	138
Gráficos y Animaciones	144
El modelo MVC.....	146
13. Excepciones.....	148
Gestión de Excepciones	148
Tipos de Excepciones	150
Throw y Throws	151
Excepciones personalizadas	152
Excepciones Frecuentes	154

14. Ficheros	155
Entrada/Salida de información. Flujos	155
Clases para lectura / escritura de texto	157
Serialización de objetos	163
Entrada/Salida estándar	165
Clase Properties.....	166
15. Colecciones	168
Interface List: Listas.....	169
Interface Set: Conjuntos	172
Interface Map: Mapas	176
Recorrer Colecciones e Iteradores	178
Comparable y Comparator.....	181
Últimas Consideraciones.....	184
16. Orientación a Objetos Avanzada.....	186
Records.....	186
Wrapper Classes (Clases Envoltorio)	187
Expresiones Regulares.....	188
Genéricos	193
Tipos de Clases Especiales	197
17. Tratamiento XML	203
Introducción.....	203
Procesar Archivos XML	204
Leer el árbol DOM	206
Modificar el Árbol DOM	209
Pasar árbol DOM a documento XML.....	212
18. Acceso a Base de Datos	213
Introducción.....	213
Conexión - Desconexión.....	213
Operaciones.....	216
Patrón DAO / Repository	222
SQLite	225
Otros Temas Interesantes	226
Instalación y uso de MySQL.....	228
19. Programación Funcional	235
Interfaces Funcionales	235
Funciones Lambda	239
API Stream.....	242
Tratar Ficheros como Streams.....	245
Últimas Consideraciones.....	246
Anexos	247
Instalación de JDK y NetBeans	247
Diferencias Tipos primitivos vs. Objetos.....	251
Conversiones entre tipos/clases.....	252
Proyecto Lombok	253
Enlaces de Interés	255

Material complementario y videos:

wirtzjava.blogspot.com



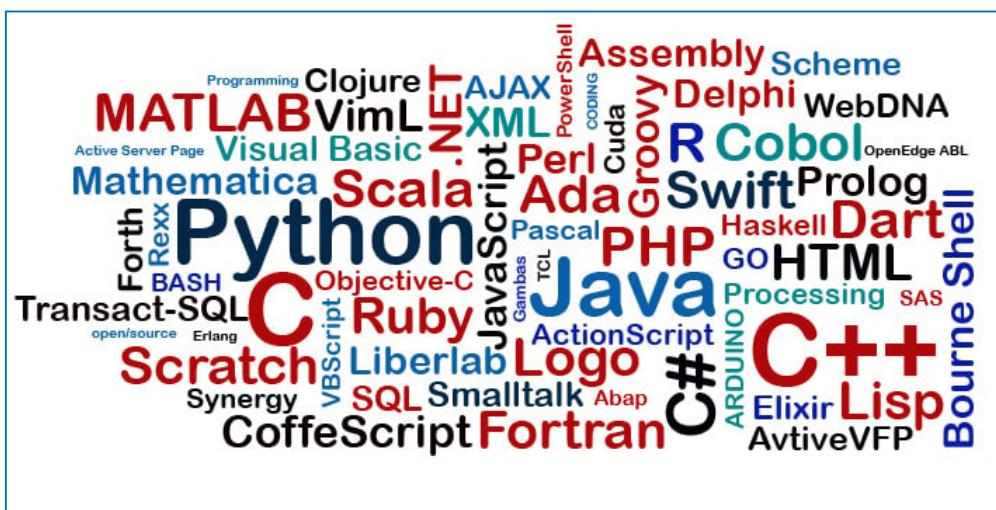
Fernando Rodríguez Diéguez
rdf@fernandowirtz.com
 Versión 2024-11-25

1. Conceptos Básicos

Introducción

La programación es el proceso de escribir instrucciones que un ordenador puede seguir para realizar tareas específicas. Es como darle al ordenador una receta que detalla paso a paso cómo realizar una actividad, desde cálculos simples hasta operaciones complejas en aplicaciones y sistemas. La programación nos permite crear software, que abarca desde aplicaciones móviles y videojuegos hasta sistemas operativos y programas de gestión empresarial.

Un lenguaje de programación es un conjunto de reglas y sintaxis que permiten a los programadores escribir código que una computadora puede entender y ejecutar. Existen muchos lenguajes de programación, cada uno con características particulares que los hacen más adecuados para ciertas tareas o entornos. Ejemplos: JavaScript para ejecutarse en los navegadores web, Kotlin para realizar apps en Android, Swift para apps en IOS, Python en el entorno de la IA, etc.

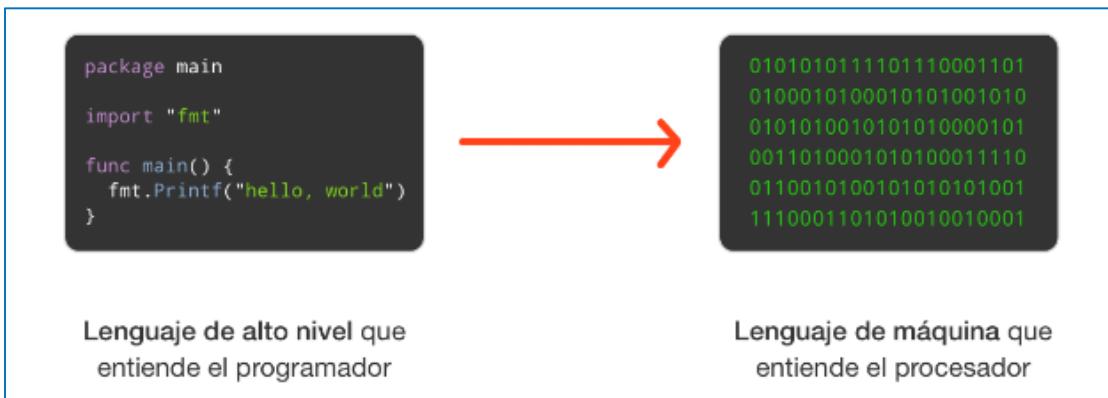


La tarea del programador será escribir el código siguiendo las reglas del lenguaje que elijamos, para su posterior ejecución. Ese código que escribe el programador es lo que denominamos **código fuente**. Por otra parte, entendemos por **código objeto** el resultado de traducir el código fuente a un formato que la computadora pueda ejecutar directamente. Este proceso de traducción puede producirse a través de un compilador o un intérprete, y no es legible por los humanos.

Lenguajes compilados vs. Lenguajes interpretados

En los **lenguajes compilados** el código fuente se traduce completamente a código objeto una sola vez generando un archivo ejecutable (en Windows archivos suelen tener la extensión .exe). Podemos ejecutar ese archivo las veces que queramos sin emplear el código fuente. Esta traducción la realiza un programa llamado compilador y es dependiente del sistema operativo, es decir, si queremos ejecutar un mismo programa en Linux y Windows debemos tener un compilador para cada uno de los dos sistemas operativos y realizar dos compilaciones independientes sobre el mismo código fuente, generando archivos ejecutables completamente diferentes. Ejemplos de lenguajes compilados son C y C++.

Ejemplo: En C, un archivo con extensión *miPrograma.c* (que en realidad es un archivo de texto, pero son sintaxis de C) se compila con un compilador de C para Windows, generando un archivo ejecutable llamado probablemente *miPrograma.exe*.



Por el contrario, en los **lenguajes interpretados**, el código fuente se traduce línea por línea a código objeto durante la ejecución, sobre la marcha, sin generar un archivo de código objeto permanente. Este proceso lo realiza un programa llamado intérprete. Esto hace que los programas de lenguajes interpretados puedan ser un poco más lentos que los compilados.

Como ventaja, no hay que compilar previamente el programa para todos los sistemas operativos en los que deseemos ejecutarlos, es suficiente con que el sistema operativo disponga del intérprete del lenguaje de programación. Ejemplos de lenguajes interpretados son Python y JavaScript.

Ejemplo: En Python, el archivo con extensión .py se ejecuta directamente a través del intérprete de Python del sistema operativo sin generar ningún ejecutable.

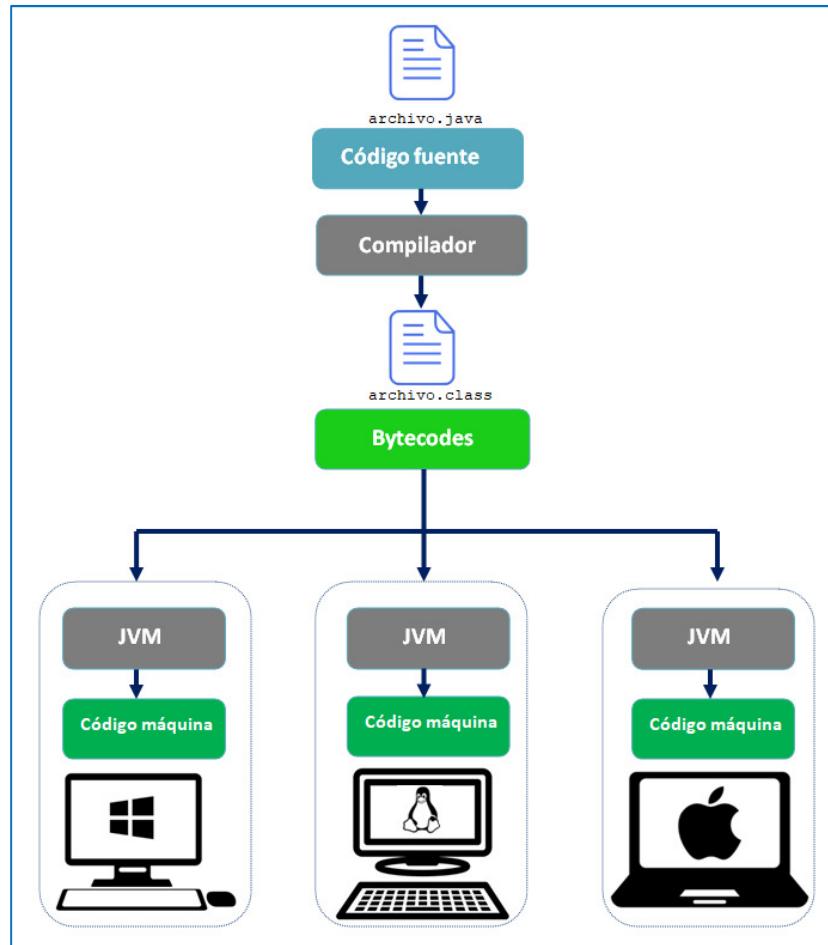
Java: ¿Compilado o Interpretado? La respuesta es que Java es un lenguaje que combina características de ambos modelos como explicaremos en el siguiente apartado.

El lenguaje Java

Java es un lenguaje de programación de propósito general, concurrente, orientado a objetos, que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo (conocido en inglés como WORA, o "write once, run anywhere"), lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra.

Las aplicaciones de Java son compiladas a **bytecode** (clase Java, no ejecutable directamente), que puede ejecutarse en cualquier máquina virtual Java (JVM) sin importar la arquitectura de la computadora subyacente. Si el compilador encuentra algún error en nuestro código nos mostrará un mensaje y si todo está bien el compilador nos creará un archivo con código byte .class, es este archivo el que será ejecutado por la JVM. Lo que tiene que hacer el hardware (+ sistema operativo) es disponer desarrollar una JVM, así podrá ejecutar el programa "semicompileado" .class. Esto hace que el mismo código pueda ser ejecutado en un ordenador Linux, Windows, etc.

Java al ser un lenguaje derivado de C tiene una sintaxis muy similar a este. Existen tres versiones de Java según la plataforma a la que va dirigido: Java SE (*Standard Edition*), Java EE (*Enterprise Edition*) orientada a aplicaciones en red y web y Java ME (*Micro Edition*) orientada a dispositivos pequeños como tabletas y móviles.



JRE y JDK

Al programar en Java, es importante entender dos componentes clave: el JDK y el JRE. Ambos son esenciales para desarrollar y ejecutar aplicaciones Java, pero tienen roles diferentes.

- **JRE (Java Runtime Environment)** es el Entorno de Ejecución de Java. Es lo que necesitas para ejecutar aplicaciones Java en tu sistema operativo. Incluye la Máquina Virtual de Java (JVM) que interpreta y ejecuta el bytecode Java, las librerías estándar necesarias para ejecutar programas y otros componentes como archivos de soporte, configuraciones, recursos, etc. **El JRE es imprescindible para ejecutar los programas Java, pero no incluye herramientas para desarrollar programas.**
- **JDK (Java Development Kit)** es el Kit de Desarrollo de Java. Es un conjunto de herramientas necesarias para desarrollar aplicaciones en Java. **Incluye el JRE** y además contiene el compilador para generar el bytecode y otras herramientas como el *debugger* (ayuda a encontrar y solucionar problemas en el código), el empaquetador de archivos jar, generador de documentación, etc. **El JDK es imprescindible para el programador en Java, pero no es necesario para los usuarios que quieran ejecutar programas Java.**

Existen distintas distribuciones de JDK entre las que destacamos:

- ✓ **Oracle JDK:** es la versión oficial del JDK proporcionada por Oracle Corporation. Incluye características adicionales y optimizaciones específicas que no siempre están presentes en otras distribuciones. Es gratuito para desarrollo, prueba y uso personal, pero requiere suscripción para entornos de producción y comerciales.

- ✓ **OpenJDK:** proporcionado por la comunidad OpenJDK, patrocinada por Oracle y otros contribuyentes con licencia GNU General Public License (GPL). Es la implementación de referencia de Java SE y es de código abierto e incluye las mismas características que Oracle JDK, pero puede carecer de algunas optimizaciones y herramientas adicionales específicas de Oracle.
- ✓ **Amazon Corretto:** otra distribución gratuita similar a OpenJDK proporcionada por Amazon Web Services (AWS).

¿Por qué Java?

Existen múltiples razones por las que es bueno aprender a programar en Java, a continuación, te mencionamos algunas.

- Es el más solicitado en las empresas de nuestra comarca.
- Las aplicaciones empresariales que se ejecutan en los servidores usan en la mayor parte de los casos Spring Framework o .NET. La primera se programa fundamentalmente en lenguaje Java y la segunda en lenguaje C.
- Si quieres comenzar a desarrollar en Android, Java es una base importante que necesitas y debes de aprender.
- Está dentro de los lenguajes más usados en la actualidad y corre en casi todas las plataformas que hay en el mercado.
- Existe gran soporte, documentación y comunidades de Java a las cuales podrás acudir si necesitas ayuda para entender mejor el lenguaje.
- Java también cuenta con una serie de librerías (nativas y de terceros) que amplían sus funcionalidades, desde manipular archivos de Office hasta reconocer huellas digitales y mucho más.
- Es un lenguaje robusto, seguro, fiable y escalable.
- Java no es un lenguaje complicado como se podría pensar, ya que es un tipo de programación orientada a objetos, comprendiendo aspectos básicos de este tipo de programación el aprendizaje de Java será de manera intuitiva.

Extensiones de archivo en Java

- **Archivo .java:** Este archivo contiene el código fuente escrito en el lenguaje de programación Java. Es el archivo de texto que el programador crea y modifica y que será compilado. En realidad, una aplicación Java se compondrá de varios (o muchos) archivos .java.
- **Archivo .class:** Este archivo contiene el *bytecode* generado por el compilador de Java. Es el resultado de compilar el archivo .java. La JVM ejecuta este bytecode para correr el programa. Este bytecode es común para los distintos sistemas operativos.
- **Archivo .jar:** Este archivo es un paquete que puede contener múltiples archivos .class y otros recursos necesarios para ejecutar una aplicación Java como archivos de configuración, elementos multimedia, etc. Es similar a un archivo ZIP y será el formato habitual en el que distribuyamos nuestras aplicaciones Java.

Herramientas necesarias para programar

De acuerdo a lo que hemos comentado, para programar en Java necesitaremos un editor de texto para escribir el código fuente y el compilador para generar el código objeto por lo tanto con un bloc de notas y el JDK sería suficiente.

Aunque la afirmación anterior es correcta, un programador necesitará que la herramienta en la que escribe el código le facilite esta labor, con sugerencias, detección de errores, prueba, etc...

Esas herramientas se denominan **IDE** (*Integrated Development Environment Entorno de Desarrollo Integrado*) y proporcionan, además de un editor de texto, multitud de herramientas ayudan al desarrollo, prueba e implantación de nuestros programas. Entre los más populares podemos citar Netbeans y Eclipse específicos para Java, y Visual Studio Code y IntelliJ Idea de propósito general.



Un programa sencillo

Comencemos viendo un primer programa, aunque no sepamos muy bien el significado de sus instrucciones ni cuál es su finalidad, nos servirá como primer contacto.

```
/* Este es un programa de ejemplo. Nuestro primer contacto con Java.
   Curso de Programación
 */
package ejemplo;
import java.util.Scanner;

public class Ejemplo {
    public static void main(String[] args) {

        int num, cont, resul;

        Scanner teclado = new Scanner(System.in);
        System.out.println("Introduce un número");
        num = teclado.nextInt();
        if (num >= 1 && num <= 9) {
            for (cont=1; cont<=10; cont++) {
                resul = num * cont;
                System.out.println(num + " x " + cont + " es " + resul);
            }
        } else { //las llaves sobrarían ya que solo tiene una instrucción
            System.out.println("Error: el número debe estar entre 1 y 9");
        }
    }
}
```

Espacios en blanco

En Java no es necesario seguir reglas especiales de indentación. Por ejemplo, el programa anterior se podría haber escrito en una línea o todas las líneas alineadas a la izquierda, siempre y cuando hubiera un carácter de espacio en blanco entre cada elemento que no estuviera ya delimitado por un operador o separador. En Java un espacio, tabulador o línea nueva son un espacio en blanco.

Estructura del programa

Como primer contacto debemos abstraernos de muchos elementos que tienen que ver con la orientación a objetos de Java. Podemos decir que en nuestro caso `public class Ejemplo` representa el nombre del programa. En Java el archivo debe llamarse igual que el programa (en realidad, igual que la clase), con la extensión `.java`. Así pues, el archivo del ejemplo anterior ha de llamarse `Ejemplo.java`.

Las primeras líneas, antes de la declaración del programa o clase, serán **package**, que se usa para indicar el contenedor (paquete) en el que está nuestro programa (vendría a ser similar al concepto de carpetas en el sistema operativo), y justo a continuación **import** para incorporar funcionalidades (clases) a nuestro programa que están en otros paquetes y queremos usar en nuestro proyecto, por ejemplo, utilidades para tratar fechas o funciones matemáticas. Estos dos apartados se verán en detalle más adelante.

También se comprueba que los programas se estructuran en bloques. Cada bloque comienza con una llave “{ ” y se cierra con “ }”. Tienen que estar totalmente emparejadas y anidadas una dentro de otra, como puedan ser los paréntesis en fórmulas matemáticas o como cuando usamos funciones dentro de funciones en las hojas de cálculo.

El código que se ejecuta es el que está dentro del bloque **main**, todo programa deberá tener este método y por ahí comenzará la ejecución, aunque luego puede redireccionarse a otros bloques de código que pueden estar en el mismo archivo u en otro. El método `main` tendrá el formato:

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

Las líneas de código se limitan con un punto y coma final salvo en las estructuras de bloque que llevan llaves de apertura o cierre.

Identificadores

Los identificadores se utilizan para los nombres de las variables, clases y de los métodos. Un identificador puede ser cualquier secuencia descriptiva de letras mayúsculas o minúsculas, números, el carácter de subrayado, o el símbolo del dólar. Un identificador no debe empezar nunca con un número, para evitar la confusión con un literal numérico.

Conviene recordar otra vez que Java distingue entre mayúsculas y minúsculas; así, el identificador **Resul** no es lo mismo que el identificador **RESUL** o **resul**.

Cuando definimos un identificador le ponemos justo antes de su nombre el tipo de dato que tiene, por ejemplo: `int num` representa una variable de tipo entero como veremos más adelante.

Comentarios

Podemos poner los comentarios que queramos en nuestros programas, los de varias líneas, que comienzan por `/*` en la primera línea y terminan por `*/` en la última, o los de una sola línea, que se representan por `//` de forma que todo lo que haya a partir de estos caracteres hasta el final de la línea no se tendrá en cuenta.

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

Palabras clave de Java

Existen unas 50 palabras clave reservadas, definidas en el lenguaje Java. Estas palabras clave, combinadas con la sintaxis de los operadores y separadores, forman la definición del lenguaje Java, y no se pueden utilizar como nombres de una variable, clase o método.

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

Ejecución de programas

Compilando el programa

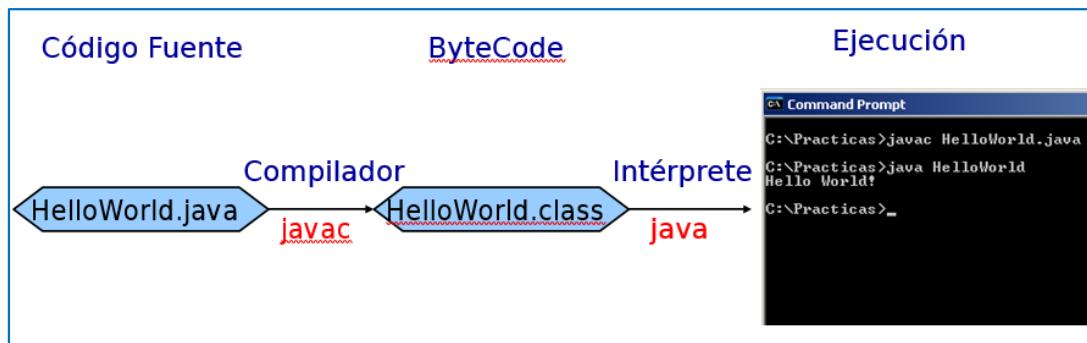
En general, se utilizarán entornos integrados de desarrollo (IDE), como pueden ser Netbeans o Eclipse, que ofrecen multitud de funcionalidades que facilitan el desarrollo y prueba de aplicaciones. Sin embargo, siendo puristas, compilar el programa Ejemplo simplemente necesitamos el compilador, **javac**, que invocaremos desde la consola especificando el nombre del archivo fuente, tal y como se muestra a continuación.

```
C:\>javac Ejemplo.java
```

El compilador **javac** crea un archivo llamado *Ejemplo.class*, que contiene la versión del programa en *bytecode*. El *bytecode* es la representación intermedia del programa que contiene las instrucciones que el intérprete o máquina virtual de Java ejecutará. Por tanto, el resultado de la compilación con **javac** no es un código ejecutable para una determinada plataforma, se podrá ejecutar en cualquiera que disponga de la máquina virtual Java (con el comando **java**):

```
C:\>java Ejemplo
```

La máquina virtual ejecutará las instrucciones que se encuentre en el bloque `main` y desde ahí podrá ejecutar instrucciones que estén en otros bloques de ese mismo archivo o en otros archivos.



Prueba a escribir este primer programa en un bloc de notas, compílalo y ejecútalo. Obviamente debes tener correctamente instalado el JDK en tu ordenador. Puedes ver el anexo final para instalar Netbeans y JDK.

```
package ejemplo;

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

Aunque lo veremos un poco más adelante en detalle, ya podemos adelantar que: `System.out.println ("Mensaje");` nos sirve para mostrar por pantalla, en modo consola, el mensaje que está entre comillas dobles.

Variables y Tipos de Datos

Los tipos primitivos

Java define ocho tipos primitivos: `byte`, `short`, `int`, `long`, `char`, `float`, `double` y `boolean`. Los tipos primitivos son llamados también tipos simples. Estos tipos pueden ser clasificados en cuatro grupos:

- **Enteros:** este grupo incluye a los tipos `byte`, `short`, `int` y `long` para almacenar números enteros positivos y negativos.
- **Números con punto decimal:** este grupo incluye a los tipos `float` y `double`, los cuales representan números con precisión decimal.
- **Caracteres:** el tipo `char` representa símbolos en un conjunto de caracteres, como letras y números.
- **Boolean:** se usa para representar los valores lógicos verdadero y falso.

Estos tipos se pueden utilizar por sí solos, o para construir arrays que veremos adelante o clases propias. Forman, por lo tanto, la base para todos los demás tipos de datos que se puedan crear.

Los tipos primitivos representan valores simples, no objetos complejos. Aunque Java es un lenguaje completamente orientado a objetos, los tipos primitivos no son objetos. Los tipos simples son análogos a los tipos simples existentes en la mayoría de los lenguajes no orientados a objetos.

Veamos cada tipo de dato.

Enteros

Java define cuatro tipos de enteros: `byte`, `short`, `int` y `long`. En todos ellos se considera el signo, valores positivos y negativos. Java no admite valores sin signo. Muchos otros lenguajes soportan enteros con signo y enteros sin signo, pero Java no.

El tamaño y rango de estos tipos enteros puede variar mucho, según se muestra en la tabla siguiente:

Nombre	Tamaño (bits)	Rango
long	64	-9,223,372,036,854,775,808 a 9,223,372,854,775,807
int	32	-2,147,483,648 a 2,147,483,647
short	16	-32,768 a 32,767
byte	8	-128 a 127

Los valores literales se pueden representar en base decimal: 1, 7, -34 pero existen otras dos bases que se pueden utilizar para literales enteros, la octal (base 8) y la hexadecimal (base 16). En Java se indica que un valor es octal porque va precedido por un 0. Por lo tanto, el valor aparentemente válido 09 producirá un error de compilación, ya que 9 no pertenece al conjunto de dígitos utilizados en base 8 que van de 0 a 7.

Una base más utilizada por los programadores es la hexadecimal, que corresponde claramente con las palabras de tamaño de módulo 8 tales como las de 8, 16, 32 y 64 bits. Una constante hexadecimal se denota precediéndola por un cero-x (0x o 0X). Los dígitos que se utilizan en base hexadecimal son del 0 al 9, y las letras de la 'A' a la 'F' (o de la 'a' a la 'f'), que sustituyen a los números del 10 al 15.

¿Es posible asignar un literal entero a alguno de los otros tipos de enteros de Java *byte* o *long*, sin que se produzca un error de incompatibilidad? La respuesta es sí: cuando se asigna una literal a una variable del tipo *byte* o *short*, no se genera ningún error si el valor literal está dentro del rango del tipo de la variable.

También se puede añadir una "L" mayúscula al final del literal para definir que es *long*, por ejemplo: x = 234123333L.

```
-78    // tipo int, dígitos sin punto decimal
034    // en octal (equivale al 28 decimal)
0x1C   // en hexadecimal (equivale al 28 decimal)
875L   // de tipo long
```

Hay que tener cuidado de elegir la variable adecuada para no asignarle valores por fuera de su rango; esta incidencia se denomina *desbordamiento* u *overflow*.

Tipos con punto decimal

Los números con punto decimal, también conocidos como números reales, se utilizan para evaluar expresiones que requieren precisión decimal. Por ejemplo, un importe en euros, que contiene dos decimales para los céntimos. Su tamaño y rango se muestran a continuación:

Nombre	Tamaño (bits)	Rango aproximado
Double	64	4.9e-324 a 1.8e+308
float	32	1.4e-045 a 3.4e+038

Los valores usan el punto como separador decimal, por ejemplo: 3.1416. También admite notación científica, con un sufijo que especifica la potencia de 10 por la que hay que multiplicar el número. El exponente se indica mediante una E o e seguida de un número decimal, que puede ser positivo o negativo; por ejemplo, 6.022E23, 3.14159E-05, y 2E+100.

Los números de punto flotante utilizan por omisión la precisión *double*. Para especificar un literal de tipo *float* se debe añadir una F o f a la constante. También se puede especificar explícitamente un literal de tipo *double* añadiendo una D o d (aunque este último es redundante, ya que por defecto los literales con decimales se crean como *double*).

```
15.2        // de tipo double
1.52e1      // el mismo valor
15.8f       // de tipo float[SEP]= 15.8F
```

Gangnam Style overflows INT_MAX, forces YouTube to go 64-bit

Psy's hit song has been watched an awful lot of times.

PETER BRIGHT - 12/3/2014, 11:32 PM



Go on, watch it.

Although it's no longer 2012, apparently people are still watching the YouTube video for Korean pop star Psy's smash hit song *Gangnam Style*.

The irritatingly catchy tune has racked up so many views that Google has been forced to upgrade YouTube's infrastructure to cope. When YouTube was first developed, nobody ever imagined that a video would be watched more than 2 billion times, so the view count was stored using a signed 32-bit integer.

The maximum value of this number type, 2,147,483,647, is well known to C programmers as INT_MAX. Once INT_MAX is reached, attempting to record another view will normally roll over to -2,147,483,648.

YouTube isn't the only software that this number is a problem for. Unix systems record time values as the number of seconds since 00:00:00 UTC on January 1, 1970. 32-bit systems use a signed 32-bit integer for this, so they will wrap around 2,147,483,647 seconds after that date. Two billion seconds is about 68 years; on January 19, 2038, at 03:14:07 in the morning, 32-bit Unix clocks will roll over.

Caracteres

El tipo de datos que se utiliza en Java para almacenar un solo carácter es **char**. A diferencia de C/C++ que un *char* ocupa un solo byte, en Java ocupa 2 bytes ya que usa codificación Unicode para representar caracteres.

Unicode define un conjunto completo e internacional de caracteres que permite la representación de todos los caracteres que se pueden encontrar en todas las lenguas del mundo. Para ello son necesarios 16 bits. Por este motivo, el tipo *char* de Java es un tipo de 16 bits. El rango de un *char* es de 0 a 65,536. No existen valores de tipo *char* negativos.

El conjunto estándar de caracteres conocido como ASCII tiene un rango que va de 0 a 127 caracteres, y el conjunto extendido de 8 bits, ISOLatin-1, va desde 0 a 255. Java utiliza Unicode para representar caracteres, ya que está diseñado para crear aplicaciones que puedan ser utilizadas en todo el mundo.

Un literal de carácter se representa dentro de una pareja de comillas simples, como, por ejemplo, 'a', 'z', y '@'. Para los caracteres que resulta imposible introducir directamente, existen varias *secuencias de escape* que permiten introducir al carácter deseado como \" para el propio carácter de comilla simple, y \"n' para el carácter de línea nueva. También se puede introducir el valor de un carácter en base hexadecimal (y octal) para ello se escribe la diagonal invertida seguida de una u (\u), y exactamente cuatro dígitos hexadecimales. Por ejemplo, '\u0061' es el carácter ISO-Latin-1 'a', ya que el bit superior es cero. '\ua432' es un carácter japonés.

Secuencia de escape	Descripción
\ddd	Carácter escrito en base octal (ddd)
\xxxxx	Carácter escrito utilizando su valor Unicode en hexadecimal (xxxx)
\'	Comilla simple
\\"	Comilla doble
\\\	Barra invertida (backslash)
\r	Retorno de carro
\n	Nueva línea o salto de línea
\f	Comienzo de página
\t	Tabulador
\b	Retroceso

Los textos los guardaremos en variables de tipo "String", pero son un tipo de datos diferentes. Los que estamos viendo hasta ahora son los tipos "primitivos". String y otros tipos de datos son ya referencias a "objetos"

Boolean

Una variable de tipo *boolean* solo puede tener dos valores, **true** o **false**. Éste es el tipo que devuelven los operadores lógicos tales como *edad > 18*. Es el tipo requerido por las expresiones condicionales que gobiernan las sentencias de control, como *if* y *for*, que veremos más adelante.

Variables

La variable es la unidad básica de almacenamiento en un programa en cualquier lenguaje de programación. Una variable se define mediante la combinación de un identificador, un tipo y un inicializador opcional.

En Java, se deben declarar todas las variables antes de utilizarlas. La forma básica de declaración de una variable es la siguiente:

```
tipo identificador [= valor] [, identificador [= valor] ...];
```

- El tipo es uno de los tipos primitivos de Java, como, por ejemplo: int, float, double (o el nombre de una clase o interfaz, pero eso lo veremos más adelante).
- El identificador es el nombre de la variable. Una convención, aunque no obligatoria, es empezar la variable por una minúscula y luego emplear la mayúscula para indicar un cambio de palabra en la variable, ejemplos: *edadAlumno*, *dniPersona*, *fechaNacimiento*, etc.
- Opcionalmente, se puede inicializar la variable mediante un signo igual seguido de un valor. La expresión a la derecha del igual debe dar como resultado un valor del mismo tipo (o de un tipo compatible) que el especificado para la variable. Para declarar más de una variable del tipo especificado, se utiliza una lista con los elementos separados por comas.

```
int a, A, edad;           // declara tres variables de enteros: a, A, y edad
int d = 3, e, f1 = 5;     // tres enteros más, inicializando d y f1
byte z = 22;              // inicializa z.
double pi = 3.14159;      // declara una aproximación de PI
char x = 'x';             // la variable x tiene el valor 'x'
```

También se puede inicializar mediante un cálculo:

```
int a = 3, b = 2;
int c = a + b;
```

Es importante destacar que el símbolo “=” se usará como asignación de variables no como la igualdad matemática. Así pues: *edad = edad + 1* tiene sentido e indica que a la variable *edad* se le asigna lo que hay a la derecha del igual, que es una suma. Si, por ejemplo, *edad* valiese 18, después de ejecutar esa instrucción valdría 19. La igualdad matemática se representa, como veremos más adelante por “==”, así pues, *edad == edad + 1* se evaluaría como *false* independientemente del valor de *edad*.

Desde Java 17 se pueden definir variables asignándoles un valor y sin especificar explícitamente el tipo y Java lo infiere del valor asignado, pero aún así es un tipo fijo que no puede cambiar posteriormente en el programa. Por eso se dice que Java es fuertemente tipado (a diferencia de otros lenguajes débilmente tipados como JavaScript). Simplemente es una forma más sencilla de escribirlo: var x = 10L; crearía x de tipo *long*.

Ámbito y tiempo de vida de las variables

Hasta el momento, todas las variables que se han utilizado se han declarado al comienzo del método *main()*. Sin embargo, Java permite la declaración de variables dentro de un bloque delimitado por llaves {}.

Como regla general, se puede decir que las variables declaradas dentro de un ámbito no son accesibles al código definido fuera de ese ámbito. Por tanto, cuando se declara una variable dentro de un ámbito, se está localizando y protegiendo esa variable contra un acceso no autorizado y/o modificación.

Las variables se pueden declarar en cualquier punto el programa, pero sólo son válidas después de ser declaradas. Obviamente, no se puede usar una variable en cualquier operación antes de ser definida, ya que no sabría el tipo de datos que tiene.

```

7
8     public class Ejemplo {
9         public static void main(String[] args) {
10            cannot find symbol
11            symbol: variable b
12            location: class Ejemplo
13            ----
14            (Alt-Enter shows hints)
15
16        }
17    }
18

```

En el ejemplo mostrado, `System.out.println(b);` es la instrucción necesaria para mostrar el valor de la variable `b` por pantalla, pero está en un bloque de código (delimitado por las llaves) y esta instrucción está fuera de este bloque.

Las variables se crean cuando la ejecución del programa alcanza su ámbito, y son destruidas cuando se abandona su ámbito. Esto significa que una variable no mantiene su valor una vez que se ha salido de su ámbito. Se recomienda definir las variables al comienzo de cada función o método.

Si definimos una variable inmediatamente después de la línea con el nombre del programa, esa variable será **global**, y esto quiere decir que se podrá usar a lo largo del todo el programa, su ámbito es todo el programa.

El siguiente ejemplo, muestra como pasar una cantidad fija de euros a dólares. Aunque no lo hemos visto todavía, el símbolo `*` representa la multiplicación.

```

double euros, dolares;
final double tasaCambio = 1.14;

euros = 15.37;
dolares = euros * tasaCambio;
System.out.println (dolares);

```

Constantes

Una constante es una variable cuyo valor no cambia durante la ejecución del programa, esto es, una vez que a una constante se le asigna un valor, este no podrá ser modificado y permanecerá así durante toda la ejecución del programa. Un ejemplo típico podría ser el número PI. Para convertir una variable en constante, simplemente se le añade el modificador **final** antes de su tipo. Ejemplo:

```
final double PI = 3.141592;
```

Existe la convención de que las constantes o variables finales se escriban con todas sus letras en mayúsculas.

En realidad, el número PI ya lo tenemos definido en Java, como `Math.PI`. Por lo que no haría falta crear esta variable.

Conversión de tipos (casting)

Es posible asignar un valor de un tipo a una variable de otro tipo. Si los dos tipos son compatibles, Java realiza la conversión automáticamente (siempre que el tipo destino sea más "grande" que el

origen, claro). Por ejemplo, siempre es posible asignar un valor del tipo *int* a una variable del tipo *long*. Sin embargo, no todos los tipos son compatibles, y, por lo tanto, no cualquier conversión está permitida implícitamente. Por ejemplo, la conversión de *int* a *short* no está definida.

```

7
8     public class Ejemplo {
9         public static void main(String[] args) {
10            int a = 3;
11            short b;
12            incompatible types: possible lossy conversion from int to short
13            ----
14            (Alt-Enter shows hints)
15
16

```

The screenshot shows a Java code editor with line numbers 7 to 16. Line 12 contains the error message: "incompatible types: possible lossy conversion from int to short". A tooltip below it says "(Alt-Enter shows hints)". The code itself is a simple class definition with a main method that assigns the value of 'a' to 'b'.

Pero se puede obtener una conversión entre tipos incompatibles. Para ello, se debe usar un **cast**, que realiza una conversión explícita entre tipos. Para ello, en la asignación se especifica entre paréntesis, después del igual, el tipo destino. Puede que ocasione pérdidas de información, por ejemplo, al pasar un *double* a *int* perdemos los decimales. En el ejemplo siguiente *resul*/valdría 3.

Promoción automática de tipos en las expresiones

Las conversiones de tipo, además de ocurrir en la asignación de valores, pueden tener lugar en las expresiones, en los cálculos a la derecha del igual. Por ejemplo:

```

byte cantidad1 = 100; byte cantidad2 = 90;
int resul = cantidad1 + cantidad2;

```

Si la suma se calculase con tipo byte (máx: 127) se hubiese producido un *overflow*, pero no se

```

public static void main(String[] args) {
    double PI = 3.141592;
    int resul;
    resul = (int) PI;
}

```

produce porque Java convierte al tipo destino el resultado de la operación. Aun así, es mejor prevenir estos problemas. Si *resul*/fuese de tipo *byte* si habría error.

Cadenas

En muchos lenguajes, incluyendo C/C++, las cadenas se implementan como listas (o Arrays) de caracteres. Sin embargo, éste no es el caso en Java. Las cadenas son realmente un tipo de objetos denominado **String**. Como se verá posteriormente, incluye un extenso conjunto de facilidades para manejo de cadenas que son, a la vez, potentes y fáciles de manejar. Podemos adelantar por ahora que podríamos definirlas como un tipo primitivo:

```
String str2 = "Los Strings son objetos";
```

O bien con notación de objetos.

```
String str1 = new String("Se construyen de diferentes modos ");
```

Los literales de cadena se encierran en comillas dobles y pueden incluir secuencias de escape como las los char.

Operadores

Operadores aritméticos

Estos son los operadores aritméticos:

<code>+</code>	Suma	<code>-</code>	Resta (también es el menos unario)
<code>*</code>	Multiplicación	<code>/</code>	División
<code>++</code>	Incremento	<code>--</code>	Decremento
<code>+=</code>	Suma y asignación	<code>-=</code>	Resta y asignación
<code>*=</code>	Multiplicación y asignación	<code>/=</code>	División y asignación
<code>%</code>	Módulo (resto)	<code>%=</code>	Módulo y asignación

Ejemplos, suponiendo que "a" es un entero con valor 13 y "b" un entero con valor 3 y "c" un float:

<code>c = a + b;</code>	c tomaría el valor 16
<code>a++;</code>	a tomaría el valor 14 (igual a: <code>a=a+1;</code>)
<code>a=a+5;</code>	a tomaría el valor 18
<code>a+=5;</code>	a tomaría el valor 18 (igual a la anterior)
<code>c = a % b; (*)</code>	c tomaría el valor 1
<code>c = a / b</code>	c tomaría el valor 4 (ojo: aunque c es float)
<code>c = (float) a / b</code>	c tomaría el valor 4.3333335

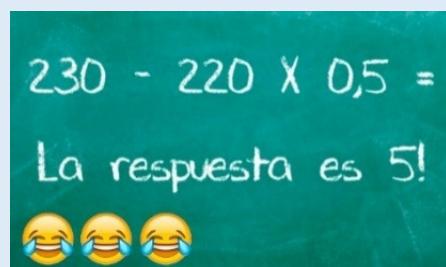
(*) % es el resto de la división entera (Excel le llama 'residuo'). Si divides 13 entre 3 te da como resultado 4,333. Pero si haces la división entera de 13 entre 3, el resultado es igual a 4 (cociente entero, sin decimales). Entonces el resto que queda en esa división es 1 (ya que 3×4 es 12, no 13). Es resto es al que llamamos módulo.

Los operadores unarios se pueden usar en dos formatos: `a++` o bien `++a`, pero puede llegar a ser confuso, compruébalo tú mismo:

```
int a=10, b;
b = a++;      //a valdría 11 y b 10
b = ++a;      //a valdría 11 y b también 11.
```

Precedencia de operadores: La precedencia de los operadores es la siguiente: primero los incrementos/decrementos `++`, `--`, luego: multiplicación, división y módulo: `*` / y `%` y finalmente sumas y restas. Obviamente, el uso de paréntesis altera esta precedencia.

Después de estas operaciones se evaluarían los operadores lógicos que veremos más adelante: `<`, `<=`, `>`, `>=`, a continuación: `==`, `!=` y finalmente el AND (`&&`) y OR (`||`).



El operador de asignación

Ya hemos hablado de él previamente “=”. No confundir con “==” que es la igualdad matemática. A la izquierda del igual siempre tendremos una sola variable (definida en ese momento o previamente) y a la derecha un valor o un cálculo evaluable (siempre de tipo compatible).

Existe una forma abreviada:

```
int a, b, c;
a = b = c = 10; //asigna a las tres variables el valor 10.
```

Veamos un ejemplo del uso del operador divisor y módulo. El siguiente ejemplo toma un importe en euros y lo descompone en los billetes de 20 euros, 5 euros y monedas de 1 euro correspondientes.

```
public class Ejemplo {
    public static void main(String[] args) {
        int billete5, billete20;
        int importe, importeRestante;

        importe = 133;
        billete20 = importe / 20;
        importeRestante = importe % 20;

        billete5 = importeRestante / 5;
        importeRestante = importeRestante % 5;

        System.out.println(billete20 + " billetes de 20");
        System.out.println(billete5 + " billetes de 5");
        System.out.println(importeRestante + " monedas");
    }
}
```

Si la variable *importe* e *importeRestante* las definimos con *long*, nos obligaría a hacer un casting en las líneas 12 y 15. Compruébalo tú mismo.

```
billete20 = (int) (importe / 20);
```

Problemas con la división entera

Como ya comentamos antes, una instrucción tan sencilla como `double res= 5 / 2;` no nos produce el esperado 2,5. Esto es debido a que Java primero calcula lo que está a la derecha del igual, sin importarle el tipo de variable a la que se asignará y que se encuentra a la izquierda de igual. Al encontrarse $5 / 2$, una operación de enteros, interpreta que el resultado también será un entero, por eso el resultado de esa operación, aun asignado a un *double*, será 2 y no 2,5.

Para solucionarlo, tenemos varias opciones: si estamos trabajando con valores literales podemos hacer que los interprete como *double* añadiendo una “d” al final, o añadiendo un cero decimal a cualquiera de los elementos de la división:

```
double res= 5 / 2.0; o bien
double res= 5d / 2;
```

Una tercera solución es hacer un casting, previo a la operación de división, también sobre cualquiera de los dos elementos de la división.

```
double res= (double) 5 / 2; o bien
double res= 5 / (double) 2; pero nunca: double res= (double) (5/2);
```

En el caso de trabajar con variables, está última solución, el casting, es la única válida:

```
double res= (double) x / y; o bien
double res= x / (double) y; pero nunca: double res= (double) (x/y);
```

Importante: Supongamos dos variables 'x' y 'z' de tipo *double*.

La instrucción `x=5/2*z;` no produce el resultado esperado, ya que comienza dividiendo 5 entre 2, y como son enteros, hace la división entera y luego multiplica 2 por 'z' y no 2,5 por 'z'.

En cambio, `x=z*5/2;` comienza multiplicando 'z' por 5, y al ser 'z' de tipo *double* produce un *double* y al dividirlo por 2 sigue siendo *double*. Esta segunda opción es la correcta, pero la primera puede producir errores que pasan desapercibidos.

Entrada y Salida por Consola

Lo último que nos hace falta para poder hacer nuestros primeros programas es la interacción con el usuario. La forma más sencilla es hacerlo mediante la consola de comandos, en la que el usuario puede introducir datos mediante el teclado, y podemos mostrar los resultados también en modo texto. Más adelante incorporaremos el entorno gráfico para hacer programas más parecidos a los que manejamos habitualmente.

Para mostrar mensajes por consola, como ya se ha ido viendo previamente es:

```
System.out.println ("Mensaje a mostrar");
```

Podemos mezclar texto y variables. Los primeros entre comillas dobles y concatenados mediante el operador "+".

```
System.out.println ("El resultado es " + importe + " euros");
```

Prueba este código:

```
int a=2,b=5;
System.out.println ("El número siguiente a dos es " + a+1);
System.out.println ("El número siguiente a cinco es " + (b+1));
```

La diferencia entre `print` y `println` radica en que `println` hace un salto de línea al final del mensaje y `print` no, por lo que los siguientes mensajes seguirían saliendo en la misma línea. Existe un tercer método: `printf` que no veremos por ahora.

Para solicitar que el usuario introduzca un valor por teclado, necesitamos hacer dos pasos previos, primero, importar la clase necesaria para emplear dichas funciones.

```
import java.util.Scanner; //antes del nombre del programa
```

A continuación, debemos crear un objeto de tipo `Scanner`:

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

A partir de este momento podemos solicitar distintos tipos de datos:

```
String texto = teclado.nextLine();
int edad =teclado.nextInt(); // nextLong();nextFloat();nextDouble();
char resp = teclado.nextLine().charAt(0);
```

Para especificar juego de caracteres en `Scanner`, por ejemplo, para evitar problemas con las eñes sería así:

```
new Scanner (System.in, "ISO-8859-1");
```

En temas posteriores se tratará en detalle la instrucción *import*, pero por ahora solo tenemos que quedarnos con la idea de que Java tiene definidas multitud de clases útiles que podemos emplear pero que pueden estar en distintos contenedores a los que llamamos paquetes. Para usar clases de otros paquetes hay que importarlas; bien importar la clase sola:

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

bien importar toda el paquete, con todas sus clases:

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

Aunque siempre será mejor la primera opción, importar solo las clases necesarias.

En Netbeans, sobre el código, botón derecho >> Fix Imports, nos incorpora todos los imports necesarios.

Por último, hay ocasiones en las que queremos que el usuario pulse la tecla <ENTER> para que el programa siga su ejecución. En este caso no valen las instrucciones anteriores, ya que esperarán que se introduzca al menos un carácter y luego <ENTER>. La solución sería:

```
System.out.println("Pulse <ENTER> para continuar");
try {System.in.read();} catch( Exception e ) {}
```

Más adelante, explicaremos esa estructura “try...catch” pero está relacionada con las excepciones o errores en tiempo de ejecución, para evitar que el programa “rompa” por meter datos incorrectos.

Aunque en el código de los programas el separador decimal es el punto, por ejemplo: 3.1416 cuando el usuario introduce un valor por consola con `nextFloat`, `nextDouble` debe usar la coma, por ejemplo: 3,1416. Esto es debido a que por defecto se usa la configuración local de España, introduciríamos punto si definiésemos Scanner de modo “americano”: `Scanner teclado = new Scanner(System.in).useLocale(java.util.Locale.US);`

“Bug” en la entrada de datos por consola

La lectura de datos por consola tiene un “bug” cuando combinamos tipos de datos distintos. Si después de leer un dato numérico con `nextInt()`, `nextFloat()`, etc., se intenta leer un texto con `nextLine()`, éste fallará porque lee el <ENTER> que dejó la lectura del numérico, y por tanto el String contendrá la cadena vacía.

Tenemos dos posibles soluciones:

- 1) Eliminar dicho <ENTER> o retorno de carro con un `nextLine` entre ambos:

```
float f = teclado.nextFloat(); teclado.nextLine();
String texto = teclado.nextLine();
```

El inconveniente de esta situación es que tenemos que recordar esta situación particular, es decir, no hay que hacerlo siempre, solo después de valores numéricos.

- 2) Leer siempre con `nextLine()` y convertir al tipo de dato que queramos (entero, doble, etc.) mediante utilidades que nos ofrece el lenguaje.

```
float f = Float.parseFloat(teclado.nextLine());
int i = Integer.parseInt(teclado.nextLine());
String texto = teclado.nextLine();
```

Más adelante explicaremos en detalle el funcionamiento de estas instrucciones, pero ya adelantamos que con `parseFloat()` debemos usar el punto como separador decimal, al contrario de lo que acabamos de decir para `nextFloat()`, `nextDouble()`.

Primer Programa

Ahora, sabiendo como pedir información al usuario, como hacer cálculos, y como mostrar datos por pantalla, estamos en condiciones de hacer nuestros primeros programas.

El siguiente ejemplo pide al usuario que introduzca dos números enteros y los sume:

```

1  /* Curso de Programación en Java */
2  package ejemplo;
3
4  import java.util.Scanner;
5  public class Ejemplo {
6
7      public static void main(String[] args) {
8          int sumando1, sumando2, resultado;
9
10         Scanner teclado = new Scanner(System.in);
11
12         System.out.print("Introduce un número entero: ");
13         sumando1 = teclado.nextInt();
14         System.out.print("Introduce otro entero: ");
15         sumando2 = teclado.nextInt();
16         resultado = sumando1 + sumando2;
17
18         System.out.println("La suma es " +resultado);
19     }
20 }
21

```

Es importante analizar el programa con calma, viendo el orden en el que se ejecutan las instrucciones:

- Línea 1: son comentarios, no se tienen en cuenta
- Línea 2: los programas se agrupan en paquetes dentro de un proyecto, sería un concepto similar al de las carpetas para agrupar archivos en nuestro ordenador.
- Línea 4: veremos más adelante que representa esta instrucción.
- Línea 5: nombre del programa. Tiene que coincidir con el nombre del archivo .java, es decir el archivo se ha de llamar *Ejemplo.java*.
- Línea 7: Comienzo del bloque de código que se ejecutará, es el programa en sí.
- Línea 8: Se crean las variables que se van a necesitar, en este caso todas de tipo entero.
- Línea 10: Creamos un “objeto” teclado para pedir datos por la consola.
- Línea 12 a 15: Pedimos los dos valores al usuario y los almacenamos en sendas variables. Si no introduce números enteros, se produce un error de ejecución y el programa se para abruptamente.
- Línea 16: Calculamos la suma y lo almacenamos en otra variable.
- Línea 18: Mostramos el resultado por pantalla.

Para probar todos los fragmentos de código de estos primeros capítulos puedes utilizar el programa anterior como plantilla y cambiar solo el contenido que hay entre la línea 8 y la 18.

2. Estructuras de Control: Condiciones

Las estructuras de control nos van a permitir determinar el flujo de la ejecución de código, mediante bifurcaciones o mediante repeticiones. En este capítulo veremos las primeras.

Java admite dos sentencias de selección: *if* y *switch*. Estas sentencias permiten controlar el flujo de ejecución del programa basado en función de condiciones conocidas únicamente durante el tiempo de ejecución.

If...Then...Else

Se utiliza para dirigir la ejecución del programa hacia dos caminos diferentes. El formato general de la sentencia *if* es:

```
if (condición) sentencia1; else sentencia2;
```

Ejemplo:

```
if (temperatura > 25) System.out.println("A la playa!!!!");
else                     System.out.println("A la montaña!!!!");
```

La *condición* tiene que ser una expresión evaluable cuyo resultado final sea verdadero o falso, son operadores típicos: <, <=, >, >=, ==, != (menor, menor o igual, mayor, mayor o igual, igual, distinto, respectivamente).

El operador **!** representa la negación, por eso == representa igual a y != distinto de.

Tanto *sentencia1* como *sentencia2* puede ser cualquier instrucción del lenguaje y puede ser una sentencia única o un conjunto de sentencias encerradas entre llaves, es decir, un bloque. La condición es cualquier expresión que devuelva un valor booleano. La cláusula *else* (con la *sentencia2*) es opcional.

La sentencia *if* funciona del siguiente modo: Si la condición es verdadera, se ejecuta la *sentencia1*. En caso contrario se ejecuta la *sentencia2* (si es que existe). En ningún caso se ejecutarán ambas sentencias.

Es una buena práctica de programación utilizar las llaves de bloque en las sentencias del *if* aunque solo haya una única instrucción y no sea obligatorio. Así, si a posteriori hay que añadir nuevas instrucciones evitaremos que se nos olvide y produzca resultados inesperados. Ejemplo:

```
int a=1, b=0, c=3;
if (a > 0) b=1;
else b=2; c=4;
```

¿Qué valor tendrá *c* después de la ejecución de este fragmento de código?

Si en la condición, trabajamos con una variable *boolean* ya no es necesario incluir la igualdad a *true*:

```
boolean expr;
if (expr == true) //es lo mismo que: if (expr)
if (expr == false) //es lo mismo que: if (!expr)
```

if anidados

Un *if* anidado es una sentencia *if* que está contenida dentro de otro *if* o *else*. Cuando se anidan *if* lo más importante es recordar que una sentencia *else* siempre corresponde a la sentencia *if* más próxima dentro del mismo bloque y que no esté ya asociada con otro *else* y que la última llave de bloque que abrimos, se corresponde con la primera que cerramos, la segunda con la penúltima, etc. (tal y como ocurre en matemáticas con los paréntesis). Veamos un ejemplo:

```
Scanner teclado = new Scanner(System.in);
System.out.println("¿Cuántos años tienes? ");
int edad = teclado.nextInt();
if (edad >= 18) (*) 
    System.out.println ("Eres mayor de edad");
else {
    if (edad >= 16) {
        System.out.println("Eres menor de edad");
        System.out.println("pero puedes trabajar");
    }
    else (*) 
        System.out.println("Sigue estudiando, aun eres un niño");
}
```

(*) Cuando solo hay una instrucción en el *if* (o en el *else*) no hacen falta llaves.

if-else-if múltiples

Una construcción muy habitual en programación es la de if-else-if múltiples. Esta construcción se basa en una secuencia de *if* anidados. Su formato es el siguiente:

```
if (condición) sentencia;
else if (condición) sentencia;
else if (condición) sentencia;
else
    sentencia;
```

La sentencia *if* se ejecuta de arriba abajo. Tan pronto como una de las condiciones que controlan el *if* sea true, las sentencias asociadas con ese *if* serán ejecutadas, y el resto ignoradas. Si ninguna de las condiciones es verdadera, entonces se ejecutará el *else* final. El *else* final actúa como una condición por omisión, es decir, si todas las demás pruebas condicionales fallan, entonces se ejecutará la sentencia del último *else*. Si no hubiera un *else* final y todas las demás condiciones fueran false, entonces no se ejecutaría ninguna acción.

Este programa calcula un descuento en función del importe proporcionado.

```
float importe = 2390;
if (importe < 1000) System.out.println ("NO HAY DESCUENTO");
else if (importe < 3000) // (importe<3000 y importe>=1000)
    System.out.println ("DESCUENTO: 3%: " + importe * 0.03);
else if (importe < 5000) // (importe<5000 y importe>=3000)
    System.out.println ("DESCUENTO: 5%: " + importe * 0.05);
else // (importe >= 5000)
    System.out.println ("DESCUENTO: 7%: " + importe * 0.07);
// no existen las palabras reservadas elseif, elif, elsif, ni equivalentes
```

Es frecuente ver operaciones de incremento en un *if*, por ejemplo, supón que *i* es una variable de tipo entero que vale 10. ¿Qué diferencia habría entre *if* (*i*+*a*>10) y *if* (*a*+*i*>10) ? Compruébalo.

Switch

La sentencia `switch` es una sentencia de bifurcación múltiple. En muchas ocasiones, es una mejor alternativa que una larga serie de sentencias `if-else-if`. El formato general de una sentencia `switch` es:

```
switch (expresión) {
    case valor1: // secuencia de sentencias; break;
    case valor2: // secuencia de sentencias; break;
    ...
    case valorN: // secuencia de sentencias; break;
    default:      // secuencia de sentencias por omisión
}
```

La expresión debe ser del tipo `byte`, `short`, `int` o `char` (desde Java7 también `String`); cada uno de los valores especificados en las sentencias `case` debe ser de un tipo compatible con el de la expresión. Cada uno de estos valores debe ser un literal único, es decir, una constante no una variable.

La sentencia `switch` funciona de la siguiente forma: se compara el valor de la expresión con cada uno de los valores constantes que aparecen en las sentencias `case`. Si coincide con alguno, se ejecuta el código que sigue a la sentencia `case`. Si ninguna de las constantes coincide con el valor de la expresión, entonces se ejecuta la sentencia `default`. Sin embargo, la sentencia `default` es opcional.

Si ningún `case` coincide y no existe la sentencia `default`, no se ejecuta ninguna acción.

La sentencia `break` se utiliza dentro del `switch` para terminar una secuencia de sentencias. Cuando aparece una sentencia `break`, la ejecución del código se desplaza hasta la siguiente instrucción después del bloque `switch`.

Este ejemplo proporciona el nombre del mes en texto a partir del número de mes:

```
public class Ejemplo {
    public static void main (String args[]) {
        int numMes = 7;

        switch (numMes) {
            case 1 : System.out.println("Enero");      break;
            case 2 : System.out.println("Febrero");     break;
            case 3 : System.out.println("Marzo");       break;
            case 4 : System.out.println("Abril");       break;
            case 5 : System.out.println("Mayo");        break;
            case 6 : System.out.println("Junio");       break;
            case 7 : System.out.println("Julio");       break;
            case 8 : System.out.println("Agosto");      break;
            case 9 : System.out.println("Septiembre"); break;
            case 10: System.out.println("Octubre");     break;
            case 11: System.out.println("Noviembre");   break;
            case 12: System.out.println("Diciembre");   break;
            default: System.out.println("Mes erróneo");
        }
    }
}
```

Si omitimos el `break` en un `case`: la ejecución continuaría por los siguientes `case`. Olvidárnosla puede provocar errores en el comportamiento del programa, pero también puede ser útil para agrupar distintos `case` que tengan las mismas instrucciones asociadas.

Este programa calcula los días que tiene los meses del año, salvo para años bisiestos.

```
int numDias=0, numMes = 5;
switch (numMes) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12: numDias = 31;    break;
    case 2:   numDias = 28;    break;
    case 4:
    case 6:
    case 9:
    case 11:  numDias = 30;    break;
}
System.out.println(numMes + " -> " + numDias + " días");
```

Otra funcionalidad que nos ofrece switch es que devuelva implícitamente un valor y se lo asignemos a una variable. Para lograr esto la sintaxis será `variable = switch { . . . };`. Dentro del switch, para devolver el valor, cada case incluirá una instrucción: `yield valor`, y no será necesario el break. Se ve más claro con un ejemplo. El código anterior podría haberse implementado así:

```
int numDias = switch (numMes) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12: yield 31;
    case 2:   yield 28;
    case 4:
    case 6:
    case 9:
    case 11:  yield 30;
    default:  yield 0;
};
```

Hay que destacar que, con este formato, los 'case' tienen que cubrir todos los casos posibles, lo que obligará a incluir la cláusula 'default' en la mayor parte de los casos.

Switch ->

Desde Java 14, existe una nueva versión de switch que sustituye los dos puntos por "->" y no necesita break, de forma que solo se ejecuta el código de cada case y no de los siguientes (como ocurría en el caso anterior si no incluíamos break). Queda el código más limpio.

```
switch (numMes) {
    case 1 -> System.out.println("Enero");
    case 2 -> System.out.println("Febrero");
    case 3 -> System.out.println("Marzo");
    case 4 -> System.out.println("Abril");
    case 5 -> System.out.println("Mayo");
    case 6 -> System.out.println("Junio");
    case 7 -> System.out.println("Julio");
    case 8 -> System.out.println("Agosto");
    case 9 -> System.out.println("Septiembre");
    case 10 -> System.out.println("Octubre");
    case 11 -> System.out.println("Noviembre");
    case 12 -> System.out.println("Diciembre");
}
```

En caso de tener varias sentencias a ejecutar para un case, debemos ponerlas entre llaves {}. Si tenemos varios case con el mismo comportamiento, separamos sus valores por comas.

```
switch (numMes) {
    case 1,3,5, 7,8,10,12 -> { numDias = 31;
                                mesLargo = true; }
    case 2                  -> numDias = 28;
    case 4,6,9,11          -> numDias = 30;
}
```

Como en el caso del switch tradicional (*case* :) podemos asignar el switch a una variable pero en este caso no es necesario la instrucción *yield* si el *case* no tiene instrucciones adicionales. En el ejemplo anterior, podríamos haber hecho:

```
int numDias = switch (numMes) {
    case 1,3,5,7,8,10,12 -> { mesLargo = true; yield 31; }
    case 2                  -> 28;
    case 4,6,9,11          -> 30;
    default                -> 0;
};
```

Al igual que en el formato tradicional, los 'case' tienen que cubrir todos los casos posibles, lo que obligará a incluir la cláusula 'default' en la mayor parte de los casos.

Operador ?

Java incluye un *operador ternario* especial que puede sustituir a ciertos tipos de sentencias if- then-else para la asignación de un valor a una variable. Este operador es ?. Puede resultar un tanto confuso en principio, pero el operador ? resulta muy efectivo una vez que se ha practicado con él. El operador ? tiene la siguiente forma general:

```
variable = condicion ? expresionTrue : expresionFalse;
```

Donde *condicion* puede ser cualquier expresión que dé como resultado un valor del tipo boolean. Si esa condición genera como resultado true, entonces se le asigna a la variable la *expresionTrue*, en caso contrario se le asigna *expresionFalse*. Es necesario que tanto la *expresionTrue* como la *expresionFalse* devuelvan el mismo tipo que no puede ser void.

Como ejemplo, el código: `if (x > y) mayor = x; else mayor = y;`

Podría escribirse como: `mayor = (x > y) ? x: y;`

Ahora que aún estamos aprendiendo a programar no recomendamos su uso, pero si tenemos que saber interpretarlo si nos encontramos con él.

Los pasos a seguir serían:

- 1) Poner la variable a asignarle valor y el igual. **mayor =**
- 2) Poner la condición entre interrogaciones y el operador "?": **mayor = (x > y) ?**
- 3) Poner el valor a asignar si la condición es true: **mayor = (x > y) ? x**
- 4) Poner los dos puntos y el valor a asignar si la condición es false: **mayor = (x > y) ? x: y;**

Otro ejemplo: Calcular la división de dos números siempre que el divisor sea distinto de cero. En caso de que el divisor sea cero el resultado a mostrar será -999.

```
if (divisor !=0) resultado = dividendo/divisor; else resultado = -999;
```

lo haríamos así:

```
resultado = (divisor !=0) ? dividendo/divisor : -999;
```

Operadores Relacionales y Lógicos

Los operadores relacionales determinan la relación que un operando tiene con otro. Específicamente, determinan relaciones de igualdad y orden. A continuación, se muestran los operadores relacionales:

Operador	Resultado
<code>==</code>	Igual a
<code>!=</code>	Diferente de
<code>></code>	Mayor que
<code><</code>	Menor que
<code>>=</code>	Mayor o igual que
<code><=</code>	Menor o igual que

El resultado de estas operaciones es un valor booleano: *true* o *false*. La aplicación más frecuente de los operadores relacionales es en la obtención de expresiones que controlan la sentencia if y las sentencias de ciclos.

Sólo se pueden comparar operandos enteros, de punto flotante y caracteres, para ver cuál es mayor o menor. Para cadenas, como son objetos usaremos métodos propios de tal objeto, pero esto lo veremos más adelante: `if (cad1.equals(cad2))`

Además de los operadores ya comentados: `<`, `<=`, `>`, `>=`, `==`, `!=` Java ofrece la posibilidad de combinarlos mediante los operadores lógicos:

`&&`, es el AND lógico, en que ambas premisas deben ser *true* para que el resultado de la expresión sea *true*.

`||`, es OR lógico, en la que con tal de que una de las expresiones sea *true*, el resultado también lo será.

var1	var2	var3	var1 && var2 && var3	var1 var2 var3
false	false	false	FALSE	FALSE
false	false	true	FALSE	TRUE
false	true	false	FALSE	TRUE
false	true	true	FALSE	TRUE
true	false	false	FALSE	TRUE
true	false	true	FALSE	TRUE
true	true	false	FALSE	TRUE
true	true	true	TRUE	TRUE

Estos operadores se evalúan en **cortocircuito**, esto es, no se evalúa el operando de la derecha si el resultado de la operación queda determinado por el operando de la izquierda. Esto es útil cuando el operando de la derecha depende de que el de la izquierda sea *true* o *false*.

Por ejemplo, si queremos hacer algo en caso de que una división valga más de 10, tenemos que asegurar que no es una división por cero (ya que causaría error de ejecución) deberemos preguntar primero si el denominador es distinto de cero.

```
if (denom != 0 && num / denom > 10)
```

Al usar la forma en cortocircuito del operador AND (`&&`) no existe riesgo de que se produzca una excepción en tiempo de ejecución, ya que, si `denom` es igual a cero, el resultado sería falso independientemente del resultado de la expresión a la derecha del `&&` por lo que ya no lo evaluaría y no realizaría la división.

La forma sin "cortocircuito", es decir, en la que se evalúan todas las expresiones de la condición, aunque al evaluar la primera ya sepamos el resultado final, se hace mediante los operadores `&` y `|` en vez de `&&` y `||` respectivamente (no es muy frecuente).

El siguiente ejemplo calcula si un año es bisiesto o no (un año es bisiesto si es múltiplo de 4 pero no múltiplo de 100. Excepcionalmente los múltiplos de 400 también lo son, a pesar de ser múltiplos de 100.

```
boolean bisiesto;
Scanner teclado = new Scanner (System.in);
System.out.println ("Introduce un año: ");
int año = teclado.nextInt();

bisiesto = (año % 4 == 0) && (año % 100 != 0) || (año % 400 == 0);
if (bisiesto) System.out.println("El " + año + " es bisiesto");
else System.out.println("El " + año + " no es bisiesto");
```

El programa anterior funciona correctamente porque el operador AND `&&` tiene más prioridad que el operador `||` y así no hacen falta paréntesis adicionales. Si tenemos dudas, siempre podemos añadir los paréntesis, aunque no sean necesarios.

Un último ejemplo, en el siguiente programa se muestra cómo resolver un problema mediante if con anidamiento y sin anidamiento.

```
Scanner teclado = new Scanner (System.in);
System.out.println("Dime tu nota, entre 0 y 10 (con coma decimal)");
float nota = teclado.nextFloat();

if (nota < 5) System.out.println (nota + " SUSPENSO");
else if (nota < 7) System.out.println (nota + " APROBADO");
else if (nota < 9) System.out.println (nota + " NOTABLE");
else System.out.println (nota + " SOBRESALIENTE");

// SIN ANIDAMIENTOS:
if (nota < 5) System.out.println (nota + " SUSPENSO");
if (nota >= 5 && nota < 7) System.out.println (nota + " APROBADO");
if (nota >= 7 && nota < 9) System.out.println (nota + " NOTABLE");
if (nota >= 9) System.out.println (nota + " SOBRESALIENTE");
```

En C/C++ el valor `boolean` falso está asimilado a cero y true como el resto de valores, pero esto no es así en Java, así que siempre debemos usar variables y expresiones de tipo `boolean` y no enteras para evaluar expresiones lógicas.

3. Estructuras de Control: Bucles

Las sentencias de iteración de Java son **for**, **while** y **do-while**. Estas sentencias crean lo que comúnmente se denominan ciclos, bucles, iteraciones, repeticiones, etc. Un ciclo ejecuta repetidas veces el mismo conjunto de instrucciones hasta que se cumple una determinada condición que lo hace parar. Si un bucle no se construye bien, puede ocurrir que nunca termine, en esos casos se suele decir coloquialmente que el programa queda “embuulado”.

For

Esta estructura se utiliza para iteraciones o bucles que se va a ejecutar un número determinado de veces, no depende de lo que ocurra en las instrucciones del interior del bloque.

La forma general de la sentencia *for* es la siguiente:

```
for (inicialización; condición; iteración) {
    // operaciones que se van a ejecutar repetidamente
}
```

Si solamente se repite una sentencia, no es necesario el uso de las llaves, al igual que ocurría en el *if*, o en su rama *else* o, como veremos a continuación, en el caso de *while* y *do-while*.

Al empezar, se ejecuta la parte de **inicialización** (generalmente, es una expresión que establece el valor de la variable de control del ciclo) Esta expresión de inicialización se ejecuta una sola vez.

A continuación, se evalúa la **condición**, que debe ser una expresión booleana, si la expresión es verdadera, entonces se ejecuta el cuerpo del ciclo. Si es falsa, el ciclo finaliza.

Cada vez que se recorre el ciclo, en primer lugar, se ejecuta la **iteración** y se vuelve a evaluar la expresión condicional, si sigue siendo verdadera, se repite el proceso. Una vez que la condición sea falsa, ya no se repite más el cuerpo y pasamos a la siguiente instrucción del programa.

Un bucle que queremos que se ejecute 10 veces podría tener una estructura como esta:

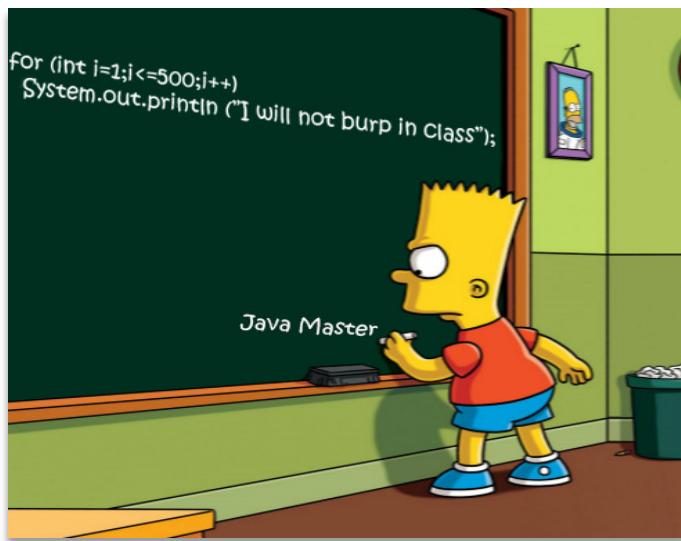
```
for (int i=1; i<=10; i++) {
    // cuerpo
}
```

Además de ejecutarse 10 veces, en cada iteración la variable va a ir cambiando, desde 1 la primera vez a 10 la última. Al salir del bucle, *i* valdría 11, pero realmente se destruye la variable. *Recordemos que el ámbito de una variable es el bloque donde está definida, para poder utilizar ese valor fuera del bucle deberíamos haberla definido antes, fuera del bucle.*

Ejemplo: mostrar la tabla de multiplicar de un número introducido previamente.

```
2 package ejemplo;
3 import java.util.Scanner;
4
5 public class Ejemplo {
6     public static void main (String args[]) {
7
8         Scanner teclado = new Scanner (System.in);
9         System.out.println("Introduce un entero positivo menor que 10:");
10        int valor = teclado.nextInt();
11
12        for (int i=1; i<=10; i++) { // se pueden omitir las llaves
13            System.out.println (valor + " x " + i + " = " + (valor * i));
14        }
15    }
16 }
```

En ocasiones puede ser necesario incluir más de una sentencia en las secciones de inicialización e iteración del ciclo **for**. Se puede hacer separando las expresiones de esas dos secciones mediante una coma. `for (int a=1,b=10;a<=10; a++,b--) System.out.println (a + " - " + b);`



While

Con este ciclo se repite una sentencia o un bloque mientras la condición de control es verdadera. Su forma general es:

```
while (condición) {
    // cuerpo del ciclo
}
```

Características del *while*:

- La condición puede ser cualquier expresión booleana, como las vistas en con el *if*.
- El cuerpo del ciclo se ejecutará mientras la expresión condicional sea verdadera.
- Cuando la condición sea falsa, la ejecución pasa a la siguiente línea de código localizada inmediatamente después de la llave de cierre del cuerpo ciclo.
- Como la condición se evalúa al principio de cada ciclo, el cuerpo del mismo no se ejecutará nunca si al comenzar la condición es falsa.
- Las llaves no son necesarias si solamente se repite una sentencia en el cuerpo del ciclo.

Ejemplo: Hacer un programa en el que usuario vaya metiendo números y los vaya acumulando, hasta que el usuario introduzca -1. Finalmente mostrará el importe acumulado.

```
float num, total = 0;

Scanner teclado = new Scanner (System.in);
System.out.println("Introduce un numero (-1 para finalizar): ");
num = teclado.nextFloat();
while (num != -1) {
    total = total + num;    //también total+=num;
    System.out.println("Introduce otro numero (-1 para finalizar): ");
    num = teclado.nextFloat();
}
System.out.println ("Total: " + total);
```

De este ejemplo podemos inferir dos conceptos nuevos relacionados con los bucles: el primero se denomina “lectura adelantada” y el de “acumulador”.

La **lectura adelantada** se refiere a que antes de evaluar el *while* debemos asegurarnos de que la condición es evaluable, es decir, que sus variables tienen valor, por ello debemos hacer una primera lectura de teclado antes de entrar en el bucle. La última instrucción del bloque del bucle debe ser una nueva lectura para volver al principio del mismo.

Si quisiésemos evitar esta doble lectura, podríamos meterla al principio del bucle, pero asegurándonos de que la primera vez la condición del *while* no va a representar ningún problema, y que cuando el usuario introduzca -1 no vamos a restarle 1 a la suma acumulada.

Así pues, sin lectura adelantada, la lectura sería la primera instrucción dentro de la repetición, pero necesitaría una condición justo a continuación para verificar que no es fin de ciclo. En el siguiente ejemplo se ve el mismo programa que el anterior, pero sin lectura adelantada. Prueba a ejecutarlo, y comprueba lo que ocurriría si se elimina el *if*.

```
import java.util.Scanner;
public class Ejemplo {
    public static void main (String args[]) {
        float num=0, total = 0;

        Scanner teclado = new Scanner (System.in);
        while (num != -1) {
            System.out.println("Introduce un numero (-1 para finalizar): ");
            num = teclado.nextFloat();
            if (num != -1)    total = total + num;    //también total+=num;
        }
        System.out.println ("Total: " + total);
    }
}
```

Otro elemento importante que ha aparecido en este ejemplo es el concepto de **acumulador**, esto es, una variable que va cambiando a medida que avanza el bucle, basándose en el valor previo y la operación realizada. Su formato es, como se ha visto:

```
acumulador = acumulador + incremento;
```

y en notación abreviada:

```
acumulador += incremento;
```

Cuando el incremento del acumulador es siempre constante, se le denomina **contador**, y tiene el formato (suponiendo la constante 2):

```
contador = contador + 2; //o contador +=2
```

En cada iteración del bucle la variable (suponiendo que se inicie en cero) tomará los valores: 2, 4, 6, 8, etc. Un caso muy habitual es en el que la constante es 1. Lo podríamos escribir de una tercera forma:

```
contador++;
```

Son frecuentes, análogamente, sentencias del tipo: `x--;` `x-=2;` etc...

Do...While

Como se acaba de ver, si la expresión condicional que controla un ciclo *while* es inicialmente falsa, el cuerpo del ciclo no se ejecutará ni una sola vez. Sin embargo, puede haber casos en los que se quiera ejecutar el cuerpo del ciclo al menos una vez, incluso cuando la expresión condicional sea inicialmente falsa. En otras palabras, puede que se desee evaluar la expresión condicional al final del ciclo, en lugar de hacerlo al principio. El ciclo *do-while* ejecuta siempre, al menos una vez, el cuerpo, ya que la expresión condicional se encuentra al final. Su forma general es:

```
do {
    // operaciones que se ejecutan repetidamente
} while (condición);
```

En cada iteración del ciclo *do-while* se ejecuta en primer lugar el cuerpo del ciclo, y a continuación se evalúa la expresión condicional. Si la expresión es verdadera, el ciclo se repetirá. En caso contrario, el ciclo finalizará.

El ciclo *do-while* es muy útil cuando se procesa un menú de selección, ya que normalmente se desea que el cuerpo del menú se ejecute al menos una vez. Ejemplo:

```
Scanner teclado = new Scanner (System.in);
char eleccion=' ';
int num=12;

do {
    System.out.println ("Elija:\n(a) Elevar al cuadrado");
    System.out.println ("(b)Raiz cuadrada\nOtra tecla para salir");
    eleccion = teclado.nextLine().charAt (0);

    if (eleccion =='a') System.out.println (Math.pow(num,2));
    else if (eleccion =='b') System.out.println (Math.sqrt(num));
} while (eleccion == 'a' || eleccion == 'b');
```

Ciclos anidados

Al igual que vimos con la sentencia **if** se pueden anidar los bucles, tanto los **while**, **do-while** como **for**. Estudia el siguiente ejemplo.

```
1  /* Curso de Programación en Java */
2  package ejemplo;
3
4  public class Ejemplo {
5      public static void main (String args[]) {
6
7          for (int i=1;i<=10;i++) {
8              for (int j=1;j<=i;j++)
9                  System.out.print ("*");
10             System.out.print ("\n");
11         }
12     }
}
```

El resultado sería el siguiente:

El primer bucle se enjutaría 10 veces, y para cada una de esas veces se ejecutaría el bucle interior. El bucle interior, la primera vez se ejecutaría una sola vez (`j` va de 1 hasta `i` que vale 1), la segunda vez se ejecutaría dos veces (`j` va desde 1 hasta `i` que ahora vale 2), y así sucesivamente. Al final de cada iteración bucle interior dibuja un salto de línea con `print ln`.

Sentencias de Salto

Java incorpora tres sentencias de salto: **break**, **continue** y **return**. Estas sentencias transfieren el control a otra parte del programa. Cada una es examinada aquí.

break

La sentencia break (además de su uso ya visto en un switch) fuerza la finalización inmediata de un ciclo, evitando la expresión condicional y el resto de código dentro del cuerpo del ciclo. Cuando se encuentra una sentencia break dentro de un ciclo, el ciclo termina y el control del programa se transfiere a la sentencia que sigue al ciclo.

Cuando la sentencia **break** se utiliza dentro de un conjunto de ciclos anidados, solamente se saldrá del ciclo en el que esté situado, si es el interno, no afecta al ciclo superior.

El siguiente ejemplo calcula si un número introducido es primo o no, recorriendo los números desde el 2 hasta el anterior a dicho número. Si alguno de esos números es divisor del número introducido (resto de su división es cero) ya no será primo, y podemos abandonar el bucle.

```
Scanner teclado = new Scanner (System.in);
System.out.println("Introduce un entero positivo:");
int valor = teclado.nextInt();
int i;
boolean primo=true;

for (i=2; i< valor; i++) {      // se pueden omitir las llaves
    if (valor % i == 0) {primo = false; break;}
}
if (primo) System.out.println ("Es primo");
else System.out.println ("No es primo");
```

El uso de **break** en los bucles puede ser considerado un mal estilo de programación ya que, entre muchas líneas de código, a veces es difícil encontrar la lógica de la condición de un bucle. Por ello, suele ser recomendable controlar la condición que provoca el *break* desde la condición de ejecución del bucle (en algunos casos convirtiendo un *for* en un *while*).

Esta sería una versión del ejemplo anterior, sin break, con un while en vez de for:

```
Scanner teclado = new Scanner (System.in);
System.out.println("Introduce un entero positivo:");
int valor = teclado.nextInt();
int i=2;
boolean primo=true;
while (primo && i < valor) {
    if (valor % i == 0) primo = false;
    i++;
}
if (primo) System.out.println ("Es primo");
else System.out.println ("No es primo");
```

Este último es muy interesante, porque utiliza el concepto de **flag**, esto es, una variable (normalmente de tipo *boolean*) que tiene un valor por defecto y en cuanto ocurre una determinada situación cambia su valor y ya no vuelve al valor inicial. Ese nuevo valor es una marca para saber más adelante si esa situación ha ocurrido o no. En el caso del ejemplo, por defecto un número dice que es *primo*, pero en cuanto se encuentre un divisor esa situación cambia, ya no es primo, y nada hará que cambie esa situación.

Este tipo de *flags* se suelen usar para enunciados del tipo: "saber si hay algún xxxx", es decir una situación, que, al darse una vez, ya nos marca la respuesta. En el caso del ejemplo: un número no es primo si hay tiene algún divisor además de 1 y él mismo.

continue

Algunas veces es útil forzar una nueva iteración de un ciclo sin concluir completamente el procesamiento de la iteración actual, es decir, un salto desde un punto del bloque hasta el final del ciclo. Eso lo hacemos con la sentencia **continue**. Al igual que el caso del *break* no es muy aconsejable abusar de su uso.

Ejemplos de estructuras repetitivas

En este apartado vamos a ver distintos ejemplos de estructuras repetitivas, que van cambiando ligeramente según lo que nos indica el enunciado.

1.- Solicitar por teclado las edades de los alumnos de una clase y obtener la edad del mayor de la clase. *Partimos de un valor ficticio muy bajo para la variable 'mayor', así en cuanto empiezamos a pedir edades por teclado, la primera pasará a ser el mayor automáticamente.*

```
Scanner teclado = new Scanner(System.in);
int edad;
final int TOTALALUMNOS = 10;
int mayor = -1;
for (int i = 1; i <= TOTALALUMNOS; i++) {
    System.out.print("Introduce una edad: ");
    edad = teclado.nextInt();
    if (edad > mayor) mayor = edad;
}
System.out.println("el mayor tiene " + mayor + " años");
```

2.- Solicitar por teclado las edades de los alumnos de una clase y obtener la edad del mayor de la clase. En el ejemplo anterior partíamos de un valor inicial ficticio para la variable 'mayor'. En este no lo hacemos así, tratamos la primera edad de forma especial, diciendo que es el mayor; luego el bucle empieza en el segundo alumno.

```
Scanner teclado = new Scanner(System.in);
int edad;
final int TOTALALUMNOS = 10;
System.out.print("Introduce una edad: ");
edad = teclado.nextInt();
int mayor = edad;
for (int i = 2; i <= TOTALALUMNOS; i++) {
    System.out.print("Introduce una edad: ");
    edad = teclado.nextInt();
    if (edad > mayor) mayor = edad;
}
System.out.println("el mayor tiene " + mayor + " años");
```

3.- Solicitar por teclado las edades de los alumnos de una clase y decir si hay algún mayor de edad, solicitando las edades de todos los alumnos de la clase (no solo hasta que encontramos un menor de edad).

```
Scanner teclado = new Scanner(System.in);
int edad;
final int TOTALALUMNOS = 10;
boolean hayMayorDeEdad = false;
for (int i = 1; i <= TOTALALUMNOS; i++) {
    System.out.print("Introduce una edad: ");
    edad = teclado.nextInt();
    if (edad >= 18)
        hayMayorDeEdad = true; //importante, no hay else
}
if (hayMayorDeEdad) System.out.println("Sí hay algún mayor de edad");
else System.out.println("No hay ningún mayor de edad");
```

4.- Solicitar por teclado las edades de los alumnos de una clase y decir si hay algún mayor de edad, solicitando las edades imprescindibles (en cuanto hay un mayor de edad ya paramos). *Solución con break, no recomendable.*

```
Scanner teclado = new Scanner(System.in);
int edad;
final int TOTALALUMNOS = 10;

boolean hayMayorDeEdad = false;
for (int i = 1; i <= TOTALALUMNOS; i++) {
    System.out.print("Introduce edad: ");
    edad = teclado.nextInt();
    if (edad >= 18) {
        hayMayorDeEdad = true;
        break;
    }
}
if (hayMayorDeEdad) System.out.println("Sí hay algún mayor de edad");
else System.out.println("No hay ningún mayor de edad");
```

5.- Solicitar por teclado las edades de los alumnos de una clase y decir si hay algún mayor de edad, solicitando las edades imprescindibles (*en cuanto hay un mayor de edad ya paramos*). *Solución sin break, recomendable.*

```
Scanner teclado = new Scanner(System.in);
int edad;
final int TOTALALUMNOS = 10;
boolean hayMayorDeEdad = false;
int i = 1;
while (i <= TOTALALUMNOS && !hayMayorDeEdad) {
    System.out.print("Introduce edad: ");
    edad = teclado.nextInt();
    if (edad >= 18) hayMayorDeEdad = true;
    else i++;
}
if (hayMayorDeEdad) System.out.println("Sí hay algún mayor de edad");
else System.out.println("No hay ningún mayor de edad");
```

6.- Solicitar por teclado las edades de los alumnos de una clase y calcular la edad media de la clase sabiendo previamente el total de alumnos.

```
Scanner teclado = new Scanner(System.in);
int edad, suma=0;
final int TOTALALUMNOS = 10;
float media;
for (int i = 1; i <= TOTALALUMNOS; i++) {
    System.out.print("Introduce una edad: ");
    edad = teclado.nextInt();
    suma+=edad;
}
media= (float)suma/TOTALALUMNOS;
System.out.println("La edad media es: " + media);
```

7.- Solicitar por teclado las edades de los alumnos de una clase y calcular la edad media solo de los mayores de edad de la clase sabiendo previamente el total de alumnos.

```
Scanner teclado = new Scanner(System.in);
int edad, sumaMayores=0,cantidadMayores=0;
final int TOTALALUMNOS = 10;
float media;
for (int i = 1; i <= TOTALALUMNOS; i++) {
    System.out.print("Introduce una edad: ");
    edad = teclado.nextInt();
    if (edad >= 18) {
        sumaMayores+=edad;
        cantidadMayores++;
    }
}
media= (float)sumaMayores/cantidadMayores;
System.out.println("La edad media es: " + media);
```

8.- Solicitar por teclado las edades de los alumnos de una clase y calcular la edad media de la clase preguntando previamente al usuario el total de alumnos.

```
Scanner teclado = new Scanner(System.in);
int edad, suma=0;
int totalAlumnos;
float media;
System.out.print("Introduce el número de alumnos: ");
totalAlumnos = teclado.nextInt();

for (int i = 1; i <= totalAlumnos; i++) {
    System.out.print("Introduce una edad: ");
    edad = teclado.nextInt();
    suma+=edad;
}
media= (float) suma/totalAlumnos;
System.out.println("La edad media es: " + media);
```

9.- Solicitar por teclado las edades de los alumnos de una clase y calcular la edad media de la clase. No sabemos la cantidad de alumnos, introducen 999 como edad para decir que no hay más alumnos.
Solución con lectura justo al empezar el bucle.

```
Scanner teclado = new Scanner(System.in);
int edad=0, suma=0;
int totalAlumnos=0;
float media;
while(edad !=999) {
    System.out.println("Introduce una edad (999 para terminar): ");
    edad = teclado.nextInt();
    if (edad!=999) {
        suma+=edad;
        totalAlumnos++;
    }
}
media= (float) suma/totalAlumnos;
System.out.println("La edad media es: " + media);
```

10.- Solicitar por teclado las edades de los alumnos de una clase y calcular la edad media de la clase. No sabemos la cantidad de alumnos, introducen 999 como edad para decir que no hay más alumnos.
Solución con lectura antes del bucle y como última instrucción dentro del bucle.

```
Scanner teclado = new Scanner(System.in);
int edad=0, suma=0;
int totalAlumnos=0;
float media;
System.out.println("Introduce una edad (999 para terminar): ");
edad = teclado.nextInt();
while(edad !=999) {
    suma+=edad;
    totalAlumnos++;
    System.out.println("Introduce una edad (999 para terminar): ");
    edad = teclado.nextInt();
}
media= (float) suma/totalAlumnos;
System.out.println("La edad media es: " + media);
```

11.- Solicitar por teclado las edades de los alumnos de una clase y calcular la edad media de la clase. No sabemos la cantidad de alumnos, después de introducir cada alumno le preguntamos si quiere introducir más y responde (S/N).

```
Scanner teclado = new Scanner(System.in);
int edad, suma=0;
int totalAlumnos=0;
float media;
char respuesta;
do {
    System.out.println("Introduce una edad: ");
    edad = teclado.nextInt();
    suma+=edad;
    totalAlumnos++;
    System.out.println("Deseas añadir más edades (S/N) ");
    respuesta=teclado.next().charAt(0);
} while(respuesta=='s' || respuesta=='S');
media= (float)suma/totalAlumnos;
System.out.println("La edad media es: " + media);
```

12.- Solicitar por teclado las edades de los alumnos de una clase y calcular la edad media de la clase. No sabemos la cantidad exacta de alumnos; introducen 999 como edad para decir que no hay más alumnos pero hay un límite máximo de 20.

```
Scanner teclado = new Scanner(System.in);
int edad=0, suma=0;
int totalAlumnos=0;
float media;
while(edad !=999 && totalAlumnos < 20) {
    System.out.println("Introduce una edad (999 para terminar): ");
    edad = teclado.nextInt();
    if (edad!=999) {
        suma+=edad;
        totalAlumnos++;
    }
}
media= (float)suma/totalAlumnos;
System.out.println("La edad media es: " + media)
```

4. Cadenas

Una cadena es una secuencia de caracteres, ya sean letras, números u otros símbolos. En Java, como ya comentamos, no son un tipo de dato primitivos, sino que son objetos, por lo que además de aprender a tratarlas nos servirán como primer contacto con el mundo de los objetos.

String

Cuando hablamos de objetos, el tipo de objeto se llama **clase**. La clase para tratar con cadenas se llama **String**. Cuando escribimos:

```
int x=3; decimos que creamos una variable llamada x que es de tipo int o entero.
```

Análogamente cuando escribimos:

```
String cad="Hola mundo"; decimos que creamos un objeto u instancia de la clase String llamada cad.
```

En realidad, la sintaxis pura para crear una instancia de un objeto, a nivel general, sería:

```
String cadena = new String("Hola");
```

Es decir, la forma de crear nuevos objetos de forma general es mediante el operador **new**. En el caso de las cadenas podemos emplear ambas sintaxis y otras clases tienen otros formatos para crearlas.

La gran diferencia que nos vamos a encontrar con una variable de un tipo primitivo es que tiene definidas una serie de operaciones (que llamaremos **métodos**) que nos facilitarán el trabajo con ellas. Por ejemplo, si queríamos saber si un número era primo o no, teníamos que hacer un bloque de código que lo calculase y nos dijese si lo era o no. En el caso de los objetos muchas de estas operaciones ya están definidas, y solo tenemos que llamarlas y recibir el valor que nos devuelven.

Como primer ejemplo, si queremos saber cuántos caracteres tiene una cadena, no tenemos que recorrerla mediante un bucle e ir contando, simplemente invocaremos al método *length()*, que devuelve precisamente la cantidad de caracteres y le asignaremos ese valor devuelto a una variable.

Lo haríamos mediante la siguiente sintaxis:

```
int longitud = cadena.length();
```

Vemos que un método se invoca con un punto y su nombre, después del nombre del objeto. Esto no es nuevo, cuando hacíamos: `System.out.println ("Hola");` estábamos invocando al método *println* de la clase *System.out*.

Los métodos pueden recibir parámetros, como una fórmula de una hoja de cálculo, es decir, valores que le hacen falta para calcular el resultado de la operación que realizan. En el caso de la longitud no necesita ningún parámetro, por eso van los paréntesis sin nada en su interior, y en el caso del *println* necesita la cadena que va a mostrar por pantalla.

El siguiente programa solicita al usuario que teclee una contraseña, que tiene que tener como mínimo 8 caracteres.

```
Scanner teclado = new Scanner (System.in);
String password = new String ();
do {
    System.out.println("Introduce una contraseña (min 8 caract)");
    password = teclado.nextLine();
} while (password.length() < 8);

System.out.println("Su contraseña es " + password);
```

Otros métodos interesantes que tenemos en las cadenas son los siguientes:

- Convertir a mayúsculas:

```
String mayusculas = cadena.toUpperCase();
```

- Convertir a minúsculas:

```
String minusculas = cadena.toLowerCase();
```

- Concatenar cadenas (también podemos concatenar caracteres individuales):

```
String nombre = "José"; String apellido = "López"
```

```
String nombreCompleto = nombre + " " + apellido;
```

También se puede usar la función "concat()" para realizar la misma acción:

```
String nombreCompleto = nombre.concat(" ").concat(apellido);
```

- Obtener una subcadena de la cadena, esto es, recuperar un fragmento del texto que contiene nuestra cadena principal. Para esto se usa un índice inicial y un índice final, tomando en cuenta que el índice inicial de una cadena es "0" y la posición del índice final no se incluye:

```
String frase = "El perro y el gato juegan juntos";
```

```
String subcadena1 = frase.substring(3, 8);
```

Subcadena1 sería igual a = "perro"

- Obtener una cadena solo a partir de un índice inicial hasta el final de la cadena:

```
String subcadena2 = frase.substring(3);
```

Subcadena2 sería igual a = "perro y gato juegan juntos"

- Encontrar la posición en la que se obtiene un carácter o una subcadena. Devuelve un entero que representa la posición de la primera ocurrencia del elemento buscado, o bien -1 si no lo encuentra. Tenemos varios formatos, dependiendo si buscamos caracteres o cadenas y si especificamos una posición de inicio de la búsqueda o no:

```
pos= frase.indexOf ('x');  
pos= frase.indexOf ('x',posini);  
pos= frase.indexOf ("texto");  
pos= frase.indexOf ("texto",posini);
```

Ejemplo:

```
String frase = "Ellos juegan juntos";  
int posic1 = frase.indexOf("ju"); // posic1 sería 6  
int posic2 = frase.indexOf("ju",8); // posic2 sería 13  
int posic = frase.indexOf("ju",14); // posic 3 sería -1
```

- Obtener el carácter que hay en una posición específica de una cadena:

```
char letra = frase.charAt (3); //en el ejemplo anterior, letra sería igual a = 'o'
```

Ejemplo de programa que muestra cada letra de una cadena en una línea distinta, saltándose los espacios en blanco:

```
String cad1 = "Este programa muestra cada letra en una línea";  
for (int i = 0; i < cad1.length(); i++) {  
    char letra = cad1.charAt(i);  
    if (letra != ' ') System.out.println(letra);  
}
```

Comparar cadenas en Java

Los String son objetos, y siempre que queramos comparar un objeto con otro, y por tanto un String con otro, debemos usar el método `equals()` al que se pasa otro objeto (otro String en este caso). El método devuelve `true` si ambas cadenas son iguales.

```
if(frase.equals(cad2)) System.out.println("Cadenas iguales");
```

Es importante destacar que la comparación mediante `==` comprueba si son la misma instancia, no si el contenido es igual. Por ejemplo:

```
String cadena1 = new String("Hola");
String cadena2 = new String("Hola");
if (cadena1 == cadena2) // sería siempre False
```

También tenemos el método `cadena.compareTo(cad2)` que devuelve 0 si es igual a la cadena comparada, <0 si la cadena inicial es menor que la pasada por parámetro o >0 si es mayor. El método `compareToIgnoreCase()` es análogo, pero ignorando mayúsculas o minúsculas.

Valor null

Como hemos comentado, un String es un objeto, y los objetos pueden no tener ningún valor (a diferencia de los tipos primitivos). Para hacer que una cadena no almacene nada le asignamos el valor `null`. `String s = null;`

También podemos preguntar si una cadena es `null`. De hecho, en algunos casos será aconsejable hacerlo antes de llamar a algún método, ya que en caso de que la cadena sea `null`, se producirá una excepción y nuestro programa terminará abruptamente.

```
if (s!=null) System.out.println(s.length());
```

String dispone de algún método que nos puede parecer que es lo mismo que preguntar por `==null` pero no producen el mismo resultado:

- `isEmpty()` devuelve true si la cadena vacía, esto es "", sin ningún carácter, pero no es lo mismo que `null`.
- `isBlank()` devuelve true si la cadena contiene solo espacios en blanco, esto es: " ", o bien " ", etc. (también si es "", sin nada dentro devuelve true).

Reemplazar caracteres

Para reemplazar caracteres en una cadena disponemos de varios métodos de la clase `String`. Como ya sabemos, los `String` son inmutables, por lo que estos métodos no modifican la cadena si no que devuelven una nueva cadena con los reemplazos realizados.

- `replace(char charbuscado, char charnuevo)` / `replace(String textobuscado, String textonuevo)` reemplaza todas las ocurrencias del primer parámetro por el segundo. Ejemplo:

```
String original = "Este barco es el barco más grande";
String nueva = original.replace("barco", "coche");
```

La cadena `nueva` contendrá: `Este coche es el coche más grande`

- `replaceAll (String expresionregular, String textonuevo)` reemplaza todos los caracteres de la cadena que encajen con la expresión regular por el texto nuevo. Ejemplo:

```
String original = "Mi numero de telefono es 900.123.456";
String nueva = original.replaceAll("[0-9]", "X");
```

La cadena `nueva` contendría: Mi numero de telefono es XXX.XXX.XXX

Una **expresión regular** es un patrón que describe a una cadena de caracteres. Todos hemos utilizado alguna vez la expresión `*.doc` para buscar los documentos en algún lugar de nuestro disco, pues bien, `*.doc` es un ejemplo de una expresión regular que representa a todos los archivos con cualquier secuencia de caracteres y que terminan con `.doc`.

Las expresiones regulares se rigen por una serie de normas y hay una construcción para cualquier patrón de caracteres, y permiten desde validar formatos de fechas, la sintaxis de una dirección de email, etc. Puedes consultar el capítulo 16 para más información.

Conversión entre tipos

Se pueden convertir en ambos sentidos, es decir, crear una cadena a partir de un tipo primitivo:

```
String cadena = String.valueOf(123);
```

Incluso en cualquier base numérica, usando los métodos siguientes:

```
String cadena = Integer.toString(255,16);      //devuelve "ff"
String cadena = Integer.toBinaryString(253);    //devuelve "11111101"
```

Y lo contrario, podemos convertir cadenas a los siguientes tipos de datos: Byte, Integer, Double, Float, Long y Short. Esto es útil, por ejemplo, si leemos un número, pero queremos tratar cada uno de sus dígitos por separado. Podemos convertirlo en una cadena y acceder a cada dígito individualmente.

```
int numeroEntero = Integer.parseInt("10");
int numeroEntero = Integer.parseInt("F7A3", 16);
float numeroFloat = Float.parseFloat("20");
double numeroDouble = Double.parseDouble("25.5");
long numeroLong = Long.parseLong("123456");
```

Recordemos que había un "bug" en la entrada de datos con Scanner cuando se hacía `.nextLine()` después de un `.nextFloat()` ó `.nextInt()`... Con `Float.parseFloat(teclado.nextLine())` se soluciona este problema, pero ojo: en este caso el separador decimal es punto y no coma.

Clase Character

En muchas ocasiones necesitaremos tratar cada una de las posiciones de la cadena (o bien alguna de ellas), para realizar alguna operación sobre la misma, por ejemplo, saber si el carácter de una posición es un dígito o es una letra minúscula.

Para ello Java nos ofrece estas "funciones":

- `Character.isLetter(ch1)` //devuelve true o false
- `Character.isDigit(ch1)` //devuelve true o false
- `Character.isSpaceChar(ch1)` //devuelve true o false
- `Character.isUpperCase(ch1)` //devuelve true o false
- `Character.isLowerCase(ch1)` //devuelve true o false
- `Character.toUpperCase(ch1)` //devuelve un char
- `Character.toLowerCase(ch1)` //devuelve un char
- `Character.toString(ch1)` //devuelve un String
- `Character.getType(ch1)` //devuelve la categoría del carácter.

Cuando avancemos en el tema de las clases y los objetos, veremos que tanto `Integer` como `Character` son clases envoltorio (wrapper class), que “envuelven” al tipo primitivo correspondiente (en este caso `int` y `char` respectivamente), con el mismo contenido que el tipo que envuelven, pero en forma de objeto con propiedades y métodos muy útiles.

También veremos que los métodos que acabamos de mencionar, con el formato `Integer.toString`, `Character.isLetter` son “de clase” o “estáticos”, es decir, que no hace falta crear una instancia de la clase para poder utilizarlos. Otros métodos, como `cadena.toUpperCase()`, requieren ser llamados sobre una instancia de una clase (en este caso `cadena` es una instancia de la clase `String`)

Format

Otra utilidad interesante es `format` para unir textos y variables con un formato definido por nosotros. Para entenderlo basta este ejemplo sencillo:

```
int edad = 28;
String nombre = "David";
String resultado = String.format("Nombre:%s,edad:%d años",nombre,edad);
```

Obtendríamos la cadena: *Nombre: David, edad:28 años*

Es decir, sustituye los símbolos `%` por las variables que van después del primer parámetro. Obviamente debe haber tantos `%` en el primer parámetro, como parámetros adicionales. En este ejemplo: dos. Pueden ser en distinto orden, pero habría que especificarlo. La sintaxis completa de los especificadores sería esta (entre corchetes opcionales):

`%[argument_index] [flags] [width] [.precision] type`

Donde:

% marca el inicio de cada uno de los argumentos. Es obligatorio.

[argument_index] indica cuál de los argumentos se va a utilizar. Se hace con un número seguido de `$`. En general no se usa este parámetro ya que se toman los argumentos en el mismo orden que aparecen, pero mostramos un ejemplo de todas formas:

```
String str = String.format("Fecha: %3$d-%2$s-%1$d",5, "Febrero", 2019) ;
```

Produciría la salida: Fecha: 2019-Febrero-5

Como comentamos, cambiando el orden de los argumentos obtenemos el mismo resultado:

```
String str = String.format("Fecha: %d-%s-%d",2019, "Febrero", 5);
```

flags: modifican apariencia:

- (guión -) : justifica el argumento a la izquierda.
- # : incluye un radical para enteros y una coma para decimales.
- + : incluye el signo (+ en los positivos, el menos se incluye siempre).
- " "
- : espacio en los valores positivos.
- 0 : se llenan los espacios con ceros en los valores positivos.
- ,
- : usa un separador de grupo específico según la zona geográfica.
- (: encierra los números negativos en paréntesis.

width: indica el número mínimo de caracteres que tendrá la salida.

.precisión: (empieza por punto) indica el número máximo de dígitos después de la coma decimal.

type: (obligatorio) indica de qué tipo es el argumento. Los más usados son:

- b: boolean
- c: carácter Unicode
- d: número entero
- f: número decimal
- s: String
- t: fecha y hora

Ejemplos:

Instrucción	Salida
<code>int num = 1234; String.format("El número es: %08d", num);</code>	00001234
<code>float num = 1.5; String.format("El número es: %08.3f", num);</code>	0001,500
<code>float num = 1.5; String.format("El número es: %+2f", num);</code>	+1,50

Existen otras clases que podemos utilizar para formatear como pueden ser `Formatter` o `DecimalFormat`, con funcionalidad similar y parecida forma de trabajo.

printf

Este método de la clase `System.out` tiene la misma funcionalidad que los vistos previamente: `print` y `println` pero siguiendo el mismo patrón de funcionamiento que el método `format` que acabamos de ver. Así podemos mostrar por consola salidas formateadas:

```
System.out.printf("cateto:%d > cateto:%d >> %.2f%n",
                   cat1, cat2, Math.hypot(cat1, cat2));
```

Con `printf`, en vez de emplear `\n` para el salto de línea, se suele usar `%n` ya que tiene mayor compatibilidad en distintas plataformas.

StringBuilder y StringBuffer

Además de `String`, existen otras clases como `StringBuffer` y `StringBuilder` que resultan de interés porque facilitan cierto tipo de trabajos y aportan mayor eficiencia en determinados contextos.

La clase `StringBuilder` es similar a la clase `String` pero presenta algunas diferencias relevantes:

- Su tamaño y contenido pueden modificarse a diferencia con los `String`.
- Debe crearse con alguno de sus constructores asociados. No se permite instanciar directamente a una cadena como sí permiten los `String`.
- Un `StringBuilder` está indexado. Cada uno de sus caracteres tiene un índice: 0 para el primero, 1 para el segundo, etc.
- Los métodos de `StringBuilder` no están sincronizados. Esto implica que es más eficiente que `StringBuffer` siempre que no se requiera trabajar con múltiples hilos (*threads*), que es lo más habitual (en caso de trabajar con hilos se recomienda `StringBuffer`).

Los constructores de *StringBuilder* se resumen en la siguiente tabla:

Constructor	Descripción	Ejemplo
<code>StringBuilder()</code>	Construye un <i>StringBuilder</i> vacío y con una capacidad por defecto de 16 caracteres.	<code>StringBuilder s = new StringBuilder();</code>
<code>StringBuilder(int capacidad)</code>	Se le pasa la capacidad (número de caracteres) como argumento.	<code>StringBuilder s = new StringBuilder(55);</code>
<code>StringBuilder(String str)</code>	Construye un <i>StringBuilder</i> en base al <i>String</i> que se le pasa como argumento.	<code>StringBuilder s = new StringBuilder("hola");</code>

Los métodos principales de *StringBuilder* se resumen en la siguiente tabla:

Retorno	Método	Explicación
<i>StringBuilder</i>	<code>append(...)</code>	Añade al final del <i>StringBuilder</i> a la que se aplica, un <i>String</i> o la representación en forma de <i>String</i> de un dato asociado a una variable primitiva
<i>int</i>	<code>capacity()</code>	Devuelve la capacidad del <i>StringBuilder</i>
<i>int</i>	<code>length()</code>	Devuelve el número de caracteres del <i>StringBuilder</i>
<i>StringBuilder</i>	<code>reverse()</code>	Invierte el orden de los caracteres del <i>StringBuilder</i>
<i>void</i>	<code>setCharAt(int indice,char ch)</code>	Cambia el carácter de la posición indicada en el primer argumento por el carácter que se le pasa en el segundo
<i>char</i>	<code>charAt(int índice)</code>	Devuelve el carácter asociado a la posición que se le indica en el argumento
<i>void</i>	<code>setLength(int nuevaLongitud)</code>	Modifica la longitud. La nueva longitud no puede ser menor
<i>String</i>	<code>toString()</code>	Convierte un <i>StringBuilder</i> en un <i>String</i>
<i>StringBuilder</i>	<code>insert(int indiceIni,String cadena)</code>	Añade la cadena del segundo argumento a partir de la posición indicada en el primero
<i>StringBuilder</i>	<code>delete(int indiceIni,int indiceFin)</code>	Borra la cadena de caracteres incluidos entre los dos índices indicados en los argumentos
<i>StringBuilder</i>	<code>deleteCharAt(int indice)</code>	Borra el carácter indicado en el índice
<i>StringBuilder</i>	<code>replace(int indiceIni, int indiceFin,String str)</code>	Reemplaza los caracteres comprendidos entre los dos índices por la cadena que se le pasa en el argumento
<i>int</i>	<code>indexOf (String str)</code>	Analiza los caracteres de la cadena y encuentra el primer índice que coincide con el valor deseado
<i>String</i>	<code>subString(int indiceIni,int indiceFin)</code>	Devuelve una cadena comprendida entre la posición inicial incluida y la final (no incluida)

La clase **StringBuffer** es similar a la clase *StringBuilder*, con los mismos métodos y constructores, pero sus métodos están sincronizados, permitiendo trabajar con múltiples hilos *threads*.

Métodos de StringBuilder vs métodos de String

El uso de los métodos de `StringBuilder` y `StringBuffer` difiere un poco de los `String`, ya que como estos últimos eran inmutables, devolvían el resultado del método, pero no modificaban el objeto en sí, por ejemplo:

```
cad.toUpperCase();
```

no modificaba `cadena` así que hay que llamarla así:

```
cad=cad.toUpperCase();
```

Esto no ocurre con los `StringBuilder/StringBuffer`, los métodos sí modifican el contenido del objeto, así podríamos hacer directamente:

```
StringBuilder sb = new StringBuilder ("abcdefg");
sb.delete(3,5);
```

Es frecuente convertir `String` a `StringBuider` si necesitamos un método de una de las clases que está disponible en la otra. Por ejemplo, si necesitamos eliminar en la tercera posición de una cadena, podríamos hacer:

```
String cadena = "abcdef";
StringBuilder sb = new StringBuilder(cadena);
sb.deleteCharAt(3);
cadena = sb.toString();
```

abreviando:

```
cadena = new StringBuilder(cadena).deleteCharAt(3).toString();
```

Errores Frecuentes:

- ✓ Hay que comparar los `String` con `equals()`, `compareTo()`, `compareToIgnoreCase()`, pero nunca con `==`.
- ✓ Los `String` comienzan en la posición cero, no en la 1.
- ✓ El último carácter de una cadena está en `length()-1`. Es erróneo `cadena.charAt(cadena.length())`.
- ✓ El método `substring (x,y)` no incluye la posición 'y', acaba en la anterior a 'y'.
- ✓ Una operación sobre un `String` crea un nuevo `String`, no modifica el actual.

5. Funciones

Las funciones en un lenguaje no orientado a objetos son bloques de código que pueden ser invocados pasándoles ciertos parámetros y que pueden devolver un valor de cualquier tipo de datos. Es frecuente que desde el cuerpo principal de un programa se llame a funciones para que realice ciertas tareas y así estructurar mejor el código.

Un ejemplo típico de función podría ser una llamada `esPrimo` a la que se le pasase como parámetro un número entero y devolviese verdadero o falso (la función sería un concepto parecido a una fórmula de Excel).

La ventaja que tiene una función es que, una vez codificada y probada, la podremos usar en multitud de programas, reutilizando código previo y quedando un código más claro. Vamos primero a ver su estructura y luego veremos un ejemplo de un programa con y sin funciones para ver sus diferencias.

En Java no existe el concepto de función, es tratado como un método estático del programa que lo usa (el programa es en realidad una clase), pero como aún no hemos llegado al tema de la orientación a objetos, vamos a obviar esto por ahora. Retomaremos esto en el tema siguiente.

La definición de una función, con lo que sabemos por ahora, la podemos simplificar así:

```
static tipodevuelto nombreFuncion (tipo param1, tipo2 param, ...) {
    //cuerpo de la función
    //return finaliza la función devolviendo un valor.
}
```

Siendo:

- *tipodevuelto*: el tipo de dato que devuelve la función, puede ser un tipo primitivo (int, boolean, etc.) o bien una clase Java (por ejemplo, String) o una clase creada por nosotros (Alumno, Teléfono, etc.)
- *nombreFunción*: es el nombre que le damos a la función: esPrimo, calcularFactura, etc... Por convenio suele empezar por minúscula, aunque no es obligatorio.
- *Tipo param1*: entre paréntesis figurarán unos identificadores precedidos de su tipo, que representan los parámetros que le pasamos a la función para que realice los cálculos necesarios.
- En el cuerpo de la función irían todos los cálculos, incluyendo uno o varios *return* que finalizan la función en ese momento, devolviendo un valor acorde a lo definido en la cabecera de la función.

La función que verifica si un número pasado como parámetro es primo sería algo así:

```
10  public static boolean esPrimo (int num) {
11      for (int i=2;i<num;i++) {
12          if (num % i ==0) return false;
13      }
14      return true;
15 }
```

Fíjate que usamos la característica del *return*, que termina abruptamente la ejecución del método, sin finalizar el bucle, para reducir el número de líneas de código con respecto al cálculo de número primo realizado al principio de este manual. Si la función llega a ejecutar todas las iteraciones del bucle, llegaría a la siguiente instrucción, que devolvería *true*.

Ahora, desde el programa (main) o incluso desde otra función podríamos llamar a esa función:

```
int x=7;
boolean primo = esPrimo (x);
if (primo) System.out.println ("Sí es primo");
```

La llamada se podría hacer en el propio if, siendo el valor devuelto por la función lo que se evalúa:

```
if (esPrimo(7)==true) System.out.println ("Sí es primo" o bien
if (esPrimo(7)) System.out.println ("Sí es primo");
```

Los **parámetros** de un método son los valores que este recibe por parte del código que lo llama. Pueden ser tipos primitivos u objetos y en la declaración del método se escriben después del nombre del método entre paréntesis indicándose el tipo de cada uno. Ejemplo:

```
sumarImpuesto (float importe, float impuesto) {
    importeTotal = importe + importe * impuesto/100f;
    . . .
}
```

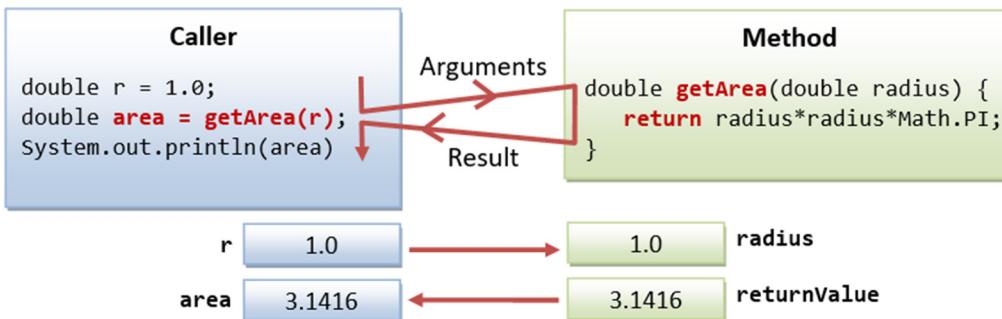
y la llamada sería algo como `sumarImpuesto (12.2 , x); //con x=21 p.ej.`

Hay que destacar que el nombre que tienen los parámetros en la cabecera de la función no tienen nada que ver con el nombre de las variables con las que son llamadas las funciones.

Es un concepto similar a los parámetros que le pasamos a las fórmulas de una hoja de cálculo, por ejemplo =SUMA(A2,B3).

Es importante destacar que, cuando se ejecuta el **return** de un método finaliza la ejecución del mismo y la expresión del *return* es lo que se devuelve a la sentencia que lo llamó. El tipo de dato devuelto tiene que coincidir con el que se especifica antes del nombre del método. Ejemplo:

Función: double sumarIva (float euros) { return euros * 1.21; }
Llamada: importeTotal = sumarIva (12.50F);



En un programa estructurado en funciones, las variables definidas en una función solo tendrán vigencia en esa función, concretamente en el bloque en el que se hayan definido (el bloque lo marcan las llaves `{ bloque }`). Si necesitamos una variable común a todas las funciones, una variable que se pueda usar a lo largo de todo el programa, debemos definirla después del nombre del programa. A este tipo de variables, visibles desde todo el programa, se le denominan **variables globales**.

En el siguiente ejemplo, la variable `teclado` (que es un objeto de la clase Scanner), se usará a lo largo de todo el programa, por eso debe ser definida como global.

Las variables globales deben llevar el prefijo "static" antes de su tipo.

```

import java.util.Scanner;
public class Ejemplo {
    static Scanner teclado;

    static void main(String[] args) {
        teclado = new Scanner (System.in);
        String nombre = pedirNombre();
        int edad = pedirEdad();
        System.out.printf("Hola %s. Tienes %d años%n", nombre, edad);
    } //fin main

    static String pedirNombre() {
        System.out.println("Introduce tu nombre:");
        return teclado.nextLine();
    }
    static int pedirEdad() {
        System.out.println("Introduce tu edad:");
        return teclado.nextInt();
    }
}

```

Para terminar este apartado de funciones vamos a escribir un mismo programa sin funciones y con funciones para comprobar lo claro que queda el código en el segundo, caso comparado con el primero.

Se trata de un programa que calcula el día siguiente a partir de una fecha introducida previamente y suponiendo que la fecha es válida. El programa debe distinguir 3 situaciones: que sea fin de año, que sea fin de mes y no fin de año y resto de los casos.

Vemos en el primer ejemplo como el código está todo mezclado y es difícil de entender.

```

public class diaSiguiente2 {
    public static int dia,mes,año; //variables globales
    public static void main(String[] args) {
        Scanner teclado =new Scanner(System.in);
        System.out.print("Introduce dia: "); dia = teclado.nextInt();
        System.out.print("Introduce mes: "); mes = teclado.nextInt();
        System.out.print("Introduce año: "); año = teclado.nextInt();
        if (dia == 31 && mes == 12) {dia=1; mes=1; año++;}
        else { int diasmes;
            if (mes == 2) {
                if (año %4==0 && año%100!=0 || año%400==0)
                    diasmes = 29;
                else diasmes = 28;
            }
            else { if (mes==4 || mes==6 || mes==9 || mes==11)
                    diasmes = 30;
                else diasmes = 31;
            }
            if (dia == diasmes ) {dia=1;mes++;}
            else dia++;
        }
        System.out.println("Sig:"+dia+"/"+mes+"/"+año);
    } //fin main
} //fin programa

```

En el segundo caso, la estructura del programa, con funciones es mucho más claro.

```
public class diaSiguiente {
    public static int dia,mes,año; //variables globales
    public static void main(String[] args) {
        Scanner teclado = new Scanner (System.in);
        System.out.print("Introduce día: ");dia=teclado.nextInt ();
        System.out.print("Introduce mes: ");mes=teclado.nextInt ();
        System.out.print("Introduce año: ");año=teclado.nextInt ();
        sumaDia();
        System.out.printf("dia siguiente:%d/%d/%d%n",dia,mes,año);
    }
    public static void sumaDia(){
        if (dia == 31 && mes ==12 ){dia=1;mes=1;año++;}
        else{
            int diasMes = cantidadDiasMes (mes,año);
            if (dia ==diasMes) {dia=1;mes++;}
            else dia++;
        }
    }
    public static int cantidadDiasMes (int mm, int aa){
        if (mm==2){if (añoBisiesto(aa)) {return 29;}
                    else {return 28;} //else opcional
        }
        if (mm==4 || mm==6 || mm==9 || mm==11) return 30;
        return 31;
    }
    public static boolean añoBisiesto (int a) {
        if (a%4==0 && a%100!=0 || a%400==0) return true;
        return false;
    }
}
```

El programa anterior no es aún así una solución perfecta, ya que la función *sumaDia()* lee y escribe las variables globales *dia*, *mes* y *año*. Lo ideal es que las funciones reciban como parámetros todo lo que necesitan para funcionar y que devuelvan su resultado en el *return*, sin emplear variables globales. Así son más fáciles de reutilizar en otros programas.

Vamos a ver a continuación como las funciones *añoBisiesto* o *CantidadDiasMes* pueden ser reutilizadas en otros programas, reduciendo nuestro trabajo de programación. Supongamos que ahora necesitamos hacer un programa que calcule el día anterior a una fecha introducida. Aprovechando las funciones creadas, el programa sería así:

```
public class diaAnterior {
    public static int dia,mes,año; //variables globales
    public static void main(String[] args) {
        Scanner teclado = new Scanner (System.in);
        System.out.print("Introduce día: ");dia=teclado.nextInt ();
        System.out.print("Introduce mes: ");mes=teclado.nextInt ();
        System.out.print("Introduce año: ");año=teclado.nextInt ();
        restaDia();
        System.out.printf("dia anterior:%d/%d/%d%n",dia,mes,año);
    }
    public static void restaDia(){
        if (dia == 1 && mes ==1 ) {dia=31;mes=12;año--;}
        else{
            if (dia==1){
                dia = cantidadDiasMes (mes-1,año);
                mes--;
            }
            else dia--;
        }
    }
}
```

Un aspecto importante en el diseño de funciones es separar la lógica de negocio (los cálculos) de la interfaz con el usuario (aplicación de consola en nuestro caso). Así, las funciones que se encarguen de lógica de negocio podremos usarlas en otro tipo de programas como entorno web, móvil, etc.

En cuanto a la codificación de funciones, al igual que los programas, se puede reducir mucho, aprovechando que `return` termina la ejecución de una función. Os dejo un chiste sobre este asunto.

```
//newbie, 11 lines
static boolean EvenNumber (int a)
{
    boolean par;
    if (a % 2 == 0) {
        par = true;
    }
    else {
        par = false;
    }
    return par;
}

//1 year later 4 lines
static boolean EvenNumber (int a) {
    if (a % 2 == 0) return true;
    else return false;
}

//2 years later + lazy writing, 1 line
static boolean EvenNumber (int a) {return a%2==0;}
```

Paso de Parámetros por Valor

En general, en los lenguajes de programación, se suele decir que hay dos posibilidades en cuanto a al paso de parámetros: por valor o por referencia.

- **Paso de parámetros por valor** significa que se pasa al método o función una copia del valor con el que se ha llamado al método, pero la variable “llamante” no es modificada, después de la ejecución del método, la variable sigue valiendo lo mismo.
- **Paso de parámetros por referencia** significa que se pasa al método o función la referencia de memoria de la variable “llamante” de forma que además de tener el valor de esa variable, podemos modificar su valor en el método.

En Java, los parámetros se pasan por valor. Lo comprobamos viendo la ejecución de este código:

```
public class Main {
    public static void main(String[] args) {
        int a = 10;
        doble (a);
        System.out.println(a);
    }
    static void doble (int x) { x = x *2;}
}
```

El programa mostaría “10”. Si los parámetros se pasasen por referencia, mostaría 20.

En el tema siguiente, veremos que en el caso de los objetos el comportamiento es ligeramente diferente.

6. Clases y Objetos

Conceptos Básicos

La programación Orientada a objetos (POO) es una forma especial de programar, más cercana a como expresaríamos las cosas en la vida real que otros tipos de programación. Por ejemplo, vamos a pensar en un coche para tratar de modelizarlo en un esquema de POO. Diríamos que el coche es el elemento principal que tiene una serie de características, como podrían ser la *matrícula*, el *color*, el *tamaño del depósito*, *cantidad actual de combustible* o la *velocidad máxima*. Además, tiene una serie de funcionalidades asociadas, como pueden ser *ponerse en marcha*, *recorrer kilómetros*, *parar* o *aparcar*.

En programación esas características se llaman **propiedades** y son similares a las variables que hemos visto previamente. Las funcionalidades se llaman **métodos** y son bloques de código que tiene una finalidad concreta. Esos métodos pueden modificar las propiedades de un objeto, por ejemplo, *recorrer kilómetros* reduciría la *cantidad actual de combustible*.

Atributos

- Valores o características de los objetos
- Permiten definir el **estado** del objeto u otras cualidades



- Velocidad
 - Aceleración
 - Capacidad de combustible
- variables**
- Marca
 - Color
 - Potencia
 - Velocidad máxima
 - Carburante
- constantes**

Métodos (u operaciones)

- Acciones que puede realizar un objeto



- Arrancar motor
 - Parar motor
 - Acelerar
 - Frenar
 - Girar a la derecha (grados)
 - Girar a la izquierda (grados)
 - Cambiar marcha (nueva marcha)
- método** **argumentos o parámetros**

En general se distinguen dos conceptos, la **clase**, y el **objeto**. La clase es la declaración o definición del objeto, es como una abstracción que determina como serán los objetos. Cuando esa abstracción toma vida, ahí hablamos del objeto. En el ejemplo anterior del coche, estábamos hablando de la *clase coche*, un objeto coche sería un coche concreto, por ejemplo, un coche con matrícula "1111-XXX", color rojo, con un depósito de 50 litros, que actualmente dispone de 30 litros y con velocidad máxima de 200 km/h.



Si sobre ese coche invocamos al método *recorrer_kilometros(100)* se reducirá unos cuantos litros el carburante disponible. Si una de las propiedades del objeto fuese su *ubicación*, dicho método también la cambiaría.

A partir de una **clase** se pueden crear múltiples objetos, todos con el mismo comportamiento (**métodos**) pero con valores propios asignados a sus **atributos**, lo que determinará el **estado** de cada objeto. Durante la ejecución del programa los objetos se crean, se ejecutan métodos sobre ellos y cuando ya no son necesarios, se destruyen.

Esto es solo lo más básico de la POO. Este paradigma contiene mecanismos como la herencia y el polimorfismo que lo dotan de gran potencia. Por ahora podemos quedarnos con la idea del **abstracción** y **encapsulamiento**, esto es, una vez definida y programada una clase podemos usarla en miles de programas distintos, pero sin conocer "las tripas" de ese objeto. La abstracción hace referencia a enfatizar las características y comportamiento de los objetos que son necesarios, prescindiendo de los que no son. El encapsulamiento complementa a la abstracción y se refiere a ocultar al exterior cómo se han programado las funcionalidades de la clase, se centra en mostrar el "qué es/qué hace" y no en "cómo lo hace" haciendo que desde el exterior solo se pueda acceder a la información relevante.

A veces se habla de "caja negra", haciendo referencia a que el comportamiento y atributos del objeto son conocidos, pero no así su trabajo interno, el cual continúa siendo un misterio. Veremos que cuando definimos una clase, le podemos asignar diferentes *modificadores de acceso*, que determinarán si ese atributo o método puede verse desde fuera o no.

Esto ocurría con los *String* vistos previamente, que como comentamos son objetos, una vez creada una cadena nosotros podíamos invocar a un método, por ejemplo

```
micadena = micadena.toUpperCase();
```

y sabíamos que realizaba una determinada operación, pero sin necesitar saber cómo funciona internamente.

Veamos otra forma de aproximarnos a los objetos: Imaginemos que hay creada una clase llamada *JuegoAjedrez*, cuyas propiedades ni siquiera conocemos pero que dispone de los métodos:

- *moverJugador*, a la que le proporcionamos como parámetro una casilla origen y una casilla destino y devuelve true si el movimiento es correcto o false si el movimiento es incorrecto.
- *moverMaquina*, el ordenador piensa y hace un movimiento.
- *esJaqueMate*, devuelve 0 si no hay jaque mate, 1 si gana blancas, 2 si gana negras
- *pintarTablero*, muestra la situación actual del tablero.

Como primer contacto, los **métodos** parecen que son como las **funciones** vistas previamente, pero aplicadas sobre una clase. En realidad, es todo lo contrario, una función es un método y la clase a la que pertenece dicho método es el programa en el que se crea.

Esto nos lleva a otra reflexión: los programas en Java son clases: sus atributos son lo que llamamos previamente variables globales del programa, y sus métodos son las funciones del programa. Más adelante descubriremos porque tanto esas variables globales como a las funciones les poníamos el modificador **static**.

Solo con esta información, y sin tener apenas ningún conocimiento sobre el ajedrez, podríamos hacer un programa en el que el ordenador jugase al ajedrez contra el usuario.

Por ejemplo, así:

```

import java.util.Scanner;
public class Ejemplo {
    public static void main (String args[]) {

        boolean fin=false, mov;
        Scanner teclado = new Scanner (System.in);
        int filaini,colini,filafin,colfin;

        JuegoAjedrez juego = new JuegoAjedrez();
        juego.pintarTablero();
        while (!fin) {
            juego.moverMaquina();
            juego.pintarTablero();
            fin = juego.esJaqueMate();
            if (!fin) {
                System.out.println("Introduce fila/col inicial y final"));
                filaini = teclado.nextInt(); colini = teclado.nextInt();
                filafin = teclado.nextInt(); colfin = teclado.nextInt();
                do {
                    mov=juego.moverJugador(filaini,colini,filafin,colfin);
                } while (!mov);
                fin = juego.esJaqueMate();
            }
            juego.pintarTablero();
        }
    }
}

```

Si analizamos el ejemplo anterior, vemos que en la línea:

```
JuegoAjedrez juego = new JuegoAjedrez();
```

se crea un objeto de tipo *JuegoAjedrez*. El método mediante el cual se crea se llama **constructor** y si nosotros no lo creamos, se crea por defecto (puede interesarnos crearlo para inicializar el objeto con unos valores determinados).

El constructor es la primera operación que debemos hacer con un objeto, y es el momento en el que nace el objeto, a partir de ese momento podremos invocar a sus métodos o leer y modificar sus propiedades. El nombre del constructor es siempre igual al de la clase y no devuelve ningún valor. Su sintaxis es:

```
NombreClase nombreObjeto = new NombreClase();
```

Una vez creado el juego, el programa es un bucle de movimientos por parte de la máquina y el jugador, comprobando si alguno hace algún movimiento ganador, y repintando el tablero cada vez que hay un movimiento. El programa funciona perfectamente y todo el código referente al juego de ajedrez ha quedado oculto en la clase *juegoAjedrez*.

El **constructor** es un método que se llama igual que su clase y que sólo puede recibir valores, pero nunca retornar ningún valor. Gracias a ellos se inicializan los atributos de objeto, bien con valores pasados como parámetro al constructor, bien con valores por defecto. Si no creamos un constructor, Java construye uno por defecto, sin parámetros, que simplemente crea el objeto, pero no inicializa ninguno de sus atributos ni hace ninguna otra tarea, solo crea el objeto.

Crear una Clase Paso a Paso

Vamos a construir una clase poco a poco, añadiéndole funcionalidad en pasos sucesivos y creando objetos, o lo que es lo mismo, **instancias** de la clase.

El primer paso sería definir el nombre de la clase y sus propiedades. Utilizamos la palabra reservada `class` y las propiedades serán variables, bien de tipos primitivos bien de otras clases:

```
class Vehiculo {
    String matricula;           //número de matrícula de cada coche
    int cantPasajeros;          //cantidad de pasajeros
    int tamDeposito;            //tamaño del depósito
    float consumo;               //consumo en litros/100km.
}
```

Ahora podríamos crear un programa que crease instancias de Vehículos. Una vez creados podemos asignar valores a cada uno de sus atributos: matrícula, cantidad de pasajeros y consumo:

El siguiente paso sería desarrollar los métodos que caracterizan el objeto o bien los métodos que son necesarios para la funcionalidad que necesitamos. También vamos a crear un constructor que se encargue de inicializar las propiedades de los objetos.

```
package ejemplo;
public class Vehiculo {
    String matricula;
    int cantPasajeros;
    int tamDeposito;
    float consumo;

    Vehiculo(String mat, int cantPas, int tamDep, float con) {
        matricula = mat;
        cantPasajeros = cantPas;
        tamDeposito = tamDep;
        consumo = con;
    }

    float autonomia(){
        return 100 * this.tamDeposito / this.consumo;
    }

    double combustibleNecesario (float kilometros) {
        return (double) this.consumo * kilometros / 100;
    }
}
```

En el constructor vemos como se inicializan las tres propiedades del objeto. Cuando hacemos referencia al propio objeto podemos utilizar opcionalmente la palabra `this` tanto en el constructor como en los métodos. Así pues, el constructor también podría haber quedado así:

```
Vehiculo(String mat, int cantPas, int tamDep, float con) {
    this.matricula = mat;
    this.cantPasajeros = cantPas;
    this.tamDeposito = tamDep;
    this.consumo = con;
}
```

Una clase puede tener varios constructores, cada uno con un número diferente de parámetros o de distinto tipo. Ejemplo: el siguiente constructor fija el número de pasajeros a 5 y ya no es necesario pasárselo como parámetro:

```
Vehiculo(String mat, int tamDep, float con) {
    this.matricula = mat;
    this.cantPasajeros = 5;
    this.tamDeposito = tamDep;
    this.consumo = con;
}
```

La pregunta que surge ahora es ¿Cómo invoco a uno u otro constructor? La respuesta es que el constructor usado vendrá determinado por los parámetros que le pasemos al mismo. Así usaríamos en un programa los constructores que acabamos de crear:

```
Vehiculo vehiculo1 = new Vehiculo ("1234ABC", 4, 50, 7.8f);
Vehiculo vehiculo2 = new Vehiculo ("7777ZZZ", 51, 8.9f);
```

Como ya comentamos, si no creamos ningún constructor, se crea uno por defecto, sin ningún parámetro ni tampoco asigna ningún valor. Sería como si escribiésemos en la clase:

```
Vehiculo() { }
```

Y en el programa crearímos los objetos así:

```
Vehiculo vehiculo3 = new Vehiculo ();
```

Obviamente habría que darle valores a los atributos posteriormente, por ejemplo:

```
vehiculo3.matricula = "AAAAA222";
```

A un constructor, también podemos pasarle un objeto de esa misma clase creado previamente, para que el objeto nuevo use esos valores, por ejemplo, para copiarlos. Añadimos a la clase:

```
Vehiculo(String mat, Vehiculo otroVehiculo) {
    this.matricula = mat;
    this.cantPasajeros = otroVehiculo.cantPasajeros;
    this.tamDeposito = otroVehiculo.tamDeposito;
    this.consumo = otroVehiculo.consumo;
}
```

Desde el programa haríamos así:

```
Vehiculo vehiculo4 = new Vehiculo ("4567ABC", 4, 50, 7.8f);
Vehiculo vehiculo5 = new Vehiculo ("4568ABC", vehiculo4);
```

A la izquierda del igual está la variable que referencia al objeto, que es la forma de acceder a ese objeto. A la derecha, el constructor hace la reserva en memoria y opcionalmente le da valores a sus atributos. Ambas operaciones son necesarias.

Desde un constructor se puede invocar a otro, para reducir el código. Así pues, este último constructor de copia se podría haber escrito así:

```
Vehiculo(String mat, Vehiculo otroVehiculo) {
    Vehiculo (mat, otroVehiculo.cantPasajeros,
              otroVehiculo.tamDeposito, otroVehiculo.consumo);
}
```

El siguiente método, *autonomía()*, vemos que tiene el tipo **float** antes del nombre del método y, ya en el bloque de código, una cláusula **return**. Lo que significa el primero es que el método devuelve un valor de este tipo, **float** en este caso, que podrá ser recibido por una variable cuando sea llamado.

El valor que devuelve es el especificado en la cláusula *return*, en este caso $100 * \text{deposito} / \text{consumo}$. Cabe destacar que este método no necesita ningún parámetro adicional para ser ejecutado ya que toda la información que necesita para calcular la autonomía está en la propia clase. Es por ello que después del nombre aparecen los paréntesis sin nada en su interior () .

El último método, `combustibleNecesario` por el contrario, sí necesita un **parámetro**, si no sabemos cuántos kilómetros va a recorrer, no podremos hacer el cálculo. Como ya comentamos, los parámetros de los métodos se definen con su tipo y el nombre que deseemos para luego, en el cuerpo del método, usarlo como necesitemos. Cuando invoquemos al método, este parámetro deberá tener un valor concreto.

Un programa que utiliza la clase `Vehiculo` con su constructor y usa sus métodos podría ser así:

```
Scanner teclado = new Scanner (System.in);
Vehiculo vehiculo1 = new Vehiculo ("1234ABC", 4, 50, 7.8f);

float autonom = vehiculo1.autonomia();
System.out.println("Autonomía del vehículo: " + autonom);

System.out.println("Introduce los km a recorrer");
float km = teclado.nextInt();

System.out.println("Combustible necesario para esa distancia: " +
    vehiculo1.combustibleNecesario(km));
```

En la siguiente imagen podemos ver una **clase** sencilla llamada '`Persona`' y como en un programa, se crea una **instancia** de esa clase llamada '`p`' invocando al **constructor** de la clase. A continuación, se invoca al **método** '`mayorDeEdad`' de dicha clase, sobre esa instancia de persona '`p`'.

```
Persona.java
Source History
1 package apuntes;
2 public class Persona {
3     public String nombre;
4     public int edad;
5
6     public Persona(String nombre, int edad) {
7         this.nombre = nombre;
8         this.edad = edad;
9     }
10
11    public boolean mayorDeEdad () {
12        if (this.edad > 18 ) return true;
13        else return false;
14    }
15
16 }
17
18

Main.java
Source History
1 package apuntes;
2 import java.util.Scanner;
3 public class Main {
4     public static void main(String[] args) {
5         Scanner teclado = new Scanner (System.in);
6         System.out.println("Introduce nombre:");
7         String nom = teclado.nextLine();
8         System.out.println("Introduce edad:");
9         int ed = Integer.parseInt(teclado.nextLine());
10
11         Persona p = new Persona(nom,ed);
12         boolean mayor = p.mayorDeEdad();
13         if (mayor)
14             System.out.printf("%s es mayor de edad\n", p.nombre);
15         else
16             System.out.printf("%s no es mayor de edad\n", p.nombre);
17     }
18 }
```

Objetos vs. Tipos primitivos

Cuando en un programa se muestra el valor de una variable de tipo primitivo, por ejemplo `System.out.print(x);` suponiendo `int x = 7;` o lo comparamos con un valor `if (x==10)` sabemos que se está accediendo realmente al valor que tiene almacenado la variable, esto es obvio.

Esto no es así en los objetos, por ejemplo, si hacemos `Vehiculo v1 = new Vehiculo (4,50,6);` `v1` contiene una referencia a memoria donde está almacenado el contenido de lo objeto, no al contenido del objeto en sí.

Asimismo, si hacemos `v1 = v2;` no estaríamos copiando el contenido del objeto `v2` a `v1`, sino que haríamos que `v1` "apuntase" ahora al contenido de `v2`, las dos variables "referencian al mismo objeto".

Si no creamos ningún constructor, Java lo crea por defecto, pero sin ninguna funcionalidad añadida, solo se preocupa de crear la instancia del objeto. Si nosotros creamos un constructor, como ha sido el caso de este último ejemplo, Java no crea ese constructor por defecto, por lo tanto, en este último programa, una instrucción de tipo:

```
Vehiculo monovolumen = new Vehiculo ();
```

sería errónea.

Para solucionarlo, deberíamos crear ese constructor sin parámetros en la definición de la clase.

```
Vehiculo() {}
```

Ejemplo 1:

- `Vehiculo v1 = new Vehiculo ("1234ABC", 4, 50, 6.2);`
- `v1 = new Vehiculo ("8888ZZZ", 32, 5.1);`

Hemos perdido el acceso al vehículo "1234ABC" y se borrará, porque v1 ahora "apunta" al vehículo "8888ZZZ".

Ejemplo 2:

- `Vehiculo v1 = new Vehiculo ("1234ABC", 4, 50, 6.2);`
- `Vehiculo v2 = v1;`

No son dos vehículos. Es un único vehículo referenciado por dos variables. Si hacemos un cambio en una variable se verá también en la otra.

Ejemplo 3:

- `Vehiculo v1 = new Vehiculo ("1234ABC", 4, 50, 6.2);`
- `Vehiculo v2 = new Vehiculo (null, 0.0, 0.0);`
- `v2.matricula=v1.matricula; v2.cantPasajeros=v1.cantPasajeros;`
- `v2.consumo=v1.consumo; v2.tamDeposito=v1.tamDeposito;`

Son dos vehículos distintos e independientes, aunque tienen los mismos valores por lo que podríamos decir que son "iguales".

Por último, si hacemos `if (v1==v3)` siempre devolverá `false` si las variables referencian distintos objetos (distintas posiciones de memoria), aunque el contenido de los objetos sea idéntico, aunque los dos vehículos tengan exactamente los mismos valores en sus atributos. Veremos más adelante que usaremos `equals()`, `hashCode()` y `compareTo()` para las comparaciones.

Una última diferencia, que ya expusimos en el capítulo de cadenas, es que una variable de tipo primitivo no puede tener valor `null`, pero un objeto sí, ya que al almacenar una referencia a memoria y no un contenido, si que podemos decirle mediante `null` que no apunte a ningún contenido.

Podemos asignarle valor `null` mediante el operador de asignación `=` y también podemos preguntar si un objeto es igual a `null` mediante `==`.

Paso de objetos como parámetros a métodos

Decíamos en el capítulo anterior que los parámetros en Java se pasaban por valor, es decir, se pasaba una copia de la variable pasada. Con los objetos sigue siendo cierto, pero según lo que estamos hablando, pasamos la referencia a memoria del objeto, y esto es lo que no se puede modificar, pero si podemos modificar contenido apuntado por esa referencia. Por eso, a veces verás que se dice que **en Java los tipos primitivos se pasan por valor, pero los objetos “parece” que se pasan por referencia.**

En el siguiente ejemplo se crea un vehículo con un consumo de 7 litros, pero en el método se le aumenta a 8.

```
public class NewMain {
    public static void main(String[] args) {
        Vehiculo v1 = new Vehiculo ("4567ABC", 5, 40, 7);
        aumentarConsumo (v1);
        System.out.println(v1.consumo);
    }
    static void aumentarConsumo (Vehiculo x) { x.consumo++; }
}
```

En cambio, no funcionaría el aumento de consumo si lo hiciésemos pasando un tipo primitivo.

```
public class NewMain {
    public static void main(String[] args) {
        float consumo = 7;
        aumentarConsumo (consumo);
        System.out.println(consumo);
    }
    static void aumentarConsumo (float c) { c++; } //error
}
```

Atributos y métodos estáticos

Para terminar este primer contacto con los objetos, hay que destacar que existen propiedades y métodos que **no son propios de cada instancia, sino que son comunes a todos ellos**. Se les pone el prefijo **static**, se llaman *métodos de clase*, y se deben referenciar con el nombre de la clase y no del objeto. Siguiendo el ejemplo, podríamos añadirle al vehículo una propiedad *precioLitro* y sería de tipo estático.

```
public class Vehiculo {
    String matricula;
    int cantPasajeros;
    int tamDeposito;
    float consumo;
    static float precioLitro;
```

Y podríamos manejarlas sin instancias de clase, por ejemplo:

```
Vehiculo.precioLitro =1.37f;
```

Pero no podemos usar la palabra reservada *this* ya que no son atributos de ninguna instancia concreta.

Son de tipo estático ciertas variables que nos ofrece Java, como *Math.PI*, que se corresponde con el número PI, o métodos estáticos como *Math.sqrt (int n)*, ya que no tendría ningún sentido tener que crear una instancia de tipo *Math*.

Cuando busquemos funcionalidades en Java y encontraremos métodos estáticos que resuelvan **esas necesidades** no necesitaremos construir instancias de clases para emplearlas. Por ejemplo, en apartados anteriores, vimos métodos estáticos cuando trabajábamos con cadenas:

```
numeroEntero = Integer.parseInt (cadena);
if (Character.isDigit (ch1)==true) { . . . } (*)
```

(*) Si el método `isDigit()` no fuese static, habría que llamarlo de una forma similar a esta: `ch1 = new Character(); if (ch1.isDigit() == true) { ... };`
 Es decir, primero creando una instancia con un constructor de la clase y luego invocando al método sobre ella.

Ahora ya tenemos la explicación de por qué las **funciones y las variables globales de un programa son static**. No tiene sentido hablar de instancias del programa.

Modificadores de Acceso

Los *modificadores* de acceso nos introducen al concepto de *encapsulamiento*. El encapsulamiento busca de alguna forma controlar el *acceso a los datos* que conforman un objeto, de este modo podríamos decir que una clase (y sus objetos) que hacen uso de modificadores de acceso (especialmente privados) son objetos encapsulados.

Los modificadores de acceso permiten dar un nivel de seguridad mayor a nuestras aplicaciones restringiendo el acceso a diferentes atributos, métodos, constructores asegurándonos que el usuario deba seguir una "ruta" especificada por nosotros para acceder a la información.

Siempre se recomienda que los atributos de una clase sean privados, para que no se acceda a ellos directamente y se implementen métodos públicos `get` y `set` para para obtener y establecer respectivamente el valor del atributo.

Algo a tener en cuenta es **que siempre que se use una clase de otro paquete, esta se debe importar usando import** (esto ya lo vimos en el apartado de entrada/salida en la que importábamos la clase `Scanner`). Cuando dos clases se encuentran en el mismo paquete no es necesario hacer el `import` pero esto no significa que se pueda acceder a sus componentes directamente, dependerá de su modificador de acceso. **Import debe ir justo después de la sentencia package**, antes de la definición de clases.

private

El modificador *private* en Java es el más restrictivo de todos: cualquier elemento que sea privado puede ser accedido únicamente por los métodos y constructores de dicha clase, pero no por otras clases o programas del proyecto.

En general, se recomienda que los atributos de una clase sean privados, para que no se acceda a ellos directamente y se implementen métodos públicos `get` y `set` para para obtener y establecer respectivamente el valor del atributo. Se llaman *getters* y *setters*, y los veremos con más detalle más adelante.

Como ejemplo, vamos a poner la propiedad `cantPasajeros` como privada:

```
public class Vehiculo {
    String matricula;
    private int cantPasajeros;
    int tamDeposito;
    float consumo;
    static float precioLitro;
```

Creando los métodos para acceder a su valor y para fijarle un nuevo valor. Podemos establecer reglas a la hora de hacer el *set*, controlando mejor los valores que le permitimos tomar.

```
public void setCantPasajeros (int p) {
    if (p > 0 && p < 10) this.cantPasajeros = p;
}
public int getCantPasajeros () {
    return cantPasajeros;
}
```

El modificador por defecto (default)

Java nos da la opción de no usar un modificador de acceso y al no hacerlo, el elemento tendrá un acceso conocido como *default* o acceso por defecto que **permite que tanto la propia clase como las clases del mismo paquete accedan a dichos elementos** (de aquí la importancia de declararle siempre un paquete a nuestras clases).

Modificador protected

El modificador de acceso *protected* nos permite acceso a los elementos con dicho modificador desde **la misma clase, clases del mismo paquete y clases que hereden de ella (incluso en diferentes paquetes)**. Como el tema de herencia aún no lo hemos tratado, no nos centraremos en este modificador de acceso por ahora.

Modificador public

El modificador de acceso *public* es el más permisivo de todos, básicamente *public* si un componente de una clase es *public*, tendremos acceso a él desde cualquier clase o instancia sin importar el paquete o procedencia de ésta.

	clase	paquete	subclase	resto
private	sí	no	no	no
ninguno	sí	sí	no	no
protected	sí	sí	sí	no
public	sí	sí	sí	sí

Una clase ha de ser siempre *public* o *default*. En el primer caso deberá estar en un archivo *.java* con el mismo nombre. En el segundo caso no es necesario.

El tema de los paquetes se tratará en temas posteriores. Por ahora simplemente saber que un **paquete** es un contenedor de clases que permite agrupar las distintas clases que por lo general tiene una funcionalidad y elementos comunes, definiendo la ubicación de dichas clases en un directorio de estructura jerárquica.

Para especificar el paquete al que pertenece una clase, se pondrá la instrucción *package nombrePaquete;* y debe ser la primera línea del archivo, así pues, la estructura típica de un programa/clase Java podría ser:

```
package nombrePaquete;
import librerias;
public class nombreClase {}
```

Enumeraciones

Es un tipo de datos especial y sirve para definir un conjunto cerrado de valores constantes similares a las cadenas. Podremos crear variables que tengan como tipo la enumeración, asignarle valores, comparar, etc. Internamente se comportan como clases, y como veremos a continuación, la forma de definirlas es parecida, disponen de métodos, etc. pero siempre restringidas a ese conjunto de valores que le asignamos.

Un ejemplo podrían ser los días de la semana, o el estado civil de una persona:

```
public enum DiaSemana {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO};
public enum EstadoCivil {SOLTERO, CASADO, DIVORCIADO, VIUDO};
```

Los valores suelen escribirse en mayúsculas. Implícitamente son *public*, *final* y *static* y no pueden contener espacios en blanco.

```
DiaSemana dia1 = DiaSemana.LUNES;
if (dia1 == DiaSemana.DOMINGO) { . . . }
```

Pueden usarse como expresiones en los 'case' de un 'switch' pero no van precedidos del nombre de la enumeración. Ejemplo: case LUNES: y no case ~~DiaSemana.LUNES~~.

En cuanto a la ubicación, se pueden definir como una clase:

- Como públicas en un archivo *.java*. Desde Netbeans: New > File > Java > Java Enum.
- Con el modificador de acceso por defecto, en el mismo archivo que otra clase.
- Como atributos de otra clase / variables globales de un programa.

Métodos de las enumeraciones

Las enumeraciones tienen dos métodos predefinidos: *values()* y *valueOf()*. Sus formas generales se muestran aquí:

```
public static tipoEnum[ ] values( )
public static tipoEnum valueOf(String str)
```

El método *values()* devuelve un array con todos los elementos de la enumeración. El siguiente ejemplo mostraría todos los días de la semana:

```
for (DiaSemana ds : DiaSemana.values()) {
    System.out.println(ds);
}
```

El método *valueOf()* devuelve la constante enumerada correspondiente al valor de la cadena pasado como parámetro, si existe. Si no existe, provoca una excepción de tipo *IllegalArgumentException*.

```
System.out.println("Introduce dia de la semana:");
String dia = new Scanner (System.in).nextLine().toUpperCase();
try {
    if (DiasSemana.valueOf(dia) == DiasSemana.DOMINGO) {
        System.out.printf("es domingo");
    } else {
        System.out.printf("no es domingo");
    }
} catch (IllegalArgumentException ex) {
    System.out.println("valor incorrecto");
}
```

El orden es importante en las enumeraciones. El método `ordinal()` devuelve la posición del elemento dentro de la enumeración empezando en 0, así pue, es importante el orden el que se escriben los elementos de la enumeración. En el primer ejemplo, `DiaSemana.LUNES.ordinal()` devolvería 0.

También incorpora el método final int `compareTo (tipo-enum e)` en el que *tipo-enum* es el tipo de enumeración y *e* es el elemento que se compara con el elemento que invoca el método. Si la constante de invocación tiene un valor ordinal menor que 'e' devolverá un valor negativo, si los dos valores ordinales son iguales, se devuelve cero y si la constante de invocación tiene un valor ordinal mayor que 'e', se devuelve un valor positivo (*análogo al compareTo() de otras clases*).

Ejemplo: Se solicita al usuario que introduzca una cadena y, si el valor introducido coincide con un elemento de la enumeración, crea una variable con ese elemento.

```
System.out.println("Introduce dia de la semana");
String dia = new Scanner (System.in).nextLine().toUpperCase();
try {
    DiaSemana dia = DiaSemana.valueOf(dia);
    System.out.printf("Es el dia %d de la semana", dia.ordinal() + 1);
}
catch (IllegalArgumentException e) {
    System.out.println("Valor incorrecto");
}
```

Java dispone de diversas enumeraciones predefinidas como `DayOfWeek`.

Constructores, métodos, variables de instancia

En Java, se ha dotado a las enumeraciones de más potencia, pudiendo incluir atributos y métodos, aproximándolas a las clases con elementos estáticos.

A la hora de especificar cada elemento de la enumeración, podemos indicarle los parámetros que empleará el constructor. A continuación, definiremos opcionalmente constructores y / o métodos.

Ejemplo:

```
enum Transporte {
    COCHE (100,1000),
    CAMION (80,20000),
    AVION (700,500000),
    TREN (120, 40000),
    BARCO (60, 300000);
    public final int velocidad; //velocidad media
    public int peso; //peso medio (kg)
    Transporte (int s, int p){velocidad=s; peso=p;} //Constructor
    double horasEnRecorrer (int km) {return km/velocidad;} //método
}
```

Podremos llamar a ese constructor implícitamente:

```
Transporte tp = Transporte.AVION; //llama al constructor
```

Y emplear sus atributos y métodos públicos:

```
System.out.println("El peso medio de un avion es: " +
                    Transporte.AVION.peso);

System.out.println("Un avion recorre 10000 km en: " +
                    Transporte.AVION.horasEnRecorrer(10000));
```

En el siguiente ejemplo vemos como definir una enumeración con las iniciales de las provincias de Galicia, y como le añadimos un atributo adicional para poder obtener su nombre completo:

```
public enum Provincia {
    LU("Lugo"),
    CO("A Coruña"),
    PO("Pontevedra"),
    OU("Ourense");

    private String nom;

    Provincia(String nom) {
        this.nombre = nom;
    }
    public String getNombre() {
        return nombre;
    }
}
```

Y podríamos emplearla en nuestros programas de la siguiente forma:

```
Provincia miProvincia = Provincia.LU;
System.out.println (miProvincia.getNombre);
```

Clases Interesantes

Java nos ofrece una serie de clases para facilitarnos operaciones cotidianas como las siguientes:

Math : Operaciones matemáticas

La clase Math ofrece varias propiedades estáticas muy útiles, el número Pi: `Math.PI` y el número e: `Math.E` y también métodos estáticos:

- Valor absoluto: `Math.abs (número)` pudiendo ser el número int, long, float o double.
- Seno, coseno y tangente: `Math.sin (angulo)`, `Math.cos (angulo)` y `Math.tan (angulo)`
- Potencia y raíz cuadrada: `Math.pow (base, exponente)` y `Math.sqrt (numero)`
- Redondeo y truncado: `Math.round (num)` y `Math.floor (num)`
- Mayor y menor: `Math.min (x,y)` y `Math.max (x,y)`
- Números pseudoaleatorios: `Math.random ()` //double entre 0 y 0,9999999
`(int) (Math.random () * 10) + 1;` //enteros entre 1 y 10
 Se puede usar `Random` o `SecureRandom` para más posibilidades.

Round

El método `round` no redondea con decimales, según la cantidad de decimales debemos multiplicar por 10, 100, etc. y luego hacer la división decimal del resultado entre el mismo número.

Ejemplo, para dos decimales: `Math.round(num*100)/100d;`

Por otra parte, con datos de tipo `float` este redondeo funciona correctamente, pero con datos de tipo `double` a veces veremos resultados del tipo 12,99999999 o bien 12,000000001. Para solucionar estos problemas se podría trabajar con otras clases como `BigDecimal` aunque nosotros no lo veremos. Por ejemplo:

```
BigDecimal precioRound = new BigDecimal(precio).setScale(2, RoundingMode.HALF_UP);
```

También comentar que se pueden usar clases que no hacen redondeo, que no modifican el contenido de la variable, pero que muestran al usuario solo los decimales deseados. Ejemplo:

```
DecimalFormat df = new DecimalFormat ("##.##");
System.out.print (df.format (precio));
```

Random : Números aleatorios

Para generar números pseudoaleatorios, además del método estático `Math.random()`; disponemos de la clase `Random`, que es más eficiente y genera los números con mayor calidad, en cuanto a aleatorio se refiere, por lo que es más recomendable. Debemos primero llamar al constructor y luego invocar a sus métodos. Dispone, entre otros, de los siguientes:

- `nextInt (int limite)`: Genera números enteros entre 0 y límite (este último no incluido)
- `nextInt (int limiteInferior, int limiteSuperior)`: Genera números enteros entre el límite inferior y el límite superior (este último no incluido)
- `nextBoolean ()`: Devuelve true o false.
- `nextFloat ()`: Genera números decimales mayor o igual a 0 y menor que 1, igual que `Math.random()`;
- `nextFloat (float limite)`: Genera números con decimales entre 0 y límite (este último no incluido).
- `nextFloat (float limiteInferior, float limiteSuperior)`: Genera números con decimales entre el límite inferior y el límite superior (este último no incluido)

Ejemplo:

```
Random random = new Random();           //solo se hace una vez en el programa
for (i=1; i<= 10; i++) {
    int dado= random.nextInt (1,7);   //generará números entre 1 y 6
    System.out.println(dado);
}
```

LocalDate : Tratamiento de fechas

Hasta Java7 se utilizaban clases como `Date`, `Calendar` y `SimpleDateFormat`, pero no eran muy seguras en programación concurrente, aparte de que tenía características poco intuitivas (por ejemplo: los meses empezaban en 1 pero los días en 0). A partir de Java8 el nuevo paquete `java.time` pretende solucionar estas problemas.

Tenemos tres clases principales: `LocalDate` (fecha sin la hora), `LocalTime` (Hora sin fecha) y `LocalDateTime` (combinación de las dos anteriores: fecha+hora)

Las tres tienen distintos tipos de métodos estáticos que nos devuelven instancias de fechas por lo que podemos usarlos como constructores (*), a partir de la fecha del sistema (`now`), dándole parámetros para cada elemento de la fecha y/u hora (`of`), desde una cadena (`parse`), o a partir de otra fecha mediante un ajuste o desplazamiento temporal (`with`):

```
LocalDate fech1 = LocalDate.now();
LocalDate fech2 = LocalDate.of(2019,11,29);
LocalDateTime fecHora3= LocalDateTime.of(2109,Month.AUGUST,20,8,30, 59);
LocalDate fech4 = LocalDate.parse("1990-12-31");
LocalTime hora5 = LocalTime.parse("08:30");
LocalDateTime fechahora6 = LocalDateTime.parse("2018-10-10T11:25");
LocalTime hora7 = LocalTime.of (6,30,12);
LocalTime hora8 = LocalTime.parse("08:30");
LocalDate fech9 = fech1.with(temporalAdjuster);
```

En 'temporalAdjuster' podemos incluir métodos estáticos de la clase TemporalAdjusters como son: `firstDayOfMonth()`, `firstDayOfNextMonth()`, etc. logrando modificaciones muy interesantes sobre la fecha a la que le aplicamos el método. Ejemplo:

```
LocalDate fechaFinMes = fech1.with(TemporalAdjusters.lastDayOfMonth());
```

(*) Los métodos que crean instancias de objetos, pero no son constructores, se llaman métodos de factoría. Internamente, ellos llaman al constructor.

Métodos generales

- `getYear(); getMonth();` // Y otros similares...
- `getDayOfWeek();` // Devuelve enumeración: *Monday, Tuesday...*
- `getDayOfWeek().getValue();` // Devuelve 1 para lunes... 7 para domingo.
- `fecha1.toString();` // Obtiene una cadena a partir de la fecha
- `isLeapYear();` // devuelve true si es bisiesto
- `lengthOfMonth();`

Podemos combinar los métodos de factoría que crean una fecha y métodos que operan sobre la fecha. Ejemplos:

- 1) Obtener la fecha del 1 de enero del año actual.

```
LocalDate fecha = LocalDate.of(LocalDate.now().getYear(), 1, 1);
```

- 2) Consultar si el año actual es bisiesto:

```
if (LocalDate.now().isLeapYear()) { . . . }
```

Comparar fechas

Disponemos de métodos que nos permiten saber si la fecha es anterior, igual o posterior a una fecha pasada como parámetro.

- `isEqual (fecha);` // devuelve true/false
- `isBefore (fecha);` // devuelve true/false
- `isAfter (fecha);` // devuelve true/false

Ejemplo: Introducir por teclado dos fechas en formato AAAA-MM-DD y decir si la primera es anterior a la segunda:

```
System.out.println("Introduce fecha 1 (formato:AAAA-MM-DD): ");
String fec1 = teclado.nextLine();
System.out.println("Introduce fecha 2 (formato:AAAA-MM-DD): ");
String fec2 = teclado.nextLine();
LocalDate fechal = LocalDate.parse(fec1);
LocalDate fecha2 = LocalDate.parse(fec2);
if (fechal.isBefore(fecha2))
    System.out.println("La primera es anterior");
else if (fechal.isAfter(fecha2))
    System.out.println("La primera es posterior");
else System.out.println("Las fechas son iguales");
```

Incrementos/decrementos en fechas:

- `plus(cant, unidad);`
- `minus (cant, unidad);`

donde:

- `cant`: cantidad que queremos sumar/restar
- `unidad`: unidad en la que está esta expresada la cantidad anterior, usando la clase `ChronoUnit`. Ejemplos: `ChronoUnit.HOURS, MINUTES, DAYS, WEEKS, MONTHS, YEARS` etc. Esta clase necesita `import java.time.temporal.*`.

Ejemplos:

- 1) Introducir por teclado tres enteros que representen año, mes y día. Con ellos construir una fecha y mostrar la fecha resultante al sumarle 90 días.

```
System.out.println("Introduce día");
int dia = Integer.parseInt(teclado.nextLine());
System.out.println("Introduce mes");
int mes = Integer.parseInt(teclado.nextLine());
System.out.println("Introduce año");
int año = Integer.parseInt(teclado.nextLine());
LocalDate fecha = LocalDate.of(año,mes,dia);
LocalDate fechaMas90 = fecha.plus(90, ChronoUnit.DAYS);
System.out.printf("Fecha resultante: %s", fechaMas90);
```

También hay métodos que no necesitan este último parámetro porque va implícito en el propio nombre del método: `plusYears(cant)`, `plusMonths(cant)`, etc. Por ejemplo, la última línea de código anterior podría ser:

```
LocalDate fechaMas90 = fecha.plusDays(90);
```

- 2) Introducir por teclado una fecha en formato AAAA-MM-DD y mostrar la fecha de los 10 días siguientes:

```
System.out.println("Introduce fecha (formato:AAAA-MM-DD): ");
String fec1 = teclado.nextLine();
LocalDate fech1 = LocalDate.parse(fec1);
for (int i=1;i<=7;i++) {
    LocalDate siguiente=fech1.plusDays(i);
    System.out.println(siguiente); }
```

Diferencias entre fechas

Para obtener la resta o diferencia entre fechas podemos usar el método `until` que devuelve la diferencia entre dos fechas en la unidad que queramos. Ejemplo:

```
dias = fech1.until(fecha2,ChronoUnit.DAYS);
```

O también con `ChronoUnit.UNIDAD.between`, como se muestra a continuación:

```
long tiempo;
tiempo = ChronoUnit.SECONDS.between(fecHoral,LocalDateTime.now());
```

En este caso resta la segunda fecha menos la primera, es decir la primera debe ser menor para que el resultado devuelto no sea negativo.

Ejemplo: Calcular la edad en años de una persona de la que introducimos su fecha de nacimiento:

```
System.out.println("Introduce fecha nacimiento (formato:AAAA-MM-DD): ");
String fec1 = teclado.nextLine();
LocalDate fech1 = LocalDate.parse(fec1);
long años = ChronoUnit.YEARS.between(fech1,LocalDate.now());
System.out.printf("Edad: %d", años);
```

Formato de fechas

Disponemos de la clase `DateTimeFormatter` (`import java.time.format.*`) para dar formato a las fechas. Por ejemplo, para solicitar al usuario una entrada por teclado en un formato concreto:

```
DateTimeFormatter formato = DateTimeFormatter.ofPattern("dd/MM/yyyy");
System.out.println("Introduce fecha DD/MM/AAAA: ");
LocalDate fecha = LocalDate.parse(teclado.nextLine(),formato);
```

O para mostrar fechas en el formato deseado:

```
DateTimeFormatter formato = DateTimeFormatter.ofPattern("yyyy/MM/dd");
System.out.println(formato.format(fecha));
```

Pudiendo incluir el patrón: yyyy, yy, MM, dd, H, m, s, a (am pm) z (zona), DD (dia del año), etc. Así pues, para asignar fechas desde cadenas, por ejemplo, introducidas por el usuario, tenemos que usar esta clase para indicarle el formato en el que está, como se ha visto en el ejemplo anterior. De no indicar formato, debemos introducirlo en forma AAAA-MM-DD.

Representación en texto

Tenemos también enumeraciones que nos muestran en texto los nombres de los meses y días de la semana, en distintos formatos: nombre completo, solo en 3 letras y solo la inicial (respectivamente: TextStyle FULL, NARROW, SHORT) y en distintos idiomas (mediante Locale). Para ello debemos usar las clases **Month** y **DayOfWeek**.

Primero debemos instanciar estas clases, podemos hacerlo desde una fecha, o empleando el método of, al que le pasamos un entero entre 1 y 12 para los meses y un entero entre 1 y 7 para los días de la semana, o bien mediante una constante. Ejemplos:

```
Month mes1 = LocalDate.now().getMonth();
Month mes2 = Month.of(3);                                //marzo
Month mes3 = Month.AUGUST;
DayOfWeek dia1 = LocalDate.parse("2020-01-01").getDayOfWeek();
DayOfWeek dia2 = DayOfWeek.of(3);                         //miércoles
DayOfWeek dia3 = DayOfWeek.MONDAY;
```

Luego, para mostrar su valor, usaremos el método **getDisplayName**, al que le pasamos el formato deseado y el idioma. El formato puede tomar los valores: TextStyle.FULL, TextStyle.NARROW, TextStyle.SHORT.

El idioma puede ser una constante Locale.* por ejemplo: Locale.FRENCH para los valores que ya están definidos o bien crear una instancia de Locale para países/idiomas como español o gallego, mediante su constructor (hasta JDK18) o mediante el método **of**(a partir de JDK19, versión a partir de la cual, el constructor de Locale está *deprecated*):

```
Locale locale = new Locale("es", "ES");      //hasta jdk18
Locale locale = Locale.of("es", "ES");        //a partir de jdk19
```

Ejemplo: muestra el día de la semana en gallego que fue el 31 de diciembre del año 1999.

```
DayOfWeek dia = LocalDate.of(1999, 12, 31).getDayOfWeek();
Locale locale = new Locale("gl", "ES");           //Locale.of("gl", "ES");
System.out.println(dia.getDisplayName(TextStyle.FULL, locale));
```

Mostrando:

venres

Ejemplo: muestra en francés, los nombres de los 12 meses del año:

```
for (int i=1;i<=12;i++){
    Month mes= Month.of(i);
    System.out.println(mes.getDisplayName(TextStyle.FULL, Locale.FRENCH));
}
```

Mostrando:

janvier
février
...
...

Otras clases

Disponemos asimismo de otras clases como:

- Year, YearMonth, etc... con métodos como *lengthOfMonth()*, *isLeapYear()*, etc.
- ZonedDateTime para trabajar con zonas horarias.
- Period para trabajar con periodos entre fechas
- Duration para trabajar con la duración entre dos horas.

Error al crear una fecha

Si intentamos crear una fecha y los parámetros pasados no son correctos, por ejemplo 31 de febrero, el constructor producirá una excepción (esto es, un error en tiempo de ejecución). Podemos aprovechar esta situación para detectar si una fecha es válida. Como veremos en capítulos posteriores, mediante la instrucción try...catch podemos evitar que el programa se interrumpa de forma abrupta y notificar que la fecha no es válida.

La siguiente función se le pasaría una posible fecha en texto y como segundo parámetro el formato de la misma (por ejemplo "dd/MM/yyyy"). El método "intenta" crear un LocalDate a partir del parámetro pasado y su formato. Si el valor es coherente devolverá *true*. En caso contrario, si la fecha es inválida, se ejecutará la parte "catch" y por tanto devolverá *false*, pero no se interrumpirá la ejecución del programa.

```
static boolean validarFecha (String fec, String formato) {
    try {
        DateTimeFormatter f = DateTimeFormatter.ofPattern(formato);
        LocalDate fecha = LocalDate.parse(fec,f);
    } catch (Exception e) { return false; }
    return true;
}
```

Métodos Comunes a Todas las Clases

Cuando definimos una nueva clase, ya hemos visto que definimos sus atributos (que representan sus características y estado), sus constructores (que representa la creación de instancias de la clase con su situación inicial) y por último los métodos (que representan el comportamiento de la clase).

Existen una serie de métodos que suelen ser frecuentes y comunes a prácticamente todas las clases que definimos, algunos ya vienen predefinidos, pero podemos sobrescribirlos. Son los siguientes:

Getters y Setters

Los *getters* y *setters* son métodos de acceso público y se usan para dar acceso a los atributos que definimos de acceso *private*. Los *getters* devuelven el valor del atributo (en inglés, *get*) y los *setters* asignan un valor al atributo (en inglés, *set*). Ejemplo:

```
public class Empleado {
    private int id;
    private String nombre;
    private LocalDate fechaNacimiento;
    private double salario;
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

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

```

public LocalDate getFechaNacimiento() {
    return fechaNacimiento;
}

public void setFechaNacimiento(LocalDate fechaNacimiento) {
    this.fechaNacimiento = fechaNacimiento;
}

public double getSalario() {
    return salario;
}
public void setSalario(double salario) {
    this.salario = salario;
}
}

```

A simple vista, podríamos decir que no son necesarios. Si los atributos los declaramos públicos, podríamos acceder a ellos directamente. La misión de los getters/setters es ocultar la implementación de nuestra clase, no permitiendo el acceso directo a nuestros atributos, sino que somos nosotros los que gestionamos como se acceden y como se modifican. Podríamos incluso hacer validaciones, formateo, modificaciones, desencadenar otras acciones, etc.

Conseguimos de cara a los usuarios de nuestra clase:

- Mejorar el encapsulamiento
- Facilitar cambios de forma transparente.
- Mejorar la seguridad

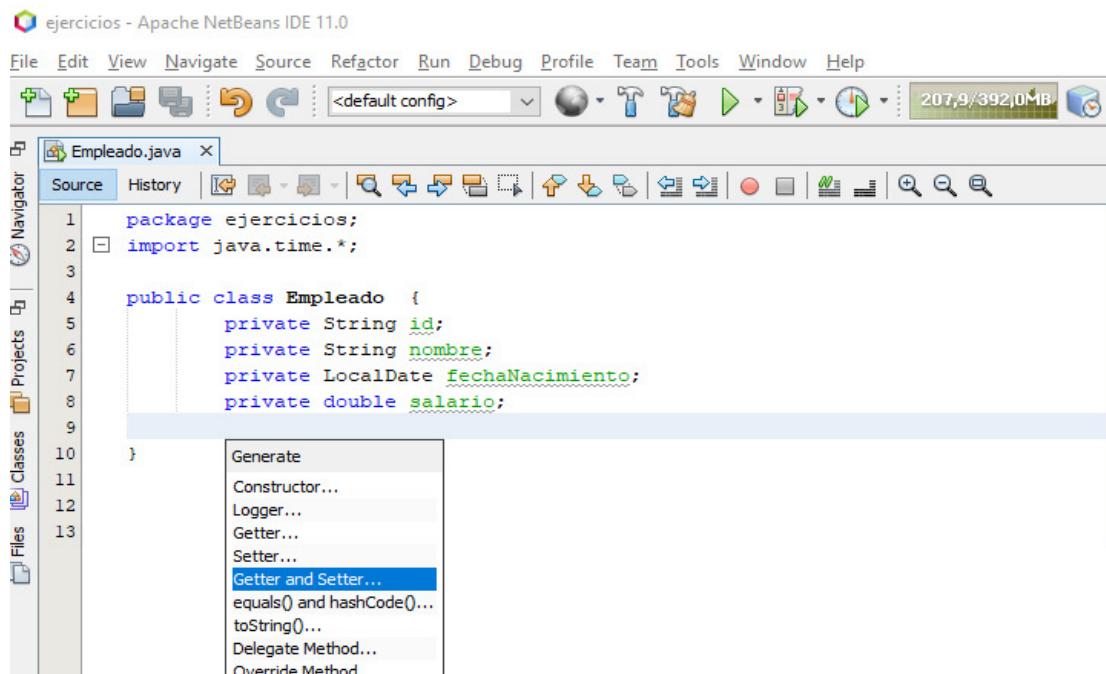
Si el atributo es boolean el *getter* suele empezar por "is" en vez de "get". Ejemplo.

```

private boolean jubilado;
public boolean isJubilado () {    return jubilado; }

```

Los IDE más populares suelen generar estos métodos de forma automática. Por ejemplo, *Netbeans* lo hace si, sobre la clase, pulsamos las teclas [Alt] + [Ins].



toString

El método `toString()`, como su propio nombre indica, se utiliza para convertir a `String` (es decir, a una cadena de texto) cualquier objeto. Este método está definido para la clase `Object`, y como todos los objetos heredan de dicha clase, todos ellos tienen acceso a este método.

Este método está redefinido para muchas clases (*se dice "sobreescrito"*) como por ejemplo `LocalDateTime` o `ArrayList` mostrando el contenido del objeto de una forma "comprendible" para el usuario. Así podemos hacer:

```
System.out.println(LocalDate.now().toString());
```

De hecho, `System.out.println` usa por defecto el método `toString()` para mostrar el contenido de su parámetro, siempre que esté definido, así pues, el ejemplo anterior podría escribirse así:

```
System.out.println(LocalDate.now());
```

Deberíamos, entonces, redefinir el método `toString()` para las clases que creamos, así facilitaremos su representación. Si no lo definimos, se mostraría una representación que incluye el nombre de su clase y una referencia de su ubicación en memoria.

```

6  public static void main(String[] args) {
7
8      Empleado e = new Empleado();
9      e.setId(10); e.setNombre("José"); e.setSalario(100000);
10     e.setFechaNacimiento(LocalDate.parse("2000-01-01"));
11     System.out.println(e);
12 }
13
14
Output - ejercicios (run)  x  run.xml  x
run:
ejercicios.Empleado@7e6cbb7a
BUILD SUCCESSFUL (total time: 0 seconds)

```

Si definimos el método `toString()`:

```
@Override
public String toString () {
    long edad = ChronoUnit.YEARS.between(fechaNacimiento, LocalDateTime.now());
    String txt = String.format("(Id:%d) %s (%d años)", this.id, this.nombre, edad);
    return txt;
}
```

La salida sería:

```

Output - ejercicios (run)  x  run.xml  x
run:
(Id:10) José (19 años)
BUILD SUCCESSFUL (total time: 0 seconds)

```

equals

Es un caso similar al `toString()` ya que está definido para la clase padre de todos los objetos `Object`, sobreescrito en otras clases como `String` y que debemos sobrescribir para nuestras clases.

Ya hemos utilizado el método `equals()` previamente, y sabemos que es la forma en que debemos comparar objetos (los objetos no se pueden comparar utilizando el operador `==`). El método `equals` está sobreescrito para la mayoría de las clases. Por ello, podemos usarlo directamente para comparar instancias de `String` o `LocalDate`, por ejemplo. Pero ¿qué ocurre en las clases que creamos nosotros?

Si probamos `empleado1.equals(empleado2)`, al no estar sobrescrito el método para la clase `Empleado`, el resultado será incorrecto. Por tanto, para comparar objetos de tipo `Empleado` debemos de sobrescribir el método en la clase:

```
@Override
public boolean equals (Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    //if (this.getClass() != obj.getClass()) return false; mejor:
    if (obj instanceof Empleado == false) return false;

    Empleado aux = (Empleado) obj;
    if (this.id==aux.id) return true;
    return false;
}
```

En este caso, si los *id* de empleado son iguales, diremos que son el mismo objeto. En otros casos habrá que comparar más atributos, o incluso todos ellos. Dependerá del contexto.

Comparación de variables

Cuando hablamos de tipos primitivos, la comparación de variables es muy sencilla, empleando operadores lógicos.

Como estamos comentando, en el caso de los objetos no es tan sencillo ya que el identificador contiene la dirección de memoria del objeto que referencia, así que, tanto para comparar como asignar estaríamos haciendo la operación con las direcciones de memoria y no con los contenidos que referencian. A modo de resumen, esta sería la forma de comparar los tipos de datos en Java:

- **Tipos primitivos:** operadores lógicos, por ejemplo: `==`.
- **Cadenas (`String`) y otras clases predefinidas,** como por ejemplo `LocalDate`: método `equals` (`==`compararía direcciones de memoria).
- **Clases definidas por nosotros:** Hay que definir qué entendemos por dos objetos iguales, depende del contexto, es decir, en qué consiste que dos instancias sean iguales ¿tienen que coincidir todos los atributos o solo alguno de ellos? De acuerdo con nuestros criterios, debemos sobrescribir el método `equals` heredado de la clase `Object`.

Si no sobrescribimos `equals()` solo devolverá `true` en caso de que las dos referencias comparadas comparten la misma ubicación de memoria, es decir apunten a la misma instancia de objeto.

hashCode

Este método se utiliza en vez de `equals` en determinadas ocasiones para determinar si un elemento de una colección de tipo `hash` (`HashSet`, `HashMap`) es igual a otra instancia, por lo que siempre que sobrescribamos `equals()` para una determinada clase, deberíamos sobrescribir también `hashCode()`. Los dos métodos tienen que ir a la par, de forma que si para dos instancias, `equals()` devuelve `true`, `hashCode()` debe devolver el mismo número, pero lo veremos en detalle en el capítulo dedicado a Colecciones.

clone

Con la asignación de objetos ocurre algo similar a la comparación. Igual que no podíamos usar == para compararlos, no podemos usar el operador de asignación = para copiar un objeto a otro. Para realizar esta operación debemos redefinir el método *clone* de la clase *Object* como se muestra a continuación.

```
class obj implements Cloneable {
    int num1;
    obj (int n) {num1 = n;} //constructor.

    @Override
    public Object clone() throws CloneNotSupportedException{
        Object clone = null;
        try { clone = super.clone(); }
        catch(CloneNotSupportedException e) { }
        return clone;
    }

    public class prueba {
        public static void main(String[] args)
            throws CloneNotSupportedException {
            obj o1 = new obj(3);
            obj o2 = (obj) o1.clone();

            System.out.println (o2.num1);
        }
    }
}
```

El código mostrado hace una copia “superficial”, es decir, copia los tipos primitivos, si el objeto tuviese otras clases entre sus atributos, habría que llamar al método *clone* de esas sub-clases, antes de hacer el *return*.

Las líneas *try...catch* sirven para capturar los posibles errores en tiempo de ejecución que se pudiesen producir (ver capítulo “Excepciones” más adelante en este manual).

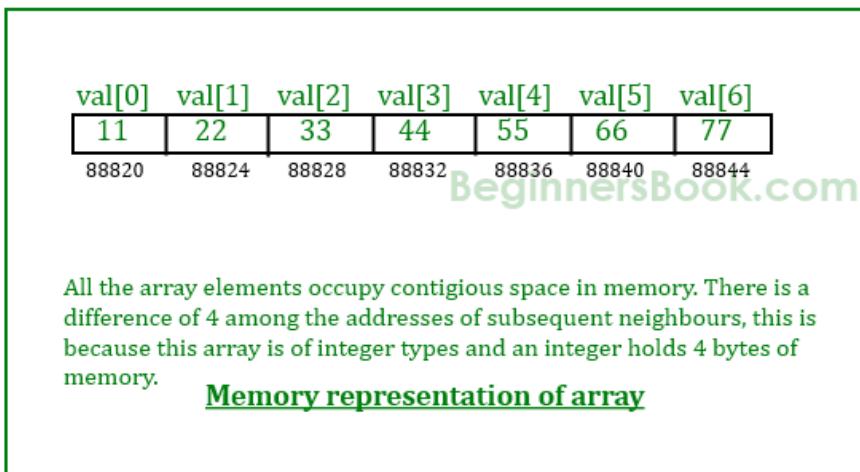
En el programa del ejemplo, el que hace la copia, hay que fijarse como la llamada a *clone()* lleva un casting al objeto destino ya que el tipo devuelto por el método *clone* es *Object*.

7. Array y ArrayList

Los arrays (o vectores, o matrices, o “arreglos”) se usan para almacenar varios valores en una sola variable, en vez de declarar varias variables con distintos nombres, una sola variable (con distintas posiciones o índices) almacenará todos esos valores. Piensa que tienes que almacenar las edades de todos los alumnos de una escuela, para calcular la edad media, o el mayor, o el menor...sería inmanejable.

```
media = (edad1 + edad2 + edad3 + . . . + edad300 ) / 300
```

En Java hay dos aproximaciones: el array tradicional, común a prácticamente todos los lenguajes de programación y la clase `ArrayList`, que implementa el array como un objeto con métodos interesantes.



Clase Array

Para trabajar con un array tenemos que hacer los pasos de declaración de la variable, la construcción (o reserva de memoria) y opcionalmente la inicialización de sus valores, estos tres pasos los podemos hacer todos juntos o por separado.

Para declarar un array seguimos el formato *tipo de dato [] nombreVariable*, es decir, como una variable cualquiera añadiendo los corchetes después del tipo.

```
int[] edad; //también está aceptado int edad[];  
String [] meses;
```

No es legal incluir el tamaño de un *array* en nuestra declaración, como en otros lenguajes, así pues esto: `int [5] edad;` no se puede hacer.

El siguiente paso es la “construcción”, la reserva de memoria en el *heap* (donde todos los objetos viven). **Para crear un objeto array, se debe saber cuantas posiciones va a tener**, cuánto espacio debe asignar, por lo que debemos especificar el tamaño del *array* en tiempo de creación. El tamaño de un *array* es el número de elementos que el *array* almacenará. Para ello usamos el operador `new`.

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

y también es válido, y muy frecuente, hacerlo los dos pasos anteriores en una sola instrucción:

```
int [] edad = new int[10];  
String [] meses = new String[12];
```

siendo `int [] edad = new int [];` un error, ya que no sabe cuanta memoria reservar.

El tercer paso, opcional, sería la inicialización, es decir, darles valores a esas posiciones del *array* que hemos reservado en una sola instrucción. Sería con la construcción:

```
int [] edad;
edad = new int[] {18, 20, 32};
```

Si hacemos los tres pasos en uno, admite también la sintaxis abreviada:

```
int [] edad = {18, 20, 32};
String [] coches = {"Volvo", "BMW", "Ford", "Mazda"};
```

pero serían un error las siguientes instrucciones:

```
int [3] edad1 = {18, 20, 32};
int [] edad2 = new int[3] {18, 20, 32};
int [] edad3; edad3 = {18, 20, 32};
```

Para acceder a los elementos de un *array*, pondremos el índice de la posición a la que queremos acceder, teniendo en cuenta que empieza en la posición cero y no uno. Así pues:

`System.out.println(coches[1]);` mostraría "BMW" por pantalla.

Los arrays tienen un comportamiento como objetos que son, y una propiedad interesante es **length** que nos indica la cantidad de elementos del array. Podríamos recorrerlos fácilmente un array entonces con un bucle:

```
for (int i = 0; i < coches.length; i++)
    System.out.println(coches[i]);
```

y también para inicializar cada una de sus posiciones:

```
for (int i = 0; i < coches.length; i++) {
    System.out.println("Introduce nombre coche");
    coches[i] = (new Scanner (System.in)).nextLine();
}
```

Ojo!! **length** es un atributo, no un método (sin paréntesis al final). La clase String dispone de un método llamado **length()** para obtener la cantidad de caracteres de la cadena, no confundir.

Si piensas en que hubiese 100 marcas de coches ya te das cuenta que los *arrays* son imprescindibles en cualquier lenguaje de programación para almacenar en memoria datos similares, poder recorrerlos y realizar operaciones sobre todos ellos. Otro ejemplo, sobre el *array* previo de edades, podría ser calcular la edad media.

```
float media = 0;
for (int i = 0; i < edad.length; i++) media += edad[i];
media = media / edad.length;
```

Al ser una operación tan habitual, el recorrido de un *array* se puede hacer con otra sintaxis más sencilla **"for-each"**, propia de Java, por ejemplo, para mostrar todas las marcas de coches del *array* definido previamente:

```
for (String c : coches)
    System.out.println(c);
```

El "for" se leería como: para cada cadena "c" del *array* coches haz...

Hay que tener presente que, recorriendo el array de esta forma, no se puede modificar el contenido del array, no funcionaría:

```
for (String c : coches) c = "Opel";
```

Habría que recorrerlo de la otra forma:

```
for (int i=0; i<coches.length; i++) coches[i] = "Opel";
```

Importante: La clase *Array* no permite cambiar el tamaño del *array* en tiempo de ejecución una vez definido. Si se necesita que ese tamaño pueda ir variando a lo largo de la ejecución del programa, debe dársele un tamaño suficientemente grande para todos los casos o bien emplear la clase *ArrayList*.

Sin embargo, en el momento de crear un objeto *Array* el tamaño no tiene que estar definido por un literal, puede ser el valor de una variable que toma valor en tiempo de ejecución. Ejemplo:

```
int [] miArray;
int tamañoArray;
tamañoArray = (new Scanner(System.in)).nextInt();
miArray = new int [tamañoArray];
```

Para mostrar el contenido de un *Array* no podemos hacerlo directamente mediante *System.out.println(coches)* ya que los arrays no tienen definido el método *toString*, y recordemos que *println* mostraba por pantalla el contenido devuelto por ese método.

Para solucionar esto, disponemos del método estático *toString* de la clase *Arrays*, al que le podemos pasar un array y obtiene una cadena con el contenido del array pasado, entre corchetes y separado por comas. Por ejemplo: *System.out.println(Arrays.toString(coches))*; mostraría:

[Volvo, BMW, Ford, Mazda]

Ejemplos de uso de Array

En este apartado vamos a ver ejemplos de distintas operaciones típicas sobre un array.

1.- Crear un programa que defina un array de enteros de 10 posiciones y que solicite al usuario que introduzca valores para todas sus posiciones:

```
Scanner teclado = new Scanner(System.in);
int [] lista = new int [10];
for (int i=0; i<lista.length;i++) {
    System.out.println("Introduce valor para posición: " + i);
    lista [i]=teclado.nextInt();
}
```

2.- Añadirle al programa anterior el código para que sume los elementos del array. Versión con contador que recorre el array:

```
int suma=0;
for (int i=0; i<lista.length;i++)
    suma+=lista[i];
System.out.printf("La suma es %d%n", suma);
```

3.- Añadirle al programa anterior el código para que sume los elementos del array. Versión con *for...each*:

```
int suma=0;
for (int elemento: lista)
    suma+=elemento;
System.out.printf("La suma es %d%n", suma);
```

4.- Añadirle al programa anterior él código para que calcule la media de los valores en las posiciones pares (*como for...each recorre todo el array, empleamos mejor el recorrido con contador*):

```
int suma=0, cont=0;
for (int i=0; i<lista.length; i+=2) {
    suma+=lista[i];
    cont++;
}
System.out.printf("La media es %.2f%n", (double) suma/cont);
```

5.- Partiendo de la clase simple (supongamos atributos públicos para que sea más sencillo):

```
Public class Persona {
    public String nombre;
    public int edad;

    public Persona(String dni, int edad) {
        this.nombre = dni;
        this.edad = edad;
    }
    @Override
    public String toString() {
        return "nombre=" + nombre + ", { edad=" + edad + ' } ';
    }
}
```

crea un array con 4 personas con unos valores cualesquiera y luego muestra todos los datos de la persona más joven:

```
Persona [] personas = new Persona [4];
personas[0]=new Persona ("Pepe",20);
personas[1]=new Persona ("Ana",10);
personas[2]=new Persona ("Luis",22);
personas[3]=new Persona ("Eva",30);
Persona joven = personas[0]; //sin constructor, apunta a un elemento del array
for (Persona persona : personas) {
    if (persona.edad < joven.edad) joven=persona;
}
System.out.println("La persona más joven es " + joven);
```

Búsqueda en Arrays

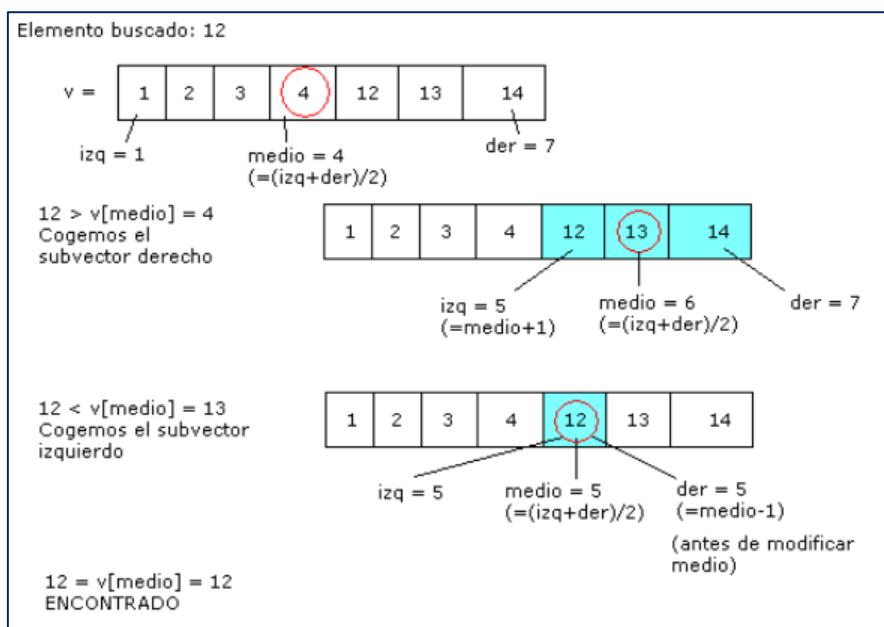
Existen dos formas de buscar elementos en un *Array*, la primera sería **secuencial**, y consistiría en ir recorrer el *Array* de principio a fin hasta encontrar el elemento buscado: es lenta pero no necesita que el *array* esté ordenado. A nivel de programación es muy sencilla, es un recorrido con un bucle bien hasta el final si no lo encuentra, bien hasta que lo encuentre. Una primera versión podría ser con un *for + break*:

```
int[] miArray = {10, 20, 12, 1, 2, 3 };
int num = (new Scanner(System.in)).nextInt();
boolean encontrado = false;
for (int i=0; i< miArray.length; i++){
    if (num == miArray[i]) {
        encontrado = true;
        break;
    }
}
System.out.println(encontrado?"encontrado":"no encontrado");
```

y otra más elegante con un *while*:

```
int[] miArray = {10, 20, 12, 1, 2, 3};
int num = (new Scanner(System.in)).nextInt();
boolean encontrado = false, fin = false;
int i=0; //contador que recorre miArray
while (!encontrado && !fin) {
    if (num == miArray[i]) encontrado = true;
    else if (i==miArray.length-1) fin = true; else i++;
}
System.out.println(encontrado? "encontrado": "no encontrado");
```

El segundo tipo de búsqueda sería **dicotómica**, necesita que el *array* está ordenado, pero es mucho más eficiente. Comienza buscando el elemento en la mitad del *array*, y si el elemento buscado es menor, pasa a buscar ahora en la mitad inferior del *array*, y si es mayor busca en la mitad superior, repitiendo el proceso sucesivas veces. Cada vez tendremos intervalos de búsqueda más pequeños hasta que no se pueda dividir más o lo encuentre. La siguiente figura explica el proceso:



El programa sería algo así:

```
import java.util.Scanner;

public class Ejemplo {
    public static void main (String args[]) {
        int[] miArray = {1, 20, 32, 41, 52, 93};
        int num = (new Scanner(System.in)).nextInt();
        boolean encontrado = false, fin = false;

        int centro, inf = 0, sup = miArray.length-1;
        while (!encontrado && !fin) {
            centro=(int)((sup+inf)/2);
            if (num == miArray[centro]) encontrado = true;
            else {
                if (num < miArray[centro])
                    sup=centro-1;
                else inf=centro+1;
                if (inf > sup) fin = true;
            }
        }
        System.out.println(encontrado? "encontrado": "no encontrado");
    }
}
```

Ordenación de Arrays

La clase *ArrayList* que veremos a continuación dispone de un método que ordena los elementos del *array* por lo que el apartado que vamos a ver ahora no sería técnicamente necesario, simplemente pasaríamos a trabajar con la clase *ArrayList* en vez de la clase *Array*, sin embargo, los algoritmos de ordenación son un elemento básico en el conocimiento y maestría de la programación por lo que vamos a estudiarlo.

Existen diferentes métodos de ordenación, algunos más rápidos, otros más intuitivos, etc. Puedes consultarlos aquí: https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento

A modo de ejemplo solo vamos a ver uno de ellos, el algoritmo de selección, ya que es muy intuitivo y sencillo de programar. Su funcionamiento es el siguiente:

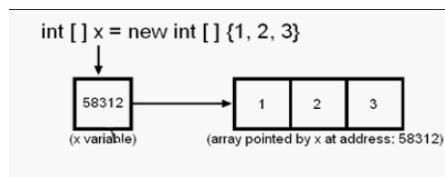
- Buscar el mínimo elemento de la lista e intercambiarlo con el primero
- Buscar el siguiente mínimo en el resto de la lista e intercambiarlo con el segundo
- Repetir el proceso hasta el penúltimo elemento de la lista.

El programa tiene dos bucles, el primero para recorrer el *Arrayy* el segundo para buscar un elemento menor que el que estamos tratando en el primer bucle. Cuando acaba la búsqueda (desde el elemento siguiente al actual hasta el final) hace el intercambio entre el actual del primer bucle con el mínimo encontrado.

```
public class Ejemplo {
    public static void main (String args[]){
        int[] miArray = {61, 120, 32, 41, 52, 93};      int posMin;
        for(int i=0; i< miArray.length-1; i++){
            //busqueda del menor
            posMin = i;
            for(int j=i+1 ; j< miArray.length; j++){
                if(miArray[j] < miArray[posMin]) {posMin=j;}
            }
            //intercambio del actual i con el menor
            int aux = miArray[i];
            miArray[i]=miArray[posMin];
            miArray[posMin]= aux;
        }
        //Mostrar resultado
        for(int i = 0 ; i< miArray.length; i++)
            System.out.println(miArray[i]);
    }
}
```

Copiar y Comparar Arrays

Al trabajar con *arrays* de tipos primitivos o de objetos se nos puede plantear la necesidad de copiar *arrays* o compararlos. La copia de *arrays* está permitida pero **no** como podemos hacer con un tipo primitivo ya que la variable que lo nombra en realidad contiene la dirección de memoria de comienzo del array, no sus elementos tal cual.



```
int[] edad = {18, 23, 13, 18, 14, 7};
int[] copia = new int[] {1,2,3,4,5,6};
copia = edad; //ahora copia es un "alias"
```

En el ejemplo anterior, efectivamente *copia* ahora referencia a *edad*, sería como un alias de *edad* (se habrán perdido los valores 1,2,3,4,5,6) y además, cualquier cambio en un elemento de *edad*, sería también un cambio en *copia*, y viceversa. Prueba a hacer `copia[0]=99;` y luego muestra `edad[0]`.

Para copiarlos deberíamos recorrer el primero y asignarlos uno a uno, o de una forma más rápida, mediante el método `System.arraycopy`, que tiene la siguiente estructura:

System.arraycopy (array origen, posición inicial, array destino, posición inicial destino, cantidad de elementos a copiar). En nuestro ejemplo:

```
System.arraycopy(edad, 0, copia, 0, edad.length);
```

Lo mismo nos ocurre cuando queremos comparar *arrays*, si hacemos algo del tipo `if (edad == copia)` estaremos comparando las direcciones de memoria del comienzo de ambos arrays, no compararemos cada uno de sus elementos. Para ello tenemos el método: `Arrays.equals(nombreArray1, nombreArray2)`, que devuelve *true* si son iguales y *false* en caso contrario.

```
if (Arrays.equals(edad, copia) == true) . . .
```

Arrays Bidimensionales

Los arrays pueden ser definidos de varias dimensiones (por ejemplo, de dos dimensiones, tendrían forma de tabla, como una hoja de cálculo, con filas y columnas).

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

```
//array para 30 alumnos y 3 evaluac.  
int [][] notas = new int[30][3];
```

Para acceder a estos elementos, sería algo como:

```
System.out.println(notas[15][1]);
```

Y para recorrerlo haría falta un doble bucle, uno primero para recorrer las filas y dentro de cada fila otro para recorrer las columnas de cada fila.

El siguiente ejemplo define un array de 4 filas y 3 columnas y lo recorre:

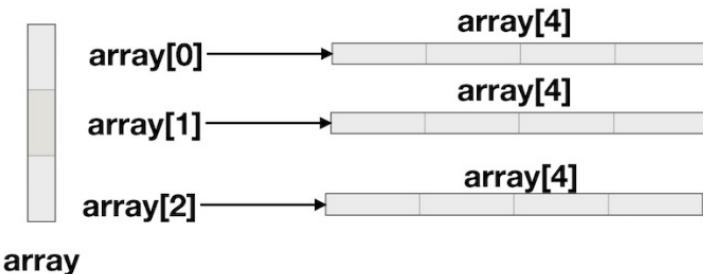
```
int[][] notas = { {10, 9, 10}, {0, 0, 1}, {6, 7, 6}, {5, 5, 4} };  
for (int fila = 0; fila < 4; fila++) {  
    for(int col = 0; col < 3; col++) {  
        System.out.println(notas[fila][col]);  
    }  
}
```

Un array bidimensional es en realidad un array unidimensional, en el que cada elemento es una fila completa y por tanto otro array. Así pues, en este ejemplo el límite del primer bucle que hemos puesto 4, podríamos haber puesto: `notas.length`, esto es, la cantidad de elementos de ese array de filas, por lo tanto, la cantidad de filas.

Y ya que cada fila es considerada un array unidimensional, `notas[0]` sería el array de la primera fila, `notas[1]` sería el array de la segunda fila, etc. y por ello podemos usar en el bucle interior la condición `notas[i].length` en vez de 3. Piénsalo con calma...

int[][] array = new int[3][4]

However, in Java, there is no concept of a two-dimensional array. A two-dimensional **array in java** is just an array of array. So below image correctly defines two-dimensional array structure in java.



También podríamos recorrerlo usando la forma “for-each” pero quizás sea más confusa:

```
for(int[] arr: notas) {
    for(int val: arr)
        System.out.println(val);
}
```

Ejemplos de uso de Arrays bidimensionales:

- Definir un array bidimensional de números con decimales, de 3 filas y 6 columnas:

```
float [][] arr = new float [3][6];
```

- Rellenar el array anterior con valores aleatorios entre 0 y 20:

```
Random random = new Random ();
for (int fila=0;fila<3;fila++) {
    for (int col=0;col<6;col++) {
        arr[fila][col]= random.nextFloat(21);
    }
}
```

- Mostrar el array anterior con todos los elementos de una fila en la misma línea:

```
for (int fila=0;fila<3;fila++) {
    for (int col=0;col<6;col++) {
        System.out.printf("%05.2f ",arr[fila][col]);
    }
    System.out.println(""); //al final de cada fila, salto de linea
}
```

- Calcular el valor máximo del array:

```
float maximo = -1f;
for (int fila=0;fila<3;fila++) {
    for (int col=0;col<6;col++) {
        if (arr[fila][col] > maximo ) maximo= arr[fila][col];
    }
}
System.out.printf("Máximo: %05.2f%n", maximo);
// se podría mejorar, en vez de poner ese valor inicial -1,
// poner la primera posición: arr[0][0]
```

- Calcular el máximo de cada fila:

```

for (int fila=0;fila<3;fila++) {
    float maxfila = -1f; //esto se ejecuta al empezar cada fila
    for (int col=0;col<6;col++) {
        if (arr[fila][col] > maxfila ) maxfila= arr[fila][col];
    }
    System.out.printf("Máx fila %d: %05.2f%n",fila, maxfila);
    //esto se ejecuta al acabar cada fila
}

// se podría mejorar, en vez de poner ese valor inicial -1 para cada
// fila, poner la primera posición de cada fila: arr[fila][0] y empezar el
// recorrido en la columna 1, en vez de la 0.

```

Argumentos en la línea de comandos

En las aplicaciones, tanto C++ como Java, al ejecutarlas, se pueden especificar argumentos en la línea de llamada, y si los programas están preparados para aceptarlos, podrán tomarlos en cuenta, tal y como pueden ser los parámetros de cualquier comando Linux. Por ejemplo:

`cp -i fich1 fich2` tiene 3 parámetros: -i, fich1 y fich2

En los programas Java, en el método **main()** que es el que se ejecuta, tenemos la siguiente firma:

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

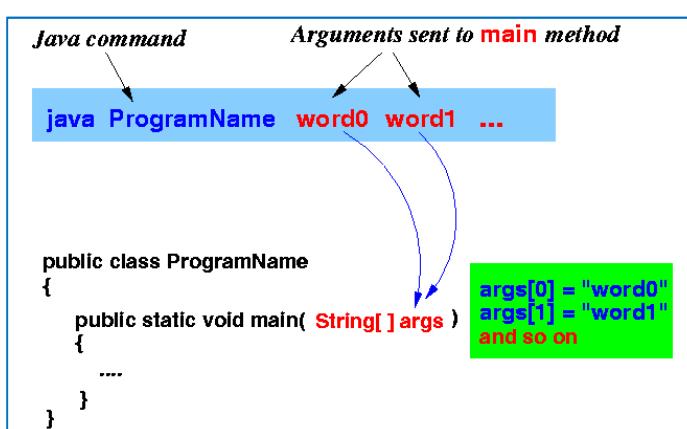
Pues los parámetros al programa serían `args[0]`, `args[1]`, ... hasta `args[args.length-1]`

En el programa podemos usar su valor directamente, como tipo String, o bien convertirlo al formato que queramos:

```

public static void main(String[] args) {
    if (args.length == 1) {
        int p1 = Integer.parseInt(args[0]);
        System.out.println("Cuadrado: " + p1 * p1);
    }
    else System.out.println ("Número de parámetros incorrecto");
}

```



Para ejecutar desde Netbeans, clicamos botón derecho sobre el proyecto y, en *Propiedades del Proyecto*, en la sección *Ejecutar*: comprobamos que la entrada 'clase main' contiene el nombre del paquete+programa que queremos ejecutar, y en 'argumentos' ponemos separados por espacios en blanco los argumentos que necesita el programa. Para ejecutar pulsamos *F6*, o bien *Ejecutar proyecto*, pero no *May+F6 (Ejecutar archivo actual)* como sí podemos hacer en otros casos.

Array en métodos (parámetro y return)

Al igual que las variables, también podemos pasar arrays a los métodos. Por ejemplo, en el programa siguiente se pasa un array al método *suma* para calcular la suma de los valores del array

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

public static void suma(int[] a) {
    int sum = 0;
    for (int i = 0; i < a.length; i++) sum+=a[i];
    System.out.println("Suma de valores del array: " + sum);
}
```

De la misma forma, un método también puede devolver un array. Por ejemplo:

```
public static void main(String args[]) {
    int [] arr = funcion(5);
    for (int i = 0; i < arr.length; i++)
        System.out.print(arr[i]+" ");
}

public static int[] funcion(int longitud){
    Random random = new Random();
    int [] a = new int [longitud];
    for (int i=0;i<longitud;i++)
        a[i]= random.nextInt(10);
    return a;
}
```

Clase ArrayList

La clase *ArrayList* en Java, es una clase que permite almacenar datos en memoria de forma similar a los Arrays, con la ventaja de que el número de elementos que almacena, lo hace de forma dinámica, es decir, que no es necesario declarar su tamaño como pasa con los Arrays. Además, ofrece muchos métodos que hacen el trabajo mucho más sencillo que con Arrays.

La forma de definir un *ArrayList* sería: `ArrayList <Object> miarray`

Y el constructor: `new ArrayList <> ()`;

Pudiendo hacerse en una sola instrucción:

```
ArrayList <Object> miarray = new ArrayList <> ();
```

Debiendo ser *Object* una clase propia de Java o una clase definida por nosotros mismos. Esta clase está incluida en la librería *util* por lo que debemos hacer un `import java.util.ArrayList;` en nuestro proyecto. En el constructor, se recomienda la sintaxis propuesta: `new ArrayList <> ()` aunque puedas ver también: `new ArrayList <Object>()` o bien `new ArrayList()`

Clases Wrapper o Envoltorio

Importante: No se pueden usar en ArrayList los tipos primitivos (int, char, etc.) pero esto no es inconveniente, ya que Java tiene definidos unas clases envoltorio (Wrapper Classes) que se corresponden con dichos tipos, pero con forma de objetos. Así, además de poder usarlos en ArrayList, nos ofrecen métodos interesantes.

Ya habíamos vimos estas clases Wrapper para la conversión de tipos de datos, con sus métodos estáticos `parseXXX`, cuando hacíamos, por ejemplo, `int i = Integer.parseInt (txt);`

En la parte final del curso se verán más en detalle, pero por ahora no van a suponer prácticamente ninguna diferencia a la hora de trabajar con los ArrayList, los trataremos como tipos primitivos a la hora de operar con ellos.

Esta es la lista de Wrapper Classes:

Primitive	Wrapper Class	Constructor Argument
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String
double	Double	double or String
long	Long	long or String
short	Short	short or String

Así pues, un ArrayList de int sería así:

```
ArrayList <Integer> miLista = new ArrayList <> ();
```

En muchas ocasiones lo veremos así:

```
List <Integer> miLista = new ArrayList <> ();
```

En el tema 15 explicaremos el por qué.

Para lo que dominéis la materia, diremos que List es una interfaz y ArrayList es una de las clases que lo implementa, y gracias al polimorfismo, se pueden crear variables de un tipo interfaz e instanciarlas con un constructor de una clase que implemente dicha interfaz.

Métodos de ArrayList

Los ArrayList tienen métodos muy útiles que facilitan el trabajo respecto a los clásicos Arrays.

size()	Devuelve el número de elementos (int)
add(X)	Añade el objeto X al final. Devuelve true.
add(posición, X)	Inserta el objeto X en la posición indicada.
get(posición)	Devuelve el elemento que está en la posición indicada.
remove(posición)	Elimina el elemento que se encuentra en la posición indicada. Devuelve el elemento eliminado.
remove(X) (*) remove (posición)	Si encuentra el objeto X, elimina su primera ocurrencia devolviendo <i>true</i> (si no lo encuentra, <i>false</i>). También puede borrar por posición. Al borrar no deja "hueco" de forma que todas las posiciones superiores a la borrada se desplazan a la izquierda (<i>para recorrer un ArrayList y eliminar algunas de sus posiciones será aconsejable usar un Iterator como veremos más adelante ya que al borrar cambia el tamaño del ArrayList</i>) for (int i=0; i< arrData.size(); i++) -if(arrData.get(i)==999) {arrData.remove(i);}
clear()	Elimina todos los elementos.
set(posición, X)	Sustituye el elemento que se encuentra en la posición indicada por el objeto X. Devuelve el elemento sustituido.
contains(X) (*)	Comprueba si la colección contiene al objeto X. Devuelve true o false.
indexOf(X) (*)	Devuelve la primera posición del objeto X. Si no existe devuelve -1
lastIndexOf(X) (*)	Devuelve la última posición del objeto X. Si no existe devuelve -1
equals (list)	Compara dos ArrayList completos. Devuelve <i>true</i> o <i>false</i>
binarySearch(arr,X)	En realidad, es un método estático de la clase <i>Collections</i> . Se invoca así: <i>Collections.binarySearch(arraylist1, X)</i> Busca X en <i>arraylist1</i> y devuelve posición. Si no encuentra, devuelve < 0. El ArrayList debe estar ordenado.
Arrays.asList (array)	Es estático. Hay que definir la variable como List, no ArrayList: <pre>List <Integer> miLista1 = new ArrayList<>(Arrays.asList (miarray)); List <Integer> miLista2 = new ArrayList<>(Arrays.asList (1,2,3));</pre> El inverso: <i>miarray= miLista1.toArray(new Integer[milista.size()]);</i>

(*) Para que funcionen tanto *remove (objeto)* como *contains(objeto)* tiene que estar definido el método *equals()* para la clase, ya que es lo que usa para comparar. String, LocalDate lo tienen definido, pero para clases definidas por nosotros mismos, habrá que definirlo.

Puedes consultar todos los métodos en:

<https://docs.oracle.com/javase/9/docs/api/java/util/ArrayList.html#method.summary>

equals y ArrayList

Como acabamos de ver la clase `ArrayList` dispone del método `indexOf(elemento_a_buscar)` que devuelve la posición de primera vez que se encuentra el elemento a buscar o -1 si no lo encuentra.

En caso de que el `ArrayList` sea de tipos primitivos, o clases que tienen implementado el método `equals()` funciona perfectamente:

```
ArrayList <String> arr = new ArrayList<>();
arr.add("aa"); arr.add("cc"); arr.add("bb"); arr.add("gg");
int pos = arr.indexOf("bb");
if (pos != -1) System.out.print("Encontrado!");
```

Pero cuando buscamos en una clase definida por nosotros esto no es así, por lo que si no sobrescribimos `equals()` los métodos `indexOf`, `contains` o `remove` no funcionarán.

En el siguiente ejemplo, definimos lo que es la igualdad en la clase `Alumno` que, en este caso, se corresponde con la igualdad en el nombre y en la edad.

```
public class Alumno {
    String nombre;
    int edad;
    public Alumno(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    @Override
    public int hashCode() {
        int hash = 7;
        return hash;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (obj instanceof Alumno == false) return false;
        Alumno other = (Alumno) obj;
        if (this.edad == other.edad && this.nombre.equals(other.nombre)) return true;
        return false;
    }
}
```

Ahora, ya podríamos hacer una búsqueda en un `ArrayList`:

```
ArrayList <Alumno> instituto = new ArrayList<>();
instituto.add (new Alumno("Pepe", 20));
instituto.add (new Alumno("Juan", 22));
instituto.add (new Alumno("Ana", 21));
instituto.add (new Alumno("Eva", 23));
Alumno aBuscar = new Alumno ("Ana", 21);
int pos = instituto.indexOf (aBuscar);
if (pos != -1) System.out.println ("Encontrado!");
```

También funcionaría en una `ArrayList` de dos dimensiones, pero recordemos que en realidad un `ArrayList` de 2 dimensiones es un `ArrayList` de `ArrayList`, por lo que la búsqueda hay que hacerla para cada `ArrayList` (lo que sería para cada "fila" del array de dos dimensiones). Sería algo así:

```

ArrayList<ArrayList<Alumno>> instituto2d = new ArrayList <>();
instituto2d.add(new ArrayList<>());
instituto2d.add(new ArrayList<>());

instituto2d.get(0).add(new Alumno("pepe",11));
instituto2d.get(0).add(new Alumno("juan",12));
instituto2d.get(0).add(new Alumno("luis",13));

instituto2d.get(1).add(new Alumno("ana",21));
instituto2d.get(1).add(new Alumno("eva",22));
instituto2d.get(1).add(new Alumno("maria",23));

Alumno b = new Alumno ("ana",21);
for (int i=0;i<instituto2d.size();i++) {
    pos = instituto2d.get(i).indexOf(b);
    if (pos!=-1)
        System.out.println("Encontrado ("+i+","+pos+")");
}

```

Operaciones sobre un ArrayList

Añadir elementos a un ArrayList

- Se puede hacer mediante el método **add()**. En el siguiente ejemplo, en un bucle for leemos 10 cadenas y las añadimos a un arrayList. Podría ser en un while, do...while, etc.

```

ArrayList <String> miLista = new ArrayList <> ();
Scanner teclado = new Scanner(System.in);
for (int i=1; i<=10; i++){
    System.out.println ("Introduzca una cadena a insertar");
    String cadena = teclado.nextLine();
    miLista.add(cadena);
}

```

Recorrer el ArrayList

- Usaremos el método **get()**. Podemos recorrerlo de forma clásica con un bucle **for**. Sobre el arraylist anterior:

```

for(int i=0;i< miLista.size();i++)
    System.out.println(miLista.get(i));

```

- O también, si lo vamos a recorrer completo en orden ascendente con un bucle **for each** y ya no es necesario get().

```

for( String cadena: miLista)
    System.out.println(cadena);

```

- Utilizando un objeto **Iterator** (esta técnica se verá en detalle, más adelante en el curso, en el tema 15). Iterator tiene como métodos:

- **hasNext()**: devuelve true si hay más elementos en el array.
- **next()**: devuelve el siguiente objeto contenido en el array.

Ejemplo:

```

ArrayList<Integer> numeros = new ArrayList <>();
//una vez insertados elementos. . .
Iterator <Integer> iterator = numeros.iterator();
while(iterator.hasNext())
    if (condición) iterator.remove();           //se crea el iterador
                                                //mientras haya elementos
                                                //se obtienen y se borran

```

Lo habitual será usar **foreach**, para recorridos completos, **for (int i...)** para recorridos específicos (solo algunas posiciones, orden descendente, etc.) y los **iteradores** para borrados, ya que no se pueden usar los dos métodos anteriores.

Insertar y modificar elementos de un ArrayList

Para insertar un elemento en un ArrayList disponemos de los métodos **add (objeto)** que añade el objeto al final y devuelve **true** y **add (posición, objeto)** que añade el objeto en la posición indicada.

Para modificar un elemento utilizamos el método **set (posición, Objeto)**.

La diferencia fundamental entre **set** y **add** (cuando le especificamos la posición) es que **set** sustituye el valor de esa posición perdiendo el valor que hubiese previamente, y **add** desplaza a la derecha todos los elementos desde esa posición, de forma que no se pierde el valor que hubiese en la posición indicada.

Es importante recordar que **como lo que se inserta o añade es un objeto, primero hay que llamar al constructor de ese objeto**, por ejemplo, sería incorrecto:

```
Persona p;
ArrayList<Persona> lista = new ArrayList <>();
lista.add (p);
```

Ya que no hemos instanciado 'p'. Deberíamos haber hecho:

```
Persona p = new Persona (); //depende de los constructores de Persona
ArrayList<Persona> lista = new ArrayList <>();
lista.add (p);
```

O bien:

```
ArrayList<Persona> lista = new ArrayList <>();
lista.add (new Persona() ); //depende de los constructores de Persona
```

Ordenar un ArrayList

ArrayList de un objeto simple creado por Java como pueda ser String, LocalDate, etc. se puede ordenar alfabéticamente de forma sencilla mediante la clase Collection (debemos hacer un: **import java.util.Collections;**):

```
ArrayList <Miclase> arraylist1= new ArrayList<>();
...
Collections.sort(arraylist1);
```

O bien, en orden inverso:

```
Collections.sort(arraylist1, Collections.reverseOrder());
```

Si es una clase con varios atributos, y queremos ordenar por uno concreto debemos añadirle un segundo parámetro al método **sort**, como explicaremos más adelante, en el capítulo 15, cuando hablemos de *Comparable* y *Comparator*. Aquí lo dejamos anotado simplemente por tener la referencia.

```
ArrayList <Miclase> arraylist1= new ArrayList<>();
...
Collections.sort(arraylist1, new Comparator<Miclase>() { (*)  
    @Override  
    public int compare(Miclase x1, Miclase x2) {  
        return x1.AtributoOrdenacion.compareTo(x2.AtributoOrdenacion);  
    }  
});
```

Ese segundo parámetro de sort es una clase que llamaremos *clase anónima*. También podremos emplear una función Lambda o una referencia a métodos. Son todo conceptos avanzados que veremos más adelante.

Copiar un ArrayList

El nombre de un ArrayList contiene la referencia al ArrayList, es decir, la dirección de memoria donde comienza el ArrayList, igual que sucede con los arrays estáticos.

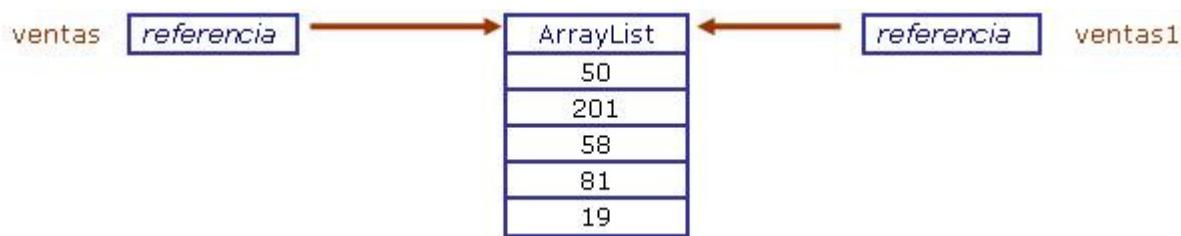
Si disponemos de un ArrayList de enteros llamado ventas:



La instrucción:

```
ArrayList<Integer> ventas1 = ventas;
```

No copia el array ventas en el nuevo array *ventas1* sino que **crea un alias**:



De esta forma tenemos dos formas de acceder al mismo ArrayList: mediante la referencia ventas y mediante la referencia ventas1.

Para hacer una copia podemos hacerlo de forma manual elemento a elemento o se puede pasar la referencia del ArrayList original al constructor del nuevo:

```
ArrayList<Integer> ventas1 = new ArrayList <>(ventas);
```



ArrayList en métodos (parámetro y return)

Un ArrayList puede ser usado como parámetro de un método o función, por ejemplo:

```
void metodo (ArrayList <String> miArray) { . . . }
```

Además, un método puede devolver un ArrayList mediante la sentencia return.

```
ArrayList <String> metodo () {
    ArrayList <String> resultado = new ArrayList<>();
    . .
    return resultado;
}
```

En el siguiente ejemplo, combina las dos funcionalidades, se le pasa como parámetro un ArrayList y devuelve un ArrayList. Lo que hace el programa es invertir el contenido de un ArrayList.

```
public static void main(String[] args) {
    ArrayList <Integer> num = new ArrayList <>();
    num.add(1); num.add(2); num.add(3);
    num = invertir(num);
}

public static ArrayList <Integer> invertir(ArrayList <Integer> n) {
    ArrayList <Integer> resul = new ArrayList<>();
    for (int i=n.size()-1; i>=0; i--)
        resul.add(n.get(i));
    return resul;
}
```

Prueba a ejecutarlo añadiendo en el `main` `System.out.println(num);` antes y después de ejecutar el método *invertir*.

ArrayList bidimensionales

Un ArrayList es un array unidimensional, pero con ellos podemos *simular* arrays de dos o más dimensiones anidando ArrayLists.

Para crear una matriz lo que creamos es un ArrayList cuyos elementos son a su vez ArrayList. Esto se puede extender sucesivamente y obtener arrays de más dimensiones. Ejemplo:

```
ArrayList<ArrayList<Integer>> miArray = new ArrayList <>();
```

Una vez creado, hay que pensar que cada elemento de *miArray* es a su vez un array, por lo que para añadir un elemento sería así: `miArray.add(new ArrayList<>());`

Ahora, para acceder a los elementos finales (fila, columna) sería así:

```
miArray.get(fila).add(valor);
miArray.get(fila).get(columna);
```

Siendo `miArray.size()`; la cantidad de filas y `miArray.get(fila).size()`; la cantidad de columnas de una fila. Así recorreríamos un arrayList de dos dimensiones:

```
for(int i=0;i<miArray.size();i++) { //para cada fila
    System.out.print("Fila: " + i + ": ");
    for(int j=0;j<miArray.get(i).size();j++) { //recorre col. de la fila
        System.out.print(miArray.get(i).get(j) + " ");
    }
    System.out.println(); //siguiente fila
}
```

Conversión Array - ArrayList

Al igual que ocurría entre un String y un StringBuffer, a veces, nos interesan características que están en la clase Array y otras veces en ArrayList por lo que podemos hacer una conversión rápida del primero al segundo mediante el método estático **Arrays.asList** y del segundo al primero con **toArray**.

Supongamos que tenemos un Array de enteros y necesitamos ordenarlo ascendente. Podríamos hacer:

```
Integer [] arr = {10, 3, 7, 2, 9, 5};
List <Integer> lista = new ArrayList<>(Arrays.asList(arr));
Collections.sort(lista);
arr = lista.toArray(new Integer[lista.size()]);
```

Clase Collections

Esta clase (no confundir con la interfaz *Collection*) es una clase con un conjunto de métodos **estáticos** que permiten operar sobre distintas colecciones como pueden ser los *ArrayList* (y otras que veremos en capítulos posteriores) que pueden ser muy útiles y que los usaremos frecuentemente.

- `Collections.max(arraylist1)` Devuelve el máximo elemento de *arraylist1*, de acuerdo al orden natural de sus elementos. También disponemos del análogo `min(arraylist1)`.
- `Collections.reverse(arraylist1)` Invierte todos los elementos del *ArrayList*. No devuelve nada.
- `Collections.shuffle(arraylist1)` Intercambia aleatoriamente todos los elementos del *ArrayList*. No devuelve nada.
- `Collections.frequency(arraylist1, obj)` Devuelve el número de veces que aparece el objeto *obj* en *arraylist1*.

Además de los ya vistos previamente:

- `Collections.binarySearch(arraylist1, obj)` Busca el objeto *obj* en *arraylist1* y devuelve posición. Si no encuentra, devuelve < 0. El *ArrayList* debe estar ordenado.
- `Collections.sort(arraylist1)` ordena *arraylist1* ascendente. También disponemos del método `reverseOrder()` para ordenar descendente como ya mencionamos previamente.

Puedes ver la lista completa en: <https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html>

Nota:

ArrayList tiene definido el método `toString()` por lo que podemos hacer `System.out.println(miArrayList)` pero no así en el caso de Array, que debeamos emplear el método estático `Arrays.toString` para convertirlo previamente en String y poder hacer `System.out.println(Arrays.toString(miArray))`.

.

8. Clases y Herencia

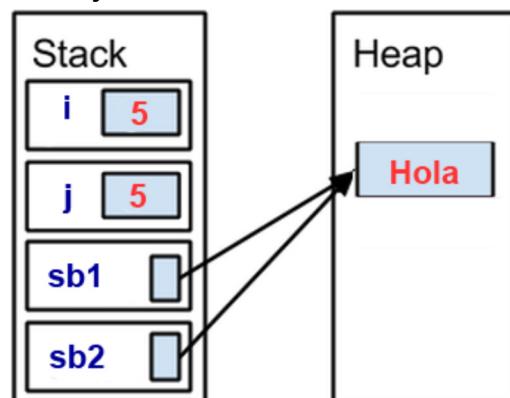
Conceptos Básicos

En el tema anterior vimos los fundamentos de las clases y objetos. Resumimos elementos básicos de la orientación a objetos:

- **Clase:** es la descripción de una identidad que queremos modelar e incluye atributos y métodos.
- **Objeto:** es la instancia de una clase. Se genera invocando a un constructor de la clase.
- **Atributo:** es cada una de las características que conforman una clase.
- **Método:** representa una acción o comportamiento de una clase. Se les pueden pasar parámetros y pueden devolver un valor.
- **Constructor:** son unos métodos especiales que se encargan de crear los objetos. Toda clase debe tener alguno. Si no lo creamos, Java lo hace por nosotros de forma implícita. **No devuelven ningún valor, no puede ser static, ni final ni abstract.**
- **Modificadores de acceso:** definen quien puede acceder a la clase/método/atributo al que califican: public, private, protected y por defecto. **Las clases sólo pueden ser: public o por defecto.** El nombre de una clase pública tiene que coincidir con el nombre del fichero en el que se guarda.
- Atributos y métodos **estáticos:** Se aplican a la clase, no a una instancia de la clase.

Otros asuntos a tener en cuenta:

- Es importante recordar también como se almacenan las instancias de las clases y las variables que las referencian. Ya sabemos que una variable es un lugar en la memoria donde se guarda un dato. Para ser exacto, este lugar en la memoria es la Pila o Stack. En el caso de los datos primitivos, como en int i = 5; hay cuatro bytes en la Pila donde se almacena el número 5. Si creamos otra variable, aún con el mismo valor: int j = 5; se asignará un nuevo espacio en la pila.
- Cuando se crea un objeto en Java, como un *StringBuilder*, el contenido del objeto se guarda en una parte de la memoria llamada *Heap*. Cuando asignamos el objeto a una variable como en `StringBuilder sb1 = new StringBuilder("Hola");` lo que guardamos en *sb1* es la dirección de memoria *Heap* donde está el objeto realmente. Y si creamos otra variable *sb2* que igualamos a *sb1* no "ocupamos" una nueva zona de memoria en el *Heap*, sino que la nueva variable referencia a la misma zona de memoria. Cambiar el contenido referenciado por una, cambiaría también el referenciado por *sb2*. Es frecuente, por tanto, cuando hablamos de variables de tipo objeto, que les llamemos **referencias** y, en general, hablamos de que una variable contiene un dato o una referencia.
- Inicialización por defecto y valor **null**: **cuando declaramos una variable, pero no la inicializamos, Java la inicializa automáticamente.** En el caso de los tipos primitivo escoge un valor por defecto, por ejemplo 0 para los numéricos como int, false para boolean, etc. En **el caso de los objetos, se inicializa con un valor o referencia llamado null.**



Principios de la POO

Existe un acuerdo acerca de qué características contempla la “orientación a objetos”. Las características siguientes son las más importantes:

Abstracción

La **abstracción** consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. Así expresaremos las características esenciales del objeto y su comportamiento esencial, eliminando lo superfluo. *Ejemplo: ¿Qué características podemos abstraer de un coche? o ¿Qué características y comportamientos semejantes tienen todos los coches?* Características: Marca, Modelo, Matrícula... Comportamiento: Acelerar, Frenar, Retroceder...

Encapsulamiento

Significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. También se suele asociar con el principio de ocultación, principalmente porque se suelen emplear conjuntamente. Esto es, solo se puede acceder a las características y comportamientos de un objeto mediante las operaciones permitidas, sin conocer realmente su funcionamiento interno. Así el usuario puede centrarse en qué hace y no como lo hace.

Herencia

Las clases no se encuentran aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento, permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Por ejemplo, en el juego del ajedrez una clase Pieza tendrá unos atributos (como su color o posición en el tablero) y unos métodos. Una clase “hija” como la clase “peón” heredará esos atributos y métodos, podrá tener nuevos métodos o redefinir alguno de su clase padre.

Polimorfismo

Las definiciones habituales de polimorfismo son bastante difíciles de entender, por ejemplo: *“Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando.”*

“El polimorfismo se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía”.

Podríamos decir que el polimorfismo es una “relajación” del sistema de tipos, de tal manera que una referencia a una clase (por ejemplo, una variable) acepta referencias de objetos de dicha clase y de sus clases derivadas (hijas, nietas, etc.).

Después de ver los apartados de herencia y polimorfismo, ya con casos concretos, podremos volver a esta definición y la entenderemos perfectamente.

Modularidad

Se denomina “modularidad” a la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. Estos módulos se pueden compilar por separado, pero tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan el modularidad de diversas formas.

Principio de ocultación

Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una "interfaz" a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas; solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no puedan cambiar el estado interno de un objeto de manera inesperada, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción.

Recolección de basura

La recolección de basura (*garbage collection*) es la técnica por la cual el entorno de objetos se encarga de destruir automáticamente, y por tanto desvincular la memoria asociada, los objetos que hayan quedado sin ninguna referencia a ellos. Esto significa que el programador no debe preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un nuevo objeto y la liberará cuando nadie lo esté usando.

Mensajes

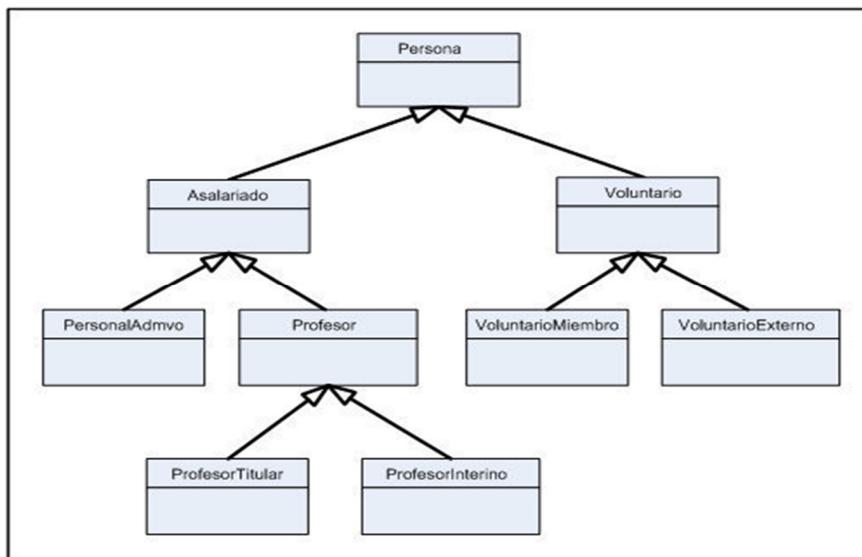
Un mensaje es una comunicación o solicitud que le hacemos a un objeto para que actúe según su comportamiento definido. En términos prácticos consiste en la invocación de un método de un objeto.

Herencia

La herencia es uno de los 4 pilares de la programación orientada a objetos (junto con la **Abstracción**, **Encapsulación** y **Polimorfismo**). Al principio cuesta un poco entender estos conceptos característicos del paradigma de la POO porque solemos venir de otro paradigma de programación como el paradigma de la programación estructurada, pero se ha de decir que la complejidad está en entender este nuevo paradigma y no en otra cosa.

Una posible definición: "La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente los métodos y atributos entre clases y sus clases sucesoras.

Ejemplo: vamos a modelar los distintos tipos de alumnos que hay en un instituto:



- **Alumno ESO:** nos interesa saber su nombre, DNI, faltas en el curso y teléfono de los padres para llamar cuando hay una falta de asistencia. Se tendrá que hacer periódicamente un recuento de faltas porque si falta más 30 sesiones, habrá que llamar a los servicios de Asuntos Sociales del ayuntamiento.
- **Alumno Ciclos:** nos interesa saber su nombre, DNI, faltas en el año, empresa donde hará las prácticas y email para notificarle cualquier aviso. Si falta más de 50 sesiones se le da de baja.

A nivel de código, tendremos lo siguiente:

```
class AlumnoESO {
    public String nombre;
    public String dni;
    public int faltas;
    public int telefono;

    AlumnoESO () {} // Flecha roja

    AlumnoESO(String nombre, String dni, int faltas, int telefono) {
        this.nombre = nombre;
        this.dni = dni;
        this.faltas = faltas;
        this.telefono = telefono;
    }

    public void resetFaltas() { this.faltas = 0; } // Flecha roja

    public void nuevaFalta(int sesiones) {
        this.faltas += sesiones;
        System.out.println("Falta registrada. Llamar a " + this.telefono);
    }
}
```

Y, por otra parte:

```
class AlumnoCiclos {
    public String nombre;
    public String dni;
    public int faltas; // Flecha roja
    public String empresa;
    public String email;

    AlumnoCiclos () {} // Flecha roja

    AlumnoCiclos(String nombre, String dni, int faltas,
                  String empresa, String email) {
        this.nombre = nombre;
        this.dni = dni;
        this.faltas = faltas;
        this.empresa = empresa;
        this.email = email;
    }

    public void resetFaltas() { this.faltas = 0; } // Flecha roja

    public void nuevaFalta(int sesiones) {
        this.faltas += sesiones;
        System.out.println("Falta registrada. Notificar a " + this.email);
        if (this.faltas > 50) System.out.println("Alumno dado de baja");
    }
}
```

Las flechas indican código idéntico en ambas clases., repetimos mucho código ya que las dos clases tienen métodos y atributos comunes, de ahí decimos que la herencia consiste en "sacar factor común" para no escribir código de más, por tanto lo que haremos será crear una clase con el "código

que es común a las dos clases" (a esta clase se le denomina en la herencia como "Clase Padre o Superclase o Clase Base") y el código que es específico de cada clase, lo dejaremos en ella, siendo denominadas estas clases como "Clases Hijas, Subclases o Clases Derivadas", las cuales heredan de la clase padre todos los atributos y métodos públicos o protegidos.

Es muy importante decir que las clases hijas no van a heredar nunca los atributos y métodos privados de la clase padre, solo heredamos elementos public y protected (en realidad los atributos y métodos privados si se heredan, pero no están accesibles). A nivel de código, las clases quedarían implementadas de la siguiente forma:

```
public class Alumno {
    public String nombre;
    public String dni;
    public int faltas;

    Alumno() { }

    Alumno(String nombre, String dni) {
        this.nombre = nombre;
        this.dni = dni;
        this.faltas = 0;
    }

    public void resetFaltas() { this.faltas = 0; }

}

class AlumnoESO extends Alumno {
    public int telefono;

    AlumnoESO() { }

    AlumnoESO(String nombre, String dni, int telefono) {
        super(nombre, dni);
        this.telefono = telefono;
    }

    public void nuevaFalta(int sesiones) {
        this.faltas += sesiones;
        System.out.println("Falta registrada. Llamar a " + this.telefono);
    }
}

class AlumnoCiclos extends Alumno {
    public String empresa;
    public String email;

    AlumnoCiclos() { }

    AlumnoCiclos(String nombre, String dni, String empresa, String email) {
        super(nombre, dni);
        this.empresa = empresa;
        this.email = email;
    }

    public void nuevaFalta(int sesiones) {
        this.faltas += sesiones;
        System.out.println("Falta registrada. Notificar a " + this.email);
        if (this.faltas > 50) System.out.println("Alumno dado de baja");
    }
}
```

Nota: vuelve al tema 6 a repasar los modificadores de acceso: public, protected, private y default.

Ahora queda un código mucho más limpio, estructurado y con menos líneas de código, lo que lo hace más legible, y lo que todavía es más importante es que es un código **reutilizable**, lo que significa que ahora si queremos añadir más clases a nuestra aplicación como por ejemplo una clase *AlumnoPrimaria*, lo podemos hacer de forma muy sencilla ya que en la clase padre tenemos implementado parte de sus datos y de su comportamiento y solo habrá que implementar los atributos y métodos propios de esa clase.

Encontramos dos palabras reservadas nuevas:

- **extends:** Como ya imaginaremos, indica a la clase hija cual va a ser su clase padre, en el ejemplo, la clase *AlumnoESO* tiene como parente a *Alumno*, de forma que la primera hereda todos sus atributos y métodos públicos o protegidos.
- **super:** es una llamada al constructor de la clase parente. Si hacemos *super ()* o llamaría al constructor por defecto de la clase parente, esto es a: *Alumno()*, y en el caso del ejemplo, *super (nombre, dni)* llama al constructor de dos parámetros *Alumno (String nombre, String dni)*.
- Con *super ()* también podemos llamar a métodos de la clase parente, por ejemplo *super.nombreMétodo (parámetros del método parente)*;

La creación de instancias de cada clase se hace como si no hubiese herencia, en nuestro caso podríamos hacer un programa con el siguiente código:

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

    Scanner teclado = new Scanner(System.in);
    AlumnoESO alum1 = new AlumnoESO ("Juan Pérez", "32233N", 981900900);
    AlumnoCiclos alum2 = new AlumnoCiclos ("Ana López", "77700K", "ABanca", "ana@gmail.com");
    alum1.nuevaFalta(3); alum1.nuevaFalta(20); alum1.nuevaFalta(12);
    alum2.nuevaFalta(7); alum2.nuevaFalta(20); alum2.nuevaFalta(33);
} //fin main
```

Prueba a codificar todo esto a ver qué salida obtienes. Puedes juntar en un único archivo las clases definidas y el programa que contenga un *main()* como el que acabamos de mostrar.

Aunque una clase pública tiene que ir en un archivo independiente, con el mismo nombre que la clase y extensión *.java*, podemos crear varias clases sin modificador de acceso en un mismo archivo e incluir el programa que las usa, el nombre del archivo será el del programa (*.java*). Estas clases solo serán accesibles desde otras clases/programas del mismo paquete (modificador de acceso por defecto) pero es una forma cómoda de hacer nuestros ejercicios.

Constructores y herencia

Viendo el ejemplo podemos establecer las siguientes consideraciones:

- Es posible que tanto las superclases como las subclases tengan sus propios constructores, entonces ¿qué constructor es responsable de construir un objeto de la subclase, el de la superclase, el de la subclase o ambos? El constructor de la superclase construye la porción de la superclase del objeto, y el constructor para la subclase construye la parte de la subclase.
- Esto tiene sentido porque la superclase no tiene conocimiento ni acceso a ningún elemento en una subclase. Por lo tanto, su construcción debe estar separada.
- Cuando solo la subclase define un constructor, el proceso es sencillo: simplemente construye el objeto de la subclase. La porción de superclase del objeto se construye automáticamente utilizando el constructor predeterminado de la superclase.

- **this ()** se usa para llamar a uno de sus constructores (depende del número de parámetros que le pasemos llamará a uno o a otro). Es decir, desde un constructor puedo llamar a otro de la misma clase.
- **super ()** se usa para llamar a uno de los constructores de la clase padre (depende del número de parámetros que le pasemos llamará a uno o a otro). Es decir, desde un constructor de la clase hija puedo llamar a cualquiera de los constructores de su padre, incluyendo el constructor por defecto.
- El constructor de una clase hija siempre llama al constructor de la clase padre. Si no se indica explícitamente, se llama al constructor sin argumentos de forma implícita.
- En un constructor solo puede haber un **super ()** o un **this ()** y obligatoriamente tiene que ir en la primera línea del constructor.
- Los constructores no son heredados por subclases, pero el constructor de la superclase puede invocarse desde la subclase con **super ()**.
- Si queremos impedir la herencia en una clase, es decir, que no se puedan derivar clases hijas, le asignaremos el modificador **final**. Por ejemplo, si no queremos permitir clases hijas de *AlumnoCiclos* como podría ser *AlumnoCicloMedio*.

```
public final class AlumnoCiclos extends Alumno {.....}
```

Ejemplo: ¿Qué error tiene ese código?

```
class Padre {
    public String nombre;

    public Padre (String nombre) {this.nombre = nombre; }
}
class Hijo extends Padre {
    public int edad;

    public Hijo () {edad=10;}
}
```

El error consiste en que en la primera línea del constructor hijo no hay una llamada explícita al constructor padre mediante **super ()** por lo que se llama al constructor por defecto **Padre () {}**, pero este constructor no existe. Para que Java cree ese constructor por defecto en el padre no debe haber ningún otro constructor.

El error se podría solucionar de dos formas diferentes.

- a) En la primera línea del constructor hijo, llamar al constructor padre que sí existe, por ejemplo, así: **super ("noname") ;**
- b) Crear en la clase padre, el constructor por defecto: **Padre () {}**

No producirían el mismo resultado las dos soluciones, pero ambas evitarían el error de compilación.

Herencia en métodos y sobrescritura

Los métodos heredados se pueden usar directamente tal como son en las clases hijas, pero también se pueden crear nuevos métodos en las subclases. En nuestro ejemplo, podríamos añadir a *AlumnoCiclos* un método para saber si va a hacer las prácticas FCT:

```
public boolean accedeFCT (int []notas) {
    for (int i: notas) if (i<5) return false;
    return true;
}
```

La sobreescritura de un método consiste en escribir un método en la subclase que tenga la misma “firma” que el de la superclase y también que devuelva el mismo tipo de dato de retorno (o al menos un subtipo de este). En este caso, para la subclase, no se ejecutaría el método padre, se ejecutaría el sobreescrito en ella misma.

La **firma** o **signatura** de un método es su nombre y los parámetros que recibe. Cuando decimos que dos métodos tienen la misma firma nos referimos a que tiene el mismo nombre y el mismo número de parámetros y éstos con el mismo tipo de datos.

En nuestro caso, vemos que una nueva falta tiene un tratamiento común para ambas clases hijas (incrementar el atributo *faltas*) pero luego hay un tratamiento distinto del límite de faltas en cada una de las subclases. Ese incremento lo podríamos hacer en la clase padre:

```
public void nuevaFalta (int sesiones) { faltas += sesiones;}
```

Y en las clases hijas podemos volver a definir el método, llamando primero al método del padre y luego hacer las tareas propias del método hijo.

Así en la clase *AlumnoESO* tendríamos:

```
public void nuevaFalta (int sesiones) {
    super.nuevaFalta(sesiones);
    System.out.println ("Falta registrada. Llamar a " + telefono);
    if (faltas > 30) System.out.println ("Llamar a asuntos sociales");
}
```

Y en la clase *AlumnoCiclos* tendríamos:

```
public void nuevaFalta (int sesiones) {
    super.nuevaFalta(sesiones);
    System.out.println ("Falta registrada. Notificar a " + email);
    if (faltas > 50) System.out.println ("Alumno dado de baja");
}
```

Siendo puristas, aunque sintácticamente sí es una sobreescritura de métodos, al llamar a *super.nuevaFalta (sesiones)* como primera línea, en realidad no estamos “sustituyendo” el método, lo estamos “extendiendo” o “ampliando”.

En la **sobreescritura** nos debemos fijar en que la estructura del método sea igual a la de su superclase, no solo el mismo nombre sino el mismo número de argumentos y tipo de retorno (o al menos una subclase de este), así como no tener un nivel de acceso más restrictivo que el original (que en la clase padre sea *protected* y en la hija sea *private*, por ejemplo) tampoco se pueden sobreescibir métodos *static* ni *final*.... (ya que *static* representa métodos globales y *final/constantes*.)

Algo que también debemos mencionar es que podemos identificar un método sobreescrito cuando tiene la anotación **@Override**, sin embargo, no es obligatorio ponerlo, pero si es recomendable (pues de esta manera el compilador reconoce que este método está sobreescribiendo un método de una superclase de ésta, ayudando a que, si nos equivocamos al construirlo, el compilador nos avisaría).

Adicionalmente si tenemos la anotación inmediatamente podremos saber que se está aplicando el concepto, algo muy útil cuando trabajamos con código de otras personas.

Las **anotaciones** son metadatos que se pueden asociar a clases, miembros, métodos o parámetros. No cambian las acciones de un programa, pero dan información sobre el elemento que tiene la anotación y permiten definir cómo queremos que sea tratado por distintas herramientas (en la compilación, documentación, ejecución, etc.) Comienzan siempre por @.

Algunas de las más comunes son @Override, @SuprressWarnings, @Deprecated, etc.

Por último, si queremos impedir la sobrescritura de un método en las subclases le asignaremos el modificador **final** (es análogo a lo que hacíamos con las clases, que al definirlas *final* impedíamos crear subclases)

getClass y instanceof

getClass es un método que devuelve la clase en tiempo de ejecución del objeto sobre el que se llama. Si lo imprimimos, muestra una referencia que incluye el nombre de la clase y el paquete en el que se encuentra, pero podemos obtener más información, como el nombre concreto de la clase (sin el paquete):

```
obj.getClass().getSimpleName();
```

El operador *instanceof* es parecido al método anterior, y nos permite preguntar el tipo de la variable. Ejemplo:

```
if (a3 instanceof AlumnoCiclos)
    System.out.println( ((AlumnoCiclos)a3).empresa );
```

instanceof devuelve un valor booleano indicando si la instancia pertenece a la clase o no. Hay que resaltar que también devuelve *true* para todas las clases ancestro (padre, abuelo, etc.) de la clase a la que pertenece.

En nuestro ejemplo, si la variable 'a3' es un 'AlumnoCiclos':

a3 instanceof AlumnoCiclos *se evaluará como true*, pero, además:

a3 instanceof Alumno también será true. Finalmente:

a3 instanceof AlumnoESO Obviamente será false.

Para saber la clase exacta de una instancia en vez de *instanceof* es mejor emplear *getClass().getSimpleClassName()*. Puedes investigar en internet su uso.

Últimas consideraciones sobre la herencia

- **Superclase predeterminada:** excepto la clase **Object**, que no tiene superclase, cada clase tiene una y solo una superclase directa (herencia única). En ausencia de cualquier otra superclase explícita, cada clase es implícitamente una subclase de la clase *Object*.

Una superclase puede tener cualquier cantidad de subclases, pero una subclase solo puede heredar de una superclase. Esto se debe a que Java no admite herencia múltiple con clases. Más adelante veremos las *interfaces*, donde sí se permite la herencia múltiple.

- **Variable superclase / constructor subclase:** Una última consideración sobre la herencia, y relacionada con el polimorfismo que hablaremos más adelante, es el hecho de que en muchas ocasiones nos encontraremos variables del tipo de la superclase que son asignadas a constructores de las subclases.

```
Alumno alum1 = new AlumnoCiclos("Ana López", "77700K", "ABanca", "ana@gmail.com");
```

Java permite estas asignaciones y ofrecen mucha flexibilidad. Por ejemplo, podríamos crear un ArrayList de alumnos, y añadir tanto alumnos de la ESO como de ciclos. Lo veremos más adelante. Por ahora quedarnos solo con la idea de que podemos encontrárnoslo.

- **this.:** En capítulos anteriores vimos cómo, para los métodos de una clase, podíamos preceder con this. a los atributos de la misma. Pues bien, seguiremos haciendo lo mismo, incluso cuando esos atributos no hayan sido creados en esa clase si no que provengan de su superclase, esto es, su clase padre.
- **Getters/Setters:** aunque una subclase no hereda los miembros privados de su clase principal, si la superclase tiene métodos públicos o protegidos (como getters y setters) para acceder a esos atributos privados, estos métodos sí pueden ser utilizados por la subclase y así, acceder a los miembros privados. Esta es una situación muy habitual.

Como resumen:

- Una clase hija incorpora la estructura (atributos public/protected) y comportamiento (métodos public/protected) de la clase padre.
- La clase hija puede añadir nuevos atributos y métodos.
- La clase hija puede redefinir métodos de su clase padre (sobrescritura).
- Una clase solo puede tener una y solo una clase padre (Object es la clase primigenia).
- Los constructores no se heredan.
- El constructor de la clase hija puede llamar explícitamente al constructor que deseé de su clase padre, pero debe ser en la primera línea del constructor hijo.
- Si el constructor de la clase hija no llama explícitamente a un constructor de su clase padre, se llama implícitamente al constructor por defecto de la clase padre.
- Los atributos no se redefinen, o se heredan o no.
- "final" aplicado a una clase impide ser heredada, a un método ser redefinido.

9. Polimorfismo

Tipos de Polimorfismo

Como primera aproximación podríamos ver alguna definición comentada ya previamente, aunque con ejemplos prácticos lo entenderemos mucho mejor.

"Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando."

El polimorfismo hace referencia a "tener varias formas" y en la programación orientada a objetos, tiene varias vertientes:

- Polimorfismo puro.
- Polimorfismo ad hoc o sobrecarga.
- Polimorfismo de inclusión (redefinición o sobrescritura).
- Polimorfismo paramétrico (generalidad).

En realidad, no nos importa tanto la nomenclatura ni las definiciones, como su funcionalidad en la programación y las ventajas que nos reporta. Vamos a ver a continuación las dos primeras, la tercera ya la hemos visto en la sobrescritura de métodos del capítulo anterior, y la última la dejamos para el siguiente trimestre, bajo el título de "*Genéricos*".

Polimorfismo Puro

Ya sabemos que una variable de tipo objeto en realidad contiene una referencia a memoria donde realmente está almacenado el objeto al que apunta. También sabemos que una vez que declaramos una variable de un determinado tipo (de tipo objeto o tipo primitivo) no se puede cambiar ese tipo. Por ejemplo, una variable llamada *a1* de tipo *Alumno* no puede ser más delante de tipo *TelefonoMovil*. Y, por último, sabemos también que el valor referenciado por una variable puede ir cambiando en un programa (salvo que la declaremos como *final*), es decir la variable *a1* puede apuntar a un alumno, y más tarde en el programa a otro alumno.

Lo nuevo que vamos a ver ahora es que una variable de un determinado tipo puede referenciar objetos de este tipo, pero también objetos de clases hijas de ese tipo, por eso decimos que es polimórfica.

Estas variables polimórficas pueden ir cambiando de un tipo a otro en tiempo de ejecución (eso sí, no a cualquier tipo, solo dentro de la jerarquía de herencia).

Partiendo del ejemplo previo de herencia (*Alumno* -> *AlumnoESO*, *AlumnoCiclos*), una variable polimórfica sería la definida de tipo *Alumno*, y en tiempo de ejecución asignarle instancias tanto *AlumnoESO* como de *AlumnoCiclos*. Java nos permite entonces instrucciones como esta:

```
Alumno a1 = new Alumno ();
Alumno a2 = new AlumnoESO ("Juan Pérez", "32233N", 981900900);
Alumno a3 = new AlumnoCiclo ("Ana López", "77700K", "ABanca", "ana@gmail.com");
```

Una vez instanciada la variable o referencia de la superclase o clase "padre" con una clase derivada o clase "hija", solo puedo acceder a los atributos y métodos definidos en la superclase. Si trato de acceder a atributos/métodos definidos en la subclase pero no en la superclase, se producirá un error de compilación.

Por ejemplo, produciría un error de compilación:

```
System.out.println (a3.empresa);
```

Ya que 'empresa' no es un atributo de la clase Alumno, es de AlumnoCiclo. Para solucionar esta limitación, debemos hacer un casting de esa variable al tipo de la subclase.

```
System.out.println ( ((AlumnoCiclo)a3).empresa );
```

El único caso en el que no es necesario el casting para acceder a métodos de la subclase, es si estos métodos son una sobrescritura de un método de la superclase.

Vamos a ver con un ejemplo distintos casos:

```
class Padre {
    public String nombre;

    public Padre (String n) {nombre = n;}
    void cambiaNombre () { nombre = nombre.toUpperCase(); }
    void minusc () { nombre = nombre.toLowerCase(); }
}

class Hijo extends Padre {
    public int edad;
    public Hijo (String n) {super(n); edad=10; }

    @Override
    void cambiaNombre () { this.nombre += " (es Hijo)"; }
    void cumple () {edad++; }
}
```

Si creamos una variable: Padre p1 = new Hijo ("Juan") tendríamos los siguientes casos:

- **p1.minusc();** llamaría al método de la clase padre, pasando el nombre a minúsculas.
- **((Hijo)p1).cumple();** Hay que hacer un casting a *Hijo* ya que *p1* es de tipo *Padre* y no tiene ningún método llamado *cumple()*;
- **p1.cambiaNombre();** Ejecuta el método de la clase hija, y sin necesidad de casting, ya que este método existe en la clase padre. Hay que tener cuidado con esto, al contrario de lo que pudiese parecer, no ejecuta el *cambiaNombre()* de la clase padre. **Quien decide los métodos que se aplican es la clase real, no el tipo de su referencia.**

Java permite estas asignaciones y ofrecen mucha flexibilidad. Por ejemplo, podríamos crear un ArrayList de alumnos, y añadir tanto alumnos de la ESO como de ciclos, como veremos en el siguiente apartado.

Siguiendo con el ejemplo anterior, en muchos casos, nos encontraremos con una variable y, debido al polimorfismo, puede que no sepamos a qué tipo de objeto referencia. Para solucionar esto, disponemos del operador ya comentado previamente *instanceof* que nos permite preguntar el tipo de la variable. Ejemplo:

```
if (a3 instanceof AlumnoCiclos)
    System.out.println ( ((AlumnoCiclos)a3).empresa );
if (a2 instanceof AlumnoESO)
    AlumnoESO aESO = (AlumnoESO)a2;
```

ArrayList Polimórfico

Podemos crear un ArrayList de *alumnos*, definiéndolo de la clase padre, con lo que nos va a permitir añadir elementos de esta clase, pero, lo que es más interesante, también de las clases hijas.

Podremos recorrer el ArrayList y ejecutamos sus métodos “comunes” como *setNombre()*, también los redefinidos como *nuevaFalta()*. Dinámicamente él ejecuta el método correspondiente a su clase, sea una clase hija o padre.

Este proceso, llamado **Vinculación dinámica**, es el mecanismo que utiliza la máquina virtual de Java para averiguar, en tiempo de ejecución, a qué método debe llamar, a partir del tipo del objeto asignado a una referencia.

En el siguiente ejemplo, cuando llamamos al método *nuevaFalta()* Java decide en tiempo de ejecución qué método ejecutar, en función si se trata de un alumno de tipo *Alumno*, *AlumnoESO* o *AlumnoCiclos*.

```
public static void main(String[] args) {
    ArrayList <Alumno> instituto = new ArrayList<>();

    AlumnoESO     alum1 = new AlumnoESO ("Juan Pérez", "32233N", 981900900);
    AlumnoESO     alum2 = new AlumnoESO ("Eva Gómez", "123123A", 881111222);
    AlumnoCiclos alum3 = new AlumnoCiclos ("Ana López", "77700K", "ABanca", "ana@gmail.com");
    AlumnoCiclos alum4 = new AlumnoCiclos ("Luis Cal", "987654L", "Indra", "luis@gmail.com");
    Alumno        alum5 = new Alumno      ("Plantilla", "0000000A");
    instituto.add(alum1); instituto.add(alum2); instituto.add(alum3);
    instituto.add(alum4); instituto.add(alum5);

    for (Alumno i : instituto) {
        i.setNombre(i.nombre.toUpperCase()); //no definimos getter, nombre es público
        i.nuevaFalta(60);
    }
    for (Alumno i : instituto)
        System.out.println(i.nombre + " " + i.faltas);
} //fin main
```

Caso diferente es el de ejecutar métodos que solo existen en algunas de las instancias del ArrayList. Por ejemplo, no podemos ejecutar el método *setEmpresa()* para alumnos que no sean de la clase hija *AlumnoCiclos*.

Para saber la clase a la que pertenece una determinada instancia de clase, usaremos **instanceof** del que ya hablamos en el apartado anterior.

```
for (Alumno i : instituto)
    if (i instanceof AlumnoCiclos)
        ((AlumnoCiclos) i).setEmpresa ("Por definir");

for (Alumno i : instituto)
    if (i instanceof AlumnoCiclos)
        System.out.println("Empresa: " + ((AlumnoCiclos) i).Empresa);
```

En el capítulo siguiente veremos el concepto de interfaz, pero ya adelantamos para los que ya las conoczáis, que también hay polimorfismo con ellas: podemos crear una variable de tipo de una interfaz e instanciarla con clases que implementen dicha interfaz.

Sobrecarga de Métodos

La sobrecarga de un método (*overload*) se llama también polimorfismo estático y se produce si los parámetros del método son:

- Distinto número de parámetros.
- Mismo número de parámetros, pero de distinto tipo.
- Mismo número, del mismo tipo, pero en distinto orden.

En cualquiera de estos tres casos siempre devuelve el mismo tipo de dato, no cambia. Si no, no lo llamaríamos sobrecarga.

Por ejemplo: partiendo del método: *int metodo (int a, int b){}* el método: *float metodo (int a, int b){}* no representa una sobrecarga del primero, ya que sólo se diferencian en el valor de retorno no valdría la sobrecarga.

El método *int metodo (float a, int b) {}* sí representa sobrecarga.

La sobrecarga de métodos es un mecanismo muy útil que permite definir en una clase varios métodos con el mismo nombre (de hecho, la hemos utilizado previamente en los constructores).

```
public void sumar (int a, int b) {  
    int suma= a + b;  
    System.out.println("la suma es: "+suma);  
}  
  
public void sumar (double x, double y) {  
    double sum= x + y;  
    System.out.println("la suma es: "+sum);  
}
```

Para la sobrecarga, no es necesario que haya herencia, sí para la sobrescritura.

10. Clases Abstractas e Interfaces

Clases Abstractas

Supongamos una clase *Trabajador* de la que heredan *Empleado* y *Consultor*. Todo trabajador será o bien *Empleado* de la empresa o bien un Consultor contratado temporalmente, es decir, que no vayan a existir instancias de la clase *Trabajador*. Entonces, ¿qué sentido tendría tener una clase *Trabajador*?

El sentido está en ordenar las clases de nuestra jerarquía. También puede servir para definir un ArrayList de la superclase como vimos en el apartado anterior, y luego introducir en él instancias de cualquiera de sus subclases.

Estas clases de las que no se crean instancias, que son simplemente un nivel de abstracción, de forma que las instancias se crearán a partir de clases hijas, se llaman abstractas y se definen así:

```
public abstract class Trabajador { ... }
```

Para este tipo de clases, no es posible instanciar la clase, es decir, no resulta posible crear objetos de ese tipo. Sin embargo, sigue funcionando como superclase de forma similar a como lo haría una superclase "normal".

Pensemos análogamente en **métodos abstractos**. Supongamos que estamos haciendo un programa que juegue al ajedrez y la clase *Pieza* tiene un método llamado *mover ()*. No tiene sentido implementar en la clase padre dicho método ya que cada clase hija (*Rey*, *Caballo*, *Torre*, etc.) lo hará de forma distinta. Estos métodos son abstractos y tienen estas características:

- No tienen cuerpo (llaves): sólo constan de signatura con paréntesis y punto y coma final.
- Sólo pueden existir dentro de una clase abstracta. Visto de otra manera, si una clase incluye un método abstracto, forzosamente la clase será una clase abstracta.
- Por el contrario, una clase abstracta no tiene por qué tener algún método abstracto.
- Los métodos abstractos forzosamente habrán de estar sobrescritos (definidos) en las subclases.

```
public abstract class Pieza {
    //atributos
    ...
    //constructores y métodos
    ...
    public abstract mover ();
}
```

Y en la clase hija:

```
public class Rey extends Pieza {
    //atributos
    ...
    //constructores y métodos
    ...
    public mover () {
        ...
    }
}
```

En el siguiente cuadro podemos ver un resumen de todas las características que definen una clase como abstracta.

Clases Abstractas: Resumen

- Es una clase que lleva el prefijo **abstract**.
- No permite crear instancias de ella (por lo tanto, no puede ser final).
- Tiene algún método abstracto (firma, pero sin cuerpo) que las subclases lo implementarán. Si la subclase no los implementa también será abstracta.
- Puede tener métodos no abstractos, métodos "normales".
- Desde Java8, puede tener métodos de tipo static que, como ya sabemos, se invocan sobre la clase y no son necesarias instancias para invocarlos.
- Una variable puede ser de tipo de superclase abstracta y podrá referenciar instancias de subclases no abstractas (polimorfismo).
- Los métodos abstractos son como un "compromiso", es decir, se definen las "plantillas" de los métodos que luego las clases hijas se verán obligadas a implementar, cumpliendo ese compromiso.

En el siguiente ejemplo, la clase abstracta *Empleado* tiene el método abstracto *calcularSalario()*, pero son las subclases las que lo implementen.

```
public abstract class Trabajador {
    public String nombre;
    public int bonus;
    public abstract int calcularSalarioAnual ();

    public Trabajador(String nombre, int bonus) {
        this.nombre = nombre; this.bonus = bonus;
    }
}
public class Empleado extends Trabajador {
    public int salarioBase;

    public Empleado(String nombre, int bonus, int salarioBase) {
        super(nombre, bonus);
        this.salarioBase = salarioBase;
    }
    @Override
    public int calcularSalarioAnual () {
        return salarioBase + bonus;
    }
}
```

```

public class Consultor extends Trabajador {
    public int HorasTrabajadas;
    public int PrecioHora;

    public Consultor(String nombre, int bonus, int HorasTrabajadas, int PrecioHora) {
        super(nombre, bonus);
        this.HorasTrabajadas = HorasTrabajadas;
        this.PrecioHora = PrecioHora;
    }
    @Override
    public int calcularSalarioAnual () {
        return bonus + PrecioHora * HorasTrabajadas;
    }
}

```

Podríamos crear ahora referencias a Trabajador y llamar al método abstracto:

```

Trabajador t1 = new Empleado("Juan", 2000, 30000);
Trabajador t2 = new Consultor("Bob", 3000, 500, 20);
System.out.printf
    ("Salario %s -> %d%n", t1.nombre, t1.calcularSalarioAnual());
System.out.printf
    ("Salario %s -> %d%n", t2.nombre, t2.calcularSalarioAnual());

```

Podemos preguntarnos **¿Qué beneficios aporta una clase abstracta?** En el ejemplo anterior podríamos pensar que si implementamos *calcularSalarioAnual()* directamente en las clases hijas, sin definirlo en la clase padre.

La respuesta a esta pregunta es que tenemos dos beneficios fundamentales:

Por una parte, **si el día de mañana creamos una nueva clase hija, el compilador nos obligará a implementar el método abstracto**, mientras que si no hubiésemos definido ese método como abstracto en la superclase, alguna subclase podría no implementarlo.

Por otra parte, como ya vimos en el capítulo de polimorfismo, solo puedo invocar a los métodos definidos en la clase de la variable (clase padre) y no de la clase con la que hemos instancia (clase hija) así que **para invocar los métodos que están en las clases hijas pero no en la superclase debo hacer un casting**.

Por lo tanto, sin el método abstracto anterior, no funcionaría:

```

System.out.printf
    ("Salario %s -> %d%n", t1.nombre, t1.calcularSalarioAnual());

```

Deberíamos hacer:

```

if (t1 instanceof Empleado)
    System.out.printf
        ("Salario %s -> %d%n", ((Empleado) t1).nombre,
         ((Empleado) t1).calcularSalarioAnual());
if (t1 instanceof Consultor)
    System.out.printf
        ("Salario %s -> %d%n", ((Consultor) t1).nombre,
         ((Consultor) t1).calcularSalarioAnual());

```

Métodos que requieren instancias

Como acabamos de comentar, no se pueden crear instancias de una clase abstracta, por lo que los métodos que requieren instancias como parámetro pueden ser problemáticos. Ejemplos de esta situación son los típicos métodos de ArrayList como puede ser *contains* o *remove*.

Siguiendo con el ejemplo anterior, imaginemos que tenemos un ArrayList de Trabajador y en un momento dado queremos eliminar

Interfaces

Una interfaz es similar a una clase abstracta en el sentido que define métodos abstractos que las clases vinculadas a ella deberán implementar. Podríamos definir una interfaz como una clase abstracta pura, en el sentido de que todos sus métodos son abstractos, no implementando ninguno de ellos, además no contendrá ningún atributo como las clases, ni otros métodos no abstractos.

Una clase que implementa una interfaz se compromete a implementar todos esos métodos abstractos definidos en la interfaz.

Ejemplo: una interfaz llamada *Avion* enumeraría los siguientes métodos: *despegar()*, *aterrizar()*, etc.

```
public interface Avion {
    int despegar(int metros);
    void aterrizar ();
}
```

Luego, clases como *AvionComercial*, *Avioneta*, etc. desarrollarán esos métodos de acuerdo a la firma especificada en la interfaz.

```
public class AvionComercial implements Avion {
    public String matricula;
    public float altura;

    @Override
    public int despegar (int metros) {
        this.altura+=metros;
        return this.altura;
    }

    @Override
    public void aterrizar () {
        this.altura=0;
    }
}
```

Podemos considerar las interfaces como un “compromiso” o como un contrato que obliga a las clases que la implementan a desarrollar el código de todos los métodos de la interfaz, o bien ser una clase abstracta, siendo las clases hijas (o nietas...) las que desarrollen.

Así pues, si *AvionComercial* no pudiese escribir el código del método de *aterrizar*, entonces *AvionComercial* debería ser abstracta a la espera de que un descendiente no abstracto lo escribiera.

Consideraciones sobre las interfaces:

- **Polimorfismo en interfaces:** Una interfaz se comporta como una clase abstracta: una variable o referencia puede ser de tipo interfaz y ser instanciada con un método de una clase que implemente dicha interfaz.

Por ejemplo, *List* es una interfaz y *ArrayList* es una clase que la implementa, por lo que será muy frecuente definir así un ArrayList:

```
List <Integer> miLista = new ArrayList <>();
```

Esto hace que el código sea más flexible ya que podríamos cambiar a otro tipo de clase que implemente la interfaz de forma muy cómoda:

```
List <Integer> miLista = new LinkedList <>();
```

En el ejemplo anterior:

```
Avion miAvion = new AvionComercial();
```

- Las interfaces, al igual que las clases, sólo pueden tener dos tipos de modificador de acceso: *public* o *default*, para las públicas, el nombre tiene que coincidir con el nombre del fichero en el que se guarda.
- Los métodos de las interfaces serán siempre *public* y *abstract* y no es obligatorio indicarlo. Hay una excepción: los métodos *default*, *private* o *static* que no son *abstract* de los que hablaremos a continuación.
- Cuando una clase implementa los métodos de una interfaz, tiene que hacerlos públicos explícitamente.
- Una interfaz puede heredar de otra interfaz usando la palabra reservada *extends*.
- Si una clase implementa más de una interfaz, se escriben todos los nombres de las interfaces separadas por comas. Por ejemplo: *public persona implements Cantante, Nacionalidad {...*

Las últimas versiones de Java, han incorporado tres tipos de métodos nuevos, que sí incluyen la implementación de los mismos en la propia interfaz, a diferencia de los métodos abstractos puros. En cierto modo, estos métodos desvirtúan la función de una interfaz, pero tienen su utilidad como veremos más adelante.

- **Métodos por defecto**: (modificador *default*) Se le añade un código general común para todas las clases. Aquellas clases que no implementen el método podrán usar esta implementación.
- **Métodos estáticos** (modificador *static*). Son métodos comunes a todas las clases que implementan la interfaz y por ello no tiene sentido que estén desarrollados en las clases.
- **Métodos privados** (modificador *private*). Son métodos desarrollados en la propia interfaz para ser usados en ella misma, ni en las clases que la implementan ni en subinterfaces. El beneficio clave de un método privado de interfaz es que permite que dos o más métodos por defecto utilicen una pieza común de código, evitando la duplicación de código.

Además de las firmas de los métodos, en las interfaces también podemos definir esos atributos estáticos y finales y podemos omitir *public static final*, ya son así implícitamente. Eso sí, tienen que estar inicializadas con un valor en la misma línea.

```
public interface Cantante {           //public abstract
    String formatoCancion = "mp3";     //public static final
    void cantar();                     //public abstract
    default double tarifa () {return 0;}
}
class Persona implements Cantante {      //public abstract
    //añadiríamos atributos y métodos de persona y ...
    @Override
    public void cantar() {
        System.out.println("La laa la raa laaa!");
    }
    @Override
    public double tarifa() {return 1000d;}
}
class Canario implements Cantante {      //no implementa método tarifa
    //añadiríamos atributos y métodos de canario y...
    public void cantar() {
        System.out.println("Pio Pio Pio");
    }
}
```

Pregunta: Supongamos que tenemos una interfaz ya creada y varias clases no abstractas que implementan sus métodos. ¿Qué ocurre si añadimos un nuevo método a la interfaz?

Respuesta: Se produciría un error de compilación en las clases, ya que no implementan el nuevo método y es algo obligatorio (si una clase no implementa todos los métodos de la interfaz, debe marcarse como abstracta).

Una solución para evitar estos errores es definir el método en la interfaz como **default**, con una implementación por defecto, así no se produciría el error y “daríamos tiempo” a los desarrolladores para que implementasen el nuevo método en las clases.

}

¿Para qué usar interfaces?

En principio podríamos pensar que una interfaz es de poca ayuda, ya que no aporta ningún código “a heredar” para la clase que la implemente, solo le aporta obligaciones o compromisos. En realidad no es así, las interfaces, como su nombre indica, se usan como intermediarias entre dos clases, ocultando la implementación de una a la otra, haciendo que el código sea menos dependiente (se dice menos *acoplado*) y más fácilmente portable y testeable. En el ejemplo del principio de este apartado, cuando hacemos:

```
Avion miAvion = new AvionComercial();
```

Luego usaremos los métodos de Avion, ya que la variable referencia a ese tipo, aunque haya sido instanciado con un AvionComercial.

```
miAvion.terrizar();
```

sin preocuparnos del código implementado en dicho método. Si en un futuro, cambiamos la instancia por otra clase que implemente la interfaz:

```
Avion miAvion = new Avioneta();
```

la llamada al método anterior seguiría funcionando correctamente.

```
miAvion.terrizar();
```

Sin interfaces, *Avioneta* podría haber llamado a ese método de otra forma: *aterrizo()*, *landing()*, etc... lo que haría que el cambio de *AvionComercial* a *Avioneta* implicaría cambios en su utilización.

Clases Abstractas vs. Interfaces

Existen varias diferencias entre una clase abstracta y una interfaz:

1. Una clase abstracta puede heredar de una sola clase (abstracta o no) mientras que una interfaz puede heredar de varias interfaces a la vez.
2. Una clase abstracta puede tener métodos que sean abstractos y otros que no lo sean, mientras que las interfaces sólo podían definir métodos abstractos. *Desde Java 8 interfaces tienen métodos estáticos, privados y por defecto, que incumplen esa afirmación.*
3. En una clase abstracta es obligatoria la palabra *abstract* para definir un método abstracto (así como la clase). En interfaces, esta palabra es opcional ya que se infiere en el concepto de interfaz.
4. En una clase abstracta, los métodos abstractos pueden ser *publico* o *protected*. En una interfaz solamente puede haber métodos públicos.

5. En una clase abstracta pueden existir variables de instancia y variables *static*, y ambas con cualquier modificador de acceso (*public*, *private*, *protected* o *default*). En una interfaz sólo puedes tener constantes (*public static final*).
6. Una clase abstracta puede heredar de cualquier clase (independientemente de que esta sea abstracta o no) o implementar interfaces, mientras que una interfaz solamente puede heredar de otras interfaces.

Una pregunta muy frecuente es **¿Qué debo usar interfaces o clases abstractas?**

No hay una respuesta válida para todos los casos. Se usan clases abstractas cuando no necesitamos herencia múltiple y cuando hay mucha compartición de código entre superclase y subclases. Las interfaces van más orientadas a ese “compromiso” o normas de cómo se comportarán las clases subyacentes, no centrándose tanto en compartir código.

Herencia Múltiple mediante Interfaces

A veces se dice que Java permite herencia múltiple (es decir, tener varias superclases o varias clases base de las que heredar atributos/métodos) mediante las interfaces. Esto es cierto, pero con matices.

Hemos visto que las interfaces tienen herencia múltiple, es decir, una clase puede implementar distintas interfaces (*implements*), mientras que solo puede heredar de una (*extends*). Pero hay dos diferencias fundamentales entre lo que se entiende por herencia múltiple de clases y la herencia que proporcionan las interfaces:

- Desde una interfaz, una clase solo “hereda” constantes, métodos estáticos, pero no puede heredar ni atributos no estáticos ni implementaciones de métodos, porque las interfaces no disponen de ellos. (*)
- La jerarquía de interfaces es independiente de la jerarquía de clases, pudiendo varias clases implementar la misma interfaz y no pertenecer a la misma jerarquía de clases. Cuando se habla de herencia múltiple, las clases pertenecen a la misma jerarquía.

(*) *Desde Java8, en la definición de interfaces se permiten métodos “default” en los que podemos hacer una implementación por defecto de los métodos para aquellas clases que implementen la interfaz, pero no desarrollen el método en cuestión. Aunque la primera intención de los métodos por defecto no era esta, es en cierta forma una forma de implementar la herencia múltiple, ya que este código si se heredaría. De todas formas, no resolvería el problema de la herencia de atributos.*

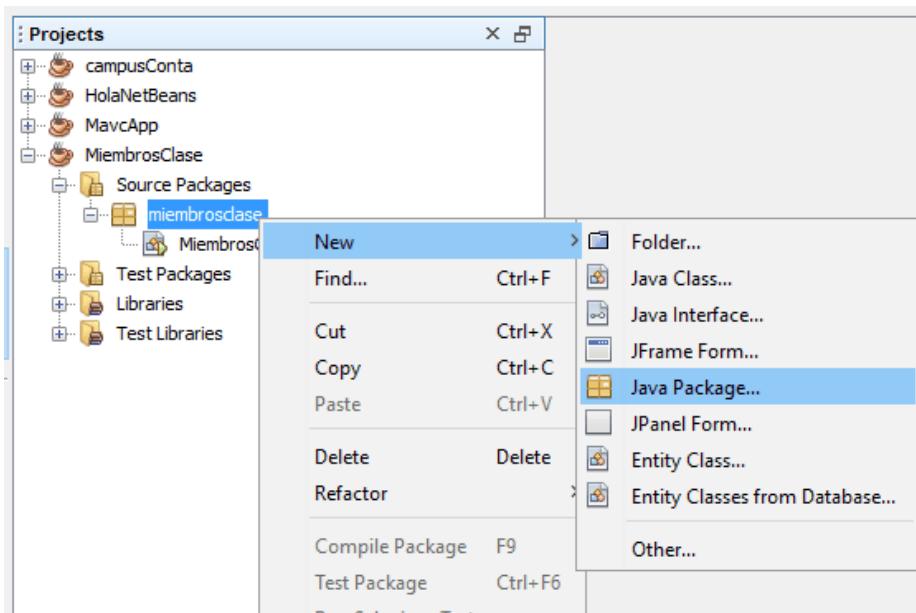
11. Paquetes

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

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

Creación de Paquetes

Un paquete Java se genera incluyendo la palabra clave *package* al inicio de los módulos de código en los que se definen las clases que formarán parte del mismo. Trabajando en un proyecto con NetBeans, comprobaremos que en la ventana *Projects* los paquetes se representan con un ícono específico y actúan como nodos contenedores, alojando los módulos .java con el código fuente. El menú contextual del proyecto nos ofrece la opción *New>Java Package*, que será el que usemos habitualmente para crear un nuevo paquete.



Cada vez que se crea un nuevo proyecto con NetBeans se propone la definición de un nuevo paquete, cuyo nombre sería el mismo del proyecto, donde se alojarían los módulos de código. En proyectos complejos, no obstante, puede ser necesaria la creación de paquetes adicionales.

Un paquete puede contener, además de definiciones de tipos como las clases e interfaces, otros paquetes, dando lugar a estructuras jerárquicas de contenedores. La denominación de los “subpaquetes”, paquetes contenidos en otros, se compondrán del identificador del contenedor seguido de un punto y el nombre del subpaquete. De existir niveles adicionales se agregarían los distintos identificadores separados por puntos, formando así el nombre completo del paquete.

Así pues, cuando hacemos `import java.time.format.*;` nos referimos al paquete `format` que es un subpaquete del paquete `time`.

Notas:

- El concepto de "subpaquete" se aplica solo a nivel organizativo de clases, pero para Java no hay ninguna relación especial entre paquete y subpaquete, es como si fueran dos paquetes totalmente independientes (a efectos de permisos, acceso...)
- Comentar finalmente que, si no se indica ningún paquete, las clases se crean en un paquete por defecto, aunque esta situación no es recomendable.

Import

Como ya hemos visto, con `import` incorporamos al archivo actual las definiciones de otro paquete para poder usarlas según el procedimiento habitual sin necesidad de especificar luego en el código, el paquete origen del elemento.

La cláusula `import` puede utilizarse para importar un elemento concreto de un paquete, facilitando el nombre de este seguido de un punto y el identificador de dicho elemento. Por ejemplo, para importar la clase `Math` del paquete `java.lang`, pudiendo así acceder a la constante `PI` y las funciones matemáticas que aporta, bastaría con la siguiente línea `import java.lang.Math;`

Es habitual que al importar un paquete nos interesen muchas de las clases definidas en el mismo. En este caso podríamos importarlas individualmente, usando la sintaxis anterior, o bien podríamos recurrir a la siguiente alternativa. Esto nos permitiría usar la clase `Math`, así como la clase `System`, la clase `Thread` y muchas otras definidas en el paquete `java.lang`: `import java.lang.*;`

En concreto '`java.lang`' se importa por defecto, por lo que no es necesario importarla.

En ocasiones, como ocurre con la clase `Math`, importamos una clase para acceder directamente a sus miembros estáticos (constantes y métodos), no para crear instancias a partir de las clases del paquete, podemos recurrir a la sintaxis `import static paquete.clase.*;`; cuya finalidad es incluir en el ámbito actual los miembros estáticos de la clase indicada y no sería necesario emplear el prefijo de la clase en nuestro código, como se muestra en la figura.

```

pruebaImport.java
Source History ▾
1 package prueba;
2 import java.lang.Math.*;
3 public class pruebaImport {
4     public static void main(String[] args) {
5         double area = 2 * 10 * Math.PI;
6     }
7 }
8

pruebaImportStatic.java
Source History ▾
1 package prueba;
2 import static java.lang.Math.*;
3 public class pruebaImportStatic {
4     public static void main(String[] args) {
5         double area = 2 * 10 * PI;
6     }
7 }
8

```

Paquetes y sistema de archivos

En Java existe una estrecha relación entre las definiciones de paquetes y clases y el sistema de archivos local, en el que se almacenan los módulos conteniendo el código. Es un hecho que puede pasarnos totalmente inadvertido al trabajar con un IDE como el de NetBeans, ya que sus opciones se encargan de ocultar estos detalles.

Cada archivo de código de un proyecto Java únicamente puede contener una clase pública, cuyo nombre, respetando mayúsculas y minúsculas, debe coincidir con el del archivo. Es decir, existe una correspondencia directa entre los identificadores de clase y archivos en los que se alojan.

El nombre del paquete establece además el nombre de la carpeta donde se alojan los módulos de código contenidos en el paquete. Los nombres de paquete pueden ser compuestos, usándose el punto como separador de las diferentes porciones del identificador. Cada parte entre puntos identificaría a una subcarpeta en la jerarquía.

Visibilidad de las Clases en Paquetes

Como ya comentamos previamente, existen 3 modificadores de acceso, además del "default" y esta es su visibilidad, en relación a los paquetes.

Relación	Private	Default	Protected	Public
Visible dentro de la misma clase	Sí	Sí	Sí	Sí
Visible dentro del mismo paquete por subclase	No	Sí	Sí	Sí
Visible dentro del mismo paquete por no-subclase	No	Sí	Sí	Sí
Visible dentro de diferente paquete por subclase	No	No	Sí	Sí
Visible dentro de diferente paquete por no-subclase	No	No	No	Sí

Así pues, podemos decir que cualquier elemento, salvo que sea *private*, es accesible desde cualquier miembro de su mismo paquete.

Por otra parte, si hablamos de distintos paquetes, solo los elementos *public* serán accesibles (y también los *protected* si hablamos subclases).

Hay que tener en cuenta que la jerarquía de paquetes no afecta para la visibilidad de sus clases, a todos los efectos, un paquete y un subpaquete del mismo son dos paquetes totalmente independientes.

Estamos hablando de elementos, y en ese término estamos agrupando clases, interfaces, atributos y métodos. Hay que recordar que una clase o una interface solo puede ser *publico default*.

Las clases que públicas y default de otros paquetes de nuestro proyecto podemos usarlas directamente sin el prefijo del paquete siempre que hagamos su *import*. Si no hacemos *import*, podemos usarlas igualmente, pero hay que poner el nombre con el prefijo del paquete al que pertenece.

12. Interfaz Gráfica de Usuario

Introducción

Java permite realizar aplicaciones de escritorio en entorno gráfico. Al contrario que en los programas que hemos desarrollado hasta ahora, los programas con interfaz gráfica seguirán un flujo guiado por los eventos que ocurran, generalmente ocasionados voluntariamente con las acciones del usuario, por ejemplo, cuando se pulse un botón. Para ello utilizaremos Swing, que es una herramienta de interfaz gráfica de usuario (GUI) ligera que incluye un amplio conjunto de widgets (botones, cuadros de texto, menús, etc) para aplicaciones Java, y es independiente de la plataforma.

Podemos crear cada nuestra aplicación gráfica mediante código, pero es más cómodo crearlo a partir de las utilidades de un IDE como Netbeans o Eclipse. Usaremos ambos métodos, pero, cuando creamos el interfaz gráfico desde el IDE, comprobaremos el código que genera.

Programación Guiada por Eventos

¿Qué es un evento?

Es todo hecho que ocurre mientras se ejecuta la aplicación. Normalmente, llamamos evento a cualquier interacción que realiza el usuario con la aplicación, como puede ser:

- Pulsar un botón con el ratón,
- Hacer doble clic,
- Pulsar y arrastrar,
- Pulsar una combinación de teclas en el teclado,
- Pasar el ratón por encima de un componente,
- Salir el puntero de ratón de un componente,
- Abrir una ventana, etc.

¿Qué es la programación guiada por eventos?

Imagina la ventana de cualquier aplicación, por ejemplo, la de un procesador de textos. En esa ventana aparecen multitud de elementos gráficos interactivos, de forma que no es posible que el programador haya previsto todas las posibles entradas que se pueden producir por parte del usuario en cada momento.

Con el control de flujo de programa de la **programación imperativa**, el programador tendría que estar continuamente leyendo las entradas (de teclado, o ratón, etc) y comprobar para cada entrada o interacción producida por el usuario, de cual se trata de entre todas las posibles, usando estructuras de flujo condicional para ejecutar el código conveniente en cada caso. Para cada opción del menú, para cada botón o etiqueta, para cada lista desplegable, y por tanto para cada componente de la ventana, incluyendo la propia ventana, habría que comprobar todos y cada uno de los eventos posibles, por lo que las posibilidades son casi infinitas, y desde luego impredecibles. Por tanto, de ese modo es imposible solucionar el problema.

Para abordar el problema de tratar correctamente las interacciones del usuario con la interfaz gráfica de la aplicación hay que cambiar de estrategia, y la **programación guiada por eventos** es una buena solución.

El proceso sería algo así: cada vez que el usuario realiza una determinada acción sobre una aplicación que estamos programando en Java: un clic sobre el ratón, presionar una tecla, etc, se produce un evento que el sistema operativo transmite a Java, creando un objeto de una determinada clase de evento, y este evento se transmite a un determinado método para que lo gestione. Ejemplos de fuentes de eventos pueden ser:

- Botón sobre el que se pulsa o pincha con el ratón.
- Campo de texto que pierde el foco.
- Campo de texto sobre el que se presiona una tecla.
- Ventana que se cierra.
- Etc.

Un ejemplo típico será la selección de un elemento, bien haciendo click con el ratón sobre él o pulsando *[Enter]* cuando tiene el foco. Suponiendo un botón llamado *jButton1* sería así:

```
jButton1.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        TareaAlPulsarBoton(evt);
    }
});

private void TareaAlPulsarBoton (java.awt.event.ActionEvent evt) {
    // Aquí nuestro código que se ejecuta al pulsar el botón
}
```

A nosotros lo que nos interesa es lo que está resaltado con fondo amarillo, el resto será generado por el IDE. El parámetro del método 'evt' nos proporciona mucha información útil sobre el evento, como se verá más adelante.

Otro evento típico será pulsar una tecla:

```
addKeyListener(new java.awt.event.KeyAdapter() {
    public void keyPressed(java.awt.event.KeyEvent evt) {
        formKeyPressed(evt);
    }
});

private void formKeyPressed(java.awt.event.KeyEvent evt) {
    // Aquí el código:
    switch (evt.getExtendedKeyCode()) {
        case KeyEvent.VK_UP: /* flecha arriba*/ ;break;
        case KeyEvent.VK_DOWN: /* flecha abajo*/ ;break;
    }
}
```

Ojo!! el *addKeyListener* no lo aplicamos a un elemento, iría a toda la ventana (o al contenedor)

Creando Una Aplicación Gráfica

Antes de ver el detalle de todos los elementos que podemos incluir en nuestra aplicación gráfica y ver los eventos que se pueden producir, vamos a ver como crearíamos una aplicación gráfica sencilla, primero utilizando el asistente de Netbeans y luego codificando los componentes desde cero.

Aplicación Gráfica con el asistente de NetBeans

Pasos a seguir:

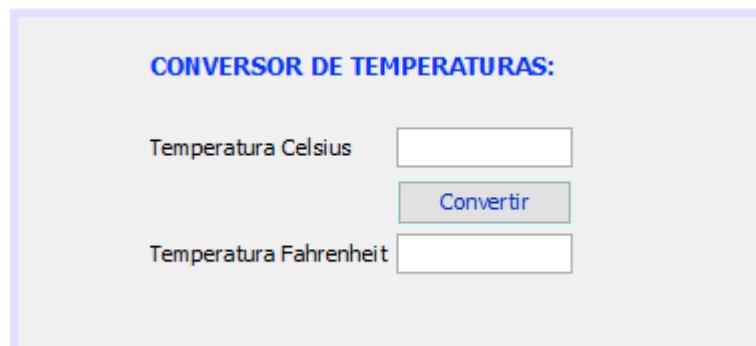
1. Crear un proyecto nuevo, por ejemplo, de tipo Maven.
2. Crear un objeto contenedor, que incluirá el resto de componentes, en nuestro caso, sobre el proyecto, botón derecho > **New > JFrame** (otros contenedores son: *JPanel*, *JApplet*, *JDialog*, *JInternalFrame*). Generará una clase *.java*, de la que podremos ver tanto el código fuente que la describe como su diseño.
3. En la ventana de proyectos, sobre el nombre del proyecto: botón derecho > *Propiedades* > *Run > Main Class* y seleccionamos el objeto creado en el punto anterior.

También se podría hacer que nuestro proyecto sí tuviese una clase principal independiente, como en los proyectos sin interface gráfica. Esta clase tendría una única misión, que sería visualizar el *JFrame*. En tal caso, el *main()* de esa clase principal tendría un código similar al método *main()* del *JFrame* generado por Netbeans.

Sobre el *JFrame* creado, ir a la pestaña *Diseño*. Veremos la nueva ventana, vacía por el momento. Este sería el código generado:

```
package ejemploSwing;
public class NewJFrame extends javax.swing.JFrame {
    public NewJFrame() {
        initComponents();
    }
    @SuppressWarnings("unchecked")
    // Generated Code
    public static void main(String args[]) {
        Look and feel setting code (optional)
        /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new NewJFrame().setVisible(true);
            }
        });
    }
    // Variables declaration - do not modify
    // End of variables declaration
}
```

4. Ir a la ventana *Paleta* (generalmente ubicada a la derecha) y arrastrar los componentes que deseemos que formen esta ventana inicial de nuestra aplicación:



En la imagen anterior podemos ver *etiquetas* que simplemente muestran un determinado texto, dos cajas de texto cuya función es que el usuario introduzca valores (en el primer caso) o para mostrar el resultado de los cálculos realizados (en el segundo caso). También tiene un botón, el cual, al ser pulsado, realizará los cálculos solicitados.

5. Clicando sobre cada elemento, podremos cambiar sus propiedades en la ventana de *Propiedades*. Unos de los cambios habituales será la propiedad que muestra en pantalla (generalmente *text*) aunque cada elemento tendrá sus particularidades:

- *Texto* para botones, etiquetas, etc.
- *Título* de la ventana para los *JFrame*.
- Grupo de botones para agrupar los *radiobutton*.
- *Horizontal resizable* y *Vertical resizable* para indicar si permitimos que el tamaño de los componentes se adapte al tamaño de sus contenedores.
- Etc.

Podremos cambiarle el nombre de la variable en la pestaña *Código*, o botón derecho sobre el mismo y *Change variable name* para darle un nombre más significativo.

Netbeans genera nuevo código que añade a la ventana, para los nuevos elementos y sus propiedades:

```
private void initComponents() {

    jLabel1 = new javax.swing.JLabel();
    jLabel2 = new javax.swing.JLabel();
    jTextField1 = new javax.swing.JTextField();
    jTextField2 = new javax.swing.JTextField();
    jButton1 = new javax.swing.JButton();
    jLabel3 = new javax.swing.JLabel();

    setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

    jLabel1.setText("Temperatura Celsius");

    jLabel2.setText("Temperatura Fahrenheit");

    jTextField1.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jTextField1ActionPerformed(evt);
        }
    });

    jTextField2.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jTextField2ActionPerformed(evt);
        }
    });

    // Variables declaration - do not modify
    private javax.swing.JButton jButton1;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JTextField jTextField1;
    private javax.swing.JTextField jTextField2;
    // End of variables declaration
}
```

6. El último paso será la generación de las acciones que queramos que realice ante los distintos eventos, eso lo haremos con botón derecho sobre cada elemento > *Events* > *Action* > ... En nuestro caso, lo haremos sobre el botón *[Convertir]*, concretamente *Events* > *Action* > *Action performed* (que se corresponde con pulsar el botón) e introduciremos el siguiente código:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    double celsius= Double.parseDouble(jTextField1.getText());
    double fahrenheit = (int) Math.round((celsius *9/5 +32)*100)/100d;
    jTextField2.setText(Double.toString(fahrenheit));
}
```

Es fácil de entender lo que hace este código: obtiene lo que haya en la primera caja de texto (*jTextField1*), que será de tipo String y lo convierte a número. Luego hace los cálculos necesarios e introduce el resultado en la segunda caja de texto (*jTextField2*). Veremos un poco más adelante los métodos interesantes de cada objeto gráfico.

Trataremos de separar, en la medida de los posible, la parte gráfica de la lógica de nuestro programa, de forma que las acciones desencadenadas desde los eventos sean llamadas a métodos y funciones situadas en otras clases. Así lograremos mayor portabilidad, por ejemplo, podremos hacer una aplicación web o para otros dispositivos, y la lógica del programa podremos reutilizarla completamente, solo cambiando el interfaz gráfico.

Si hubiésemos seguido esta filosofía en el ejemplo anterior, podríamos haber creado una clase llamada *Temperatura* con un método estático que pasase de Celsius a Fahrenheit y del evento quedaría algo así:

```
double celsius= Double.parseDouble(jTextField1.getText());
double fahrenheit = Temperatura.convertirCelsiusFar(celsius);
jTextField2.setText(Double.toString(fahrenheit));
```

7. Con esto ya podemos ejecutar nuestra aplicación. Al principio se ejecutará el código que está en el constructor de la ventana (JFrame) y por defecto es solo el método *initComponents()* aunque podríamos añadir más operaciones. Luego, al pulsar el botón se ejecutará el código añadido en el paso anterior.

Por último, comentar que Netbeans genera archivos XML con extensión *.form* para almacenar información sobre la parte gráfica de cada JFrame. No es necesario distribuirlos con la aplicación ya que solo los emplea el IDE.

Aplicación Gráfica codificada desde cero

Vamos a ver como se crearía una aplicación gráfica con Swing sin ningún asistente, programada "a mano".

1.- Primero crearemos una clase de tipo ventana (hija de JFrame) que contendrá distintos elementos sobre los que luego programaremos acciones en los diferentes eventos.

```
import java.awt.EventQueue;
import javax.swing.*;
import java.awt.event.*;

public class EjemploJFrame extends JFrame implements ActionListener {
```

2.- Dentro de la clase declararíamos las variables de los elementos presentes en la ventana:

```
private JLabel etiq;
private JTextField campoTexto;
private JButton btnPulsame;
private JPanel PanelContenido;
```

Y el *main()* que lanza la aplicación, en una versión sencilla:

```
public static void main(String[] args) {
    EjemploJFrame frame = new EjemploJFrame();
    frame.setVisible(true);}
```

O bien de forma más sofisticada:

```
public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                EjemploJFrame frame = new EjemploJFrame(); frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}
```

3.- Dentro de la clase *EjemploJFrame*, con el constructor de la ventana, en la que se definen ciertas características de la ventana, se definen también los componentes de la misma (etiquetas, botones, etc.) y se añaden a la ventana.

Realmente los elementos se añaden a un Panel, que es un contenedor, un elemento intermedio entre ellos y el *JFrame*.

```
public EjemploJFrame() {
    setTitle("Titulo de la ventana"); // Título opcional
    setBounds(400, 200, 655, 520); // Cordenadas 'x' 'y', altura, long
    setDefaultCloseOperation(EXIT_ON_CLOSE); // Al cerrar la ventana finaliza
    setVisible(true);
    PanelContenido = new JPanel(); //Creamos el panel
    PanelContenido.setLayout(null); //Indicamos su diseño
    setLocationRelativeTo(null); //Centrada en pantalla
    setContentPane(PanelContenido); //asigno el pannel a la ventana

    //Componentes
    etiq = new JLabel("Texto de la etiqueta");
    etiq.setBounds(369, 32, 229, 25);
    PanelContenido.add(etiq);

    campoTexto = new JTextField();
    campoTexto.setBounds(371, 68, 193, 26);
    PanelContenido.add(campoTexto);

    JButton btnPulsame = new JButton("Pulsame");
    btnPulsame.setBounds(43, 70, 89, 23);
    PanelContenido.add(btnPulsame);
    //Ahora definiríamos los eventos y sus acciones asociadas
} //fin constructor
```

Componentes Swing

Swing ofrece dos tipos de elementos clave: *componente* y *contenedor*. Sin embargo, esta distinción es principalmente conceptual debido a que todos los contenedores también son jerárquicamente componentes. La diferencia entre los dos se encuentra en su propósito: como el término se emplea comúnmente, un *componente* es un control visual independiente, como un botón o un deslizador. Un *contenedor* aúna a un grupo de componentes. Por ello, un contenedor es un tipo especial de componente que está diseñado para poseer a otros componentes.

Además, para que un componente sea desplegado debe ser colocado en un contenedor. Por ello, todas las interfaces gráficas hechas con Swing contienen al menos un contenedor por defecto. Debido a que los contenedores son componentes, un contenedor puede también poseer a otros contenedores. Esto le permite a Swing definir lo que se denomina una *contención jerárquica*, en cuyo nivel más alto se encuentra lo que se denomina un *contenedor raíz*.

Componentes

En general los componentes de Swing se derivan de la clase JComponent. Las únicas excepciones a esto son los cuatro contenedores raíz que se describen en la siguiente sección. Todos los componentes de Swing están representados por clases definidas en el paquete javax.swing. Observe que todas las clases comienzan con la letra J. Por ejemplo, la clase para crear una etiqueta es JLabel; la clase para un botón es JButton; y la clase para un scrollbar es JScrollPane.

Contenedores

Swing define dos tipos de contenedores. El primero son los contenedores raíz: JFrame, JApplet, JWindow y JDialog. A diferencia de los otros componentes de Swing, los contenedores raíz son componentes pesados y son los que interactúan con el sistema operativo. Como el nombre lo indica un contenedor raíz debe estar en lo alto de una jerarquía de contención y no está contenido ningún otro componente. Además, todas las jerarquías de contención deben comenzar con un contenedor raíz. El contenedor utilizado más comúnmente en aplicaciones es JFrame. El contenedor utilizado para applets es JApplet.

El segundo tipo de contenedores soportados por Swing son los contenedores ligeros. Los contenedores ligeros heredan de la clase JComponent. Por ejemplo, JPanel, el cual es un contenedor de propósito general. Los contenedores ligeros se utilizan a menudo para organizar y administrar grupos de componentes relacionados debido a que un contenedor ligero puede ser contenido por otros contenedores. Así, el programador puede utilizar contenedores ligeros como JPanel para crear subgrupos de controles relacionados que están contenidos en un contenedor exterior.

Ventana de Aplicación

JFrame: Es el contenedor de alto nivel más empleado, tiene las funcionalidades típicas de una ventana (maximizar, cerrar, título, etc.). Algunos métodos importantes son:

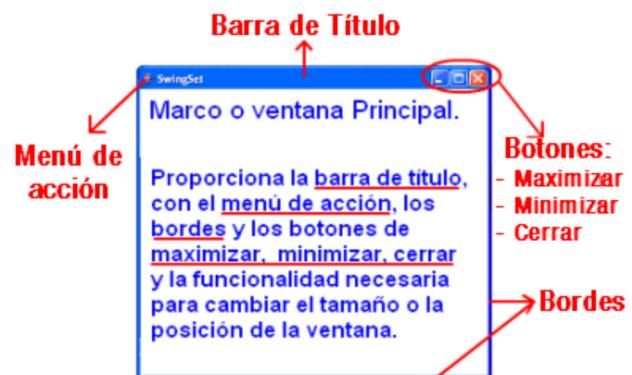
- `setSize (w,h);` asigna las dimensiones de la ventana (ancho y alto).
- `setTitle (str), getTitle ();`
- `setVisible (bool);`
- `setDefaultCloseOperation (constante);` controla la acción al pulsar la "X" de cierre de ventana. Su valor habitual suele ser JFrame.EXIT_ON_CLOSE.
- `setLocationRelativeTo(null);` centra la ventana en la pantalla.

Una vez creado el objeto de ventana, hay que establecer su tamaño, establecer la acción de cierre y hacerla visible.

```
import javax.swing.*;
public class VentanaTest {
    public static void main(String[] args) {
        JFrame f = new JFrame("Titulo de ventana");
        f.setSize(400, 300);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
}
```

Acciones de cierre:

- `JFrame.EXIT_ON_CLOSE`: Abandona aplicación.
- `JFrame.DISPOSE_ON_CLOSE`: Libera los recursos asociados a la ventana.
- `JFrame.DO NOTHING_ON_CLOSE`: No hace nada.
- `JFrame.HIDE_ON_CLOSE`: Cierra la ventana, sin liberar sus recursos

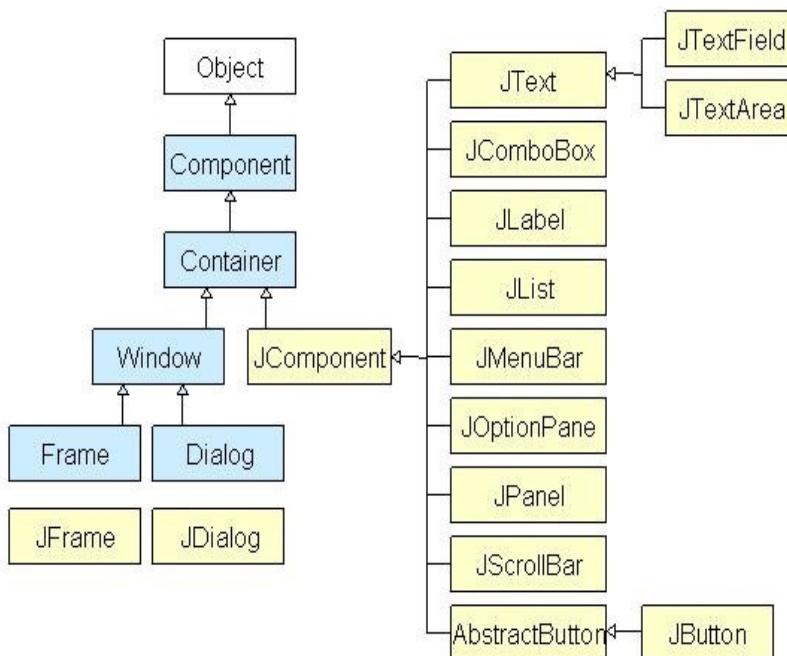


Por último, para cerrar una ventana liberando todos sus recursos, emplearemos el método `dispose()`. Si tenemos un botón de salir en nuestra ventana, su código podría ser: `this.dispose()`.

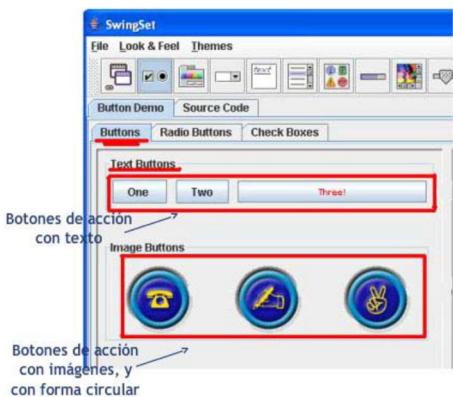
Componentes básicos

Vamos a introducir aquí los componentes habituales de una ventana de aplicación, con su descripción, apariencia y métodos interesantes. Estos componentes se añadirán a la ventana o, mejor dicho, a un panel de la ventana. Si no hay ningún panel, se añaden al panel por defecto.

Esta es la jerarquía de componentes de Swing.



JButton: Botón de acción. Para realizar alguna acción o proceso.



El texto que muestra se establece con el método `setText()`.

Ojo: Si en Netbeans, modificamos el texto directamente en el botón y no mediante la propiedad Text de la Paleta, utiliza el método `setLabel()` en vez de `setText()` y es un método obsoleto por lo que a la hora de compilar nos informará de un aviso.

Antes vimos que `ActionPerformed` era el evento típico de seleccionar/pulsar el botón.

```
jButton1.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        // Código que se ejecuta al pulsar el botón
    }
});
```

Si deseamos detectar el evento de clicar con el botón derecho, este sería el evento:

```
jButton1.addMouseListener(new java.awt.event.MouseAdapter() {
    public void mouseClicked(java.awt.event.MouseEvent evt) {
        TareaAlPulsarRaton(evt);}});

private void TareaAlPulsarRaton (java.awt.event.MouseEvent evt) {
    if (((java.awt.event.MouseEvent) evt).getButton() ==
        java.awt.event.MouseEvent.BUTTON3 ){
        // Aquí el código:
    }
}
```

JCheckBox: Casilla de verificación. Se usa normalmente para indicar opciones que son independientes, pudiéndose seleccionar algunas, todas o ninguna.

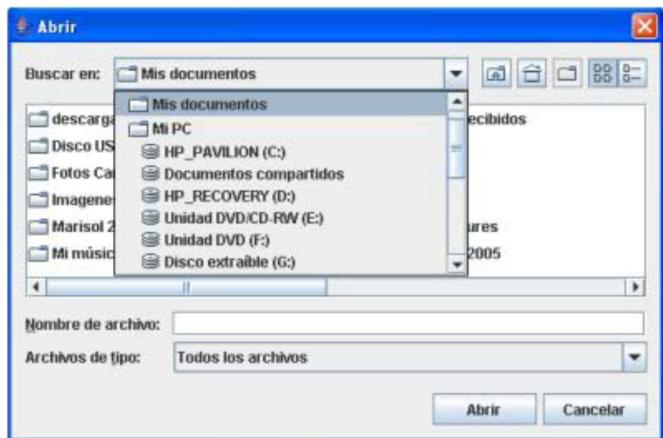


El método que más nos interesa de esta clase es: `isSelected()` que devuelve true si el check-box está seleccionado y false en caso contrario.

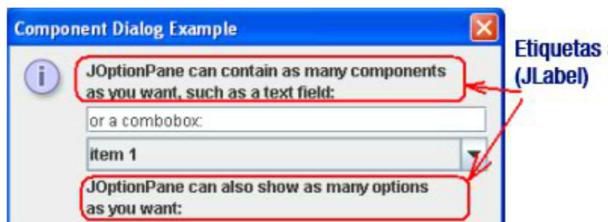
JColorChooser: Panel de controles que permite al usuario seleccionar un color.



JFileChooser: Permite al usuario elegir un archivo para la lectura o escritura, mostrando una ventana que nos permite navegar por los distintos discos y carpetas de nuestro ordenador



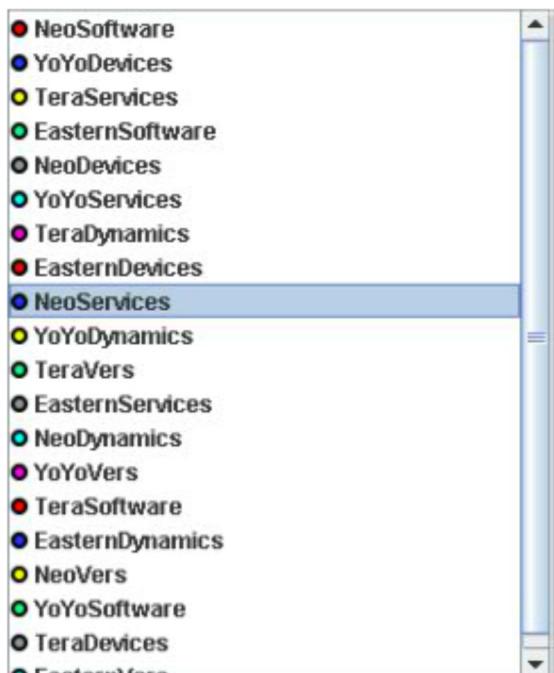
JLabel: Una etiqueta que puede contener un texto corto, o una imagen, o ambas cosas. En general suelen ser estáticas, de forma que desde el asistente de Netbeans le asignaremos un valor y ya no lo modificaremos más, pero siempre podremos emplear los métodos `setText()` y `getText()` para modificar y obtener el texto mostrado, respectivamente.



```
jLabel1.setText ("Temperatura Celsius");
```

Se suele utilizar esta etiqueta para incorporar imágenes (veremos más adelante cómo hacerlo).

JList: Un cuadro de lista para seleccionar uno o varios elementos de una lista



Los valores contenidos en la lista se gestionan desde una colección (similar a un array) llamada `model` y que dispone de métodos para añadir / modificar / eliminar elementos y otras operaciones.

Por ejemplo, podemos definir en nuestro *frame*, como atributo global:

```
DefaultListModel listamodelo;
```

Al inicializar los componentes:

```
listamodelo = new DefaultListModel();
jList1.setModel (listamodelo);
```

y luego podemos emplear los métodos siguientes para trabajar con la lista:

```
listamodelo.addElement(str);           //añadir a la lista, generalmente String
listamodelo.clear();
listamodelo.removeElementAt(position);
listamodelo.setElementAt(string, position);
listamodelo.getElementAt(posistion);
listamodelo.size();
```

Para obtener el índice o valor de la lista seleccionado por el usuario, haremos respectivamente:

```
int index = jList1.getSelectedIndex();
String s = (String) jList1.getSelectedValue();
```

Si no hay ningún elemento seleccionado, *getSelectelIndex* devuelve -1.

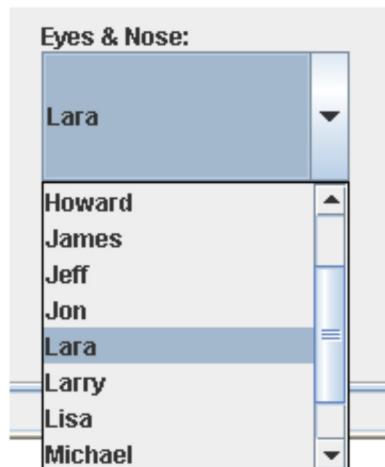
En cuanto a los eventos, nos interesa detectar cuando el usuario selecciona un valor, esto lo hacemos con *jListValueChanged ()*. Por ejemplo:

```
private void jList1ValueChanged(javax.swing.event.ListSelectionEvent evt) {
    String s = (String) jList1.getSelectedValue();
}
```

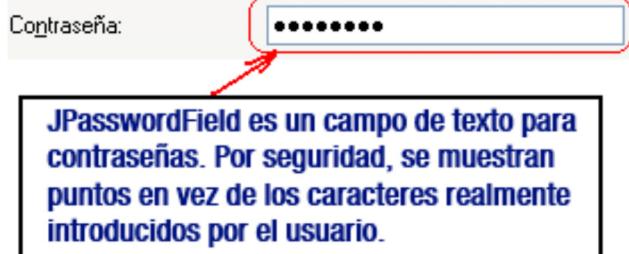
JComboBox: Lista desplegable con características similar a *JList*, solo cambia su apariencia. El *model* en este caso es de tipo: *DefaultComboBoxModel*.

```
listamodelo = new DefaultComboBoxModel();
jComboBox1.setModel (listamodelo);
listamodelo.addElement(...)
```

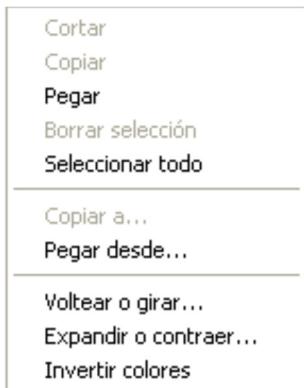
DefaultComboBoxModel tiene otro constructor al que se le puede pasar como parámetro un array de elementos. En ese caso inicializa la combo con dichos elementos, evitándonos tener que hacer *addElement* para cada uno.



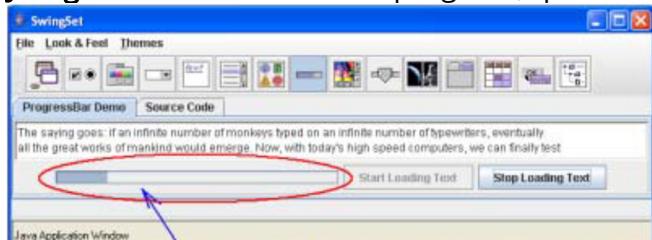
JPasswordField : Un cuadro de texto donde no se muestran los caracteres que realmente se introducen, sino algún otro (puntos o asteriscos). Se usa para introducir contraseñas de forma que otras personas no puedan ver el valor introducido



JPopupMenu: Un menú emergente. La imagen del ejemplo está sacada de una aplicación de dibujo



JProgressBar: Una barra de progreso, que visualiza gráficamente un valor entero en un intervalo

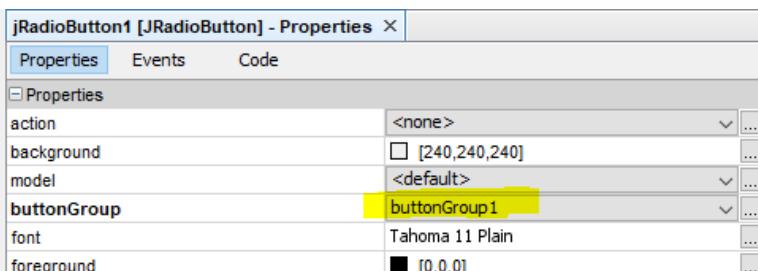


Barra de progreso (JProgressBar). Se asocia a una tarea para mostrar de una forma gráfica la parte de tarea que ya se ha realizado y la que queda pendiente. Al mismo tiempo sirve para que el usuario compruebe que la tarea progresá, que el ordenador no se ha quedado bloqueado. En este ejemplo se asocia a la carga del texto en el editor.

JRadioButton: Crea un botón de radio. Se usa normalmente para seleccionar una opción de entre varias excluyentes entre sí



Los botones de radio deben estar agrupados para que, cuando se seleccione uno, el resto del grupo estén seleccionados (es lo que lo diferencia de un check button). Para ello, necesitamos añadir un nuevo componente de tipo *buttonGroup* (*a nivel gráfico es invisible*). Luego a cada *radioButton*, le asignaremos el *buttonGroup* creado en la propiedad *buttonGroup*:



También se podría hacer por código, en el constructor de la ventana, después de *initComponents()*:

```
ButtonGroup bg=new ButtonGroup();
bg.add (radiobutton1);
bg.add (radiobutton2);
bg.add (radiobutton3);
```

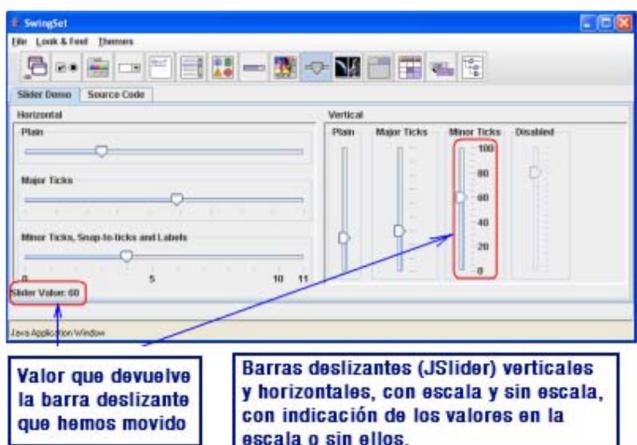
Como en el caso de los check-buttons, el método `isSelected()` nos informará si un botón está seleccionado o no.

```
if (!jRadioButton1.isSelected() && !jRadioButton2.isSelected()) {
    JOptionPane.showMessageDialog(null, "Seleccione una opción");
    return;
}
```

Al igual que con JButton, el evento que ocurre cuando el usuario selecciona un botón de radio es:

ActionPerformed.

JSlider : Barra deslizante. Componente que permite seleccionar un valor en un intervalo deslizando un botón



Para obtener el valor seleccionado tenemos el método `getValue()` y el evento para detectar cambios en un Slider sería:

```
jSlider1.addChangeListener(new javax.swing.event.ChangeListener() {
    public void stateChanged(javax.swing.eventChangeEvent evt) {
        // Aquí el código
    }
});
```

JSpinner : Un cuadro de entrada de una línea que permite seleccionar un número (o el valor de un objeto) de una secuencia ordenada. Normalmente proporciona un par de botones de flecha, para pasar al siguiente o anterior número, (o al siguiente o anterior valor de la secuencia).

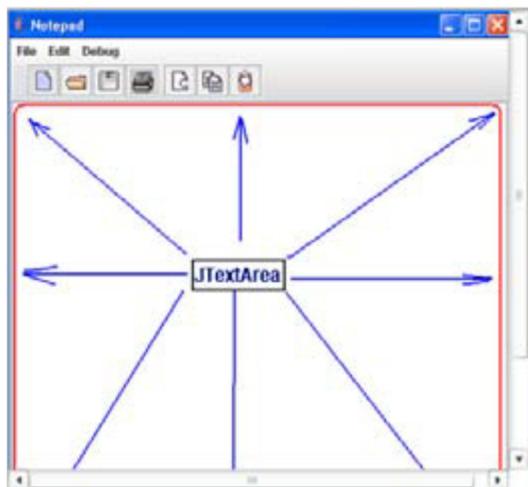
En NetBeans, mediante la propiedad `model` podemos controlar sus valores máximos y mínimos.



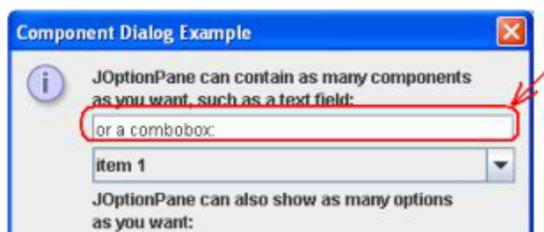
JSpinner permite seleccionar un valor de una secuencia ordenada de valores posibles. Normalmente se usa con datos numéricos y las flechas permiten seleccionar el anterior (abajo) o siguiente número o valor de la secuencia. También puede escribirse el valor directamente.

JTable: Una tabla bidimensional para presentar datos

JTextArea : Crea un área de texto de dos dimensiones para texto plano



JTextField : Crea un cuadro de texto de una dimensión.

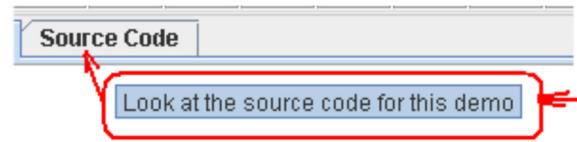


Cuadro de texto (JTextField)

Los dos métodos que más nos interesan son el que obtienen el texto que hay escrito en el componente: `getText()` y el que escribe texto en el componente: `setText()`.

El valor devuelto por `getText()`; es de tipo String por lo que, si queremos hacer operaciones numéricas el mismo, debemos convertirlo con los métodos estáticos que ya conocemos, como `Integer.parseInt()`.

JToolTip : Una especie de etiqueta emergente, que aparece para visualizar la utilidad de un componente cuando el cursor del ratón pasa por encima de él deteniéndose brevemente, sin necesidad de pulsar nada.



Texto informativo emergente asociado a un elemento, que se visualiza al pasar y parar el ratón por encima del elemento.

JTree : Visualiza un conjunto de datos jerárquicos en forma de árbol



Métodos comunes

Existen una serie de métodos comunes a los elementos que acabamos de ver, de uso frecuente:

- `setBounds (x, y, w, h);` Este es común a todos los componentes e indica la posición y el tamaño del “rectángulo” que forma la apariencia del componente. Los primeros dos parámetros son las posiciones X e Y de la esquina superior izquierda (es la posición respecto a su contenedor). El tercer parámetro es el ancho del elemento y el cuarto el alto del elemento.
- `setLocation (x, y);` y `setSize (w, h);` representan lo mismo que el anterior, por separado.
- `setResizable (bool);` Indica si se permite cambiar el tamaño del componente o no.
- `setVisible (bool);`
- `setEnabled (bool);` si acepta eventos o no (aparecerá en gris)
- `setText (String n);` para establecer el texto que aparece en el elemento (etiqueta, botón, caja de texto, etc.)
- `setName (String n);` para asignarle un nombre al elemento.
- `getName();` para obtener el nombre del objeto.

Ventanas de Diálogo

Las ventanas de diálogo son ventanas sencillas, que se muestran en forma de “pop-up”, que muestran información y pueden pedir información al usuario. Otra característica fundamental es que son modales, es decir, paran la ejecución de nuestro programa hasta que el usuario cierre el diálogo seleccionando alguna de las opciones.

Las ventanas de diálogo se gestionan en Swing con la clase **JOptionPane** que contiene los métodos básicos para cada tipo de diálogo. Todos los métodos que vamos a ver tienen dos parámetros comunes a todos ellos:

- **parentComponent:** Apuntador al padre (ventana sobre la que aparecerá el diálogo) o bien `null`. En muchos casos podrá ser `this`, la ventana donde está definido.
- **message:** El mensaje a mostrar, habitualmente un `String`, aunque vale cualquier `Object` cuyo método `toString()` esté definido.

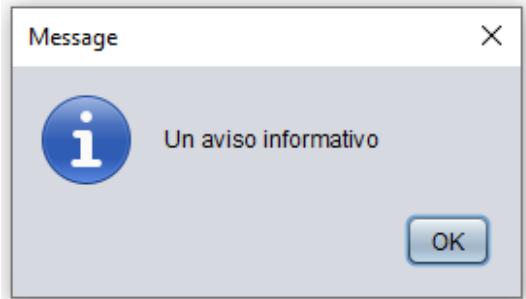
Estos son los métodos y los diálogos que generan:

JOptionPane.showMessageDialog()

Este es el diálogo más sencillo de todos, sólo muestra una ventana de aviso al usuario. La ejecución se detiene hasta que el usuario cierra la ventana. El método está sobrecargado, con más o menos parámetros, en función de si aceptamos las opciones por defecto o deseamos asignarle un título y un ícono. Ejemplo:

```
JOptionPane.showMessageDialog(this, "Un aviso informativo");
```

Mostrando:



No devolviendo ningún valor y continuando la ejecución del programa una vez se pulse *Aceptar*.

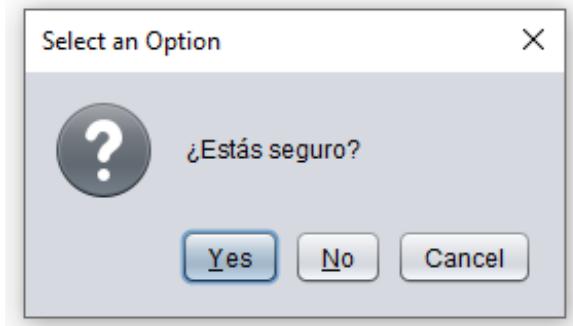
JOptionPane.showConfirmDialog()

Este método muestra una ventana pidiendo una confirmación al usuario, estilo "*¿Estás seguro?*" y da al usuario opción de aceptar o cancelar esa operación. El método devuelve un entero indicando la respuesta del usuario. Los valores de ese entero pueden ser alguna de las constantes definidas en *JOptionPane*: *YES_OPTION*, *NO_OPTION*, *CANCEL_OPTION*, *OK_OPTION*, *CLOSED_OPTION*.

El siguiente ejemplo de código

```
int respuesta = JOptionPane.showConfirmDialog(this, "¿Estás seguro?");
if (respuesta == JOptionPane.OK_OPTION)
    System.out.println("El usuario confirma la operación");
else
    System.out.println("El usuario cancela la operación");
```

muestra la siguiente imagen:



Podríamos configurar las opciones de respuesta con nuestros propios valores, utilizando el método *showOptionDialog* que veremos más adelante.

JOptionPane.showInputDialog()

A diferencia del anterior, este diálogo permite una respuesta del usuario. También está sobrecargado, admitiendo más o menos parámetros, según queramos aceptar o no las opciones por defecto. Los parámetros y sus significados son muy similares a los del método *showOptionDialog()*, pero hay una diferencia.

Si usamos los métodos que no tienen array de opciones, la ventana mostrará una caja de texto para que el usuario escriba la opción que desee (un texto libre). Si usamos un método que tenga un array de opciones, entonces aparecerá en la ventana un *JComboBox* en vez de una caja de texto, donde estarán las opciones que hemos pasado. Vemos las dos posibilidades en estos dos ejemplos:

```
String txt = JOptionPane.showInputDialog(this, "Introduce tu nombre");
System.out.println("El usuario ha escrito "+ txt);
```

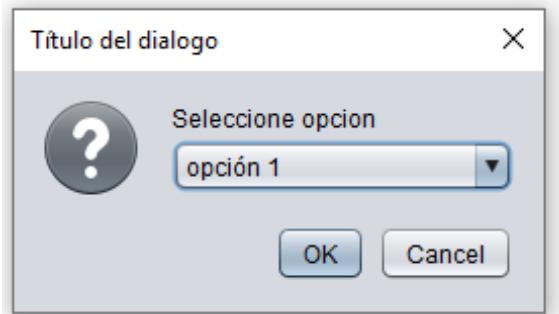
y la imagen que obtenemos con este código:



En este segundo ejemplo, incluimos *JComboBox* con las opciones:

```
Object txt = JOptionPane.showInputDialog(null, "Seleccione opcion",
    "Título del dialogo", JOptionPane.QUESTION_MESSAGE, null,
    new String[] { "opción 1", "opción 2", "opción 3" },
    "opcion 1");
System.out.println("El usuario ha elegido "+ txt);
```

y esta es la imagen que se obtiene.



En ambos casos, si se pulsa *Cancel*/o la X de cierre de ventana, el valor obtenido es *null*.

JOptionPane.showOptionDialog()

Este método muestra la ventana más configurable de todas, en ella debemos definir todos los botones que lleva. De hecho, las demás ventanas disponibles con *JOptionPane* se construyen a partir de esta. Por ello, al método debemos pasarle muchos parámetros:

- **parentComponent**: Apuntador al padre (ventana sobre la que aparecerá el diálogo) o bien *null* (ventana actual).
- **message**: El mensaje a mostrar, habitualmente un *String*, aunque vale cualquier *Object* cuyo método *toString()* esté definido.
- **title**: El título para la ventana.
- **optionType**: Un entero indicando qué botones tendrá el diálogo. Los posibles valores son las constantes definidas en *JOptionPane*: *DEFAULT_OPTION*, *YES_NO_OPTION*, *YES_NO_CANCEL_OPTION*, o *OK_CANCEL_OPTION*.

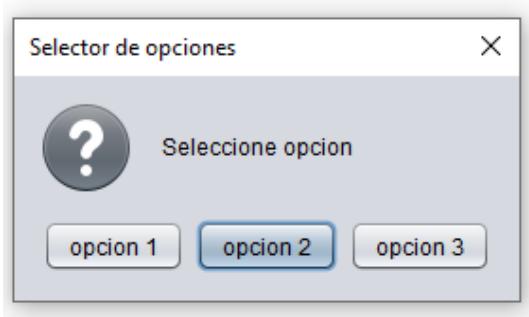
- **messageType:** Un entero para indicar qué tipo de mensaje estamos mostrando. Este tipo servirá para que se determine qué ícono mostrar. Los posibles valores son constantes definidas en *JOptionPane*: *ERROR_MESSAGE*, *INFORMATION_MESSAGE*, *WARNING_MESSAGE*, *QUESTION_MESSAGE*, o *PLAIN_MESSAGE*
- **icon:** Un ícono para mostrar. Si ponemos *null*, saldrá el ícono adecuado según el parámetro *messageType*.
- **options:** Un array de *objects* que determinan las posibles opciones. Si los objetos son componentes visuales, aparecerán tal cual como opciones. Si son *String*, el *JOptionPane* pondrá tantos botones como *String*. Si son cualquier otra cosa, se les tratará como *String* llamando al método *toString()*. Si se pasa *null*, saldrán los botones por defecto que se hayan indicado en *optionType*.
- **initialValue:** Selección por defecto. Debe ser uno de los *Object* que hayamos pasado en el parámetro *options*. Se puede pasar *null*.

La llamada a *JOptionPane.showOptionDialog()* devuelve un entero que representa la opción que ha seleccionado el usuario. La primera de las opciones del array es la posición cero. Si se cierra la ventana con la cruz de la esquina superior derecha, el método devolverá -1.

Aquí un ejemplo de cómo llamar a este método:

```
int seleccion = JOptionPane.showOptionDialog(
    null, "Seleccione opcion", "Selector de opciones", //ventana, mensaje, título
    JOptionPane.YES_NO_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE,
    null, // icono: null para ícono por defecto
    new Object[]{"opcion 1","opcion 2","opcion 3" }, // opciones
    null //null para YES, NO y CANCEL
    "opcion 2" //opción con el foco
);
```

y la ventana que se obtiene:



JDialog

Por último, comentar que disponemos de la clase *JDialog*, que es una clase base para hacer ventanas de diálogo totalmente personalizadas. Son un elemento más que se puede añadir a nuestro *JFrame* y podemos añadirle cualquier componente como cajas de texto, botones de opción, etc. Exactamente igual que hacemos con una ventana o con un panel.

A nivel de código haremos lo siguiente:

- En el constructor del *JDialog* añadiremos:

```
setLocationRelativeTo(null);
this.setVisible(true);
```

- El *JDialog* deberá tener un botón de cierre con el código:

```
this.setVisible(false);
```

- Definiremos las variables que queremos que el usuario asigne valor como atributos del *JDialog* y al cerrar la ventana tomarán valores a partir de los componentes incluidos en el diálogo. También podemos hacer validaciones y deshabilitar ese botón mientras no se cumplan ciertas condiciones.
- Para finalizar, desde el *JFrame* de nuestra aplicación llamaremos al diálogo. Por ejemplo, si le habíamos llamado *MyDialog*:

```
MyDialogCheckIn myD = new MyDialog(new javax.swing.JFrame(), true);
```

- Al cerrar el diálogo tendremos disponibles los valores de los atributos para tratarlos en nuestra aplicación: *myD.atributo1*, *myD.atributo2*, etc.

Menús

JMenuBar: Una barra de menú, que aparece generalmente situada debajo de la barra de título de la ventana



JMenu: Un menú de lista desplegable, que incluye *JMenuItem*s y que se visualiza dentro de la barra de menú (*JMenuBar*)



JMenuItem: Cada una de las opciones de un menú concreto



Gestión de Eventos

Ya hemos visto en los apartados anteriores eventos típicos como seleccionar un botón o detectar los cambios en un Slider. A continuación, vamos a profundizar un poco más en la gestión de eventos.

Eventos comunes a varios componentes

Netbeans crea código de eventos para cada componente por separado. Un enfoque diferente sería hacer métodos generales comunes a varios elementos y mediante **getSource** obtener el elemento concreto que lo ha producido.

Cuando tenemos Arrays de elementos, esta es la forma habitual de trabajar (por ejemplo, en un tablero, los dígitos de una calculadora, etc). No tendría sentido duplicar de varios elementos que realizan la misma función (en la calculadora, la tarea a realizar si pulsas el 1 o el 2 o 3... es la misma, solo necesitamos identificar el botón que se ha pulsado).

Analiza este ejemplo, que dibuja un tablero de damas y le asigna a cada casilla (que en realidad es un JButton) un nombre compuesto por la fila y columna en la que está ubicado.

```
import javax.swing.*; import java.awt.Color;

public class damas extends javax.swing.JFrame {
    private JButton tablero [][]; //el tablero es un array bidimens. de botones

    public damas() {
        boolean blanco=true;
        tablero = new JButton [8][8];
        for (int f=0;f<8;f++){
            for (int c = 0;c<8; c++) {

                // para cada posición del array añade un botón completo, eso
                // implica crear el botón con su evento y su nombre
                tablero [f][c] = new JButton();
                tablero [f][c].addActionListener(new java.awt.event.ActionListener()
                    { public void actionPerformed(java.awt.event.ActionEvent evt)
                        {tareaSipulsa(evt);}});

                // el nombre se construye con la fila y colum: 00,01..07, 10,11..17
                tablero [f][c].setName(Integer.toString(f)+Integer.toString(c));

                // una vez creado el botón añade el botón al JFrame (this)
                // y le indica su posición dentro del JFrame.
                this.add(tablero[f][c]);
                tablero [f][c].setBounds(c*30,f*30,34,34);

                // le asigna el color alternando blanco y negro
                if (blanco) {tablero [f][c].setBackground(Color.white);blanco=false;}
                else {tablero [f][c].setBackground(Color.black);blanco=true;}

            }
            if (blanco) blanco=false; else blanco=true;
        }
    }
    private void tareaSipulsa (java.awt.event.ActionEvent evt) {
        evt.getSource();
        //((JButton) evt.getSource()) sería el botón pulsado
    }
...
}
```

En este ejemplo, hemos situado los botones “a mano” en la pantalla, mediante el método `setBounds()`. Más adelante, cuando veamos los Layouts, veremos que hay formas más eficientes y rápidas de colocar los objetos con la disposición que queramos. El caso de los tableros, la **disposición GridLayout o GridBagLayout** será la más aconsejable.

Lista de Eventos

NOMBRE LISTENER	DESCRIPCIÓN	MÉTODOS	EVENTOS
ActionListener	Se produce al hacer click en un componente, también si se pulsa Enter teniendo el foco en el componente.	public void actionPerformed (ActionEvent e)	JButton: click o pulsar Enter con el foco activado en él. JList: doble click en un elemento de la lista. JMenuItem: selecciona una opción del menú. JTextField: al pulsar Enter con el foco activado.
KeyListener	Se produce al pulsar una tecla. según el método cambiara la forma de pulsar la tecla.	public void keyTyped(KeyEvent e)	Cuando pulsamos una tecla, segun el Listener: keyTyped: al pulsar y soltar la tecla.
		public void keyPressed (KeyEvent e)	keyPressed: al pulsar la tecla.
		public void keyReleased (KeyEvent e)	keyReleased: al soltar la tecla.
FocusListener	Se produce cuando un componente gana o pierde el foco, es decir, que esta seleccionado.	public void focusGained (FocusEvent e) public void focusLost (FocusEvent e)	Recibir o perder el foco.
ItemListener	Cambios en checkbox	ItemStateChanged()	Clicar en una checkbox
MouseListener	Se produce cuando realizamos una acción con el ratón.	public void mouseClicked (MouseEvent e)	Según el Listener: mouseClicked: pinchar y soltar.
		public void mouseEntered (MouseEvent e)	mouseEntered: entrar en un componente con el puntero.
		public void mouseExited(MouseEvent e)	mouseExited: salir de un componente con el puntero
		public void mousePressed (MouseEvent e)	mousePressed: presionar el botón.
		public void mouseReleased (MouseEvent e)	mouseReleased: soltar el botón.
		public void mouseDragged (MouseEvent e)	Según el Listener: mouseDragged: click y arrastrar un componente.
		public void mouseMoved (MouseEvent e)	mouseMoved: al mover el puntero sobre un elemento
TextListener	Cambios en texto	textValueChanged();	

Paneles y Layout Managers

Los paneles son contenedores para otros componentes y los usamos por diversos motivos:

- Para organizar los componentes. Si nuestro JFrame contiene muchos elementos, podemos dividirlos en distintas áreas, cada una con su panel.
- Para aplicaciones multiventana. Tendremos un solo JFrame con varios paneles (que normalmente ocuparán todo el JFrame) y según las distintas opciones que seleccione el usuario se mostrará uno y otro panel.
- Agrupar acciones sobre varios componentes. Si queremos hacer un tratamiento especial sobre un conjunto de elementos (por ejemplo, moverlos, hacerlos invisibles, etc.) si están en un panel, podremos hacerlo sobre el panel en vez de cada elemento uno a uno.
- Controlar la disposición de los componentes gráficos: en fila, haciendo un tablero, etc.
- Capas. Si nuestra aplicación muestra distintas capas que se superponen, los paneles pueden hacer esta función.

Ya comentamos que todo JFrame tiene un panel por defecto, pero podemos crear nosotros distintos paneles mediante la clase JPanel.

En Netbeans podemos crear un panel de varias formas: arrastrándolo desde la paleta como cualquier otro componente o bien creando en el proyecto un nuevo elemento, con botón derecho sobre el paquete deseado: *New > JPanel Form*, tal y como haríamos con un JFrame.

Una vez creados, trabajamos con ellos como un JFrame. Podemos mostrarlos u ocultarlos con el método *setVisible(boolean)*.

Para probarlo, crea un JFrame con:

```
import javax.swing.JPanel;
```

y añádele dos atributos de tipo JPanel:

```
private JPanel jPanel1;
private JPanel jPanel2;
```

Luego instanciamos los paneles y los añadimos al JFrame.

```
jPanel1 = new JPanel(null);
jPanel1.setBounds(10, 50, 200, 200);
jPanel1.setBackground(java.awt.Color.GREEN);
this.add(jPanel1);
jPanel1.setVisible(true);

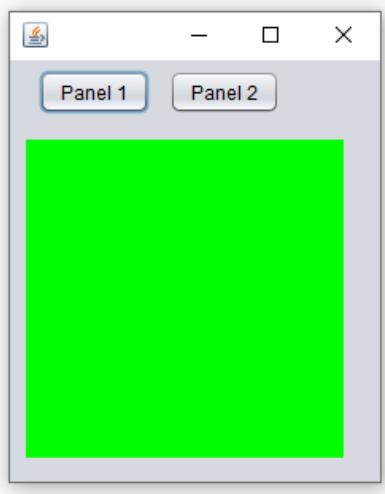
jPanel2 = new JPanel(null);
jPanel2.setBounds(10, 50, 200, 200);
jPanel2.setBackground(java.awt.Color.RED);
jPanel2.setVisible(false);
this.add(jPanel2);
```

Podríamos añadir al JFrame dos botones que hiciesen que estuviese visible uno u otro panel:

```
private void jButton1ActionPerformed
    (java.awt.event.ActionEvent evt) {
    jPanel1.setVisible(true);
    jPanel2.setVisible(false);
}

private void jButton2ActionPerformed
    (java.awt.event.ActionEvent evt) {
    jPanel1.setVisible(false);
    jPanel2.setVisible(true);
}
```

La forma de trabajar con los JPanel es como con los JFrame, es decir, añadiendo los componentes al mismo mediante el método *add()*.



Layout

Los paneles nos ofrecen distintas formas de organizar, de manera automática, la posición y tamaño de los componentes dentro de los contenedores, mediante lo que se conoce como *Layout Managers* o gestores de aspecto.

FlowLayout: Es el más simple y el que se utiliza por defecto en todos los paneles si no se indica el uso de alguno de los otros. Los componentes añadidos a un panel con FlowLayout se encadenan en forma de lista. La cadena es horizontal, de izquierda a derecha, y se puede seleccionar el espaciado entre cada componente.

Si el contenedor se cambia de tamaño en tiempo de ejecución, las posiciones de los componentes se ajustarán automáticamente, para colocar el máximo número posible de componentes en la primera línea.

Los componentes se alinean según se indique en el constructor. Si no se indica nada, se considera que los componentes que pueden estar en una misma línea estarán centrados, pero también se puede indicar que se alineen a izquierda o derecha en el contenedor.

BorderLayout: Esta composición (de borde) proporciona un esquema más complejo de colocación de los componentes en un panel. Es el layout o composición que utilizan por defecto JFrame y JDialog. La composición utiliza cinco zonas para colocar los componentes sobre ellas:

- Norte: parte superior del panel (NORTH o PAGE_START)
- Sur: parte inferior del panel (SOUTH o PAGE_END)
- Este: parte derecha del panel (EAST o LINE_END)
- Oeste: parte izquierda del panel (WEST o LINE_START)
- Centro: parte central del panel, una vez que se hayan rellenado las cuatro partes (CENTER)

Este controlador de posicionamiento resuelve los problemas de cambio de plataforma de ejecución de la aplicación, pero limita el número de componentes que pueden ser colocados en contenedor a cinco; aunque, si se va a construir un interfaz gráfico complejo, algunos de estos cinco componentes pueden contenedores, con lo cual el número de componentes puede verse ampliado.

En los cuatro lados, los componentes se colocan y redimensionan de acuerdo a sus tamaños preferidos y a los valores de separación que se hayan fijado al contenedor.

Es muy importante indicar:

- Tamaño prefijado: `setPreferredSize(new Dimension (WIDTH, HEIGHT));`
- Tamaño máximo: `setMaximumSize(new Dimension(WIDTH, HEIGHT));`
- Tamaño mínimo: `setMinimumSize(new Dimension(WIDTH, HEIGHT));`

ya que un botón o panel puede ser redimensionado a proporciones cualesquiera; sin embargo, el diseñador puede fijar un tamaño preferido para obtener una mejor visualización de cada componente.

CardLayout: Este es el tipo de composición que se utiliza cuando se necesita una zona de la ventana que permita colocar distintos componentes en esa misma zona. Este layout suele ir asociado con botones de selección, de tal modo que cada selección determina el panel (grupo de componentes) que se presentarán.

GridLayout: Esta composición proporciona gran flexibilidad para situar componentes. El controlador de posicionamiento se crea con un determinado número de filas y columnas y los componentes van dentro de las celdas de la tabla así definida.

Si el contenedor es alterado en su tamaño en tiempo de ejecución, el sistema intentará mantener el mismo número de filas y columnas dentro de los márgenes de separación que se hayan indicado. En este caso, estos márgenes tienen prioridad sobre el tamaño mínimo que se haya indicado para los componentes, por lo que puede llegar a conseguirse que sean de un tamaño tan pequeño que sus etiquetas sean ilegibles.

GridBagLayout: Es igual que la composición de GridLayout, con la diferencia que los componentes no necesitan tener el mismo tamaño. Es quizás el controlador de posicionamiento más sofisticado de los que actualmente soporta Swing.

BoxLayout: El controlador de posicionamiento BoxLayout es uno de los dos que incorpora Java a través de Swing y permite colocar los componentes a lo largo del eje X o del eje Y. También posibilita que los componentes ocupen diferente espacio a lo largo del eje principal.

En un controlador BoxLayout sobre el eje Y, los componentes se posicionan de arriba hacia abajo en el orden en que se han añadido. Al contrario, en el caso del GridLayout, aquí se permite que los componentes sean de diferente tamaño a lo largo del eje Y, que es el eje principal del controlador de posicionamiento, en este caso.

En el eje que no es principal, BoxLayout intenta que todos los componentes sean tan anchos como el más ancho, o tan alto como el más alto, dependiendo de cuál sea el eje principal. Si un componente no puede incrementar su tamaño, el BoxLayout mira las propiedades de alineamiento en X e Y para determinar dónde colocarlo.

OverlayLayout: El controlador de posicionamiento OverlayLayout, es el otro que se incopora a Java con Swing, y es un poco diferente a todos los demás. Se dimensiona para contener el más grande de los componentes y superpone cada componente sobre los otros.

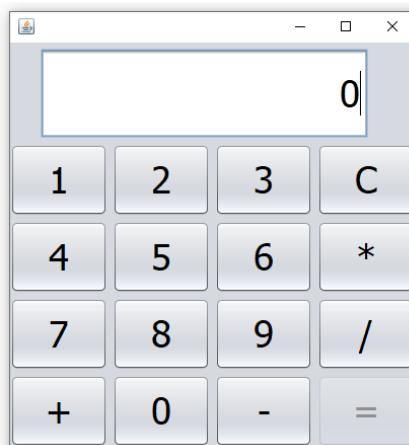
La clase OverlayLayout no tiene un constructor por defecto, así que hay que crearlo dinámicamente en tiempo de ejecución.

Por último, comentar que es posible construir nuestros propios layouts (CustomLayout).

Netbeans usa su propio esquema: GroupLayout y se basa en anidar los componentes por las dos dimensiones (horizontal y vertical) por separado.

Ejemplo: Calculadora con GridLayout

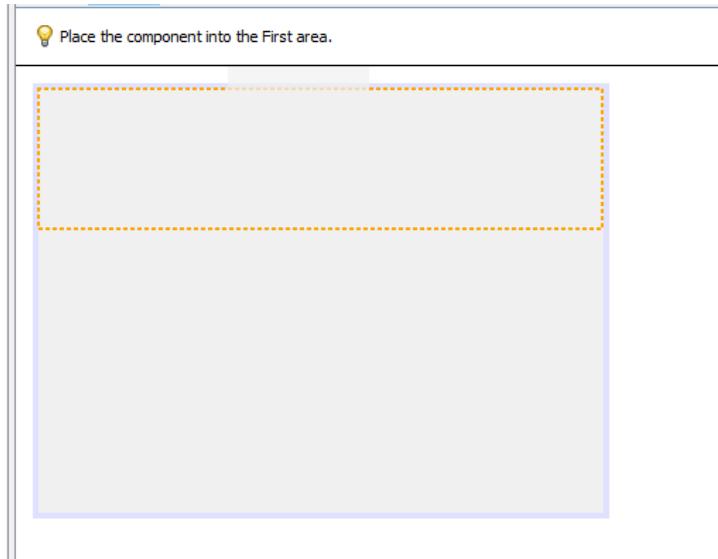
Vamos a hacer paso a paso una sencilla calculadora, empleando diferentes Layouts. La apariencia final que buscamos es como la siguiente figura.



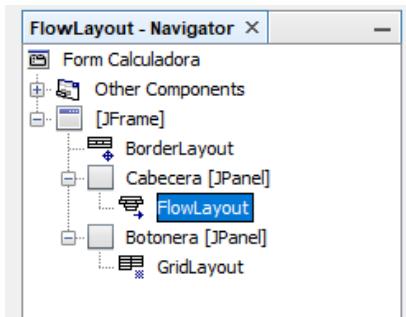
En ella tendremos un JPanel para la parte superior y otro JPanel para la botonera. Este segundo panel será tipo GridLayout de 4 filas x 4 columnas. Al añadir al panel los botones se irán colocando uno a continuación de otro ocupando las 16 posiciones.

1.- Sobre el paquete de nuestro proyecto, botón derecho > *New Frame*. A continuación, desde la ventana navegador, botón derecho sobre ella > *Set Layout > Border Layout*.

2. – Arrastramos un JPanel desde la paleta a la parte superior del JFrame, y lo soltamos cuando la línea de puntos amarilla nos muestre la zona superior del *frame*. Esto indica que estamos situando el panel en el área *North*. Desde la ventana Navigator, botón derecho sobre este panel: lo llamamos *Cabecera* cambiándole el nombre de la variable y decimos que tiene *BorderLayout*. Luego veremos que contendrá la caja de texto en la que se muestran operandos y resultados.



3.- Arrastramos otro JPanel desde la paleta a la parte central del *frame*, y le llamaremos *Botonera*. Desde la ventana *Navigator*, botón derecho sobre este panel: *Set Layout > GridLayout*. Luego le asignaremos todos los botones.



En las propiedades de ambos paneles podemos fijar su *PreferredSize* para establecer su tamaño, 400 x 100 para el primero y 400 x 300 para el segundo.

4.- Arrastramos al panel de cabecera un JTextField y cambiamos en la paleta su formato como tamaño de letra, alineación, etc.

5.- Modificamos el código del JFrame para añadir los 16 botones que forman la calculadora al panel *Botonera* e incorporamos las variables globales para hacer los cálculos:

```
public class Calculadora extends javax.swing.JFrame {

    long num1 = 0;
    long num2 = 0;
    char oper = '=';
    JButton[] tablero; //definimos el array con los 16 botones

    public Calculadora() {
        initComponents();

        setLocationRelativeTo(null); //ventana centrada en pantalla
        tablero = new JButton[16]; //definimos los 16 botones
        for (int i = 0; i < 16; i++) {
            tablero[i] = new JButton();
            tablero[i].addActionListener(new java.awt.event.ActionListener() {
                public void actionPerformed(java.awt.event.ActionEvent evt) {
                    FActionPerformed(evt);
                }
            });
            tablero[i].setName("jButton" + Integer.toString(i));
            tablero[i].setFont(new java.awt.Font("Tahoma", 0, 36));
            if (i < 10) {
                tablero[i].setText(Integer.toString(i));
            } else {
                switch (i) {
                    case 10: tablero[i].setText("+"); break;
                    case 11: tablero[i].setText("-"); break;
                    case 12: tablero[i].setText("C"); break;
                    case 13: tablero[i].setText("*"); break;
                    case 14: tablero[i].setText("/"); break;
                    case 15: tablero[i].setText("="); break;
                }
            }
        }
        //añadimos los botones al Grid Panel en el orden que queremos que los muestre
        Botonera.add(tablero[1]); Botonera.add(tablero[2]); Botonera.add(tablero[3]);
        Botonera.add(tablero[12]); Botonera.add(tablero[4]); Botonera.add(tablero[5]);
        Botonera.add(tablero[6]); Botonera.add(tablero[13]); Botonera.add(tablero[7]);
        Botonera.add(tablero[8]); Botonera.add(tablero[9]); Botonera.add(tablero[14]);
        Botonera.add(tablero[10]); Botonera.add(tablero[0]); Botonera.add(tablero[11]);
        Botonera.add(tablero[15]);
        tablero[15].setEnabled(false); //boton "=" desabilitado
    }
}
```

6.- Para finalizar, incorporamos el código que se ejecuta al pulsar un botón, esto es, el método *FActionPerformed(evt)*. Será un código común para los 10 botones que representan dígitos.

```

private void FActionPerformed(java.awt.event.ActionEvent evt) {
    String nomBoton = ((JButton) evt.getSource()).getName();
    int numBoton = Integer.parseInt(nomBoton.substring(7, nomBoton.length()));
    if (numBoton <= 9) { //si es un botón numérico
        if (jPantalla.getText().equals("0"))
            jPantalla.setText(((Integer) numBoton).toString());
        else
            jPantalla.setText(jPantalla.getText() +
                ((Integer) numBoton).toString());
    } else {
        switch (numBoton) {
            case 10://botón +
                num1 = Long.parseLong(jPantalla.getText());
                oper = '+';
                jPantalla.setText("0");
                tablero[15].setEnabled(true);
                break;

            case 11: //botón -
                num1 = Long.parseLong(jPantalla.getText());
                oper = '-';
                jPantalla.setText("0");
                tablero[15].setEnabled(true);
                break;

            case 12://letra "C" borra el último digito introducido
                int longit = jPantalla.getText().length();
                if (longit > 1)
                    jPantalla.setText(
                        jPantalla.getText().substring(0, longit - 1));
                else {jPantalla.setText("0"); }
                break;

            case 13: // botón *
                num1 = Long.parseLong(jPantalla.getText());
                oper = '*';
                jPantalla.setText("0");
                tablero[15].setEnabled(true);
                break;

            case 14: // botón /
                num1 = Long.parseLong(jPantalla.getText());
                oper = '/';
                jPantalla.setText("0");
                tablero[15].setEnabled(true);
                break;

            case 15: // botón =
                long resul = 0;
                num2 = Long.parseLong(jPantalla.getText());
                switch (oper) {
                    case '+': resul = num1 + num2; break;
                    case '-': resul = num1 - num2; break;
                    case '*': resul = num1 * num2; break;
                    case '/': if (num2 != 0) resul = num1 / num2;
                               else resul = 0;
                               break;
                }
                jPantalla.setText(Long.toString(resul));
                tablero[15].setEnabled(false);
                break;
        }
    }
}

```

Gráficos y Animaciones

La animación gráfica queda fuera del ámbito de este curso ya que es un tema muy especializado y existen librerías específicas para optimizar tanto el desarrollo como la ejecución. Sin embargo, vamos a ver utilidades sencillas que pueden resultarnos útiles en nuestras aplicaciones, para hacerlas más vistosas o programar juegos sencillos.

Imágenes

La forma más sencilla de incluir imágenes en nuestras aplicaciones es añadiendo una etiqueta `JLabel`, eliminando su texto y añadiéndole la imagen mediante el método `setIcon` (en Netbeans: propiedad "Icon"). Podemos incluir imágenes de diferentes tipos incluidos gifanimados.

Para que cuando generemos el `.jar` con el ejecutable no haya problemas con la ubicación de la imagen, esta tiene que estar en la estructura de carpetas del proyecto. Para ello, la forma de trabajar con imágenes será distinta según el tipo de proyecto que hayamos creado.

Para proyectos Ant:

- Primero crearemos una carpeta en el proyecto/src: Botón derecho sobre el proyecto/src > `New > Others > Folder`, y le llamamos como queramos, por ejemplo: `recursos`.
- Cuando asignamos la imagen a la propiedad `Icon` de la etiqueta, pulsamos en `Import image to Project`, incorporándola a la carpeta del paso anterior: `recursos`.

El código generado será:

```
jLabel1.setIcon(new
    javax.swing.ImageIcon(getClass().getResource("/recursos/img.gif")));
```

Para proyectos Maven:

- Primero crearemos una carpeta en el proyecto: Botón derecho sobre el proyecto `/src/main/` > `New > Others > Folder`, y le llamamos `resources` (es el nombre que se espera, sino habría que modificar el archivo de configuración `pom.xml`) y añadimos la imagen en esa carpeta.
- Cuando asignamos la imagen a la propiedad `Icon` de la etiqueta, pulsamos en `Image within Project` la seleccionamos directamente.

El código generado será:

```
jLabel1.setIcon(new
    javax.swing.ImageIcon(getClass().getResource("/img.gif")));
```

En ambos casos, para ejecutar desde Netbeans, es necesario compilar y ejecutar todo el proyecto (`Project > Clear & Build` y `Project > Run`, y no [May]+[F6] sobre el `JFrame`).

Swing.Timer

En nuestras aplicaciones con interfaz visual es muy frecuente que tengamos que ejecutar algo periódicamente, por ejemplo, un contador, un reloj, o el movimiento de un objeto por la ventana, etc. Para ello tenemos varias opciones: uso de hilos (`Threads`), la clase general `Timer` (del paquete `util`) y la clase `Timer` (del paquete Swing). La primera es la más completa pero también la más compleja.

La segunda y la tercera opción son más sencillas, y la ventaja principal de la última es que el código se programa en la propia clase del contenedor, por lo que tenemos disponibles los objetos que lo

componen sin necesidad de tener que pasárselos por parámetro, que sería lo que habría que hacer en caso de usar `util.Timer`.

Así pues, en nuestra aplicación gráfica, cuando necesitemos una tarea repetitiva, por ejemplo, definiríamos una instancia de Timer como atributo global: `Timer myTimer;`, importando previamente: `import javax.swing.Timer;`

Y luego invocaríamos a su constructor (por ejemplo, al iniciar la aplicación). El constructor de Timer necesita dos parámetros: el primero es el intervalo de ejecución en milisegundos y el segundo es la tarea a realizar (es de clase ActionListener, que a su vez se apoya en ActionEvent), quedando el código así:

```
myTimer = new Timer (1000, new java.awt.event.ActionListener () {
    public void actionPerformed(java.awt.event.ActionEvent e) {
        //Aquí la tarea a repetir, por ejemplo:
        // jTextField1.setText(LocalDateTime.now().toString());
    }
});
```

Y luego podríamos arrancar o parar la tarea con:

```
myTimer.start(); y pararla con myTimer.stop();
```

Moviendo elementos

Un aspecto fundamental en las aplicaciones gráficas es el movimiento de los elementos que aparecen en pantalla. Para ello, modificaremos su posición mediante el método `setLocation(x,y)` representando 'x' e 'y' las coordenadas de la esquina superior izquierda del elemento.

Estas coordenadas no son absolutas respecto a la aplicación, sino que son relativas al contenedor en el que esté ubicado el elemento, ya sea el JFrame o un JPanel/concreto.

Si lo que necesitamos es un desplazamiento, tendremos que saber las coordenadas actuales para incrementarlas o decrementarlas. Esa información la obtenemos con los métodos `getX()` y `getY()` y finalmente necesitaremos saber los límites por los que nos podemos desplazar. Lo haremos con los métodos `getHeight()` y `getWidth()` pero del contenedor del elemento, no del propio elemento (en muchos casos será `this` ya que programamos eventos en el contenedor).

El siguiente ejemplo movería la etiqueta en diagonal hacia abajo.

```
jLabel1.setLocation(jLabel1.getX()+1, jLabel1.getY()+1);
```

Este movimiento puede ser automático, con Swing.Timer o provocado por acciones del usuario mediante los eventos vistos previamente.

Dibujando figuras

Para dibujar figuras el proceso que seguiremos será el siguiente:

- 1.- Definir un panel (una nueva clase que extiende JPanel).
- 2.- Sobrescribir el método `paintComponent()` del panel
- 3.- Añadir el panel al JFrame
- 4.- Cada vez que haya un cambio, llamar a `repaint()` del panel, el cual llama internamente a `paintComponent()`

En la clase `Graphics` disponemos de métodos para dibujar figuras básicas. Por ejemplo:

```
@Override
public void paintComponent(Graphics g) {
    super.paintComponent(g); // siempre es la primera instrucción
    g.drawString("Ejemplo de texto", 30, 20); // texto y posición
    g.setColor(Color.RED); // color de primer plano
    g.fillRect(40, 50, 60, 70); // rectángulo sólido
    g.setColor(Color.BLUE); // cambia el color de primer plano
    g.drawRect(80, 90, 100, 120); // dibuja un rectángulo sin relleno
    g.drawLine(0, 0, 70, 70); // línea de esquina sup. izq. a inf. dcha
}
```

Ver esta serie de videos sobre cómo hacer el juego “Pong” con Swing en la que se ven los conceptos vistos previamente: <https://www.youtube.com/watch?v=fnJQLBPemcl>



01 PONG en Java / Creando la ventana y pelota

51.662 visualizaciones • Hace 4 años

Proyecto: <https://mega.nz/#IMYc2S0rS!ScmoXdZHvu49fwkxL4ngU2U>
SIGUEME EN LAS REDES SOCIALES n_n Sigue me en Twitter ► <https://>

El modelo MVC

Modelo-vista-controlador (MVC) es un patrón desarrollo que separa los datos y la lógica de negocio de la aplicación de su representación y el módulo encargado de gestionar los eventos. Con ello optimiza la reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

De manera genérica, los componentes de MVC se podrían definir como sigue:

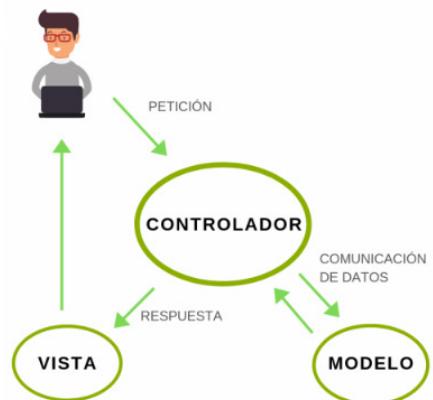
Modelo: Es la representación de la información con la cual el sistema opera, por lo tanto, gestiona todos los accesos a dicha información, tantas consultas como actualizaciones (lógica de negocio). Envía a la ‘vista’ aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario). Las peticiones de acceso o manipulación de información llegan al ‘modelo’ a través del ‘controlador’.

Controlador: Responde a eventos (usualmente acciones del usuario) e invoca peticiones al ‘modelo’ cuando se hace alguna solicitud sobre la información (por ejemplo, realizar un cálculo o modificar un registro en una base de datos). También puede enviar comandos a su ‘vista’ asociada si se solicita un cambio en la forma en que se presenta el ‘modelo’ (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos), por tanto, se podría decir que el ‘controlador’ hace de intermediario entre la ‘vista’ y el ‘modelo’.

Vista: Presenta el ‘modelo’ (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz de usuario), por tanto, requiere de dicho ‘modelo’ la información que debe representar como salida.

El proceso sería algo así:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, pulsando un botón).
2. El controlador recibe la notificación de la acción solicitada por el usuario y accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario).
3. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista.
4. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.



Este sistema es bueno como primera aproximación, pero se queda "corto" en cuanto a la división de responsabilidades, por ejemplo, no queda clara una división entre las entidades que forman el sistema y las reglas de negocio. Una evolución del patrón MVC son las divisiones en capas (Controlador, Servicio, Repositorio, etc...) que se emplean actualmente y que se denominan "arquitecturas limpias". Dentro de ellas, es común hablar de Arquitectura Hexagonal y Arquitectura DDD.

13. Excepciones

Gestión de Excepciones

En Java los errores en tiempo de ejecución (cuando se está ejecutando el programa) se denominan **excepciones**, y esto ocurre cuando se produce un error en alguna de las instrucciones de nuestro programa, como por ejemplo cuando se hace una división entre cero, cuando un objeto es 'null' y no puede serlo, cuando no se abre correctamente un fichero, etc. Cuando se produce una excepción se muestra en la pantalla un mensaje de error y finaliza la ejecución del programa.

En Java (al igual que en otros lenguajes de programación), existen muchos tipos de excepciones y enumerar cada uno de ellos sería casi una labor infinita.

Cuando en Java se produce una excepción se crea un objeto de una determinada clase (dependiendo del tipo de error que se haya producido), que mantendrá la información sobre el error producido y nos proporcionará los métodos necesarios para obtener dicha información. Estas clases tienen como clase padre la clase **Throwable**, por tanto, se mantiene una jerarquía en las excepciones. A continuación, mostramos algunas de las clases para que nos hagamos una idea de la jerarquía que siguen las excepciones, pero existen muchísimas más excepciones que las que mostramos:



La clase **Error** se utiliza para representar problemas graves fuera del control del programa (*OutOfMemoryError*, por ejemplo) y no se suelen capturar ni tratar de forma alguna. Se ha creado esta clase para diferenciarlas semánticamente del resto de las excepciones, consideradas menos graves y que son con las que trabajaremos a continuación.

La estructura empleada por Java para la gestión de excepciones es el uso del bloque: **try/catch/finally**

La técnica básica consiste en colocar las instrucciones que podrían provocar problemas dentro de un bloque `try`, y colocar a continuación uno o más bloques `catch`, de tal forma que si se provoca un error de un determinado tipo, lo que haremos será saltar al bloque `catch` capaz de gestionar ese tipo de error específico. El bloque `catch` contendrá el código necesario para gestionar ese tipo específico de error. Suponiendo que no se hubiesen provocado errores en el bloque `try`, nunca se ejecutarían los bloques `catch`.

```
try {
    division = num1 / num2;      //instrucción que puede provocar excepción.
}
    catch (Exception e) { division = 0; //si se produce excepción
}
```

En el caso anterior, si el denominador es igual a cero, se produciría una excepción y en vez de romper el programa, le asignaríamos cero al resultado de la división, y el programa continuaría normalmente. Se puede añadir una última parte al bloque, y que es opcional llamada: `finally` que se ejecuta finalmente, haya o no haya habido excepciones.

```
1 package Main;
2
3 public class Main {
4
5     public static int numerador = 10;
6     public static Integer denominador = 0;
7     public static float division;
8
9     public static void main(String[] args) {
10         System.out.println("ANTES DE HACER LA DIVISIÓN");
11         try {
12             division = numerador / denominador;
13         } catch (ArithmaticException ex) {
14             division = 0; // Si hay una excepción doy valor '0' al atributo 'division'
15             System.out.println("Error: "+ex.getMessage());
16         } finally {
17             System.out.println("División: "+division);
18             System.out.println("DESPUES DE HACER LA DIVISIÓN");
19         }
20     }
21 }
```

The screenshot shows an IDE interface with the following details:

- Code Area:** Shows the Java code for the `Main` class, including the `try`, `catch`, and `finally` blocks.
- Console Output:**
 - Line 1: <terminated> Main (5) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_20.jdk/Contents/Home/bin/java ()
 - Line 2: ANTES DE HACER LA DIVISIÓN
 - Line 3: Error: / by zero ← (highlighted by a red arrow)
 - Line 4: División: 0.0
 - Line 5: DESPUES DE HACER LA DIVISIÓN

Como vemos, capturamos la excepción en un objeto al que llamamos “`ex`” de la clase “`ArithmaticException`” y podemos obtener el mensaje de error que nos proporciona esa clase a través del método `getMessage()`. Vemos también que el programa termina su ejecución de forma correcta, aunque se haya producido una excepción.

También podríamos haber puesto simplemente `catch (Exception ex)` y así capturaríamos cualquier tipo de excepción. Por el contrario, si queremos distinguir las distintas excepciones que se pueden producir, podemos, dentro de una misma estructura `try`, definir todos los `catch` que queramos. En el caso anterior hemos definido solo la excepción “`ArithmaticException`”; pero, por ejemplo, podemos definir también la excepción “`NullPointerException`”, por si nos viene un valor a ‘null’ al hacer la división:

```

1 package Main;
2
3 public class Main {
4
5     public static int numerador = 10;
6     public static Integer denominador = null;
7     public static float division;
8
9     public static void main(String[] args) {
10         System.out.println("ANTES DE HACER LA DIVISIÓN");
11         try {
12             division = numerador / denominador;
13         } catch (ArithmeticsException ex) {
14             division = 0; // Si hay una excepción doy valor '0' al atributo 'division'
15             System.out.println("Error: " + ex.getMessage());
16         } catch (NullPointerException ex) {
17             division = 1; // Si si la excepción es de un null doy valor '1' al atributo 'division'
18             System.out.println("Error: " + ex.getMessage());
19         } finally {
20             System.out.println("División: " + division);
21             System.out.println("DESPUES DE HACER LA DIVISIÓN");
22         }
23     }
24 }

```

Problems @ Javadoc Declaration Console

<terminated> Main (5) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_20.jdk/Contents/Home/bin/java (12/10/2014 16:45)

ANTES DE HACER LA DIVISIÓN
Error: null
División: 1.0
DESPUES DE HACER LA DIVISIÓN

El objeto excepción, como vemos en el ejemplo, dispone de métodos que nos proporcionan información adicional sobre el error, el más importante es el ya mencionado `getMessage()`, pero disponemos de otros como: `getClassName()`, `getName()`, etc.

Hay muchísimas excepciones en Java por lo que se irán aprendiendo según nos las vayamos encontrando. Para saber el nombre de la excepción a tener en cuenta en cada caso, podemos provocarla previamente, cuando desarrollamos el programa y aplicar luego el nombre obtenido.

: Output - dam000prog (run)

```

run:
Introduce número 1: 6
Introduce número 2: 0
Exception in thread "main" java.lang.ArithmeticsException: / by zero
        at dam000t2.dam000t2e301.main(dam000t2e301.java:12)
C:\Users\Usuario\AppData\Local\NetBeans\Cache\11.0\executor-snippets\run.xml:111: The following
C:\Users\Usuario\AppData\Local\NetBeans\Cache\11.0\executor-snippets\run.xml:68: Java returned:
BUILD FAILED (total time: 3 seconds)

```

Tipos de Excepciones

En Java se distinguen dos tipos de excepciones: excepciones *checked* y *unchecked*. Una excepción *checked* representa un error del cual podemos recuperarnos y que debemos tener en cuenta obligatoriamente en nuestro código. Por ejemplo, una operación de lectura/escritura en disco puede fallar porque el fichero no existe, porque este se encuentre bloqueado por otra aplicación, etc. Todas estas situaciones, además de ser inherentes al propósito de la aplicación, son totalmente ajenas al propio código, y deben ser declaradas y manejadas mediante excepciones de tipo *checked* y sus mecanismos de control.

Por tanto, todas las excepciones de tipo *checked* deben ser capturadas (try...catch) o relanzadas (*throws*) hacia "arriba", hacia el método que haya llamado al código que produce la excepción (en ese caso, el "llamante" deberá capturarla o volver a lanzarla hacia "arriba"). Un ejemplo típico de estas excepciones son los errores de entrada/salida al leer un fichero.

Ejemplo que capture la excepción:

```
public void leerFichero() {
    try {
        List<String> lineas = Files.readAllLines(Paths.get("archivo.txt"));
    }
    catch (IOException e) {
        System.out.println("Error I/O: " + e.getMessage());
    }
}
```

Ejemplo que envía la excepción al método que lo invocó y que este deberá gestionarla:

```
public void leerFichero() throws IOException {
    List<String> lineas = Files.readAllLines(Paths.get("archivo.txt"));
}
```

Una excepción de tipo *unchecked* representa un error de programación, por ejemplo, acceder a elementos de un array en posiciones mayores que su tamaño. Estas excepciones no es necesario capturarlas en el código y en caso de no controlarlas, tampoco es necesario incluir *throws* en la firma del método.

Malas prácticas:

```
try {
    // Código que declara lanzar excepciones
} catch(Exception ex) { }
```

El código anterior ignorará cualquier excepción que se lance dentro del bloque *try*, o mejor dicho, capturará toda excepción lanzada dentro del bloque *try* pero la silenciará no haciendo nada (frustrando así el principal propósito de la gestión de excepciones). Cualquier error de diseño, de programación o de funcionamiento en esas líneas de código pasará inadvertido tanto para el programador como para el usuario. Algo mejor sería así:

```
try {
    // Código que declara lanzar excepciones
} catch(Exception ex) {
    ex.printStackTrace();
}
```

printStackTrace () mostraría el código de error por consola, pero no pararía la ejecución del código, es una buena fórmula para seguir con la ejecución, pero detectando posibles errores.

Throw y Throws

Hasta ahora hemos visto las excepciones que lanza el propio Java cuando se encuentran situaciones de error en la ejecución de un programa, pero también nos puede interesar que nuestros programas, bajo determinadas circunstancias, lancen excepciones. La palabra reservada *throw* nos permite hacer esto, nos permite lanzar una excepción de forma explícita.

Por ejemplo:

```
public void metodo (int valor) {
    if( valor < 0 )
        throw new RuntimeException("Error: valor negativo");
}
```

El flujo de la ejecución se detiene inmediatamente después de la sentencia `throw`, y nunca se llega a la sentencia siguiente. El control pasa al código que llamó a este método y debe gestionar la excepción como ya hemos descrito previamente.

¿y entonces throws? Los métodos que incluyan un lanzamiento de excepción mediante `throw` pueden incluir en su firma la palabra reservada `throws` indicando a todos los programadores que lanzan excepciones y es obligatorio en caso de las excepciones *checked*. Esto nos servirá para identificar métodos que, como generan excepciones, deben ser invocados teniendo en cuenta esta situación, por ejemplo, invocándolos dentro de un bloque `try...catch`.

Puede darse el caso que queramos lanzar distintas excepciones, en ese caso, las separaremos por comas después del `throws`. El siguiente ejemplo toma dos cadenas de texto, y en caso de que sean numéricas las divide. En caso contrario puede generar dos distintas excepciones (los textos introducidos no son números o división por cero):

```
public class gestionExcepciones {
    public static void main(String[] args) {
        Scanner scanner =new Scanner(System.in);
        System.out.println("Introduce un entero");
        String str1= scanner.nextLine();
        System.out.println("Introduce otro entero");
        String str2= scanner.nextLine();
        try {
            System.out.println(division(str1, str2));
        }
        catch(NumberFormatException ex){
            System.out.println("Se han introducido caracteres no numéricos");
            //o bien: System.out.println(ex.getMessage());
        }
        catch(ArithmaticException ex){
            System.out.println("División entre cero");
        }
    }

    static int division(String str1, String str2)
        throws NumberFormatException, ArithmaticException{
        int num1=Integer.parseInt(str1);
        int num2=Integer.parseInt(str2);
        return (num1/num2);
    }
}
```

Excepciones personalizadas

Además de las excepciones propias del sistema, es una buena práctica que nuestras aplicaciones lancen excepciones cuando se encuentran situaciones inesperadas, por ejemplo, que un usuario no introduzca los datos correctamente. Los programadores noveles suelen evitar las excepciones, y hacen que los métodos que llegan a una situación de error devuelvan `null` o valores por defecto.

Para las excepciones personalizadas tenemos dos opciones: o bien crear la propia excepción o bien usar *RuntimeException* con un mensaje personalizado. Para crear una excepción podemos seguir las mismas reglas que para otra clase, es decir, irán en archivo *.java* independiente, y generalmente en un paquete donde agruparemos todas las excepciones de la aplicación.

Completando el ejemplo anterior, podríamos añadir una excepción para el caso en que el usuario introduzca unos valores fuera de un intervalo predefinido:

```
public class IntervaloException extends RuntimeException {
    public IntervaloException () {
        super("Error: Número fuera del intervalo válido");
    }
}
```

Vemos que su código es tan sencillo como crear un constructor que invoca al constructor padre, pasándole un texto, que será el mensaje de la excepción, y al que tendremos acceso mediante el método *getMessage()*.

Una vez creada, se trata como cualquier otra excepción. El ejemplo anterior se modificaría añadiendo el lanzamiento de la excepción:

```
static int division(String str1, String str2)
    throws NumberFormatException, ArithmeticException, IntervaloException {
    int num1=Integer.parseInt(str1);
    int num2=Integer.parseInt(str2);
    if (num1 <0 || num1> 100 || num2 < 0 || num2 > 100)
        throw new IntervaloException();
    return (num1/num2);
}
```

Y su captura en el *main*:

```
catch(IntervaloException ex){
    System.out.println(ex.getMessage());
}
```

Podríamos hacer nuestra excepción más compleja, pudiendo mostrar distintos mensajes en función de los distintos parámetros a la hora de crearla. Haríamos un constructor que recibiría un parámetro y que en función de ese parámetro enviase a *super()* un mensaje u otro.

```
public class IntervaloException extends RuntimeException {
    ExpcionIntervalo (int parametro) {
        if (parametro==1)
            super("Error: Número por debajo del intervalo válido");
        else
            super("Error: Número por encima del intervalo válido");
    }
}
```

A la hora de lanzar la excepción, lo haríamos indicándole el parámetro que determinará el mensaje:

```
if(num1<0) throw new ExpcionIntervalo(1);
if(num1>100) throw new ExpcionIntervalo(2);
```

Si en nuestra aplicación se pueden producir muchas excepciones podría llegar a ser muy laborioso y complejo crear un número elevado de excepciones. Lo ideal es crear excepciones para los errores graves o importantes en la lógica de nuestra aplicación. El resto de errores, los menos importantes, los podemos resolver mediante *RuntimeException*. Para el caso anterior, podríamos haberlo resuelto así, sin crear la excepción *IntervaloException*.

```
if (num1 <0 || num1> 100 || num2 < 0 || num2 > 100)
    throw new RuntimeException("Error: Número fuera del intervalo válido");
```

Excepciones Frecuentes

Nombre de la Excepción	Descripción
<code>FileNotFoundException</code>	Lanza una excepción cuando el fichero no se encuentra.
<code>ClassNotFoundException</code>	Lanza una excepción cuando no existe la clase.
<code>EOFException</code>	Lanza una excepción cuando llega al final del fichero.
<code>ArrayIndexOutOfBoundsException</code>	Lanza una excepción cuando se accede a una posición de un array que no existe.
<code>NumberFormatException</code>	Lanza una excepción cuando se procesa un número, pero este es un dato alfanumérico. Se puede producir, por ejemplo, al hacer: <code>Integer.parseInt(txt)</code>
<code>NullPointerException</code>	Lanza una excepción cuando intentando acceder a un miembro de un objeto para el que todavía no hemos reservado memoria.
<code>IOException</code>	Generaliza muchas excepciones anteriores. La ventaja es que no necesitamos controlar cada una de las excepciones.
<code>Exception</code>	Es la clase padre de IOException y de otras clases. Tiene la misma ventaja que IOException.
<code>InputMismatchException</code> (import <code>java.util.*</code>)	Error en la entrada de datos por teclado, cuando para un entero, por ejemplo, se introduce un texto.
<code>ArithmaticException</code>	Errores en operaciones aritméticas
<code>ClassCastException</code>	Intento erróneo de convertir una clase en otra
<code>IllegalArgumetnException</code>	Argumento ilegal en la llamada a un método

Todas aquí: <https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html>

14. Ficheros

Entrada/Salida de información. Flujos

Con las estructuras de almacenamiento vistas hasta ahora sólo se podían guardar los datos durante la ejecución del programa. Para que los datos puedan perdurar de unas ejecuciones a otras, la solución es lo que se denomina “persistencia” y consiste en almacenar esos datos en disco y no en estructuras de memoria RAM como la ya vistas: variables, Arrays, etc. La forma habitual de lograr esa persistencia es mediante ficheros o bien bases de datos. En este tema trataremos los primeros.

Fichero: se define como una colección de información, que almacenamos en un soporte físico no volátil, para poderla manipular en cualquier momento.

Atendiendo a la forma en la que se graban los registros podemos clasificarlos en **ficheros secuenciales y aleatorios**. En los primeros se escriben y leen de forma secuencial, en orden de principio a fin, y en los segundos se accede directamente a la información deseada, mediante un índice.

Flujos: Determinan la comunicación entre un programa y el origen o destino de cierta información, es decir, es un objeto que hace de intermediario entre el origen y destino de la información. Podemos definir flujo como una 'abstracción' que proporciona Java y que identifica a una secuencia de bytes desde un dispositivo de entrada hacia un dispositivo de salida.

A través de dichos flujos vamos a poder realizar lecturas y escrituras sobre diferentes dispositivos sin que el programador necesita saber nada acerca de ellos, ya que será el núcleo de Java el que se entenderá con el S.O. para realizar las operaciones correspondientes.

En Java, dichos flujos están identificados por una jerarquía de clases que se encuentran en el paquete **java.io**, por lo que añadiremos siempre a nuestros programas y clases el import:

```
import java.io.*;
```

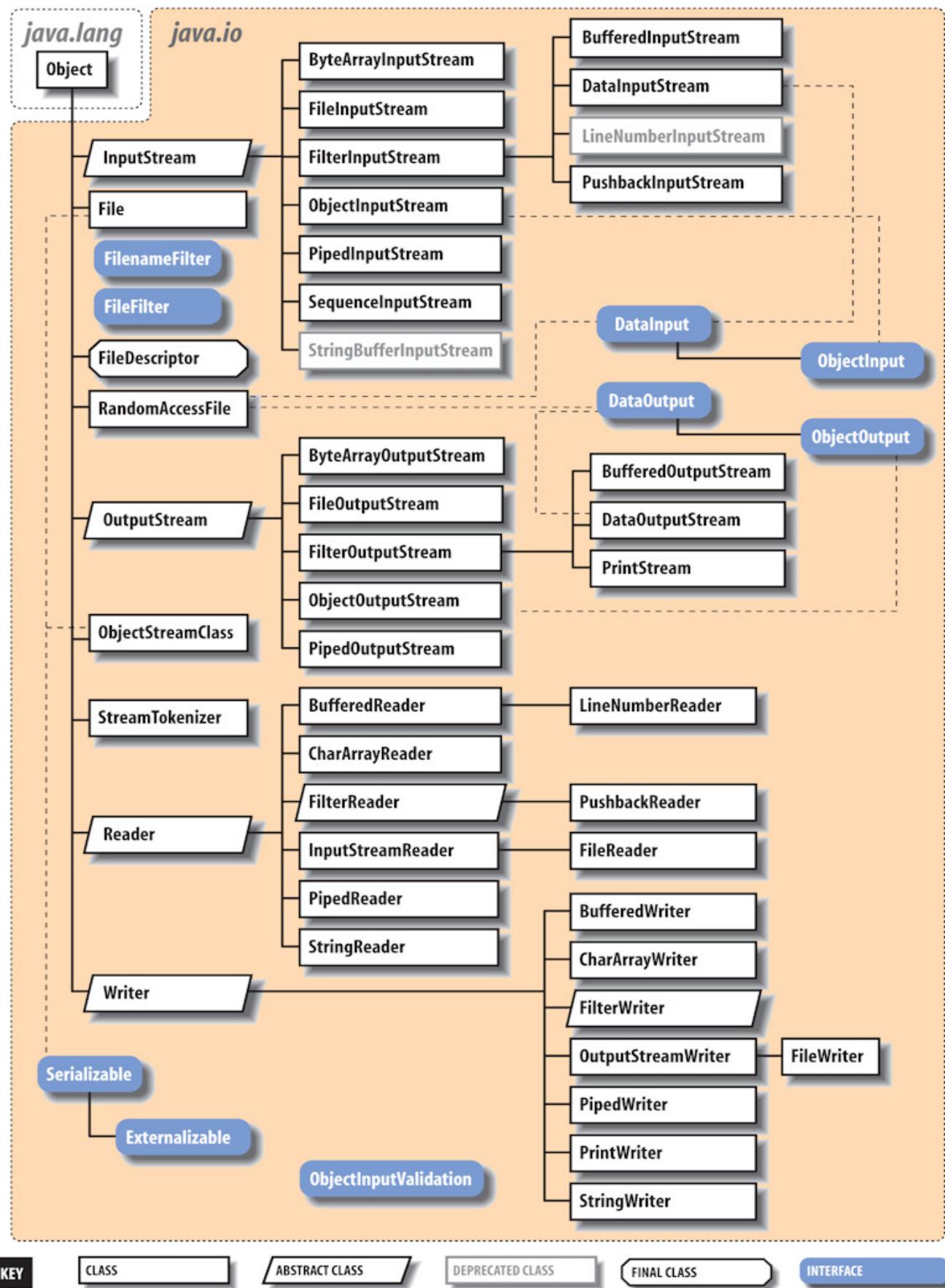
Quizás podemos entender más claramente el concepto de flujo si nos fijamos cuando escribimos por teclado y desde nuestro programa guardamos dicha información o cuando mandamos escribir por consola un texto determinado.

En Java, las letras (información) que estamos tecleando llegan a nuestro programa a través de un 'flujo' de entrada, denominado 'flujo de entrada estándar' y posteriormente enviamos información a la consola a través de un 'flujo de salida estándar'. En caso de encontrar algún error también enviará un mensaje a la consola (como la salida) por medio de otro flujo de datos. Estos flujos en Java los tenemos implementados en los objetos **System.in**, **System.out** y **System.err**.

Para tratar un archivo en la forma tradicional siempre vamos a tener que realizar las siguientes operaciones:

- Abrir el archivo para iniciar la lectura/escritura
- Leer /Escribir en el archivo (de forma repetitiva hasta llegar al final del mismo o aleatoria a un punto concreto)
- Cerrar el archivo.

Todas estas operaciones pueden producir excepciones de tipo **IOException** por lo que los programas y clases que trabajen con ficheros deberán realizar estas operaciones en bloques **try-catch** capturando este tipo de excepciones o bien “lanzarlas” hacia arriba, hacia el método que lo invocó con **throws**.

**KEY****CLASS****ABSTRACT CLASS****DEPRECATED CLASS****FINAL CLASS****INTERFACE**

---- implements

— extends

Esta figura muestra todas las clases e interfaces de las que dispone Java para trabajar con ficheros... ¡Un verdadero lío! Pero trataremos de simplificarlo en los siguientes apartados.

Buffers: En la jerarquía anterior vemos que hay ciertas clases que implementan un buffer (BufferedReader, BufferedInputStream,...). Un buffer es una zona de almacenamiento, que se utiliza de 'puente' entre un dispositivo de entrada y otro de salida, para aumentar la velocidad en las operaciones de lectura y escritura, así como el rendimiento.

Si usamos sólo las clases básicas para trabajar con ficheros (FileInputStream, FileOutputStream, FileReader o FileWriter, etc.) cada vez que hagamos una lectura o escritura, se hará físicamente en el disco duro. Si escribimos o leemos pocos caracteres cada vez, el proceso se hace costoso y lento, con muchos accesos a disco duro. Los BufferedReader, BufferedInputStream, BufferedWriter y BufferedOutputStream añaden un buffer intermedio para aumentar el rendimiento.

Cuando leamos o escribamos, esta clase controlará los accesos a disco:

- Si vamos escribiendo, se guardarán los datos hasta que tenga bastantes datos como para hacer la escritura eficiente.
- Si queremos leer, la clase leerá muchos datos de golpe, aunque sólo nos dé los que hayamos pedido. En las siguientes lecturas nos dará lo que tiene almacenado, hasta que necesite leer otra vez.

Esta forma de trabajar hace los accesos a disco más eficientes y el programa correrá más rápido. La diferencia se notará más cuanto mayor sea el fichero que queremos leer o escribir.

Para estudiar este tema de forma estructurada vamos a dividirlo en dos grandes bloques: lectura de ficheros por un lado y la escritura por otro.

Clases para lectura / escritura de ficheros

Lectura de ficheros de texto

Para leer ficheros de texto secuencialmente tendremos en cuenta los siguientes criterios:

- Usaremos la clase **FileStreamReader** con **InputStreamReader** y no **FileReader** ya que esta última tiene la limitación de no poder especificar la codificación del fichero (UFT-8, ISO-8859-1, etc.). Siempre debemos indicar el juego de caracteres, de lo contrario, se usará el configurado por defecto en el Java instalado en el ordenador, por lo que el mismo programa ejecutado en equipos distintos, provoque resultados distintos.
- Para mejorar su eficiencia, **InputStreamReader** será tratado a través de un **BufferedReader**.
- En casi todas las clases de lectura de datos vamos a encontrar distintos métodos para leer: **read()** para leer un solo carácter, **read(char[] b)** que lee de una sola vez los caracteres indicados por el tamaño del array, etc. pero nosotros usaremos **readLine()** que nos ofrece **BufferedReader** forma que leeremos líneas completas, y que luego trataremos como un String cualquiera.
- El fichero se abre en el constructor.
- Deberíamos cerrar el fichero de forma explícita con el método **close()**, aunque desde la versión JDK7 no es necesario si usamos **try-with-resources**.

try-with-resources: Es una forma “especial” de usar *try* de modo que el cierre del flujo se realiza automáticamente tanto si ha habido alguna excepción como si no. Y por lo tanto no se hace necesaria la llamada al método *close()*. La sintaxis es la habitual de un *try*, pero añade la definición de los recursos entre paréntesis justo después de la palabra reservada *tryy* antes de la llave de comienzo de bloque.

```
try (FileReader fr = new FileReader("fichero.txt") ) {  
    //tratamiento  
}  
catch (IOException ex) {  
    System.err.printf("%nError:%s",ex.getMessage());  
}
```

De lo contrario, deberíamos hacer algo así:

```
FileReader fr=null;  
try { fr = new FileReader("fichero.txt");  
    //tratamiento  
} catch (IOException ex) {  
} finally {  
    try{ if (fr !=null) fr.close(); } //close() tb puede generar excepcion  
    catch(IOException e){ System.err.printf("Error:%s",e.getMessage()); }  
}
```

Teniendo en cuenta estas consideraciones, la estructura que emplearemos habitualmente para la lectura secuencial de ficheros de texto será esta:

```
File f = new File("fichero.txt"); String cadena;  
try (FileInputStream fis = new FileInputStream(f);  
     InputStreamReader isr = new InputStreamReader(fis,"UTF-8"); //ISO-8859-1"  
     BufferedReader bfr = new BufferedReader(isr)) {  
    while((cadena=bfr.readLine()) != null)  
        //tratamiento, por ejemplo: System.out.println(cadena);  
}  
catch (IOException ex) {  
    System.err.printf("Error:%s\n",ex.getMessage());  
}
```

Podemos ver como se llama a los constructores y se repite el proceso de lectura hasta el final del archivo.

Clase Scanner

Esta clase ya la hemos utilizado durante el curso como forma de obtener información a través del teclado. **No es una clase que represente un flujo, sino que hace uso de las clases comentadas en el gráfico inicial.** Así podemos ver como utilizar otros flujos de entrada que no sea el teclado. Este sería un ejemplo de su uso mediante el método *hasNextLine()* para saber cuándo se ha acabado de leer el fichero (existen múltiples hasXXXXX disponibles en esta clase).

```
String salida="";  
try (FileReader fr = new FileReader("fichero.txt");  
     Scanner sc = new Scanner(fr)){  
    while(sc.hasNextLine()){  
        salida +=sc.nextLine() + '\n';  
    }  
    System.out.println(salida);  
}  
catch (IOException ex) {  
    System.err.printf("%nError:%s",ex.getMessage()); }
```

Aspectos a tener en cuenta:

- Las llamadas a los constructores llevan implícita la apertura del fichero.
- En el nombre de la ruta podemos indicarle un path o ruta donde está ubicado (ruta absoluta o relativa). La ubicación actual en caso de rutas relativas es la raíz del proyecto cuando lo ejecutamos desde el IDE o la ubicación del jar cuando ya está en producción.
- El salto de línea en Windows son dos caracteres (\r\n) mientras que en Linux es solo uno (\n). Java nos ofrece un método que devuelve un String de acuerdo a nuestro sistema operativo por lo que para aplicaciones multiplataforma es recomendable su uso: `System.getProperty("line.separator");`
- La barra que separa las carpetas en el path o ruta de un archivo es '\' en sistemas Windows y '/' en sistemas Linux. Al igual que en el caso anterior, Java nos ofrece una forma de evitar esta dualidad. Tenemos dos formas de obtener el String correspondiente: mediante la constante: `File.separator;` o mediante el método: `System.getProperty("file.separator");`
- Cuando trabajamos con rutas Windows, como la '\' es un carácter de escape deberíamos duplicarla, por ejemplo: `String ruta = "c:\\misarchivos\\nombre.txt";`
- La codificación de archivos por defecto suele ser **UTF-8** pero podemos indicarle en el `InputStreamReader` **ISO-8859-1** para codificación **ANSI**. Existen otras codificaciones como **Unicode** (emplea 2Bytes por carácter): **UTF-16**

Archivos csv

Un archivo '.csv' (*comma-separated values*) es un archivo de texto separado cada campo por un símbolo (en muchos casos, un punto y coma) y cada línea termina por un salto de línea. Es un formato habitual para convertir hojas de cálculo a archivos de texto. El siguiente ejemplo podría representar una archivo csv con un nombre, edad y número de teléfono.

```
Angel:23:981111111
Pedro:45:234232314
```

La clase `String` dispone de un método llamado `split()` que obtiene de una cadena un array con las sub-cadenas comprendidas entre el delimitador pasado como parámetro. Así que como lo que leemos en el fichero de texto, son líneas, podríamos tratar cada línea así:

```
String[] partes = linea.split(";");
if (partes.length == 3)
    System.out.printf("%s,%s,%s\n",partes[0],partes[1],partes[2]);
else System.out.println("Error de formato");
```

El parámetro que recibe `split` puede ser una expresión regular. Aunque las veremos en detalle en el tercer trimestre, podremos decir que permite especificar patrones, por ejemplo, si queremos obtener las palabras que componen una frase, el separador será el espacio en blanco, pero una o varias veces: `String[] partes = linea.split(" +");` Si queremos indicar que el punto o la coma también son separadores: `String[] partes = linea.split("[.,]+");`

Clase Files

La clase `Files`, incorporada en la versión 7 de Java, incluye muchos métodos estáticos para la manipulación de archivos (copiar, renombrar, consultar atributos, etc.) e incorpora también métodos que permiten leer y escribir ficheros completos en una sola operación, a partir de una estructura en memoria. Serían los siguientes:

- `readAllLines (Path, CharSet)`: Lee todas las líneas y las devuelve en una `List<String>`.
- `readAllBytes (Path)`: Lee todos sus bytes y devuelve un array de byte con todo su contenido.
- `write (Path, byte[], options)`: Escribe el array de bytes pasado en el fichero. Devuelve el path. En las opciones se le puede especificar si añade o sobreescribe el fichero.
- `write (Path, Colección de líneas, Charset)`: Escribe las líneas de texto al archivo.

En este curso nos centraremos solo en el primero:

```
public static List<String> leerFicheroTexto (String rutaArchivo) {
    try {
        Path path = Paths.get(rutaArchivo);
        return Files.readAllLines(path); // UTF8
    //return Files.readAllLines(path, StandardCharsets.ISO_8859_1); //ANSI
    }
    catch (IOException ex) {System.err.printf("%nError:%s",ex.getMessage());}
        return null;
}
```

O bien, sin `try...catch`:

```
public static List<String> leerFicheroTexto (String rutaArchivo)
    throws IOException {
    Path path = Paths.get(rutaArchivo);
    return Files.readAllLines(path); // UTF8
    //return Files.readAllLines(path, StandardCharsets.ISO_8859_1); //ANSI
}
```

En un solo paso, leemos todo el archivo y lo pasamos a memoria. Obviamente, si el fichero es muy grande, la `List` en memoria podría ser poco eficiente. En el ejemplo expuesto, la colección devuelta sería una lista con un elemento por cada línea leída.

Clase File

La clase `File` es una representación de un fichero o un directorio, permitiendo operaciones como:

- Borrar un archivo.
- Crear un archivo.
- Establecer fecha y hora de modificación del archivo.
- Crear un directorio.
- Listar el contenido de un directorio.

Por ejemplo, podemos hacer uso del método `exists()` para determinar si un fichero/directorio existe antes de hacer uso del mismo (hasta ahora hacíamos uso de la excepción `FileNotFoundException`).

Dispone de diversos constructores:

```
File (String ruta_completa);           File fichero=new File ("archivos\\texto.txt");
File (String ruta, String nombre);   File fichero=new File ("archivos","texto.txt");
```

Y métodos como:

<code>exists()</code>	Devuelve true si el fichero ya existe
<code>length()</code>	Devuelve el tamaño del fichero
<code>delete()</code>	Borra el fichero
<code>renameTo()</code>	Renombra el fichero especificado por el objeto File
<code>toString()</code>	Devuelve la ruta especificada cuando se creó el objeto File
<code>createNewFile()</code>	Crea el fichero (devuelve <i>false</i> si ya existe el fichero)

Podemos enviarlo como un flujo de entrada de un `FileInputStream`, `FileOutputStream`, `FileReader`, `Scanner` como vimos anteriormente.

Un ejemplo:

```
File fichero = new File("fichero.txt");
if (!fichero.exists()){
    System.err.println("El fichero no existe.");
    try { fichero.createNewFile(); }
    catch (IOException ex) { System.err.println("Error:" +ex.getMessage()); }}
```

Copiar un archivo

La clase `File` no ofrece ningún método para copiar archivos. Podríamos hacerlo "a mano", creando un archivo con un flujo de escritura e ir leyendo del origen y escribiendo en el archivo destino.

Una forma más cómoda de hacerlo es mediante la clase `java.nio.file.Files` que tiene método `copy`, a la que le pasaremos ruta origen y ruta destino. Ejemplo:

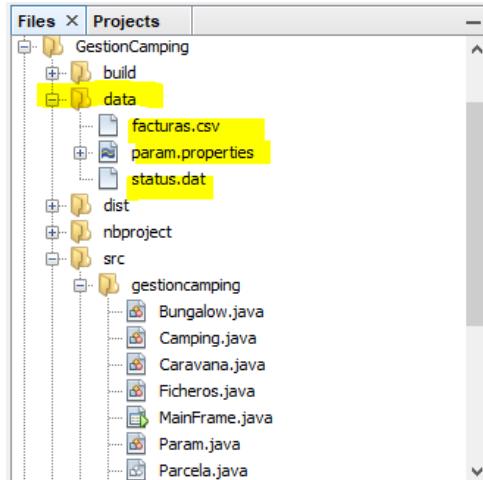
```
import java.nio.file.*;
...
File fo = new File("fichOrigen.txt");
File fd = new File("fichDestino.txt");
try {if (fd.exists()) fd.delete();
    Files.copy(f.toPath(), b.toPath());
} catch (IOException ex) {System.err.printf("Error:%s",ex.getMessage());}
```

Ubicación física de los ficheros:

Cuando distribuyamos nuestra aplicación crearemos un archivo `.jar` mediante la opción "Build" de nuestro IDE. Ese archivo contendrá las clases compiladas a bytecode (archivos `.class`) y otros recursos como imágenes o iconos, pero no los archivos de datos.

Los archivos deberán estar en el sistema de archivos de nuestro sistema, no en `.jar`, por ello, lo más habitual es crear en el IDE una carpeta para los archivos en la raíz de nuestro proyecto y referenciarla con posicionamiento relativo, por ejemplo: `File fichero=new File ("data/facturas.csv");`

Al distribuir la aplicación, podemos generar un archivo comprimido con el `.jar` y la carpeta con los ficheros.



Otras clases

Existen muchas otras clases con características y métodos específicos que pueden ser útiles en determinados contextos: CharacterArrayReader, StringReader, PipedReader, PushbackReader, LineNumberReader, etc.

Clases para escritura en ficheros de texto

Para la escritura secuencial de ficheros de texto tendremos en cuenta una serie de premisas análogas a las mencionadas para la lectura de ficheros.

- Usaremos la clase ***FileOutputStream*** con ***OutputStreamReader*** y no ***FileWriter*** ya que esta última tiene la limitación de no poder especificar la codificación del fichero (UFT-8, ISO-8859-1, etc.). Opcionalmente, dispone de un segundo parámetro, de tipo *boolean* que en caso de valer *true* añade los datos al final del fichero y en caso de ser *false* (o no estar presente este segundo parámetro) borrará los datos que existiesen previamente.

Ejemplos:

- ***FileOutputStream fs=new FileOutputStream ("fichero.dat");*** Esta instrucción crea el fichero con nombre texto.txt. Si ya existiera borraría su contenido.
- ***FileOutputStream fs=new FileOutputStream ("fichero.dat", true);*** Esta instrucción si no existiera el fichero, lo crea y si ya existe añade datos al final.
- Para mejorar su eficiencia, *OutputStreamReader* será tratado a través de un ***BufferedWriter***.
- Usaremos ***write(cadena)*** que nos ofrece *BufferedWriter* forma que escribiremos líneas completas. Podemos usar también ***newLine()*** para escribir los saltos de línea.
- El fichero se abre en el constructor.
- Deberíamos cerrar el fichero de forma explícita con el método *close()*, aunque desde la versión JDK7 no es necesario si usamos ***try-with-resources***.

Teniendo en cuenta estas consideraciones, la estructura que emplearemos habitualmente para la escritura secuencial de ficheros de texto será esta:

```
File f = new File("fichero.txt");
try {
    FileOutputStream fos =new FileOutputStream(f, true);
    OutputStreamWriter osw =new OutputStreamWriter(fos, "UTF-8"); //ISO-8859-1"
    BufferedWriter bfw = new BufferedWriter(osw)) {
        bfw.write("Esto es un texto"); bfw.newLine();
        bfw.write("Esto es otro texto con eñe");
    } catch (IOException ex) {
        System.err.printf("Error:%s", ex.getMessage());
    }
}
```

PrintWriter

Permite enviar cadenas de caracteres con un formato a un flujo de salida (Writer, File, OutputStream o Cadena con nombre del fichero) con métodos que ya utilizamos para la salida por consola: *print()*, *println()*, *printf()*.

Tiene varios constructores en los que le podemos pasar como parámetro un *File* o un *OutputStream/FileWriter*. En el primer caso se le puede pasar adicionalmente el tipo de codificación y en segundo caso un *boolean* que representa si el 'autoflush' está activado o no. Ejemplo:

```
File fichero = new File("fichero.txt");
try(PrintWriter pw = new PrintWriter(fichero, "UTF-8")){ }
```

Los métodos de escritura de esta clase no lanzan excepciones IOException en caso de error. El programador debe hacer uso de los métodos *checkError()* para comprobar si la llamada al método produjo un error y *clearError()* para eliminar ese error y poder chequear otro posterior.

Si está habilitado el 'autoflush', cada vez que se invoque a los métodos 'println', 'printf' o 'format' se escribirá en el fichero. Sino será necesario llamar al método 'flush' o cerrar el flujo para que se envíen.

Como en casos anteriores, podemos hacer que un *FileWriter* sea 'envuelto' por un 'BufferedWriter' para hacer uso de un buffer y aumentar el rendimiento y a su vez, el *BufferedWriter* será 'envuelto' por un *PrintWriter* para poder enviar al flujo de salida diferentes tipos de datos, como flotantes, booleano o cualquiera de los que soporta la clase *PrintWriter*.

De esta forma nos evitamos tener que estar convirtiendo los datos a enviar al flujo.

Fijarse que el orden en el que se envuelven las clases es importante, ya que podríamos haber hecho que la clase *BufferedWriter* envolviera a la *PrintWriter*, pero de esa forma sólo, si quisieramos utilizar el buffer sólo lo podríamos conseguir llamando a los métodos de *BufferedWriter*, no a los de *PrintWriter*.

```
File f = new File("fichero.txt"); double num=2.3232d;
try {
    FileOutputStream fos =new FileOutputStream(f, true);
    OutputStreamWriter osw =new OutputStreamWriter(fos, "UTF-8"); //ISO-8859-1"
    BufferedWriter bfw = new BufferedWriter(osw);
    PrintWriter pw = new PrintWriter(bfw, true)) {
        pw.printf("num=%06.1f\n", num);
        pw.println("linea nueva");
    } catch (IOException ex) {
        System.err.printf("Error:%s", ex.getMessage());
    }
}
```

Serialización de objetos

En Java disponemos de clases que permiten establecer un flujo de entrada (lectura) / salida (escritura) de objetos. La **serialización** es el proceso por el que un objeto se convierte en una secuencia de bytes, permitiendo guardar su contenido en un archivo:

- Cuando escribimos un objeto a disco lo que hace la clase **ObjectOutputStream** es convertir el contenido de cada uno de los campos a binario y lo guarda en disco.
- Cuando leemos un objeto de disco, lo que hace la clase **ObjectInputStream** es leer un flujo de bytes que después, mediante un cast, guardará en cada uno de los campos del objeto.

Para que un objeto pueda ser serializado debe de implementar la interface **java.io.Serializable**. Esta interface no define ningún método, pero toda clase que la implemente, informará a la JVM que el objeto será serializado.

Todos los tipos primitivos de datos en Java son serializables al igual que los arrays. Si una clase tiene como 'dato' algún objeto de otra clase, esa otra clase debe implementar la interface **Serializable**.

Podemos indicarle a Java que un atributo de una clase no sea serializado (y por lo tanto no lo guardará) con la palabra clave **transient** de la forma:

```
private transient String ejemplAtributo;
```

Importante:

La serialización no permite añadir objetos a un archivo conservando los que tuviera previamente. Esto es debido a que cada vez que se añade un objeto habiendo cerrado el archivo, se añade una cabecera. Si añadiésemos, se crearía una nueva cabecera cada vez, y los archivos tendrían un número indeterminado de cabeceras. Un ObjectInputStream solo va a leer una cabecera.

Cuidado con el método `readObject()`. Este método no indica cuando se acaba el fichero, por lo que si estamos haciendo un bucle while leyendo objetos, la condición del while para salir podría ser llamando al método `available()` de un flujo de bytes asociado al fichero (leeríamos mientras el método devuelva un valor mayor que 0). Otra forma sería capturar la excepción `EOFException`.

Cuando leemos un objeto, tenemos que hacer un cast a la clase a la que pertenece. Sin embargo, aplicando el polimorfismo, podemos hacer un cast a la clase 'común' pero aún así, el objeto tendrá toda la información de su clase original, de tal forma que si lo convertimos a su clase original (con otro cast) podremos acceder a todos sus métodos y propiedades.

Veamos en un ejemplo práctico. Partiendo de la siguiente clase:

```
public class Persona implements Serializable { //import.java.io.*  
    public String nombre;  
    public String telefono;  
    public float sueldo;  
    public Persona(String nombre, String telefono, float sueldo){  
        this.nombre = nombre;  
        this.telefono = telefono;  
        this.sueldo = sueldo;  
    }  
    @Override  
    public String toString(){  
        return String.format("Nombre:%s%nTelefono:%s%nSueldo:%.2f%n",  
                            nombre, telefono, sueldo);  
    }  
}
```

Para escribir en fichero un array de objetos de la clase Persona llamado `pers[]`:

```
Persona[] pers = {new Persona("Pedro", "981222333", 1234.34f),  
                  new Persona("Juan", "982444555", 2222.34f)};  
try( FileOutputStream fos = new FileOutputStream("fichero.dat", false);  
     // No podemos añadir, siempre vacía contenido previo si lo hubiese  
     BufferedOutputStream bos = new BufferedOutputStream(fos);  
     ObjectOutputStream oos = new ObjectOutputStream(bos) ){  
    for(int cont=0; cont < pers.length; cont++)  
        oos.writeObject(pers[cont]);  
}  
catch (IOException ex) {System.err.println("Error:"+ ex.getMessage()); }
```

La operación contraria, leer las instancias de Persona desde el disco sería como se muestra a continuación: (*vamos a suponer que el número de objetos leídos nunca será superior a 100*)

```

Persona[] personas = new Persona[100];
boolean eof = false;
File fichero = new File("fichero.dat");
try( FileInputStream fis = new FileInputStream(fichero);
    BufferedInputStream bufis = new BufferedInputStream(fis);
    ObjectInputStream ois = new ObjectInputStream(bufis)) {
    int cont = 0;
    while(!eof) { //while(bufis.available()>0
        personas[cont] = (Persona)ois.readObject();
        if (++cont > personas.length) break;
    }
} catch (EOFException e) {eof = true;}
} catch (IOException ex) { System.err.println("Error:"+ ex.getMessage()); }
} catch (ClassNotFoundException ex) { System.err.println("Err:"+ ex.getMessage()); }

```

Entrada/Salida estándar

A nivel de sistema operativo (sobre todo en Linux/Unix) se identifica la entrada estándar (stdin) con el teclado y la salida estándar (stdout) como la pantalla. También se dispone de la salida de errores (stderr) que también va a la pantalla.

Java tiene acceso a estos flujos estándar a través de la clase System. Así:

- **Stdin:** Es un objeto de la clase InputStream y es el flujo de entrada estándar (lectura por teclado). Dicho flujo se puede redirigir con el método System.setIn(InputStream). Podemos acceder a él de la forma: System.in.
- **Stdout:** Es un objeto de la clase PrintStream. Se pueden utilizar los métodos print y println. Se puede redirigir llamando al método System.setOut(PrintStream). Podemos acceder a él de la forma: System.out.
- **Stderr:** Es un objeto de la clase PrintStream. Se pueden utilizar los métodos print y println. Se puede redirigir llamando al método System.setErr(PrintStream). Podemos acceder a él de la forma: System.err.

Si cambiamos la entrada / salida / errores estándar y queremos volver a su 'valor original', tendremos que emplear la clase FileDescriptor para tener una referencia a los flujos originales de entrada, salida y errores de la forma: FileDescriptor.in, FileDescriptor.out y FileDescriptor.err.

Para redireccionar a la salida estándar por defecto, podríamos poner: System.setOut(new PrintStream(new FileOutputStream(FileDescriptor.out)));

En el comienzo de este curso vimos que, para leer del teclado, hacíamos uso de la clase Scanner. Utilizando los conceptos vistos de salida y entrada estándar podemos hacer una redirección de la salida estándar (podríamos hacerlo con la salida de errores o la entrada estándar) a un fichero.

Para ello debemos hacer uso del método setOut() de la clase System.

```

FileReader f = null; int c;
PrintStream pStreamSalida=null;
try {pStreamSalida = new PrintStream("fichero.txt");
    System.setOut(pStreamSalida);
    f = new FileReader("fichero.txt");
    while((c = f.read())!= -1) {
        System.out.print((char)c);
    }
} catch (IOException ex) {System.err.println("Error I/O");}

```

```

    finally {try {
        if (f != null)
            f.close();
    } catch (IOException ex) {System.err.println("Error I/O"); }
}

```

Cualquier System.out que hagamos después de redirigir la salida estándar será enviado al fichero "fichero.txt"

Clase Properties

Java dispone de librerías específicas para trabajar con ficheros de configuración, esto es ficheros que se componen de parejas de variables y valores, típicos en casi cualquier aplicación, servicio, etc. Como todos siguen patrón similar, es la librería la que se encarga de acceder al fichero a bajo nivel y nosotros sólo tenemos que indicar la propiedad a leer/escribir.

Ejemplo:

```

# Fichero de configuración
# Thu Feb 13 10:49:39 CET 2020
user=usuario
password=mypassword
server=localhost
port=3306

```

Escribir Fichero de Configuración

```

Properties config = new Properties(); // (import.java.util.*)
config.setProperty("user", miUsuario);
config.setProperty("password", miContrasena);
config.setProperty("server", elServidor);
config.setProperty("port", elPuerto);
try {config.store(new FileOutputStream("myconf.props"), "Fichero de config.");}
catch (IOException ioe) {ioe.printStackTrace();}

```

Si el fichero ya existe y solo queremos modificar una propiedad de configuración, podemos cargarlo previamente, hacer el set de la propiedad y guardar:

```

config.load(new FileInputStream("myconf.props"));
config.setProperty("server", nuevoServidor);
config.store(new FileOutputStream("myconf.props"), "Fichero de config.");

```

Leer ficheros de configuración

A la hora de leerlo, en vez de tener que recorrer todo el fichero como suele ocurrir con los ficheros de texto, simplemente tendremos que cargarlo e indicar de qué propiedad queremos obtener su valor con `getProperty(String)`.

```

Properties config = new Properties();
try {
    config.load(new FileInputStream("myconf.props"));
    usuario = config.getProperty("user");
    password = config.getProperty("password");
    servidor = config.getProperty("server");
    puerto = Integer.valueOf(config.getProperty("port"));
} catch (IOException ioe) {ioe.printStackTrace();}

```

Tanto para escribir como para leer este tipo de ficheros, hay que tener en cuenta que, al tratarse de ficheros de texto, toda la información se almacena como si de un String se tratara. Por tanto, todos aquellos tipos Date, boolean o incluso cualquier tipo numérico serán almacenados en formato texto. Así, habrá que tener en cuenta las siguientes consideraciones:

- Para el caso de las fechas, deberán ser convertidas a texto cuando se quieran escribir. Por el contrario, deben ser convertidas a LocalDate (por ejemplo, con su método estático `parse`) cuando se lean del fichero como String.
- Para el caso de los tipos boolean, podemos usar el método `String.valueOf(boolean)` para pasarlo a String cuando queramos escribirlos. En caso de que queramos leer el fichero y pasar el valor a tipo boolean podremos usar el método `Boolean.parseBoolean(String)`.
- Para el caso de los tipos numéricos (Integer, Float, Double) es muy sencillo ya que Java los convertirá a String cuando sea necesario al escribir el fichero. En el caso de que queramos leerlo y convertirlos a su tipo concreto, podremos usar los métodos `Integer.parseInt(String)`, `Float.parseFloat(String)` y `Double.parseDouble()`, según proceda.
- Las líneas que comienzan por '#' o '!' se interpretan como comentarios.
- En las parejas de propiedad valor, el separador entre la clave y el valor puede ser '=' o bien ':'.

15. Colecciones

Las colecciones representan grupos de objetos, denominados elementos que podemos tratar de una forma conjunta, por ejemplo, recorriéndolos o accediendo a ellos individualmente. Un ejemplo de colección, con la que hemos trabajado a lo largo de este curso son los `ArrayList`.

Podemos encontrar diversos tipos de colecciones, según si sus elementos tienen una posición determinada o no, o si se permite repetición de elementos o no. En todos los casos, el tamaño de las colecciones es dinámico, esto es, podremos añadir y eliminar los elementos que sea necesario.

Para usar estas colecciones haremos uso del Java Collections Framework (JCF), el cual contiene un conjunto de clases e interfaces del paquete `java.util` para gestionar colecciones de objetos. Una limitación de las colecciones es que solo se aplican a objetos, no a tipos primitivos.

Todas las colecciones implementan la interfaz `Collection`, en la que encontramos una serie de métodos que nos servirán para acceder a los elementos de cualquier colección de datos, sea del tipo que sea. Estos métodos generales son:

- boolean `add` (`Object o`): Añade un elemento (objeto) a la colección. Nos devuelve `true` si se ha añadido el elemento correctamente, o `false` en caso contrario.
- void `clear` () Elimina todos los elementos de la colección.
- boolean `contains` (`Object o`) Indica si la colección contiene el elemento (objeto) indicado.
- boolean `isEmpty()` Indica si la colección está vacía (no tiene ningún elemento).
- Iterator `iterator()` Proporciona un iterador para acceder a los elementos de la colección (lo veremos más en detalle)
- boolean `remove` (`Object o`) Elimina un elemento (objeto) de la colección, devolviendo `true` si dicho elemento estaba contenido en la colección, y `false` en caso contrario.
- int `size()` Nos devuelve el número de elementos que contiene la colección.
- Object [] `toArray()` devuelve la colección como un array de objetos. Para llamarlo, crearemos e instanciaremos previamente el array destino.

```
String [] cadenas = new String[MiColeccion.size()];
MiColeccion.toArray(cadenas);
```

Esta interfaz es muy genérica, y por lo tanto no hay ningún tipo de datos que la implemente directamente, sino que implementarán subtipos de ellas.

En Java las principales interfaces de las que disponemos para trabajar con colecciones son: `Set`, `List`, y `Map` y las clases más interesantes y que vamos a utilizar son las siguientes:

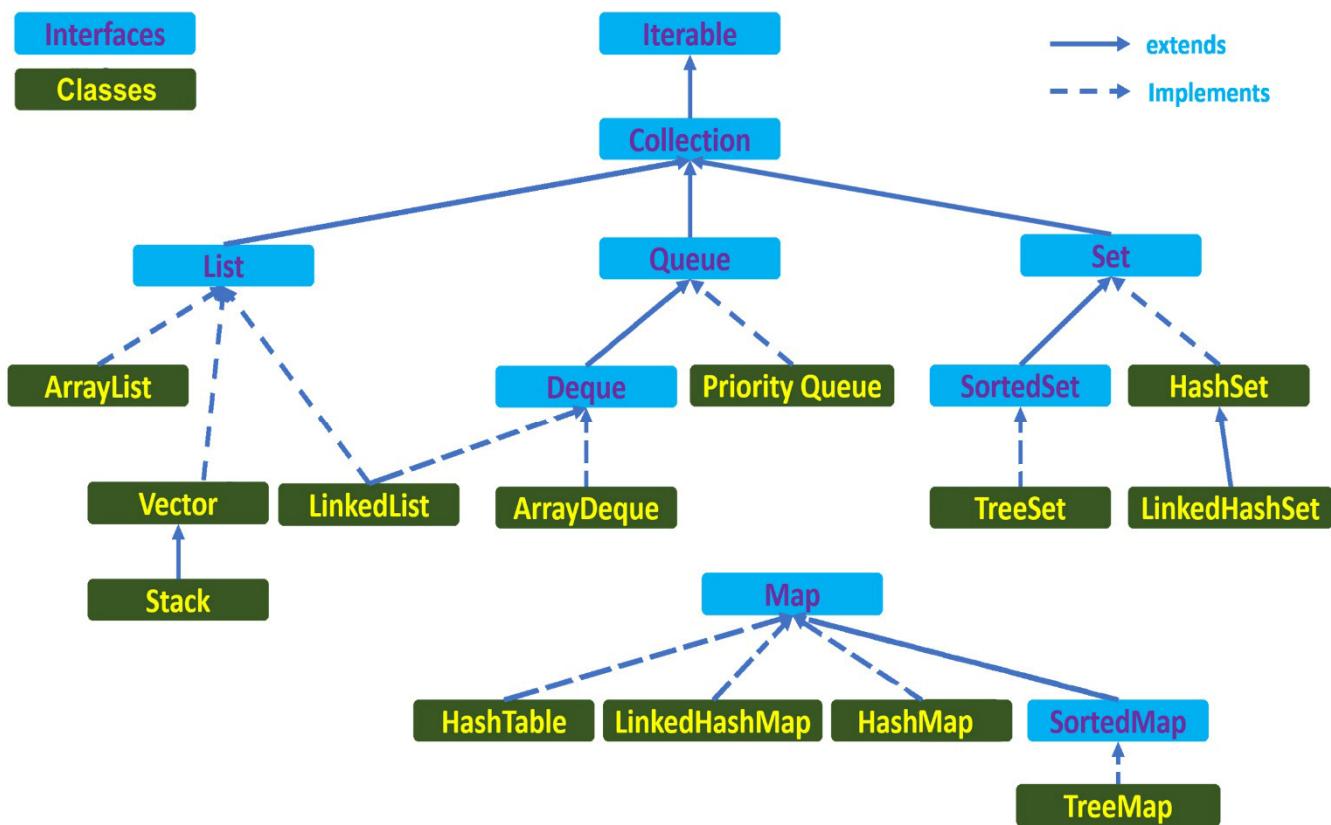
`List` → `ArrayList`, `Vector` y `LinkedList`

`Set` → `HashSet`, `TreeSet` y `LinkedHashSet`

`Map` → `HashMap`, `TreeMap` y `LinkedHashMap`

`Queue` → `Priority Queue`, `ArrayDeque`

A grandes rasgos, las clases que implementan `List` almacenan los elementos en cierto orden, admiten duplicados y permiten acceder a ellos por su posición. Las clases de `Set` tienen como principal característica que no admiten duplicados y, por último, los `Map` permite acceder por claves y valores en vez de por posiciones.

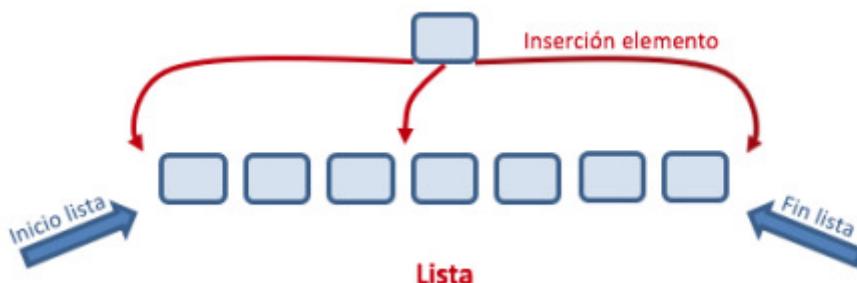


Como ya hablamos en el capítulo de `ArrayList`, hay métodos que necesitan comparar objetos para ver si son iguales (por ejemplo: `contains(Object)`) compara el elemento pasado con cada elemento de la colección y devuelve true si lo encuentra, `remove(Object)` borra el objeto si lo encuentra, etc.). y decíamos que esos métodos requerían redefinir `equals(Object)`. Eso ocurre con el resto de colecciones y, adicionalmente, deberemos redefinir `hashCode()` como explicaremos más adelante.

Sería inabordable hablar de todas las colecciones, con sus particularidades y métodos, vamos a ver las más importantes, dividiéndolas en las tres interfaces principales: `List`, `Set` y `Map`.

Interface List: Listas

Este tipo de colección se refiere a listas en las que los elementos de la colección tienen un orden, existe una secuencia de elementos. En ellas cada elemento estará en una determinada posición (índice) de la lista.



Incorpora métodos nuevos, no disponibles en otras colecciones, que hacen referencia a estos índices:

void add (int índice, Object obj): Inserta un elemento (objeto) en la posición de la lista dada por el índice indicado.

Object get (int índice): Obtiene el elemento (objeto) de la posición de la lista dada por el índice indicado.

int indexOf (Object obj): Nos dice cuál es el índice de dicho elemento (objeto) dentro de la lista. Nos devuelve -1 si el objeto no se encuentra en la lista.

Object remove (int índice): Elimina el elemento que se encuentre en la posición de la lista indicada mediante dicho índice, devolviéndonos el objeto eliminado.

Object set (int índice, Object obj): Establece el elemento de la lista en la posición dada por el índice al objeto indicado, sobrescribiendo el objeto que hubiera anteriormente en dicha posición. Nos devolverá el elemento que había previamente en dicha posición.

Podemos encontrar diferentes clases que implementan esta interfaz: ArrayList ya vista en capítulos anteriores, LinkedList y las obsoletas Vector y su hija Stack.

Clase ArrayList

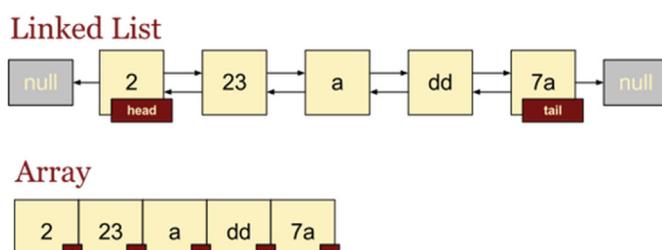
Esta clase, ya vista en capítulos previos, implementa una lista de elementos mediante un array de tamaño variable. Conforme se añaden elementos el tamaño del array irá creciendo si es necesario. El array tendrá una capacidad inicial, y en el momento en el que se rebasa dicha capacidad, se aumentará el tamaño del array.

Las operaciones de añadir un elemento al final del array (add), y de establecer u obtener el elemento en una determinada posición (get/set) tienen un coste temporal constante. Las inserciones y borrados tienen un coste lineal dependiente del número de elementos del array.

Clase LinkedList

Es como un ArrayList, pero los elementos están conectados con el anterior y el posterior permitiendo gestión tanto por el principio como el final de la lista. Cuando realicemos inserciones, borrados o lecturas en los extremos inicial o final de la lista el tiempo será constante, mientras que para cualquier operación en la que necesitemos localizar un determinado índice dentro de la lista deberemos recorrer la lista de inicio a fin, por lo que el coste será lineal con el tamaño de la lista.

Array vs. Linked List



Para aprovechar las ventajas que tenemos en el coste temporal al trabajar con los extremos de la lista, se proporcionan métodos propios para acceder a ellos en tiempo constante:

- void **addFirst**(Object o) / void **addLast**(Object o): Añade el objeto indicado al principio / final de la lista respectivamente.
- Object **getFirst()** / Object **getLast()**: Obtiene el primer / último objeto de la lista respectivamente.

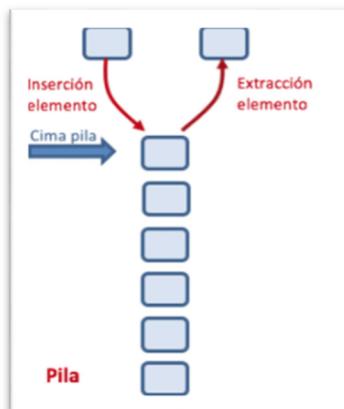
Object **removeFirst()** / Object **removeLast()**: Extrae el primer / último elemento de la lista respectivamente, devolviéndonos dicho objeto y eliminándolo de la lista.

Hemos de destacar que estos métodos nos permitirán trabajar con la lista como si se tratase de una **pila o de una cola**. En el caso de la pila realizaremos la inserción y la extracción de elementos por el mismo extremo, mientras que para la cola insertaremos por un extremo y extraeremos por el otro.

```
LinkedList <String> lista = new LinkedList<>();
if (lista.size()<100) {lista.addFirst("Pepe");}
String n = lista.removeLast();
if (!parking.isEmpty()) System.out.println (parking.getFirst());
```

También implementa la interfaz Queue, con operaciones típicas de colas:

- **poll()** / **pop()**: elimina y devuelve la cabeza de la pila (el primer elemento). Sería similar a **removeFirst()**.
- **push()**: añade un elemento en la cabeza de la pila (el primer elemento). Sería similar a **addFirst()**.
- **peek()**: devuelve la cabeza de la pila - como **poll()**/pop() – pero sin eliminarlos.



Las colecciones desde Java9 tienen un método estático de factoría (hace la función de constructor) llamado 'of', y se usa así:

```
List<String> list = List.of ("Alpha", "Bravo", "Charlie");
Set<String> set = Set.of ("Alpha", "Bravo", "Charlie");
```



Interface Set: Conjuntos

Los conjuntos son grupos de elementos en los que **no se permite ningún elemento repetido**. Consideramos que un elemento está repetido si tenemos dos objetos o1 y o2 iguales, comparándolos mediante el operador `o1.equals(o2)`.

Implementa los métodos vistos para la interfaz *Collection*. El método `add` añadirá el elemento si no

Interfaces y clases:

Como ya comentamos en el capítulo de polimorfismo e interfaces, se suelen crear variables o referencias del tipo de la interfaz y se instancian con una clase que implemente la interfaz. La limitación de esta técnica es que los atributos/métodos que podremos usar serán los definidos a la interfaz o nos veremos obligados a hacer castings.

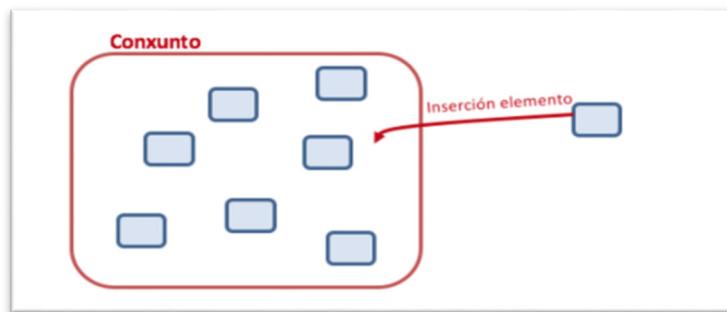
Así pues, será frecuente ver definidas colecciones de esta forma:

```
List <String> lista1 = new LinkedList<>();
List <String> lista2 = new ArrayList<>();
```

está repetido, devolviendo *true*, y no lo añadirá en caso de ya existir previamente en el conjunto, devolviendo *false*.

Un conjunto podrá contener a lo sumo un elemento null y podremos recorrerlo con un *for-each* o mediante un *Iterator*. Esta última forma la veremos en apartados posteriores.

Esta interfaz es implementada por distintas clases, de las que destacaremos: *HashSet*, *LinkedHashSet* y *TreeSet*.



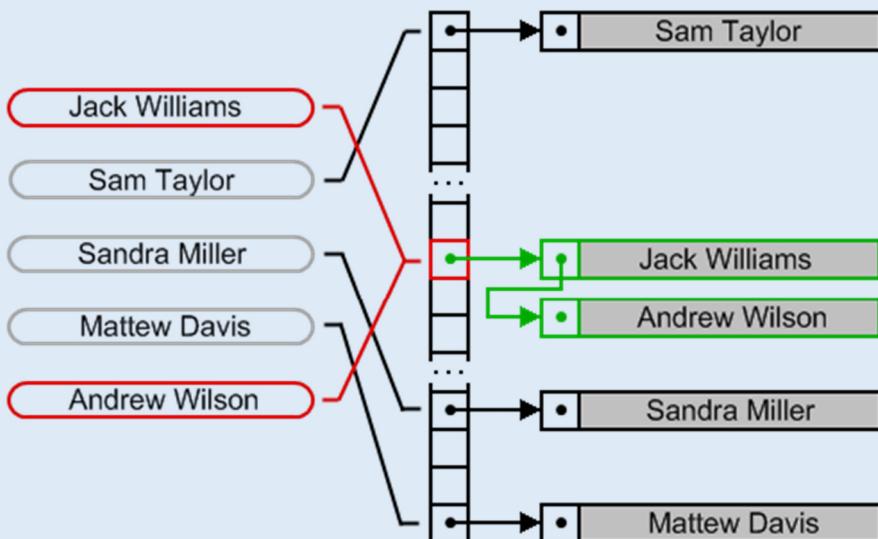
Clase HashSet

Los objetos de esta clase (como todas las que llevan la partícula hash en su nombre) se almacenan en una tabla de dispersión (hash).

En una **estructura hash** se almacena cada dato en una posición calculada a partir de una fórmula, una operación sobre sus datos. Así, los datos se dispersan y su acceso es más eficiente que en una estructura no ordenada.

Llevado a la vida real, imagina que tienes que guardar 30 DNI físicamente en una caja. Para acceder a ellos de forma rápida podrías guardarlo ordenadamente pero cada vez que te diesen uno nuevo, insertarlo ordenado sería laborioso (costoso en tiempo). Sabiendo que la letra de los DNI españoles puede tener 23 valores distintos, lo que podrías hacer, es tener 23 cajas, cada una etiquetada con una letra, y meter en cada una los DNI cuya letra sea la de la caja. Los DNI se "dispersarían" por las cajas. La búsqueda sería rápida y añadir nuevos DNI también. En este caso, obtener la letra es la función hash. Si tuvieras 1000 DNI cada caja tendría muchos DNI, lo que te ralentizaría las búsquedas, por lo que te interesaría tener más cajas, necesitarías una función hash distinta. Las "cajas" en Java se llaman "buckets" y que dos DNI vayan a la misma caja se llama "colisión". Lo óptimo es reducir el número de colisiones.

En el siguiente ejemplo, vemos como a partir del nombre se calcula una posición en la estructura, y si hay una colisión (*como Jack Williams y Andrew Wilson*), se enlazan unos con otros.



Como todas las clases que implementan la interface Set, **no admite duplicados**. Para identificar los duplicados **utilizará los métodos equals() y hashCode()**, que habrá de redefinir como ya comentamos previamente. Recordemos que la definición por defecto de estos métodos en la clase *Object*, compara la referencia de cada objeto, y dos objetos distintos aun con todos los atributos iguales, produciría un *false* en el *equals()* y distinto valor de *hashCode()*.

El método **hashCode()** deberá devolver un valor para cada objeto, de forma que dos objetos que consideremos que son iguales, tengan el mismo *hashCode()*, yendo a la par con *equals()*. Los generadores de código de los IDE nos ayudan en esta labor.

Siempre que redefinimos `equals()` hay que redefinir `hashCode()`, ya que si dos objetos son iguales según `equals()`, sus métodos `hashCode()` deben devolver lo mismo.

Ejemplo de `equals()` y `hashCode()` generado por Netbeans para una clase `Producto`, a partir de su atributo `nombre`. Las operaciones que se pueden ver en el método buscan la mayor “dispersión” de los elementos.

```
class Producto {
    String nombre;
    double precio;

    Producto (String n, double p){nombre=n; precio=p;}

    @Override
    public int hashCode() {
        int hash = 5;
        hash = 11 * hash + Objects.hashCode(this.nombre);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        final Producto other = (Producto) obj;
        if (!Objects.equals(this.nombre, other.nombre)) return false;
        return true;
    }
}
```

Netbeans usa una forma con números primos para lograr más dispersión, pero podríamos generarlo de forma más sencilla, así:

```
public int hashCode() { return Objects.hash(nombre); }
```

Por último, comentar que esta implementación no trabaja con índices y **no garantiza el orden de los elementos a través del tiempo**. Como ejemplo, podríamos pensar en una lista de la compra ya que no hay elementos repetidos y no nos importa el orden en el cual encontramos los elementos en la lista.

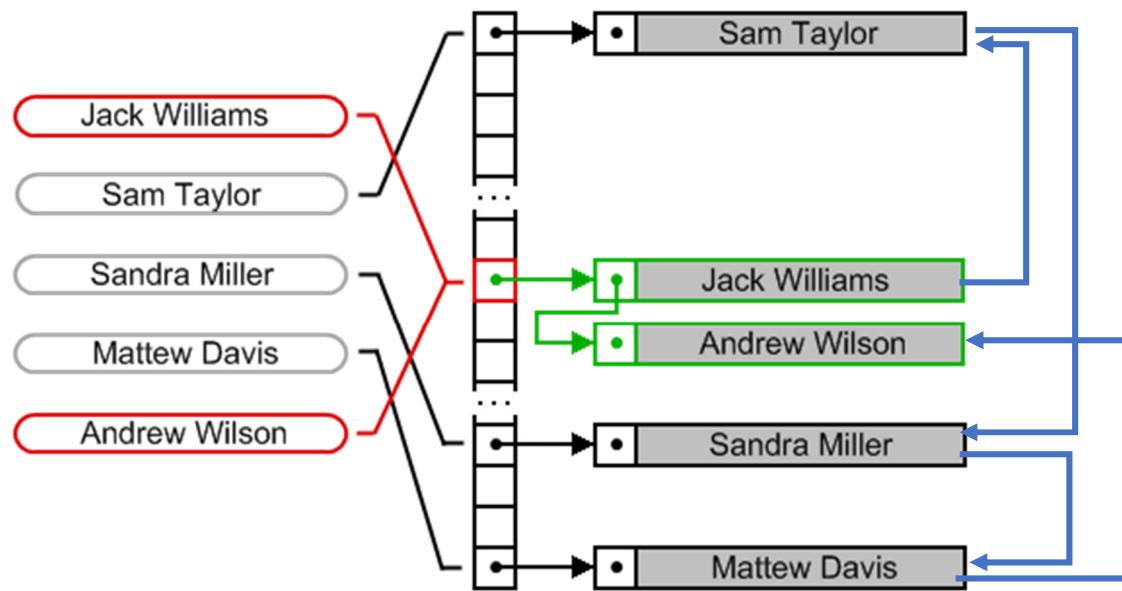
Ejemplo: **Lista de la compra** (no nos importa el orden, pero no hay repetidos).

```
HashSet<Producto> listaCompra = new HashSet<>();
if (listaCompra.add(new Producto("Platanos", 2.5)))
    System.out.println("Añadido correctamente");
else System.out.println("No se puede añadir. Repetido");
double total=0;
for (Producto p : listaCompra) total += p.precio;
```

Clase `LinkedHashSet`

Es similar al `HashSet`, pero los elementos, además del almacenamiento de tipo `hash`, están enlazados entre sí según el orden de inserción, lo que representa una mejora en rendimiento en caso de querer recorrer todo el conjunto, mientras que las operaciones básicas seguirán teniendo coste constante similar a `HashSet` (salvo la carga adicional que supone tener que gestionar los enlaces).

Si hacemos un recorrido for...each garantizamos que los recorrerá en el orden de inserción, algo que no podríamos garantizar en un HashSet.



En el gráfico, las flechas de la derecha representan los enlaces entre los elementos, por orden de inserción.

Clase TreeSet

Similar a un HashSet, no admite duplicados y está ordenado ascendentemente. Para ambas tareas (duplicados y ordenación) emplea el método `compareTo()` de la clase contenida en el Set o bien el `Comparator` indicado en el constructor. En un apartado posterior detallaremos estas dos interfaces: Comparable y Comparator y explicaremos ese método en detalle.

A diferencia de TreeSet, HashSet usaba `equals()` y `hashCode()` para evitar duplicados.

En secciones posteriores de este capítulo veremos en detalle `Comparable` y `Comparator`. Para su almacenamiento usa un árbol en vez de una tabla hash. Por lo tanto, el coste para realizar las operaciones básicas será logarítmico con el número de elementos que tenga el conjunto.

Diferencias entre conjuntos

Difference between HashSet and TreeSet

Property	HashSet	TreeSet
Ordering or Sorting	HashSet doesn't provide any ordering guarantee.	TreeSet provides ordering /sorting guarantee.
Comparison and Duplicate detection	HashSet uses <code>equals()</code> method for comparison.	TreeSet uses <code>compareTo()</code> method for comparison
Underlying data structure	HashSet is backed by hash table	TreeSet is backed by Red-Black Tree.
Null element	HashSet allows one null element	TreeSet doesn't allow null objects.
Implementation	Internally implemented using HashMap	Internally implemented using TreeMap.
Performance	HashSet is faster	TreeSet is slower for most of the general purpose operation e.g. add, remove and search

¿Cuándo usar cada uno de los tipos de conjuntos?

En todos los casos, usaremos conjuntos cuando no queramos repetidos ni tampoco queramos tener los elementos localizables y accesibles por un índice. Usaremos:

- **HashSet**: Si no importa el orden y queremos un acceso rápido.
- **LinkedHashSet**: Si nos hace falta mantener el orden de inserción, penalizando ligeramente la inserción.
- **TreeSet**: Si nos hace falta mantener el conjunto ordenado por un determinado criterio, penalizando el tiempo de inserción ya que hay que gestionar ese orden.

Interface Map: Mapas

Aunque muchas veces se hable de los mapas como una colección, en realidad podemos considerarlas con tal desde un punto conceptual, pero no heredan de la interfaz `Collection`.

Un mapa es un objeto que relaciona una clave (*key*) con un valor. Contendrá un conjunto de claves, y a cada clave se le asociará un determinado valor. En versiones anteriores este mapeado entre claves y valores lo hacía la clase `Dictionary`, que ha quedado obsoleta. Tanto la clave como el valor puede ser cualquier objeto.

Un ejemplo típico puede ser una estructura para almacenar País -> Cantidad de Habitantes, en el que el acceso es hace por "país", no por un índice, como tendríamos que hacer con un `ArrayList`.

Los métodos básicos para trabajar con estos elementos son los siguientes:

- **get (clave)**: Nos devuelve el valor asociado a la clave indicada
- **put (clave, valor)**: Inserta una nueva clave con el valor especificado. Si ya existía esa clave, la reemplaza con el nuevo valor y nos devuelve el valor que tenía antes dicha clave. Si no existía la clave, la añade y devuelve `null`.
- **remove (clave)**: Elimina una clave, devolviéndonos el valor que tenía dicha clave.
- **keySet ()**: Nos devuelve el conjunto *Set* de claves registradas
- **size ()**: Nos devuelve un entero con el número de parejas (clave,valor) registradas.
- **containsKey (clave)** devuelve true la clave *key* está dada de alta en el Map, false en caso contrario.
- Object **getOrDefault(clave, valorPorDefecto)** es una variante de get, de forma que, si no encuentra la clave pasada como parámetro, devuelve el valor por defecto.

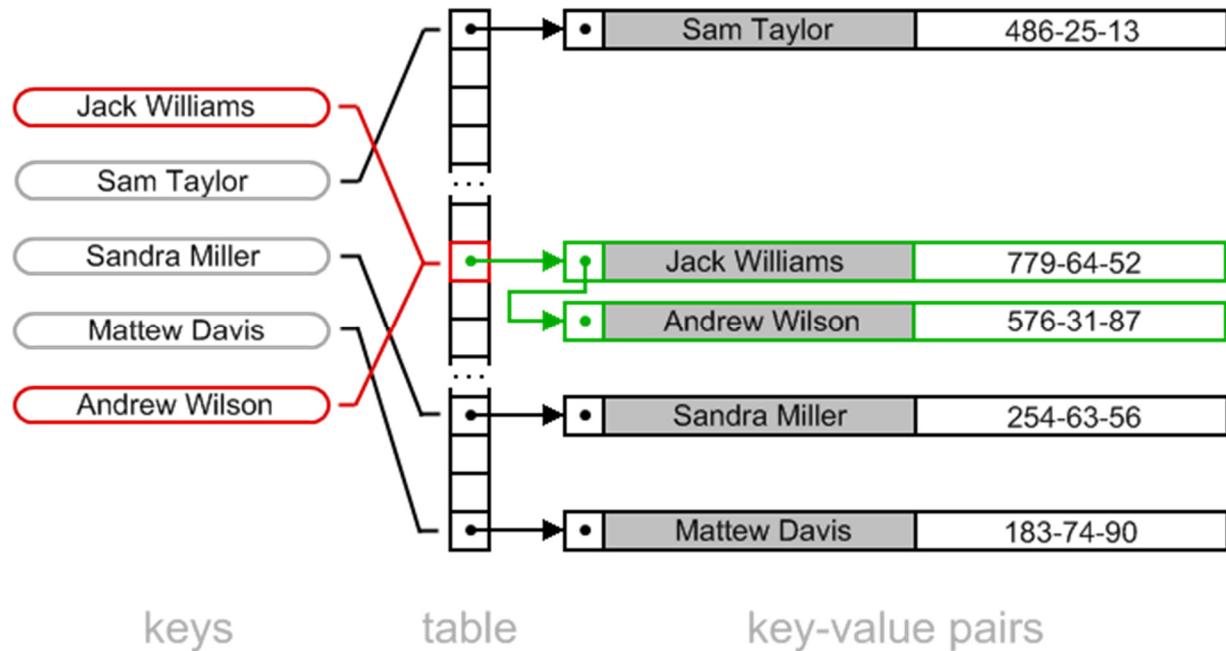
Encontramos distintas implementaciones de los mapas como son `HashMap` y `TreeMap`.

Cada elemento del mapa será una tupla de tipo `Map.Entry`, que dispondrá de los métodos **getKey ()** y **getValue ()**.



HashMap

Utiliza una tabla de dispersión para almacenar la información del mapa con la técnica de hashing que describimos previamente. Las operaciones básicas (*get y put*) se harán en tiempo constante siempre que se dispersen adecuadamente los elementos. Es coste de la iteración dependerá del número de entradas de la tabla y del número de elementos del mapa.



Características:

- No se garantiza que se respete el orden de las claves y permite una sola clave igual a *null* y múltiples valores iguales a *null*.
- Existe otra clase llamada '**HashTable**' similar a **HashMap**, que a diferencia de éste no admite ningún valor nulo, ni en la clave ni en el valor y además está sincronizada, lo que la hace aconsejable para aplicaciones multihilo.
- La clave y el valor tienen que ser clases, no tipos primitivos.
- La clase de la clave del mapa tiene que tener definido el método **hashcode()** ya que, a la hora de introducir una nueva clave, es la función que determina si ya existe (y por tanto la sustituirá) o no existe (y la añadirá). Las clases típicas como Integer, String, etc. ya lo tienen definido.

Ejemplo: País con cantidad de habitantes.

```
HashMap<String, Integer> mapaPaises = new HashMap<>();
mapaPaises.put("España", 47000000);
if (mapaPaises.containsKey("Portugal"))
    mapaPaises.put("Portugal", mapaPaises.get("Portugal") + 100000);
else mapaPaises.put("Portugal", 0);
```

Para recorrerlos podremos utilizar un **for...each** (o un Iterador como veremos más adelante).

```
for (String k : mapaPaises.keySet())
    System.out.println(k + " tiene " + mapaPaises.get(k) + " habitantes.");
```

TreeMap

Utiliza un árbol para implementar el mapa de forma que los elementos se encontrarán ordenados por orden ascendente de clave.

Ya que está ordenado incorpora métodos que no tienen sentido para un *HashMap* pero sí para un mapa ordenado:

firstKey(), **firstEntry()**, **lastKey()**, **lastEntry()** y otros similares.

Ver: https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html#method_summary

```
Entry <String, String> ent = miTreeMap.firstEntry();
System.out.println(ent.getKey() + " ==> " + ent.getValue());
```

Análogamente a lo que ocurre con *TreeSet*, la clase contenida en la clave del mapa debe implementar **Comparable** y desarrollar el método **compareTo()**, en cambio no necesita **equals()** ni **hashCode()**. Más adelante detallamos esta interfaz y el método.

El funcionamiento será igual *HashMap*, con un rendimiento peor en las operaciones básicas que el *HashMap* en inserción ya que tiene que mantener adicionalmente el árbol de ordenación.

Recorrer Colecciones e Iteradores

(Apartado opcional)

Hasta ahora hemos recorrido cualquier colección mediante un bucle *for...each*. En realidad, lo que estamos haciendo es utilizar implícitamente el **iterador** que tiene definida esa colección.

Un iterador se puede definir como una clase que usa para recorrer estructuras dinámicas como arrays o Colecciones. Los iteradores que usamos en el *for...each* están definidos como **clase internas** de la clase sobre la que iteran.

Una **clase interna** es una clase no estática que se define en el cuerpo de otra clase. La gran funcionalidad de estas clases internas es que pueden acceder a los atributos de la clase que las envuelve. Ejemplo:

```
public class Externa {
    // atributos clase externa
    // constructor y métodos

    public class Interna {
        // atributos clase interna
        // constructor y métodos de la clase interna
    } //fin clase interna

} // fin clase externa
```

Para crear una instancia de la clase interna hay que emplear esta fórmula (suponiendo constructores sin parámetros):

```
ClaseExterna e = new ClaseExterna();
ClaseExterna.ClaseInterna i = e.new ClaseInterna();
```

O bien todo junto:

```
ClaseExterna.ClaseInterna j = new ClaseExterna().new ClaseInterna();
```

Viendo la definición de clase interna del cuadro azul previo, crearemos una instancia del iterador de una colección así:

```
ArrayList<String> lista= new ArrayList<>();
Iterator<String> iterator = lista.iterator();
```

Los iteradores siempre proporcionan tres métodos:

- Object item = iterator.next() para obtener el siguiente elemento de la colección.
- iterator.hasNext() para saber si quedan más elementos que leer.
- iterator.remove(); para borrar el último elemento que hayamos leído.

Podremos utilizarlo para recorrer la colección e incluso eliminar aquellos que cumplan ciertas condiciones:

```
while (iterator.hasNext()) {
    String item = iterator.next();
    if(condicion_borrado(item)) iterator.remove();
}
```

Como ya mencionamos estas operaciones las podemos hacer con una sintaxis más sencilla empleando for...each. Hay que destacar que el for...each no permite el borrado que sí lo podemos hacer con el método remove() del iterador.

Tenemos también el método *descendingIterator()* para instanciar el iterador.

```
Iterator<String> iterator = list.descendingIterator();
while (iterator.hasNext()) System.out.println(iterator.next());
```

Crear nuestro propio iterador

En algunos casos nos puede interesar nuestro propio iterador, bien para una colección o bien para una estructura dinámica definida por nosotros. Vamos a distinguir dos situaciones: iteradores externos sobre Colecciones o iteradores internos de clases creadas por nosotros.

Iterador sobre Colección

Las colecciones ya tienen su propio iterador definido (el que usamos con el for...each). Si queremos hacer una iteración distinta lo haremos con un iterador de una clase separada. Este sería el código para hacer un recorrido de un ArrayList de Integer, solo para las posiciones pares: 0,2,4,....

```
class IteradorPosPares implements Iterator<Integer> {
    private List<Integer> lista;
    private int posActual=0;

    public IteradorPosPares(List<Integer> li) {
        lista = li;
    }
    public boolean hasNext() {
        return posActual < lista.size();
    }
    public Integer next() {
        Integer result = lista.get(posActual);
        posActual += 2;
        return result;
    }
}
```

para utilizarlo:

```
List<Integer> lista = Arrays.asList(1, 2, 3, 4, 5, 6);
Iterator it = new IteradorPosPares(lista);
while (it.hasNext()) System.out.println(it.next());
```

Iterador sobre una clase propia:

Para nuestras clases, podemos hacerlo como clase interna, y así hacer que funcione, no solo con la estructura de iterador que acabamos de ver, sino que también con un *for...each*.

Las condiciones para hacerlo son las siguientes:

- La clase sobre la que queremos iterar debe implementar la interfaz iterable. Las colecciones lo hacen también: `public interface Collection<E> extends Iterable<E> {}`
- La clase sobre la que queremos iterar debe definir el método *iterator()* de la interfaz Iterable. Ese método *Iterator()* tiene las siguientes características:
 - o Define una clase interna que implementa la interfaz Iterator.
 - o Esta clase interna debe escribir los métodos *hasNext* y *next()* de la interfaz Iterator.
 - o El método devolverá una instancia de esa clase interna.
- Una vez hecho esto, podremos usar un *for...each* para recorrer los elementos de nuestra clase.

Este es un ejemplo que define una colección llamada MiColección que tiene un array de 20 posiciones, y un iterador para recorrer solo los pares.

```
class MiColeccion implements Iterable <Integer> {

    private final static int TAM = 20;
    private Integer[] arr = new Integer[TAM];

    MiColeccion() {
        for (int i = 0; i < TAM; i++) arr[i] = i * 10;
    }

    @Override
    public Iterator<Integer> iterator() {
        Iterator<Integer> it = new MiIterador();
        return it;
    }
    class MiIterador implements Iterator <Integer> {

        private int sig = 0;

        @Override
        public boolean hasNext() {
            if (sig < TAM) return true;
            return false;
        }

        @Override
        public Integer next() {
            int val = arr[sig];
            sig += 2;
            return val;
        }
    }
}
```

Y lo usaríamos así:

```
MiColección m = new MiColección();
for (Integer i : m) System.out.println(i);
```

La clase `MiIterador` se puede definir como anónima, pero aún no sabemos cómo hacer eso, lo veremos en capítulos posteriores.

Recorrer mapas

Vamos a ver las 2 formas de recorrer un `HashMap`, usando `entrySet()` o usando `keySet()`.

Con `keySet()` lo que se obtiene como indica el nombre de la función son las claves y mediante un iterador se recorre la lista de claves. De esta forma si queremos saber también el valor de cada elemento tenemos que usar la función `get(clave)`.

```
HashMap<String,Float> listaProductos;
Iterator<String> productos = listaProductos.keySet().iterator();
while (productos.hasNext()) {
    String clave = productos.next();
    System.out.println(clave + " - " + listaProductos.get(clave));
}
```

La otra opción es `entrySet()` con la que se obtienen los valores y al igual que en el caso anterior con un iterador se recorre el `HashMap`, pero de esta forma hay que crear una variable de tipo `Map.Entry` para almacenar el elemento y con los métodos `getKey()` y `getValue()` de `Map.Entry` se obtienen los valores.

Se puede usar un iterador del tipo que vamos a coger en este caso `Map.Entry` de la misma forma que se hizo en el método anterior, o sino usar un iterador genérico y luego hacer el casting a `Map.Entry`.

```
HashMap<String, Float> listaProductos;
Iterator iterador = listaProductos.entrySet().iterator();
Map.Entry producto;
while (iterador.hasNext()) {
    producto = (Map.Entry) iterador.next();
    System.out.println(producto.getKey() + " - " + producto.getValue());
}
```

Con esta segunda forma es necesario usar una variable `Map.Entry`. El resultado es el mismo, aunque esta segunda forma es más eficiente puesto que mientras que en la primera solo obtenemos la clave y luego hay que buscar el contenido asociado con la función `get(clave)` con esta segunda forma ya tenemos ambos valores y no hay que realizar esa búsqueda adicional, aunque quizás la primera forma sea más sencilla.

Comparable y Comparator

Interfaz Comparable

`Comparable` es una interfaz que nos permite ordenar los elementos de una colección según los criterios que queramos. Esta interfaz debe ser implementada por la clase que forma la colección. La forma de ordenar (que atributos son los que definen el orden, orden ascendente o descendente, etc.) la marca el método abstracto `compareTo(Object o)`.

La clase debe redefinir este método de forma que devuelva un entero negativo si el objeto es menor que el pasado como parámetro, un entero positivo si el objeto es mayor que el pasado como parámetro, 0 si son iguales. Y ese será el criterio de ordenación.

Este método es usado por TreeSet y TreeMap (implementando *Comparable*) para mantener su orden interno y evitar duplicados, y también será el que se emplee para ordenar listas cuando hacemos *Collection.sort (milista);*

Este ejemplo ordena una lista de películas ascendente por su año.

```
public class Main {
    public static void main(String[] args) {
        List<Peli> lista = new ArrayList<>();
        lista.add(new Peli("Episode 7: The Force Awakens", 2015));
        lista.add(new Peli("Episode 4: A New Hope", 1977));
        lista.add(new Peli("Episode 1: The Phantom Menace", 1999));

        Collections.sort(lista);
        for (Peli p: lista) System.out.println(p);
    }
}

class Peli implements Comparable {
    public String nombre;
    public int año;

    @Override
    public int compareTo(Object o) {
        Peli m = (Peli) o;
        return this.año - m.año;
    }

    public Peli(String n, int a) {
        this.nombre = n; this.año = a; }

    @Override
    public String toString () {
        return "("+this.año +") " + this.nombre;
    }
}
```

¡Ojo! Un TreeSet de Peli estaría ordenado también por año, pero por otra parte no habría duplicados por año...algo un poco extraño, así que *compareTo()* debería ser análogo a *equals()* y *hashCode()* y por ejemplo, en este caso, trabajar sobre 'nombre' y no 'año'.

El problema de la interfaz comparable es si queremos ordenar la colección por distintos criterios, es decir imaginemos que queremos que la lista de películas en algún momento esté ordenada por el año, pero en otros casos, por el nombre. La solución la proporciona la interfaz Comparator.

Interfaz Comparator

Aunque la misión de Comparator es igual a la de Comparable, lo hace mediante una clase externa, no implementada mediante la clase objeto de ordenación. Por lo tanto, podemos crear tantas clases que la implementen, cada una con sus criterios.

Esa es la gran diferencia: con Comparable solo podemos tener un criterio de ordenación para una clase (criterio definido en la propia clase), y con Comparator podemos ordenar por distintos criterios, (criterios definidos en clases independientes) y usaremos el que queramos en cada ocasión.

Los pasos a seguir serán:

1.- Crear las clases que implementan **Comparator** y redefinen **compare()** con los mismos criterios que expusimos previamente para **compareTo()**.

2.- **Crear instancias** de las clases creadas en el paso anterior.

3.- Llamar a **Collection.sort()** incluyendo como segundo parámetro la instancia que queramos que marque la ordenación.

Siguiendo el ejemplo anterior, si queremos mostrar las películas primero ordenadas por año y luego por nombre, haríamos lo siguiente:

```

public class Main { public static void main(String[] args) {
    List<Peli> lista = new ArrayList<>();
    lista.add(new Peli ("Episode 7: The Force Awakens", 2015));
    lista.add(new Peli ("Episode 4: A New Hope", 1977));
    lista.add(new Peli ("Episode 1: The Phantom Menace", 1999));

    ComparaNombre compNombre = new ComparaNombre ();
    Collections.sort(lista, compNombre);
    for (Peli p: lista) System.out.println("Por nombre: " + p);

    ComparaAño compAño = new ComparaAño ();
    Collections.sort(lista, compAño);
    for (Peli p: lista) System.out.println("Por año: " + p);
}

class Peli {
    public String nombre;
    public int año;

    public Peli(String n, int a) {
        this.nombre = n; this.año = a; }

    @Override
    public String toString () {
        return "("+this.año+") " + this.nombre;
    }
}

class ComparaAño implements Comparator {
    public int compare(Object o1, Object o2) {
        Peli p1 = (Peli) o1; Peli p2 = (Peli) o2;
        return p1.año - p2.año;
    }
}

class ComparaNombre implements Comparator {
    public int compare(Object o1, Object o2) {
        Peli p1 = (Peli) o1; Peli p2 = (Peli) o2;
        return p1.nombre.compareToIgnoreCase(p2.nombre); (*)}
}
}

```

(*) **compareToIgnoreCase()** es un método de **String** que compara la instancia que invoca el método con la cadena pasada como parámetro sin tener en cuenta mayúsculas o minúsculas. Devuelve negativo si la instancia es menor alfabéticamente que el parámetro, 0 si iguales, o positivo si la instancia es mayor, o sea, que encaja perfectamente con lo que queremos hacer en el método **compare** de **Comparator**.

Fíjate que para ordenar por Comparable hay que modificar la clase que queremos que sufra la ordenación: hay que decir que implementa Comparable y hay que sobreescribir el método `compareTo()`. Esto no ocurre con Comparator, no hace falta modificar la clase, está todo en las clases que implementan Comparator. **Este puede ser otro motivo para emplear Comparator en vez de Comparable, si no podemos o no queremos modificar la clase que va a sufrir la ordenación.**

Se podrá ver la clase de que implementa Comparator de forma anónima, sin crear la clase como tal, escribiendo su código como parámetro de `Collections.sort()`, con una sintaxis más compacta:

```
Collections.sort(lista, new Comparator<Peli>() {
    @Override
    public int compare(Peli p1, Peli p2) {
        return p1.año-p2.año;
    }
});
```

Explicaremos esta estructura más adelante, en el apartado de "Clases anónimas".

Últimas Consideraciones

Polimorfismo en Colecciones

Como vimos en temas anteriores, podemos crear instanciar una clase sobre una variable de una superclase. Lo mismo ocurre con las interfaces y por lo tanto con las colecciones. Por ejemplo, todas las clases de listas implementan la interfaz List. Por lo tanto, en un método que acepte como parámetro un objeto de tipo List podremos utilizar cualquier tipo que implemente esta interfaz, independientemente del tipo concreto del que se trate.

Es por lo tanto recomendable hacer referencia siempre a estos objetos mediante la interfaz que implementa, y no por su clase concreta. De esta forma posteriormente podríamos cambiar la implementación del tipo de datos sin que afecte al resto del programa. Lo único que tendremos que cambiar es el momento en el que se instancia.

```
List<Peli> lista = new ArrayList<>();
//List<Peli> lista = new LinkedList<>();
lista.add(new Peli ("Episode 4: A New Hope", 1977));
```

Conversión entre Colecciones

Para hacer conversiones entre uno tipos de colecciones y otros siempre podemos hacerlo "a mano", esto es, un bucle for...each o un iterador e insertar cada elemento de la colección origen en la destino, pero existen ciertas conversiones que podemos hacer de forma más sencilla.

- Convertir un Mapa en dos ArrayList, uno para claves y otro para los valores:

```
HashMap<String, Integer> mapaPaises = new HashMap<>();
ArrayList<String> keyList = new ArrayList<String>( mapaPaises.keySet());
ArrayList<Integer> valueList = new ArrayList<Integer>( mapaPaises.values());
```

- El proceso inverso lo podremos hacer también, pero con Streams y una función Lambda como veremos al final de este manual.
- Crear una lista (*interfaz List*) a partir de un array con:

```
List<String> lista = Arrays.asList(array);
```

- Crear un conjunto desde una lista

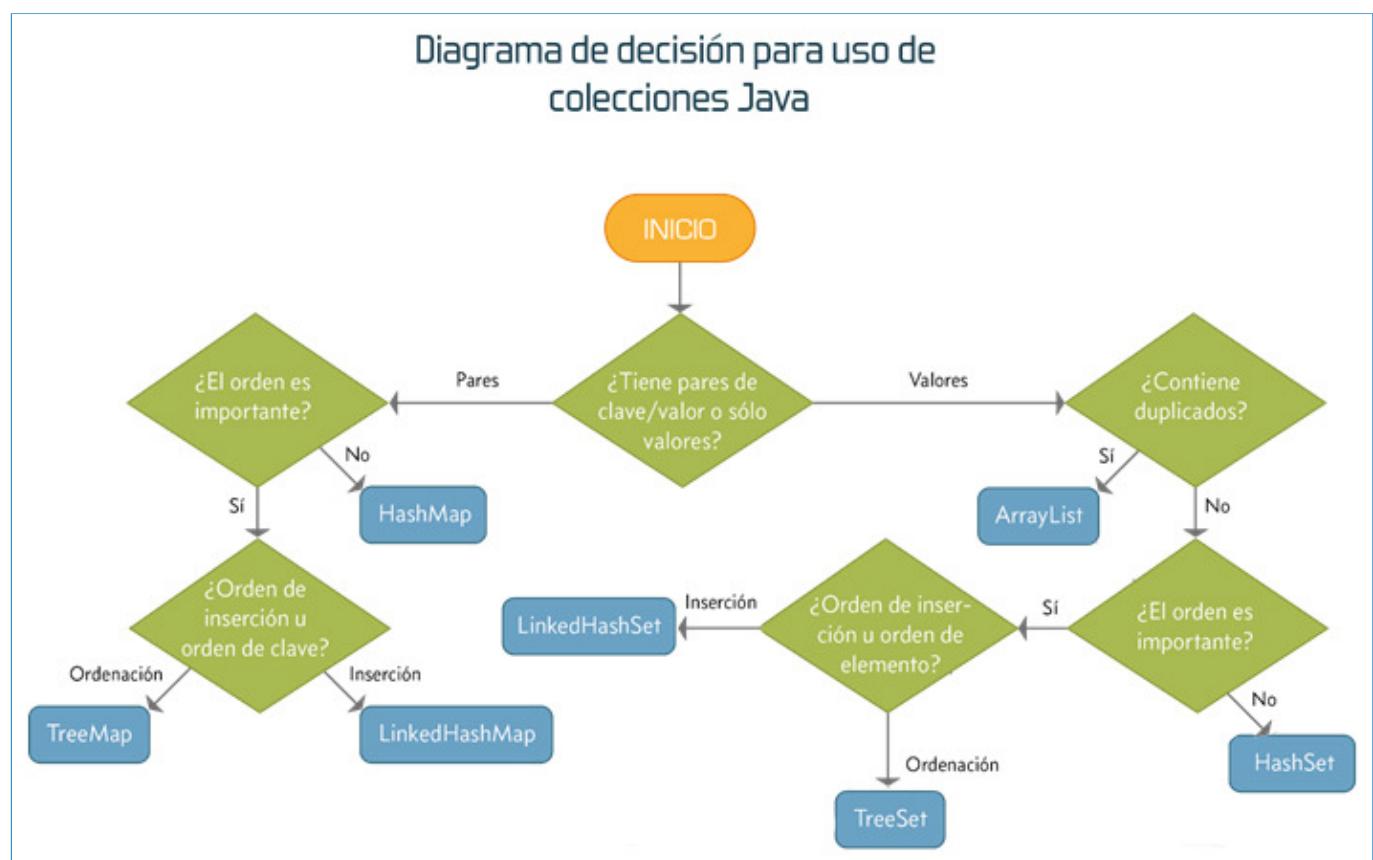

```
Set conjunto = new HashSet (lista);
```

Esto es una forma rápida de ver si una lista contiene duplicados. Hacemos un conjunto con una instrucción como la anterior y como un Set no puede tener duplicados, la comparación de `size()` de las dos colecciones nos dirá si hay duplicados o no.

Elección de Colección

A la hora de elegir la colección adecuada para resolver un problema emplearemos diferentes criterios. El primero será la estructura necesaria: admite duplicados o no, queremos que tengan un índice, está ordenado, es de tipo par clave/valor, etc.

El siguiente gráfico puede ser de ayuda para la elección.



Con respecto al rendimiento en las distintas operaciones, este podría ser un resumen:

- Las operaciones mediante *hash* son muy eficientes en operaciones básicas (insertar, borrar y buscar) sobre todo si la función de dispersión es adecuada. La iteración a través de sus elementos es más costosa ya que no mantiene ningún enlace entre los elementos.
- Las colecciones que incorporan *link* penalizan un poco las operaciones básicas ya que tienen que gestionar los enlaces entre los elementos para mantener el orden de inserción, pero mejoran las iteraciones respecto al hash.
- Los *Tree* incorporan un sistema de enlaces más complejo y solo son adecuadas cuando necesitamos ordenación.

16. Orientación a Objetos Avanzada

En el primer trimestre “conocimos” las clases y objetos, con su estructura: atributos, constructores y métodos. En el segundo trimestre profundizamos en la herencia, polimorfismo, clases abstractas e interfaces.

Este capítulo cierra todo el contenido del curso respecto a la orientación a objetos, tratando distintos aspectos importantes a modo de “cajón de sastre”, pero que hemos dejado para este último tramo del curso ya que hay que tener bien afianzados los conceptos previos.

Records

Los *records* son una nueva funcionalidad introducida en Java 14 para definir clases inmutables. Por clase inmutable entendemos aquella que, una vez asignados valores a sus atributos en el constructor, estos no pueden ser modificados.

Si bien podemos definir clases inmutables con *class*, los records lo hacen de una forma mucho más concisa, generando en una sola línea la clase con sus atributos, con visibilidad privada e incluyendo de forma implícita los métodos: *equals*, *hashCode*, *toString* y los *getters* sobre los atributos, estos últimos con el mismo nombre que cada atributo (no dispone de *setters* por ser inmutable).

A nivel de almacenamiento, al igual que las clases públicas, se guardan en archivos .java con el mismo nombre que el record. Por ejemplo, el archivo Punto.java contendría simplemente:

```
public record Punto(double x, double y) {}
```

Ahora podríamos instanciar variables de tipo *Punto* y emplear sus métodos predefinidos:

```
Punto punto1 = new Punto(2.5, 3.2);
Punto punto2 = new Punto(9.0, 8.12);
System.out.println("Coordenada X del primer punto: " + punto1.x());
System.out.println("Datos del segundo punto: " + punto2.toString());
if (punto1.equals(punto2))
    System.out.println("Ambos puntos tienen las mismas coordenadas");
```

Aunque ya incorpora los métodos que acabamos de ver, podríamos definirle otros métodos:

```
public record Punto(double x, double y) {
    // Constructor personalizado
    public Punto { if (x < 0 || y < 0)
        throw new IllegalArgumentException("Error: coordenadas negativas");
    }

    // Método adicional para calcular la distancia a otro punto
    public double distancia(Punto otro) {
        double deltaX = this.x - otro.x;
        double deltaY = this.y - otro.y;
        return Math.sqrt(deltaX * deltaX + deltaY * deltaY);
    }
}
```

Wrapper Classes (Clases Envoltorio)

Hemos visto que en Java cualquier tipo de datos es un objeto, excepto los tipos de datos básicos: *boolean, int, long, float, double, byte, short, char*.

Cuando trabajamos con colecciones de datos los elementos que contienen éstas son siempre objetos, por lo que en un principio no podríamos insertar elementos de estos tipos básicos. Para hacer esto posible tenemos una serie de objetos que se encargarán de envolver a estos tipos básicos, permitiéndonos tratarlos como objetos y por lo tanto insertarlos como elementos de colecciones. Estos objetos son los llamados **wrappers**, y las clases en las que se definen tienen nombres similares al del tipo básico que encapsulan, con la diferencia de que comienzan con mayúscula: *Boolean, Integer, Long, Float, Double, Byte, Short, Character*.

Autoboxing: Es una característica que evita al programador tener que establecer correspondencias manuales entre los tipos simples (*int, double, etc*) y sus correspondientes wrappers o tipos complejos (*Integer, Double, etc*). Podremos utilizar un *int* donde se espere un objeto complejo (*Integer*), y viceversa.

Por ejemplo, si tenemos un objeto *Integer* llamado *io*, la operación *io++*; es válida.

Estas clases, además de servirnos para encapsular estos datos básicos en forma de objetos, nos proporcionan una serie de constantes y métodos e información útiles para trabajar con estos datos. Nos proporcionarán métodos por ejemplo para convertir cadenas a datos numéricos de distintos tipos y viceversa, así como información acerca del valor mínimo y máximo que se puede representar con cada tipo numérico.

Ejemplos:

```
Integer.MIN_VALUE, Integer.MAX_VALUE, Float.MIN_VALUE, Float.MAX_VALUE
Float.SIZE (tamaño en bits, no en Bytes)
Float.NEGATIVE_INFINITY,
```

Métodos

Estas clases disponen de diversos métodos muy útiles:

valueOf(): Para crear un objeto de tipo wrapper no se usan constructores, se recomienda que use el método **valueOf()**, que es un miembro estático de todas las clases wrappers y todas las clases numéricas admiten formas que convierten un valor numérico o una cadena en un objeto. Por ejemplo, aquí hay dos formas compatibles con *Integer*:

```
Integer oI1 = Integer.valueOf(23);
Integer oI2 = Integer.valueOf("123"); //throws NumberFormatException si error.
```

Y tomando como ejemplo *Integer*, pero los hay análogos para todos los wrappers:

parselnt (String s): devuelve un entero obtenido a partir de la cadena 's' (es estático). Acepta como segundo parámetro la base, pudiendo así convertir de números hexadecimales a enteros.

byteValue(): devuelve el valor del entero como byte.

doubleValue() : devuelve el valor del entero como double (también **floatValue()**, **longValue()**...).

int compareTo(int i) Compara el entero con 'i' devolviendo 0 si ambos son iguales, valor negativo si el Integer que invoca al método es menor que 'i', y positivo si es mayor.

`int compare(int num1, int num2)` (estático). Compara los dos enteros pasados como parámetros devolviendo valores análogos a `compareTo()`

`toHexString(int i):` (estático) Convierte a cadena hexadecimal el entero pasado como parámetro. Hay también `toOctalString()` and `toBinaryString()`.

Expresiones Regulares

Una expresión regular es un patrón que describe a una cadena de caracteres. Cuando usamos la expresión `*.doc` para buscar archivos en el sistema operativo, estamos empleando un patrón que representa a todos los archivos con extensión *doc*, en cierto modo esto sería una expresión regular sencilla.

Una expresión regular nos servirá para buscar patrones en una cadena de texto, por ejemplo, comprobar que una fecha tiene la estructura correcta (dos dígitos, luego un punto, después otros dos dígitos para el mes, otro punto y cuatro dígitos para el año), también para comprobar que un email está bien escrito (letra y números, luego una arroba, más texto, un punto final y otro texto más), etc. Para cada uno de estos casos existe una expresión regular que los representa.

Hay que destacar que la expresión regular solo va a chequear la sintaxis, es decir el formato, pero no va a validar la lógica semántica, por ejemplo, en una fecha no va a verificar si un 31 de abril es correcto o no.

Uso

Las expresiones regulares se aplican a distintas tareas, aunque la más habitual será chequear si una cadena cumple con un determinado patrón representado por la expresión regular. En Java utilizaremos las clases `Matcher` y `Pattern` del paquete `java.util.regex` (incluyendo la excepción `PatternSyntaxException` que se producirá si la sintaxis de la expresión no es válida).

Veamos con un ejemplo las tres formas de comprobar si una cadena cumple una expresión regular. En este caso comprobaremos si la cadena `txt` cumple con el formato especificado por `regexp`.

```
boolean resultado1, resultado2, resultado3;
String txt="12345678A";
String regexp="\d{8}[A-Z]";

//Opción 1
resultado1 = txt.matches (regexp);

//Opción 2
resultado2 = Pattern.matches(regexp, txt);

//Opción 3
Pattern p = Pattern.compile (regexp);
Matcher m = p.matcher (txt);
resultado3 = m.matches();
```

En el ejemplo se está verificando si la cadena tiene 8 dígitos numéricos y a continuación una letra mayúscula (como debería ser un DNI). En los tres casos devuelve ‘true’ si se cumple y false en otro caso. Con cualquiera de las tres formas descritas, en caso de que se cumpla la expresión, el resultado será *true* y si no se cumple será *false*, por eso está asignándose el resultado a variables *boolean*. El último de los formatos tiene funcionalidades adicionales, que veremos en próximos apartados.

Elementos de una expresión regular

Veamos algunos ejemplos más:

- **Verificar una fecha** en formato dd/mm/aaaa:

```
String regexp = "\d{1,2}/\d{1,2}/\d{4}";
// Lo siguiente devuelve true
Pattern.matches(regexp, "11/03/2020");
Pattern.matches(regexp, "11/3/2020");

// Los siguientes devuelven false
Pattern.matches(regexp, "11/12/14");
Pattern.matches(regexp, "11//2014");
```

Explicación de la expresión regular: \d representa un dígito numérico y entre llaves indicamos el número de elementos. Por lo tanto \d{1,2} significa 1 ó 2 dígitos. Luego vendría la "/" separadora entre día y mes que tiene que llevar la fecha, etc.

Siempre que en Java metemos un carácter \ dentro de una cadena delimitada por "", debemos "escapar" esta \ con otra \, por ello todas nuestras \ en la expresión regular, se convierten en \\ en nuestro código.

- **Verificar un DNI** de una forma más precisa que el ejemplo inicial (entre 5 y 8 dígitos numéricos más una letra mayúscula que no sea I, O, U):

```
String regexp = "\d{5,8}[A-HJ-NP-TV-Z]";
```

Explicación de la expresión regular: \d representa un dígito numérico y entre llaves indicamos el número de veces, entre 5 y 8 veces. Entre corchetes ponemos rangos de valores posibles, si valiese cualquier letra sería [A-Z] pero como queremos excluir I,O,U hacemos esos rangos A a H, J a N, P a T y V a Z. Si admitiésemos también minúsculas habría que incluirlas:

```
[A-HJ-NP-TV-Za-hj-np-tv-z]
```

- **Verificar un email:** no existe una expresión regular para email que sea 100% fiable, puesto que hay muchos formatos válidos de email y muy complejos. Aquí vamos a usar una expresión regular más o menos sencilla: `[\^@]+@[^\^@]+\.\[a-zA-Z]{2,}`.

El símbolo ^ representa la negación, por lo tanto ^@ es cualquier carácter menos @. El + indica una o más veces así: `[\^@]+` uno o más caracteres que no sean @.

Luego iría la @ del email y otra vez uno o más caracteres que no sean arroba. Finalmente, un punto \.

- `[\^@]+` cualquier carácter que no sea @ una o más veces seguido de
- `\.` un punto seguido de
- `[a-zA-Z]{2,}` dos o más letras minúsculas o mayúsculas

```
String emailRegexp = "[\^@]+@[^\^@]+\.\[a-zA-Z]{2,}";
```

Podemos, a partir de los ejemplos anteriores, hacer un resumen de los elementos más frecuentes:

- `\w` una palabra (estaría delimitada por espacios en blanco, punto, coma, etc)
- `\d \s` un dígito, un espacio en blanco
- `[abc]` o 'a' o 'b' o 'c' (un solo carácter).
- `[^abc]` ni 'a' ni 'b' ni 'c'.
- `[a-g]` un carácter entre 'a' y 'g'.
- `[A-Dh-k]` un carácter entre 'A' y 'D' o bien entre 'h' y 'k'.
- `\. * \\` caracteres especiales: el punto, el asterisco y la barra.
- `a* a+ a?` *: cero o más veces, +: una o más veces, ?: cero o una vez.
- `a{5} a{5,8} a{5, }` {5} cinco veces exactas, {5,8} entre 5 y 8, {5, }5 ó mas.
- `ab | cd` o bien 'ab' o bien 'cd'

Además de estas disponemos de los paréntesis para agrupar elementos, por ejemplo:

`([a-f]*){3}` representa una letra entre la 'a' y la 'f' seguida de un asterisco, todo ello tres veces, de forma que `a*a*c*` cumpliría la expresión, pero `ab**a*` no.

Por último, a veces nos encontraremos los caracteres de comienzo `^` y final de expresión `$` para delimitarla y que coincida la expresión con la totalidad de la cadena y no solo con un subconjunto de la misma, por ejemplo: `^([a-f]*){3}$` aunque en los métodos que hemos empleado no tiene relevancia ya que no responde ante subconjuntos, digamos que ambos caracteres van implícitos.

En <https://docs.oracle.com/javase/10/docs/api/java/util/regex/Pattern.html> disponemos de una lista completa de los parámetros disponibles para expresiones regulares y también adjuntamos una *cheatsheet* resumen al final de este apartado.

En <http://regexp.com> podemos hacer pruebas y practicar con distintos ejemplos de una forma rápida e independiente del lenguaje. La página ofrece una explicación de cada paso que sigue en la interpretación de las expresiones que escribimos.

The screenshot shows the regexp.com interface. On the left, there's a "Cheatsheet" panel with various regex elements and their descriptions. In the center, the "Expression" field contains the regex pattern `/\d{5,8}[A-Za-z]/g`. Below it, the "Text" tab is selected, showing the input string `12345678A`. The "Tests" tab is also visible. At the bottom of the interface, there's a "Tools" section with detailed explanations for different regex components like digits, quantifiers, character sets, and ranges.

Extraer partes de la cadena

Una vez validado un patrón, podemos extraer parte de ese patrón, por ejemplo, las cifras de la fecha (día, mes y año). Cambiemos el ejemplo. Queremos extraer los sumandos y el resultado de una cadena así "`xxx+yy=zzzz`" donde 'x', 'y' y 'z' representan dígitos y pueden ser en cualquier número.

Con `\d+` indicamos uno o más dígitos. La expresión regular para ver si una cadena cumple ese patrón puede ser `\d+\+\d+=\d+`. Puesto que el `+` tiene un sentido especial en los patrones -indica uno o más-, para ver si hay un "+" en la cadena, tenemos que "escaparlo" con el `\` delante.

Las partes que queramos extraer, debemos meterlas entre paréntesis. Así, la expresión regular quedaría `(\d+)\+(\d+)=(\d+)`.

Utilizaríamos el método `find()` del matcher y `group(num.grupo)` para obtener cada parte, pero antes hay que verificar que se cumple el patrón, ya que si no se cumple ocurrirá la excepción: `IllegalStateException`. Entonces ejecutaremos primero el método `matches()` y a continuación `reset()` (para volver el Matcher al inicio de la cadena). Sería así:

```
Pattern patron = Pattern.compile("(\\d+)\\+(\\d+)=(\\d+)");
Matcher matcher = patron.matcher("990+22=1012");
if (matcher.matches()) {
    matcher.reset(); //verificar patrón
    matcher.reset(); //volver matcher al inicio
    matcher.find(); //obtener grupos
    System.out.println(matcher.group(1)); //muestra grupo 1: 990
    System.out.println(matcher.group(2)); //muestra grupo 2: 22
    System.out.println(matcher.group(3)); //muestra grupo 3: 1012
}
```

Expresiones regulares en otros métodos

Las expresiones regulares se pueden usar en otros métodos ya vistos previamente, como `replaceAll()` o `split()` ambos de la clase de String. El siguiente ejemplo separa los términos de una ecuación, esto es, bloques de texto separados o bien por un '+' o bien por un '-'.

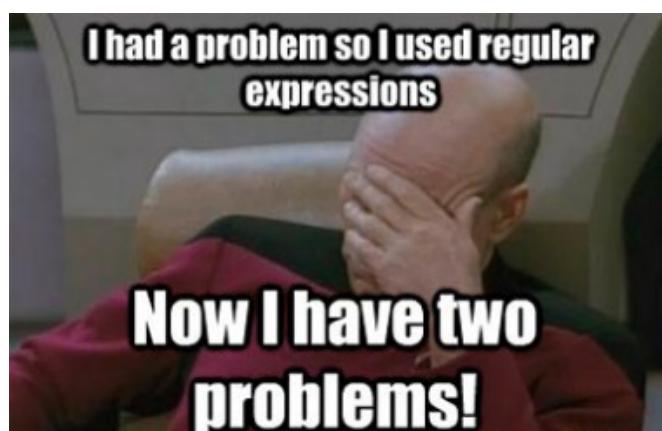
```
String ecuacion = "3x^1+2x^2-4x^3";
String[] terminosEcuacion = ecuacion.split("[+-]");
for (String tE : terminosEcuacion) System.out.println(tE);
```

Mostrando: `3x^1` `2x^2` `4x^3`

Este otro ejemplo, elimina todos los espacios en blanco.

```
String txt = "Esto      es una prueba de expresiones regulares";
System.out.println(txt.replaceAll("\\s+", ""));
```

Mostrando: `Estoesunapruebadexpresionesregulares`



Cheatography

Regular Expressions Cheat Sheet

by Dave Child (DaveChild) via cheatography.com/1/cs/5/

Anchors	Assertions			Groups and Ranges
^ Start of string, or start of line in multi-line pattern	?= Lookahead assertion			. Any character except new line (\n)
\A Start of string	?!= Negative lookahead			(a b) a or b
\$ End of string, or end of line in multi-line pattern	?<= Lookbehind assertion			(...) Group
\Z End of string	?!= or ?<! Negative lookbehind			(?:...) Passive (non-capturing) group
\b Word boundary	?> Once-only Subexpression			[abc] Range (a or b or c)
\B Not word boundary	?() Condition [if then]			[^abc] Not (a or b or c)
\K Start of word	?() Condition [if then else]			[a-q] Lower case letter from a to q
\E End of word	?# Comment			[A-Q] Upper case letter from A to Q
Character Classes				
\c Control character	*	0 or more	{3}	Exactly 3
\s White space	+	1 or more	{3,}	3 or more
\S Not white space	?	0 or 1	{3,5}	3, 4 or 5
Add a ? to a quantifier to make it ungreedy.				
Quantifiers				
Escape Sequences				
\ Escape following character				Ranges are inclusive.
\Q Begin literal sequence				g Global match
\E End literal sequence				i * Case-insensitive
"Escaping" is a way of treating characters which have a special meaning in regular expressions literally, rather than as special characters.				
POSIX				
[:upper:] Upper case letters				m * Multiple lines
[:lower:] Lower case letters				s * Treat string as single line
[:alpha:] All letters				x * Allow comments and whitespace in pattern
[:alnum:] Digits and letters				e * Evaluate replacement
[:digit:] Digits				U * Ungreedy pattern
[:xdigit:] Hexadecimal digits				* PCRE modifier
[:punct:] Punctuation				
[:blank:] Space and tab				
[:space:] Blank characters				
[:cntrl:] Control characters				
[:graph:] Printed characters				
[:print:] Printed characters and spaces				
[:word:] Digits, letters and underscore				
Common Metacharacters				
^ [. \$				
{ * (\				
+) ?				
< >				
The escape character is usually \				
Special Characters				
\n New line				
\r Carriage return				
\t Tab				
\v Vertical tab				
\f Form feed				
\xxx Octal character xxx				
\xhh Hex character hh				



By **Dave Child** (DaveChild)
cheatography.com/davechild/
www.getpostcookie.com

Published 19th October, 2011.
 Last updated 12th May, 2016.
 Page 1 of 1.

Sponsored by **CrosswordCheats.com**
 Learn to solve cryptic crosswords!
<http://crosswordcheats.com>

Genéricos

Los tipos genéricos (introducidos en Java 5) permiten forzar la seguridad de los tipos, en tiempo de compilación y eliminan el uso de casting necesarios para utilizar las colecciones, que hasta este momento eran de tipo *Object*. Permite que podamos definir una clase y la podamos instanciar de tipos totalmente diferentes.

ArrayList, por ejemplo, está definida de forma que podemos crearla de *Integer*, *String*, clases creadas por nosotros, etc.

```
ArrayList <Integer> aL1 = new ArrayList<>();
ArrayList <Alumno> aL2 = new ArrayList<>();
```

Al crear una clase/interface con genéricos, se le proporciona a la clase un parámetro "inventado" al que no se le especifica su tipo. Será luego, cuando se cree la instancia de la clase, cuando se defina ese tipo (que podrá ser una clase cualquiera). Ahí es donde sabremos de qué tipo es el elemento creado.

```
public class MiClase <T> {
    private T dato;                                // T: parámetro que se reemplazará
                                                    // Declara un atributo de tipo T

    MiClase(T param) {this.dato = param;}          // El constructor recibe param T

    T getData() {                                    // Devuelve tipo T
        return dato;
    }

    void verTipo() {
        System.out.print("Tipo: " + dato.getClass().getName());
    }
}
```

Y luego podremos crear instancias de esa clase, por ejemplo, de *Integer*:

```
MiClase <Integer> m1 = new MiClase <>();
m1.verTipo();
```

O también de *String*:

```
MiClase <String> m2 = new MiClase <>("Hola");
m2.verTipo();
```

Aunque en el ejemplo anterior solo se le proporciona un parámetro a la clase (T), podrían proporcionársele más. Y también en el ejemplo, el parámetro se usa en el constructor, pero puede no ser así, como veremos en el siguiente ejemplo:

Imaginemos que necesitamos una clase, a la que llamaremos *Caja*, que contiene dos atributos del mismo tipo, por ejemplo *int*, y con los que podemos hacer varias operaciones, como puede ser añadir un valor a cada uno de los atributos (setters), recuperar el valor almacenado en ellos (getters) y por último, hacer un intercambio entre los valores almacenados en cada uno.

Podríamos crear dicha clase así:

```
public class Caja {

    private int valor1;
    private int valor2;

    public Caja () {
        valor1 = 0;
        valor2 = 0;
    }

    public int getValor1() { return valor1; }
    public void setValor1(int valor1) { this.valor1 = valor1; }

    public int getValor2() { return valor2; }
    public void setValor2(int valor2) { this.valor2 = valor2; }

    public void swap () {
        int aux = valor2;
        valor2 = valor1;
        valor1 = aux;
    }
}
```

Pero qué ocurre si necesitamos que esta clase funcione con cualquier tipo de objetos: String, LocalDate, Alumno, Coche, etc. Pues sin genéricos deberíamos crear los atributos de tipo *Object* y trabajar con castings donde fuese necesario. Con genéricos simplemente debemos proporcionarle el parámetro y trabajar sobre ese tipo ficticio:

```
public class CajaGenerica <T> {

    private T valor1;
    private T valor2;

    public CajaGenerica () {
        valor1 = null;
        valor2 = null;
    }

    public T getValor1() { return valor1; }
    public void setValor1(T valor1) { this.valor1 = valor1; }

    public T getValor2() { return valor2; }
    public void setValor2(T valor2) { this.valor2 = valor2; }

    public void swap () {
        T aux = valor2;
        valor2 = valor1;
        valor1 = aux;
    }
}
```

Así, ahora podríamos generar cajas de cualquier tipo:

```
CajaGenerica <Integer> cg1 = new CajaGenerica<>();
cg1.setValor1(1); cg2.setValor2(2);
cg1.swap();
System.out.println("valor1 -> " + cg1.getValor1());

CajaGenerica <Alumno> cg2 = new CajaGenerica<>();
cg2.setValor1 (new Alumno("Ana"))); cg2.setValor2 (new Alumno("Eva")));
cg2.swap();
System.out.println("valor1 -> " + cg2.getValor1());
```

Los genéricos nos ofrecen también la posibilidad de limitar los tipos de los que se permite crear instancias. Lo logra mediante el uso de herencia:

```
class Numerico<T extends Number> { // El argumento tiene que ser un Number o descendiente
    T num;

    Numerico(T n) { num = n; }

    double inverso() { return 1 / num.doubleValue(); }
    double parteDecimal() { return num.doubleValue() - num.intValue(); }
}
```

En este ejemplo, solo podremos instanciar la clase con clases de tipo Number o descendientes de Number, como puede ser Integer, Double, etc. pero no con otro tipo de clases.

```
Numerico<Integer> iOb = new Numerico<Integer>(5);
System.out.println("Inverso de iOb: " + iOb.inverso());
System.out.println("Decimales de jOb: " + iOb.parteDecimal());

Numerico<Double> d = new Numerico<Double>(1.2d);
System.out.println("Inverso: " + d.inverso());
System.out.println("Decimales: " + d.parteDecimal());

// Error: String no es subclase Number:
Numerico<String> sOb = new Numerico<String>("X");
```

La forma en que hemos instanciado la clase anterior nos recuerda mucho a la creación de instancias de colecciones. Las colecciones están definidas como genéricos y por eso podemos crear ArrayList de cualquier tipo:

```
public class ArrayList <T> { . . . }
```

El problema que resuelven es que si al crear una colección (u otra clase) de un tipo determinado, pongamos String, la instanciamos con un elemento de tipo entero, entonces producirá una excepción. Si crease la colección de tipo Object, necesitaría castings y podrían dar error.

Según las convenciones los nombres de los parámetros de tipo usados comúnmente son los siguientes:

- E: elemento de una colección.
- K: clave.
- N: número.
- T: tipo.
- V: valor.
- S, U, V etc: para segundos, terceros y cuartos tipos.

En el apartado de colecciones, definimos una clase que implementaba *Comparable* así:

```
class Peli implements Comparable {
    public String nombre;
    public int año;

    @Override
    public int compareTo(Object o) { //parámetro debe ser Object. Si no error.
        Peli m = (Peli) o;
        return this.año - m.año;
    }

    public Peli(String n, int a) {this.nombre = n; this.año = a; }
}
```

Como la interfaz Comparable emplea genéricos, podríamos hacerlo así, escribiendo menos código:

```
class Peli implements Comparable <Peli> {
    public String nombre;
    public int año;

    @Override
    public int compareTo(Peli p) { //parámetro debe ser Peli
        return this.año - p.año;
    }
    public Peli(String n, int a) {this.nombre = n; this.año = a; }
}
```

Las interfaces también pueden ser genéricas. Las definiríamos como las clases, empleando el tipo genérico en los métodos que deseemos:

```
interface NombreInterfaz <T> {

    T metodo1 (T op1, T op2);

    void metodo2 (T op1);
}
```

Las clases que implementen una interfaz genérica tienen dos posibilidades, ser genéricas o no.

- Clases genéricas que implementan interfaz genérica. Ejemplo:

```
public class ClaseGenerica <T> implements NombreInterfaz <T> {
    @Override
    public T metodo1 (T op1, T op2) {. . .}

    @Override
    public void metodo2 (T op1) {. . .}
}
```

En el caso, será en la instanciación en la que se determine su tipo:

```
ClaseGenerica <Integer> cG1 = new ClaseGenerica <>();
cG1.metodo1(3,2);
```

- Clases no genéricas que implementan interfaz genérica. Ejemplo:

```
public class ClaseNoGenerica implements NombreInterfaz <Integer> {
    @Override
    public Integer metodo1 (Integer op1, Integer op2) {. . .}

    @Override
    public void metodo2 (Integer op1) {return . . .}
}
```

En el caso, las instancias de la clase ya solo pueden ser del tipo determinado en la clase.

```
ClaseNoGenerica cNG1 = new ClaseNoGenerica();
cNG1.metodo1(3,2);
```

Los genéricos tienen algunas restricciones:

- No se pueden instanciar tipos genéricos con tipos primitivos.
- No se pueden crear instancias de los parámetros de tipo.
- No se pueden declarar campos static cuyos tipos son parámetros de tipo.
- No se pueden usar casts o instanceof con tipos genéricos.
- No se pueden crear arrays de tipos genéricos.
- No se pueden crear, capturar o lanzar tipos genéricos que extiendan de Throwable.

Optional

Optional es una clase genérica que “envuelve” a otra clase, y que permite gestionar los nulos de una forma más eficaz y evitar excepciones de tipo “*Null pointer exception*”. Su constructor es privado, pero podemos crear el método `of()` para su instanciación.

```
String nombre = "Daniel";
Optional<String> oNombre = Optional.of(nombre);
```

Tiene métodos como `isPresent()` o `isEmpty()` para comprobar si tiene contenido (sería equivalente a preguntar si la variable subyacente es nula o no).

```
if (oNombre.isEmpty()) { . . . }
```

El método `orElse()` devuelve un valor por defecto en caso de que la variable sea nula y `get()` nos devolverá el valor:

```
if (nombre == null) nombre="default"
System.out.print (nombre);
```

Podría ser: `System.out.print (oNombre.orElse("default"));`

Aunque en programación imperativa no le encontramos grandes beneficios, en programación funcional permite encadenar operaciones sin tener que validar si los resultados intermedios contienen valores nulos.

Más información aquí: <https://www.adictosaltrabajo.com/2015/03/02/optional-java-8/>

Tipos de Clases Especiales

Clases Anidadas

Es una clase definida dentro de otra clase. Puede ser estática o no estática, a las no estáticas las llamamos *clases internas*.

Se usan como agrupamiento lógico, si una clase se usa solo dentro de otra, queda el código mejor encapsulado. Veamos por separado las no estáticas (clases internas) y las estáticas.

Clases Internas

Ya hablamos de ellas en el apartado de los iteradores. Solo existen en el marco de una instancia de una clase externa. Una de las características más importantes es que una instancia de una clase anidada puede acceder a los atributos no estáticos de la clase que le envuelve.

Ejemplo:

```
public class Externa {
    // atributos clase externa
    // constructor y métodos de la clase externa

    public class Interna {
        // atributos clase interna
        // constructor y métodos
    } //fin clase interna
} // fin clase externa
```

Este ejemplo de estas clases ya lo hemos visto previamente, en la implementación los iteradores.

```
class MiColeccion implements Iterable <Integer> {

    private final static int TAM = 20;
    private Integer[] arr = new Integer[TAM];

    MiColeccion() {
        for (int i = 0; i < TAM; i++) arr[i] = i * 10;
    }

    @Override
    public Iterator<Integer> iterator() {
        Iterator<Integer> it = new MiIterador();
        return it;
    }

    class MiIterador implements Iterator <Integer> {

        private int sig = 0;

        @Override
        public boolean hasNext() {
            if (sig < TAM) return true;
            return false;
        }

        @Override
        public Integer next() {
            int val = arr[sig];
            sig += 2;
            return val;
        }
    }
}
```

Para crearlas tendríamos que emplear esta fórmula (suponiendo constructores sin parámetros):

```
ClaseExterna e = new ClaseExterna();
ClaseExterna.ClaseInterna i = e.new ClaseInterna();
```

O bien todo junto:

```
ClaseExterna.ClaseInterna j = new ClaseExterna().new ClaseInterna();
```

Para ver el ejemplo volver a la sección: “*Recorrer Colecciones e Iteradores*”.

Si definimos un atributo de la clase interna con el mismo nombre que un atributo de la clase que la envuelve, el atributo interno “oculta” al externo. Esto se llama “shadowing”. Es similar a la sobreescritura de métodos, pero en este caso hablando de atributos en vez de métodos.

¿Cuándo usar clases internas?

Una clase interna es una estructura un poco “extraña” en su primer contacto. Decíamos que su ventaja es que desde ella se tiene acceso a los atributos de la instancia de la clase que la envuelve. Si fuese una clase independiente y necesitase acceder a atributos habría que pasárselos como parámetros.

Imaginemos una clase *Alumno* con unos atributos referentes a sus notas y por otra parte otra clase llamada *Evaluación* con un método que hace un tratamiento de las notas de un alumno. Si *Evaluación* es una clase interna de *Alumno* podría acceder a esos atributos directamente; en caso de ser una clase independiente, al método de *Evaluación* habría que pasarle como parámetro esa instancia de *Alumno*.

Entonces, si podemos resolverlo de esta segunda forma, más sencilla ¿Cuál es la ventaja de la clase interna? La respuesta es que se logra mayor encapsulamiento, tenemos el código situado de una forma más coherente (una instancia *Evaluación* no tiene sentido por si mismo, solo tiene sentido para una instancia de *Alumno*).

Clases Anidadas Estáticas

Son similares a las internas, pero no necesitan una instancia de la clase externa, por lo tanto, solo pueden acceder a los miembros estáticos de la clase externa.

Por eso para crearlas no necesitamos una instancia de la clase externa (suponiendo constructores sin parámetros):

```
ClaseExterna.ClaseAnidadaStatic ie = new ClaseExterna.ClaseAnidadaStatic();
```

Clases Locales

Son clases que se crean dentro de un bloque de código, en general en el cuerpo de un método. En el siguiente ejemplo, la clase *Persona* tiene un método llamado *validarDNI* que valida que el DNI sea correcto, esto es, tiene 8 dígitos más la letra correspondiente según una determinada fórmula.

Dicho método contiene una clase local llamada *DniCorrecto*. Esa clase local tiene un constructor al que se le pasan los dígitos de un DNI y genera un DNI su letra correspondiente.

Así pues, para una persona concreta, el método *validarDNI* primero valida la cantidad de dígitos del DNI de esa persona y luego, para validar la letra, crea una instancia de esa clase local con los dígitos del DNI la persona y compara el DNI creado en la clase local (que sabemos que es correcto) con el DNI de esa persona.

```
class Persona {
    public String dni;      public String nombre;

    public Persona(String dni, String nombre, LocalDate fechaNacimiento) {
        this.dni = dni;
        this.nombre = nombre;
    }
    public boolean validarDni () {
        class DniCorrecto {
            private String valor;
            private static final String letras = "TRWAGMYFPDXBNJZSQVHLCKE";
            DniCorrecto (String digitos) {
                valor = digitos + letras.charAt(Integer.parseInt(digitos) % 23);
            }
        }
    }
}
```

```

        if (dni.length() != 9) return false;
        try {Integer.parseInt(dni.substring(0,8));}
        catch (Exception e) {return false;}
        DniCorrecto dc = new DniCorrecto(dni.substring(0,8));
        if (dc.valor.equals(dni))return true;
        return false;
    }
}

```

Clases Anónimas

Permiten definir e instanciar la clase a la vez, sin asignarle un nombre y para ser usadas una sola vez. Podemos crearlas en el cuerpo de un método, de una clase o como argumento de un método. Esta situación es muy frecuente, y de hecho ya las habíamos visto sin darnos cuenta.

Cuando creamos una aplicación Swing, main () contiene:

```

java.awt.EventQueue.invokeLater(new Runnable() {
    public void run() {
        new MainFrame().setVisible(true);
    }
});

```

creando una instancia de una clase que implementa la interfaz *Runnable* y de la que redefine su método *run()*.

O cuando usamos swing.Timer:

```

timer = new Timer (100,
    new java.awt.event.ActionListener () {
        public void actionPerformed(java.awt.event.ActionEvent e) {
            crono.Incrementar();
            jTextField1.setText(crono.toString());
        }
    }
);

```

creando una instancia de una clase que implementa la interfaz *ActionListener* y de la que redefine su método *ActionPerformed()*.

El constructor new puede hacer referencia a una interfaz, y entenderemos que está creando una clase que la implementa, o bien hacer referencia a una clase, y entenderemos que está definiendo una clase hija. En ambos casos crea una instancia, pero la clase creada no tiene nombre. Tampoco tendrá nunca constructor, solo llama implícitamente a *super()* pero no tiene constructor propio.

Otro ejemplo habitual, suele ser a la hora de ordenar una colección con un *Comparator*:

```

Collections.sort(lista, new Comparator<Peli>() { (*)
    @Override
    public int compare(Peli p1, Peli p2) {
        return p1.año-p2.año;
    }
});

```

(*) new Comparator<Peli> está creando una instancia de una clase (sin nombre!!) que implementa la interface Comparator (genérica). Luego abre llaves y ahí hace la implementación de la clase, en este caso solo con el método *compare()*.

En el capítulo anterior, el de Colecciones, definíamos un iterador como clase interna a la clase *MiColección*. Como solo la vamos a usar una sola vez, podríamos haberlo hecho como clase anónima. Este sería el código:

```
public class MiColección implements Iterable<Integer> {
    private final static int TAM = 20;
    private Integer[] arr = new Integer[TAM];
    MiColección() {
        for (int i = 0; i < TAM; i++) arr[i] = i * 10;
    }
    @Override
    public Iterator<Integer> iterator() {
        Iterator<Integer> it =
            new Iterator<Integer>() {
                private int sig = 0;

                @Override
                public boolean hasNext() {
                    if (sig < TAM) return true;
                    return false;
                }

                @Override
                public Integer next() {
                    int val = arr[sig]; sig += 2;
                    return val;
                }
            };
        return it;
    }
}
```

Patrón Singleton

Singleton es un patrón de diseño que define una clase de la cual solamente queremos tener una instancia. Un ejemplo de utilización podría ser un servicio del sistema operativo o la conexión a una base de datos (solo queremos una única conexión, sobre la que luego haremos las operaciones sobre la base de datos, pero no queremos una nueva instancia cada vez que hagamos una operación sobre la base de datos). Para implementarla, podemos seguir los siguientes pasos:

- Definir un único constructor como privado, así no se podrán crear instancias desde el exterior.
- Obtener siempre la instancia a través de un método estático, que llama al constructor solo la primera vez. Las veces siguientes que invoquemos a ese método devolverá esa primera instancia creada.

Un constructor devuelve la referencia en memoria de la instancia que crea. Este método estático "simula" el comportamiento de un constructor.

Ejemplo:

```
class ClaseSingleton {
    private static ClaseSingleton instance = null;

    private ClaseSingleton() {}      //Evita la instanciación directa

    public static ClaseSingleton getInstance() {
        if (instance == null)
            instance = new ClaseSingleton();
        return instance;
    }
}
```

Podemos usarlo y comprobar su funcionamiento así:

```
// ClaseSingleton cls0 = new ClaseSingleton(); //Error
ClaseSingleton cls1 = ClaseSingleton.getInstance ();
ClaseSingleton cls2 = ClaseSingleton.getInstance ();
// Comprobamos así, viendo que la referencia es la misma
System.out.println("Instancia Singleton 1: "+cls1);
System.out.println("Instancia Singleton 2: "+cls2);
```

Clases Inmutables

Son clases que una vez inicializadas con su constructor ya no pueden ser modificadas. Se diferencian de las constantes (atributos *final*) en que las constantes se definen en tiempo de compilación y las inmutables en tiempo de ejecución. *String* es un ejemplo de inmutable.

Para hacer una clase inmutable:

- Los atributos *final* y *private*
- No añadir métodos setter ni métodos que modifiquen los atributos.
- Declarar la clase como *final* (evitamos la herencia, que no haya una clase hija que pueda implementar los setters).
- Los parámetros, del constructor también serán finales, así en el momento de pasarlos también se hacen constantes y no se pueden modificar.

Ejemplo:

```
final class Cliente {
    private final int numCliente;
    private final String nombre;
    private final LocalDate fechaAlta;

    public Cliente(final int nC, final String n, LocalDate fA) {
        this.numCliente = nC;
        this.nombre = n;
        this.fechaAlta = fA;
    }
}
```

17. Tratamiento XML

Introducción

XML es un lenguaje de marcado similar a HTML. Significa Extensible Markup Language (Lenguaje de Marcado Extensible) y es una especificación de W3C como lenguaje de marcado de propósito general. Esto significa que, a diferencia de otros lenguajes de marcado, XML no está predefinido, por lo que debes definir tus propias etiquetas. El propósito principal del lenguaje es compartir datos a través de diferentes sistemas heterogéneos, como Internet.

Veamos un ejemplo de fichero XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<! -- Este archivo recoge recetas variadas-->
<recetas>
    <receta>
        <tipo definicion="postre"/>
        <dificultad>Fácil</dificultad>
        <nombre>Tarta de chocolate</nombre>
        <ingredientes>
            <ingrediente nombre="Cola-Cao" cantidad="250 gramos"/>
            <ingrediente nombre="mantequilla" cantidad="200 gramos"/>
            <ingrediente nombre="harina" cantidad="1 vaso"/>
            <ingrediente nombre="azúcar" cantidad="1 vaso"/>
            <ingrediente nombre="huevos" cantidad="3"/>
        </ingredientes>
        <calorias>600</calorias>
        <pasos>
            <paso orden="1">Mezclar el cola-cao con la mantequilla.</paso>
            <paso orden="2">Separar las yemas y mezclarlas con el azúcar.</paso>
            <paso orden="3">Unir ambas mezclas y añadir la harina.</paso>
            <paso orden="4">Montar las claras y añadirlas a la mezcla anterior.</paso>
            <paso orden="5">Untar un molde de mantequilla y espolvorear de harina.</paso>
            <paso orden="6">Precalentar el horno y hornear a 175° durante unos 40 min.</paso>
        </pasos>
        <tiempo>45 minutos</tiempo>
        <elaboracion>Horno</elaboracion>
    </receta>
    <receta>
        <tipo definicion="postre"/>
        <!--De esta receta no sabemos la dificultad -->
        <nombre>Tiramisú</nombre>
        <ingredientes>
            <ingrediente nombre="cobertura Valor" cantidad="100 gramos"/>
            <ingrediente nombre="bizcochos blandos" cantidad="24"/>
            <ingrediente nombre="azúcar" cantidad="0.5 vasos"/>
            <ingrediente nombre="café solo" cantidad="1 tazón"/>
            <ingrediente nombre="huevos" cantidad="3"/>
            <ingrediente nombre="licor" cantidad="1 chorrito"/>
            <ingrediente nombre="nata líquida" cantidad="0.3 litros"/>
            <ingrediente nombre="queso mascarpone" cantidad="250 gramos"/>
        </ingredientes>
        <calorias>450</calorias>
        <pasos>
            <paso orden="1">Echar la mitad del azúcar en la nata líquida y montarla</paso>
            <paso orden="2">Separar las claras de las yemas y montarlas a punto de nieve.</paso>
            <paso orden="3">Mezclar despacio el queso y las yemas de huevo</paso>
            <paso orden="4">Juntar con la nata y las claras a punto de nieve</paso>
            <paso orden="5">En el café disolver el chocolate y echar el licor.</paso>
            <paso orden="6">Mojar los bizcochos en el café sin dejar que se empapen</paso>
            <paso orden="7">Echar mitad de la mezcla sobre los bizcochos </paso>
            <paso orden="8">Poner una nueva capa de bizcochos mojados sobre la mezcla.</paso>
            <paso orden="9">Echar el resto de la mezcla sobre la capa de bizcochos.</paso>
            <paso orden="10">Dejar reposar al menos 6 horas antes de servir.</paso>
        </pasos>
        <tiempo>7 horas</tiempo>
        <elaboracion>Microondas</elaboracion>
    </receta>
</recetas>
```

Este archivo lo usaremos como base para todos los ejemplos del capítulo.

Para que un documento cumpla el estándar XML tiene que cumplir las siguientes reglas:

- Debe de tener una cabecera donde se especifica la versión de XML que cumple el documento. También suele aparecer tipo de codificación, normalmente UTF-8 o ISO-8859-1.
- El documento debe estar estructurado en forma de etiquetas de apertura <etiqueta> y cierre </etiqueta>.
- Un documento debe de tener al menos una etiqueta de apertura y otra de cierre.
- Se pueden poner etiquetas que abran y cierren la etiqueta al mismo tiempo de la forma: <etiqueta />. Por ejemplo: <profesor nombre="Angel" />
- El orden de las etiquetas es importante. Se deben cerrar en el orden inverso a cómo fueron abiertas.
- Se distingue mayúsculas y minúsculas en la apertura y cierre de etiquetas.
- Una etiqueta puede llevar uno o más atributos. Por ejemplo: <pedidos num_pedidos="10" fecha_pedido="10/01/2012">
- Todos los atributos deben ir entre comillas dobles.
- Se pueden añadir comentarios de la forma:<!-- Texto comentario -->
- Los nombres de las etiquetas y atributos no pueden tener espacios en blanco.
- No todos los “registros” tienen por qué tener la misma información, sobre el ejemplo, no todas las recetas tienen porqué tener ingredientes o calorías.
- Podéis validar cualquier documento/página XML en el siguiente enlace: <https://validator.w3.org/>

Existen diversas formas de tratar los XML en Java, son lo que se denominan “parsers”, nosotros veremos la manera basada en DOM, la más estándar, aunque existen otras librerías como SAX o JAXB. En esta página puedes ver el detalle de métodos disponibles para diferentes tipos de parsers: https://www.tutorialspoint.com/java_xml/index.htm

En estos dos videos explica el proceso para SAX y JAXB.

SAX : <https://www.youtube.com/watch?v=CaF8w9RjVac>

JAXB: <https://www.youtube.com/watch?v=3qMVLs3K0ws>

Con las librerías estándar, cuando manejamos un documento XML o bien creamos uno nuevo, vamos a tener que recorrer su estructura.

Procesar Archivos XML

Cuando tratemos con documentos XML vamos a diferenciar dos fases:

- Comprobar que el documento XML esté bien formado en base a las reglas que indicamos en la introducción.
- Pasar dicho documento a un conjunto de objetos que podamos manejar desde Java. Este formato se denomina DOM (Document Object Model), modelo de objetos de documento. En DOM cada objeto es un nodo de una rama de un árbol jerárquico a la que vamos podemos acceder obteniendo su contenido, atributos, nombre,...

Por lo tanto, vamos a pasar el documento XML en formato texto a un árbol DOM desde donde poder acceder a cada uno de los elementos del documento XML, llamados nodos. Hay diferentes tipos de nodos, siendo los principales los siguientes:

- **Document:** Contenedor de todos los nodos del árbol. También se conoce como la raíz del documento, que no siempre coincide con el elemento raíz.
- **Element:** Cada etiqueta de apertura y cierre junto con su contenido. En un *element* puede tener otros 'element' en su contenido.
- **Attr:** Son los atributos de un elemento y sólo pueden existir en su etiqueta de apertura.
- **Text:** La información (el texto) que va entre las etiquetas de apertura y cierre.
- **Comment:** Son los comentarios del documento.

Atributos de nodos Node



Tipos Node	name	value	nodeType
Document	#document	null	9
DocumentFragment	#document fragment	null	11
DocumentType	nombre del tipo de documento	null	10
EntityReference	nombre de la entidad referenciada	null	5
Element	nombre de la etiqueta	null	1
Attr	nombre del atributo	valor del atributo	2
ProcessingInstruction	destino (<i>tag #</i>)	contenido íntegro exceptuando el <i>tag #</i>	7
Comment	#comment	contenido del comentario	8
Text	#text	contenido de nodo <i>text</i>	3
CDataSection	#cdata-section	contenido de la sección CDATA	4
Entity	nombre de la entidad	null	6
Notation	nombre de la notación declarada en el DTD	null	12

Pasar el documento XML a árbol DOM

Lo primero que tenemos que hacer es verificar que el documento XML esté bien formado. Para ello se hace uso de un parser. Hay dos tipos parser:

- Sin validación: Comprueba que el documento esté bien formado de acuerdo a las reglas de sintaxis de XML.
- Con validación: Además de comprobar que el documento está bien formado sintácticamente, comprueba el documento utilizando un *DTD*(ya sea interno o externo). Un DTD (Document Type Definition) es un documento en el que se indica cual es la estructura que tiene que tener el documento, sus elementos y atributos.

Nosotros vamos a hacer uso de un parser sin validación que además va a ser compatible con el modelo de objeto de documento (DOM) de XML. Usaremos las siguientes clases:

- ***javax.xml.parsers.DocumentBuilderFactory***: instanciánolo mediante el método *newInstance()*.
- ***javax.xml.parsers.DocumentBuilder***: Es la clase encargada de transformar el documento XML a DOM, instanciéndolo mediante el método *newDocumentBuilder()* de la clase anterior.
- ***org.w3c.dom.Document***: Representa un documento XML pasado a DOM. Una vez que el *parser* analice el documento XML creará una instancia de esta clase donde estará guardado todo el documento en formato DOM.
- El modelo de lectura podría ser así:

```
File file = new File("archivos" + File.separator + "recetas.xml");
try (FileInputStream fis = new FileInputStream(file);
     InputStreamReader isr = new InputStreamReader(fis, "UTF-8")) {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder dB = factory.newDocumentBuilder();
    Document doc = dB.parse(new InputSource(isr));
    //ahora podemos acceder a los nodos del document `doc'
}
catch (Exception e) { e.printStackTrace(); }
```

(*) El método *parse()* está sobrecargado y puede recibir un *File* o un *StreamReader* en el que podríamos especificar la codificación. Y ojo al importar *Document* que el IDE puede importar la clase *Document* de Swing en vez de *org.w3c.dom*.

Leer el árbol DOM

Obtener el elemento raíz

Para obtener el elemento raíz (nodo raíz) del documento (recordar que obligatoriamente debe existir para cumplir con el estándar XML) llamaremos al método ***getDocumentElement()***.

```
Element raiz = doc.getDocumentElement();
```

En nuestro caso será *<recetas>*. Podemos indicar que muestre el nombre del nodo ***getNodeName()***.

```
System.out.println ("Raíz: " + doc.getDocumentElement () .getNodeName () );
```

Este método devuelve un objeto de tipo *Element*, por lo que podemos aplicarle directamente el método *getNodeName()*. Veremos a continuación que mediante otros métodos obtenemos objetos de tipo *Node*, que nos obligarán a hacer castings a *Element* para utilizar determinados métodos.

Recorrer los nodos

El método ***getElementsByTagName(etiqueta)*** va a devolver la lista de nodos que tienen esa etiqueta, y lo hará devolviendo un objeto de tipo *NodeList*. Podemos ver ese *NodeList* como una colección que podemos recorrer, utilizando sus métodos *getLength()* e *item(n)* como mostramos a continuación:

```

NodeList listaRecetas = doc.getElementsByTagName("nombre");
for(int i = 0 ; i < listaRecetas.getLength() ; i++) {
    Node nodo = listaRecetas.item(i);
    System.out.println("Nombre de la receta: " + nodo.getTextContent());
}

```

Siguiendo con el ejemplo de las recetas arriba expuesto, *NodeList* contendría los dos nodos que tienen la etiqueta *nombre*:

```

<nombre>Tarta de chocolate</nombre>
<nombre>Tiramisú</nombre>

```

y con *getTextContent()* obtenemos el texto contenido en cada una de ellas, por lo que el trozo de código anterior mostraría:

```

Tarta de chocolate
Tiramisú

```

Acceder al contenido de un Element

El proceso anterior funciona correctamente para nodos que no tienen etiquetas dentro del mismo, pero si invoco al método *getTextContent()* sobre nodos que sí tienen otros nodos dentro, por ejemplo *<ingredientes>* no produciría el resultado esperado. En esos casos tenemos que hacer **un casting del nodo a Element** y podremos acceder a sus nodos internos mediante sus etiquetas, como vemos en el siguiente ejemplo.

```

NodeList listaRecetas = doc.getElementsByTagName("receta");
for(int i = 0 ; i < listaRecetas.getLength() ; i++) {
    Element element = (Element) listaRecetas.item(i);
    String n = element.getElementsByTagName("nombre").item(0).getTextContent();
    String c = element.getElementsByTagName("calorias").item(0).getTextContent();
    String e = element.getElementsByTagName("elaboracion").item(0).getTextContent();
    System.out.printf("%s: (%s) %s.%n", n, c, e);
}

```

Lo que hemos hecho en el bucle anterior tiene un problema grave: al poner *item(0).getTextContent()* le estamos diciendo que nos muestre el texto del primer elemento que tenga esa etiqueta, sin comprobar previamente que dicha etiqueta existe. Si no existiese se produciría una excepción.

Por ejemplo, si hubiésemos puesto:

```
c = element.getElementsByTagName("dificultad").item(0).getTextContent();
```

se produciría esa excepción ya que la segunda receta no tiene etiqueta *<dificultad>*. Se lo solucionaría preguntando antes si existe ese “ítem”:

```

String d;
if (element.getElementsByTagName("dificultad").item(0) != null)
    d = element.getElementsByTagName("dificultad").item(0).getTextContent();
else d = "Dificultad desconocida";

```

Si para cada *receta* lo que queremos es acceder a nodos que se repiten, como pueden ser los *<pasos>*, deberemos incluir de nuevo un bucle adicional para cada receta. Por ejemplo:

```

NodeList listaRecetas = doc.getElementsByTagName("receta");
for (int i = 0 ; i < listaRecetas.getLength() ; i++) {
    Element receta = (Element) listaRecetas.item(i);
    String n = receta.getElementsByTagName("nombre").item(0).getTextContent();
    String c = receta.getElementsByTagName("calorias").item(0).getTextContent();
    System.out.printf("%s: (%s)%n", n, c);
}

```

```

NodeList listaPasos = receta.getElementsByTagName("paso");
for (int j = 0 ; j < listaPasos.getLength() ; j++) {
    String p = listaPasos.item(j).getTextContent();
    System.out.printf("\tPaso %d: %s.%n", j+1, p);
}
}

```

Para navegar por todo el contenido de un nodo de tipo *Element*, disponemos de métodos muy útiles que además nos ayudan a evitar las posibles excepciones de intentar acceder al texto contenido de elementos inexistentes.

- ***getFirstChild***(): devuelve el nodo primer hijo, puede ser *Element*, texto, etc. Null si no hay hijos.
- ***getLastChild***() devuelve el nodo último hijo, puede ser *Element*, texto, etc.
- ***getParentNode***() devuelve el nodo padre.
- ***getPreviousSibling***(): devuelve el nodo anterior (no el nodo hijo, sería "hermano").
- ***getNextSibling***(): devuelve el siguiente nodo "hermano", no "hijo". Null si no hay más.
- ***getChildNodes***(): devuelve NodeList devuelve todos los nodos 'hijos'. Null si no hay.
- ***hasChildNodes***(): devuelve true si el nodo tiene algún hijo. Falso en otro caso.

Si para un *Node* queremos saber de qué tipo es podemos invocar a su método ***getNodeType()*** devolviéndonos un elemento enumerado: ELEMENT_NODE, TEXT_NODE, COMMENT_NODE, etc. Este método es muy útil cuando no conocemos la estructura exacta del documento.

Así pues, podemos usar las dos estrategias vistas para recorrer un archivo XML, o bien mediante bucles anidados, accediendo a las etiquetas por su nombre (si conocemos la estructura de documento y es bastante "fija") o bien mediante los métodos que acabamos de ver, en la que hacemos un recorrido exhaustivo (por todos los hijos, hermanos, etc.) aunque la estructura del archivo sea muy variable.

IMPORTANTE: Al pasar los documentos XML al árbol DOM sin validar (sea con DTD o XSD) se preservan los saltos de línea, espacios en blanco, etc. que pudiese haber en el archivo y se convierten en nodos de tipo TEXT_NODE, con nombre #text. Esto habrá que tenerlo en cuenta en el procesamiento cuando accedamos sin indicar el nombre de la etiqueta. Prueba este código para comprobarlo:

```

Node raiz = doc.getDocumentElement();
Node node = raiz.getFirstChild();
while (node != null) {
    System.out.println(node.getNodeName());
    System.out.println(node.getNodeType());
    System.out.println("-----");
    node = node.getNextSibling();
}

```

Se podría prescindir de ellos chequeando su tipo y nombre, por ejemplo, al principio del while:

```

if (node.getNodeType() != Node.TEXT_NODE || 
    !node.getNodeName().equals("#text")) {

```

Acceder a los atributos

Un atributo es un elemento opcional de la etiqueta de apertura de un nodo del documento. Consiste en una pareja de nombre del atributo y valor del atributo. En una etiqueta podría haber varios atributos.

Ejemplo:

```
<ingrediente nombre="Cola-Cao" cantidad="250 gramos"/>
```

Tiene dos atributos: *nombre* y *cantidad* con valores "Cola-cao" y "250 gramos" respectivamente.

```
<paso orden="1">Mezclar el cola-cao con la mantequilla.</paso>
```

Tiene un atributo llamado *orden* con valor '1'.

Para obtener los valores de un atributo, la clase *Element* dispone de dos métodos:

- ***getAttribute(nombreAtributo)***: Se le envía como parámetro el nombre del atributo y devuelve su valor como String. En caso de que no exista el atributo, devuelve la cadena vacía.
- ***getAttributeNode(nombreAtributo)***: Se le envía como parámetro el nombre del atributo y devuelve un objeto de la clase Attr la cual dispone de métodos para obtener el nombre del atributo, obtener su valor, modificarlo,...Devuelve null en caso de que el atributo no exista.

Para comprobar si el atributo existe se puede hacer uso del método ***hasAttribute(nombreAtributo)***, que devuelve un booleano indicando si el atributo existe o no.

El siguiente ejemplo mostraría el nombre de todos los ingredientes de todas las recetas (estando este valor almacenado en el atributo '*nombre*' de la etiqueta `<ingrediente>`):

```
NodeList listaIngredientes = doc.getElementsByTagName("ingrediente");
for (int i = 0 ; i < listaIngredientes.getLength() ; i++) {
    Element ingrediente = (Element) listaIngredientes.item(i);
    String nomIng = ingrediente.getAttribute("nombre");
    System.out.printf("Ingrediente %d: %s%n", i+1, nomIng);
}
```

Modificar el Árbol DOM

Existen otro tipo de operaciones que no vamos a ver, como la posibilidad de asociar a un nodo de un árbol un objeto a través de un valor clave asociado al nodo, mediante el método *setUserData* pero lo que sí vamos a ver son las operaciones básicas para modificar el contenido de un árbol DOM y añadir nuevo contenido.

Modificar el contenido del árbol DOM

Normalmente vamos a querer modificar o bien el texto de asociado a una etiqueta o bien el valor de un atributo. Disponemos de los siguientes métodos:

- ***setTextContent(String nuevoValor)***: Este método sustituye el texto del nodo. Ojo: elimina todos los nodos que tuviera por debajo del actual, sustituyéndolos por ese texto.
- ***setAttribute(String atributo, String nuevoValor)***: se aplica a objeto de tipo *Element*, por lo que se debe hacer un casting los nodos para poder aplicarlo. Recordar que disponemos del método *hasAttribute(String)* para comprobar si existe antes de modificarlo.

También disponemos de *setnodeValue (String nuevoValor)* que cambiar el *value* de un nodo. Ese *value* es distinto según el tipo de nodo.

Ejemplos:

El siguiente código pasaría a mayúsculas todos los pasos de nuestras recetas:

```
NodeList listaPasos = doc.getElementsByTagName("paso");
for (int i = 0 ; i < listaPasos.getLength() ; i++) {
    Element paso = (Element) listaPasos.item(i);
    paso.setTextContent(paso.getTextContent().toUpperCase());
}
```

El siguiente código modifica el valor de los atributos, sustituyendo el ingrediente "azúcar" por "sacarina" en todas las recetas:

```
NodeList listaIngredientes = doc.getElementsByTagName("ingrediente");
for (int i = 0 ; i < listaIngredientes.getLength() ; i++) {
    Element paso = (Element) listaIngredientes.item(i);
    if (paso.getAttribute("nombre").equals("azúcar"))
        paso.setAttribute("nombre", "sacarina");
}
```

Añadir nuevo contenido al árbol DOM

Igual que en el caso de la modificación, nos vamos a centrar en las operaciones para añadir/reemplazar/eliminar nuevas etiquetas o nuevos atributos asociados a una etiqueta.

Teniendo en cuenta que una etiqueta al final es un objeto de la clase *Node* podemos emplear los siguientes métodos:

- [appendChild\(nuevoNodo\)](#): Añade un nuevo hijo a un nodo
- [insertBefore\(nodoExistente, nuevoNodo\)](#): Añade un nuevo "hermano" al nodo.
- [removeChild\(nodoEliminar\)](#): Elimina un nodo.
- [replaceChild\(nodoASustituir, nuevoNodo\)](#): Sustituye un nodo hijo.

Importante: Recordar que *Node* es una interface y por tanto no podemos crear un objeto directamente. En el párrafo anterior hablamos de '*Node*' pero realmente usaremos '*Element*'.

Para utilizar los métodos anteriores debemos de seguir estos pasos:

- 1) Crear un *Element*, llamando al método del *document.createElement(nombreEtiqueta)* para crear una etiqueta.
- 2) Crear un nodo de tipo texto llamando al método del *document.createTextNode(texto)* que añade el texto a la etiqueta creada en el paso anterior.
- 3) Añadir el *Element* del primer paso en el punto de la jerarquía que queramos, mediante los métodos anteriores (*appendChild*, *insertBefore*) aplicados sobre un *Element* concreto.

El siguiente ejemplo añadiría al final de las recetas, como último nodo de elemento raíz `<recetas>`:

```
Element raiz = doc.getDocumentElement();
Element nuevoElementFabrica = doc.createElement("autor");
nuevoElementFabrica.appendChild(doc.createTextNode("Carlos Arguiñano"));
raiz.appendChild(nuevoElementFabrica);
```

Siendo este el resultado, en la parte final del archivo:

```

        <paso orden="7">Echar la mitad de la mezcla sobre
        <paso orden="8">Poner una nueva capa de bizcochos
        <paso orden="9">Echar el resto de la mezcla sobre
        <paso orden="10">Dejar reposar al menos 6 horas ar
    </pasos>
    <tiempo>7 horas</tiempo>
    <elaboracion>Microondas</elaboracion>
</receta>
<autor>Carlos Arguiñano</autor>
</recetas>
```

En el ejemplo anterior elegimos *raíz* para insertar el nodo hijo por comodidad, pero podría ser un elemento cualquiera, por ejemplo, un hijo de una receta concreta, o de todas la recetas...

Eliminar nodos: para eliminar nodos, tendremos que situarnos en el nodo padre y pasar 'una referencia' al nodo hijo que queremos eliminar.

El siguiente ejemplo recorrería todas las *recetas* y eliminaría para todas ellas su nodo (etiqueta) *<calorías>*:

```

NodeList lista = doc.getElementsByTagName("receta");
for (int i = 0; i < lista.getLength(); i++) {
    Element elementoPadre = (Element) lista.item(i);
    Element elementoHijo = (Element)
        elementoPadre.getElementsByTagName("calorías").item(0);
    elementoPadre.removeChild(elementoHijo);
}
```

Podemos hacerlo de otra forma, recorriendo directamente los nodos *calorías* y obteniendo el padre dinámicamente:

```

NodeList lista = doc.getElementsByTagName("calorías");
for (int i = 0; i < lista.getLength(); i++) {
    Element elCalorías = (Element) lista.item(i);
    elCalorías.getParentNode().removeChild(elCalorías);
    i--;
}
```

(*) *i--;* hace falta para que retroceda el bucle, ya que, al eliminar el elemento actual, la siguiente pasa a la posición actual, y al hacer *i++* en el *for*, nos lo saltaríamos. Quítalo para comprobar cómo no funciona correctamente.

Eliminar atributos: lo haremos mediante el método *removeAttribute (nombreAtributo)* aplicado sobre el *Element* que tiene ese atributo. El siguiente ejemplo elimina la *cantidad* en los *ingredientes* de todas las recetas.

```

NodeList lista = doc.getElementsByTagName("ingredientes");
for (int i = 0; i < lista.getLength(); i++)
    ((Element) lista.item(i)).removeAttribute("cantidad");
```

Árbol DOM desde cero

El proceso es similar a lo visto previamente. Primero crearemos el árbol DOM, aunque vacío:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder dB = factory.newDocumentBuilder();
```

Creamos un nuevo documento y le añadimos un elemento raíz, por ejemplo <recetas>.

```
doc = dB.newDocument();
Element raiz = doc.createElement("recetas");
doc.appendChild(raiz);
```

Luego podemos añadir elementos, hijos de ese elemento raíz. El siguiente ejemplo añade un nodo texto.

```
Element receta = doc.createElement("receta");
receta.appendChild(doc.createTextNode("Paella"));
raiz.appendChild(contacto);
```

Pero podría ser un elemento y crear "subnodos" dentro de él.

Pasar árbol DOM a documento XML

Este será el último paso necesario después de modificar el árbol DOM, llevar esos cambios al fichero XML físicamente. Lo haremos mediante el método **transform** de la clase **Transformer**. Este método tiene dos parámetros, el primero basado en el documento origen que tiene el árbol DOM que queremos escribir, y el segundo un *StreamResult* con la ruta del archivo.

Este sería el código que se ejecutaría en un try...catch e incluyendo los *imports* necesarios:

```
File ficheroSalida = new File("archivos" + File.separator + "recetas2.xml");

TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer transformer = tFactory.newTransformer();

transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
transformer.transform(new DOMSource(doc), new StreamResult(ficheroSalida));
```

18. Acceso a Base de Datos

Introducción

Una parte fundamental de cualquier aplicación informática es la persistencia de los datos, es decir, que, al acabar nuestra sesión de trabajo con nuestros programas, podamos guardar los datos relevantes de forma permanente. Esto lo podemos lograr mediante ficheros (ya vistos) o mediante una base de datos externa.

El proceso de acceso a base de datos consistirá en un primer paso de conexión con la base de datos, a continuación, la ejecución de operaciones tanto de lectura como de modificación de datos (mediante lenguaje SQL) y finalmente la desconexión.

JDBC es el interfaz común que Java proporciona para poder conectarnos a cualquier SGBD relacional. Proporciona una API completa para trabajar de forma que sea cual sea el motor de base de datos con el que conectemos, la API siempre será la misma. Simplemente tendremos que obtener el *driver* correspondiente al motor de base de datos que queramos usar, que sí que dependerá totalmente de éste. Lista de SGBD soportados por JDBC: <https://www.oracle.com/technetwork/java/index-136695.html>

Ya que, como hemos dicho, el driver es lo único que depende exclusivamente del SGBD utilizado, es muy sencillo escribir aplicaciones cuyo código se pueda reutilizar si más adelante tenemos que cambiar de motor de Base de Datos o bien si queremos permitir que dicha aplicación pueda conectarse a más de un SGBD de forma que el usuario no tenga por qué comprometerse a un SGBD concreto si la adquiere o quiere ponerla en marcha.

Para este curso utilizaremos MySQL, cuyo driver podemos descargar en la siguiente URL: <https://dev.mysql.com/downloads/connector/j/5.1.html>.

En dicha página, elegiremos la opción “*Platform independent (Architecture Independent) zip file*”, obteniendo un archivo .zip que, una vez descomprimido, obtendremos un archivo .jar.

Para la ejecución de los ejemplos de este manual y para la realización de las prácticas deberemos tener un servidor arrancado con MySQL, podemos usar XAMPP, que ya lo conocemos.

Los pasos a seguir serán los siguientes:

- 1.- Establecer los datos de conexión y conectarse con la base de datos mediante un objeto *Connection*.
- 2.- Ejecutar las consultas SQL, generalmente mediante un objeto *PreparedStatement*.
- 3.- Tratar los resultados devueltos por la consulta, será en un objeto *ResultSet*.
- 4.- Cerrar los objetos empleados (*ResultSet*, *PreparedStatement* y *Connection*).

Conexión - Desconexión

La conexión con la Base de Datos es la única parte de la programación con JDBC que depende directamente del SGBD que se vaya a utilizar. No es un cambio muy grande puesto que simplemente hay que seleccionar el drive adecuado y la cadena de conexión adecuada.

El primer paso será añadir el driver al proyecto. Este proceso difiere según el tipo de proyecto en que nos encontremos, bien Maven, bien Ant.

Proyecto Maven:

En este caso no es necesario descargar el driver, Maven lo hará por nosotros. Lo único que debemos hacer es incorporar esa dependencia en el archivo *pom.xml*. Para saber las líneas exactas a incluir nos conectamos al repositorio de Maven: <https://mvnrepository.com/>, y buscamos: *mysql*, y elegimos la última versión:

The screenshot shows the Maven Repository homepage with a search bar at the top. Below it, a chart titled 'Indexed Artifacts (17.0M)' shows the growth of indexed artifacts from 2008 to 2018. The main content area displays the details for 'MySQL Connector/J > 8.0.20'. It includes fields for License (GPL 2.0), Categories (MySQL Drivers), Organization (Oracle Corporation), HomePage (http://dev.mysql.com/doc/connector-j/en/), Date (Apr 26, 2020), Files (jar (2.3 MB) View All), Repositories (Central), and Used By (4,421 artifacts). At the bottom, a section titled 'Maven' contains the XML dependency code:

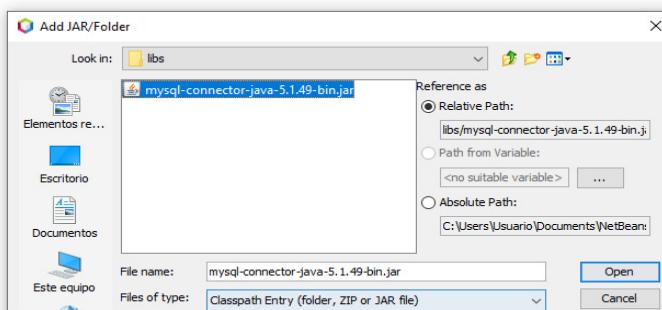
```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.20</version>
</dependency>
```

Entonces, debemos incluir en el *pom.xml*:

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.20</version>
    </dependency>
</dependencies>
```

Proyectos Ant:

En este caso sí debemos descargar el *zip* con el driver MySQL. Descomprimiendo ese *zip* obteníamos un archivo *jar*. Ya lo tenéis en el aula virtual con el nombre *mysql-connector-java-5.1.49-bin.jar*. Crearemos en nuestro proyecto una nueva carpeta (New >> Folder) llamada '*libs*', y copiaremos ese archivo *.jar* en la carpeta creada. Una vez hecho eso, añadiremos esa carpeta al proyecto, es decir, le diremos a Netbeans que cuando haga el *build* del proyecto, incluya ese *jar*. Para ello, en nuestro proyecto, botón derecho sobre *Libraries >> Add JAR/Folder* y seleccionamos el archivo.



Establecer la conexión:

Una vez hecho esto, estemos en un tipo de proyecto u otro, en nuestros programas procederemos a establecer la conexión. Para ello podemos usar dos clases distintas: **DriverManager** (más sencilla y típica para aplicaciones pequeñas, y **DataSource**, más avanzado, más complejo, más habitual para aplicaciones empresariales con mucho tráfico). Nosotros emplearemos la primera.

En esa clase, utilizaremos el método **getConnection()**, que necesita tres parámetros:

- Cadena de conexión incluyendo la URL y puerto del gestor de base de datos y el nombre de la base de datos.
- Usuario y contraseña del gestor de base de datos, con permiso sobre la base de datos incluida en el parámetro anterior.

Por sencillez, nosotros emplearemos el usuario root por defecto de MySQL que le habremos asignado una contraseña en la instalación (ver anexo). Podríamos crear otro usuario desde una herramienta como MySQL Workbench.

```
Connection conexion = null;
try {
    conexion = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/bdPrueba", "root", "contraseña");
} catch (SQLException e) {
    System.out.println("Código de Error: " + e.getErrorCode() +
        "\nSQLState: " + e.getSQLState() +
        "\nMensaje: " + e.getMessage());
}
```

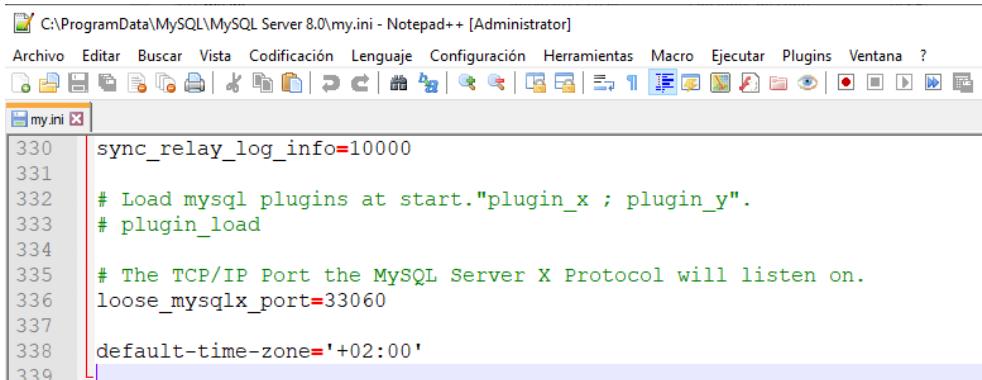
Al igual que vimos en la gestión de ficheros, podemos utilizar desde Java 7, *try with resources*, definiendo entre paréntesis la instancia de conexión. Así el sistema se encargará de hacer el *close()* de la conexión automáticamente.

```
try (conexion = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/bdPrueba", "root", "contraseña")) {
    .
    .
} catch (SQLException e) {
    System.out.println("Código de Error: " + e.getErrorCode() +
        "\nSQLState: " + e.getSQLState() +
        "\nMensaje: " + e.getMessage());
}
```

Dependiendo de la versión, con la configuración por defecto de MySQL, al hacer la conexión puede producirse el siguiente error:

```
Código de Error: 0 - SQLState: 01S00
Mensaje: The server time zone value 'Hora de verano romance' is unrecognized..
```

Para solucionarlo, tenemos dos opciones, cambiar la configuración de MySQL, en el archivo **c:\ProgramData\mysql\MySQL Server 8.0\my.ini**, en la sección **[mysqld]**, añadiendo la línea: **default-time-zone='+02:00'** y reiniciando el servidor.



O bien cambiar la cadena de conexión en el método `getConnection()` en todos los programas:

```
"jdbc:mysql://localhost:3306/bdrpueba?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC", "root", "");
```

La primera opción es más cómoda, siempre que tengamos permisos para modificar la configuración del servidor. La segunda opción hará que el programa no falle, independientemente del servidor contra el que ejecutemos la conexión.

En un entorno real, los parámetros usuario y contraseña no deberían estar escritos en el programa, deberían tomarse, por ejemplo, de un archivo. El método `getConnection()` está sobrecargado, admitiendo como segundo parámetro un objeto `Properties` que contenga `user` y `password`, aportando más seguridad (`Properties` lo vimos en el capítulo de Ficheros).

Desconexión

Al abrir la conexión con `try with resources` ya no es necesario hacerla. En todo caso, sería así:

```
try {
    conexion.close();
    conexion = null;
} catch (SQLException e) { e.printStackTrace();}
```

Operaciones

Una vez creada la base de datos, con sus tablas y resto de elementos (vistas, índices, etc.) las operaciones típicas que realizaremos desde nuestras aplicaciones serán la lectura de esos datos, la inserción, modificación y borrado. Estas operaciones las realizaremos mediante lenguaje SQL y reciben las siglas de CRUD (acrónimo de "Crear, Leer, Actualizar y Borrar" en inglés: *Create, Read, Update and Delete*).

Disponemos de dos clases básicas para ejecutar estas operaciones sobre nuestra base de datos, son `Statement` y `PreparedStatement`.

Statement provee de métodos para ejecutar operaciones, recibiendo la instrucción SQL como parámetro de tipo String. Crearemos un `Statement` a partir de la instancia de conexión:

```
Statement st = conexion.createStatement();
```

Los dos métodos que usaremos serán: `executeQuery` y `executeUpdate`.

- **executeQuery()** lo usaremos para consultas de tipo *SELECT* y devuelve un `ResultSet`.
- **executeUpdate()** se usa para *INSERT*, *UPDATE* o *DELETE* y devuelve un entero que representa el número de filas afectadas.

ResultSet es una interfaz que recoge los resultados de una consulta. Tiene una estructura de cursor (podríamos decir que es como una tabla, en la que hay una fila por cada fila devuelta por la consulta). Podemos navegar a través de esa estructura fila a fila obteniendo los datos de cada fila.

PreparedStatement es una extensión de `Statement` que previene la inyección de código con una forma distinta de componer el String que forma la consulta, de forma que JDBC "precompilará" la consulta antes de enviarla. Esta clase dispone igualmente de los métodos `executeQuery()` y `executeUpdate()`. Se genera así:

```
PreparedStatement ps = conexion.prepareStatement();
```

La **inyección de código** es una técnica de ataque a una base de datos que se basa en introducir (inyectar) código malicioso en las partes dinámicas de la consulta, por ejemplo, valores solicitados al usuario. Ejemplo: Supón que construyes esta cadena que será una consulta en tu base de datos:

`String consulta = "select * from tabla where usuario = ' " + user + " ' "` en la que `user` es una variable que introduce el usuario, con intención de que la consulta similar a:

```
select * from tabla where usuario = 'Pepe'
```

Pero imagina que el usuario introdujese: `xxx' or '1='1`

El resultado final sería: `select * from tabla where usuario = 'xxx' or '1='1'`

producido un resultado totalmente diferente.



Inyección de código a los radares de tráfico ;)

Consultas

Para lanzar instrucciones SQL de consulta, que devuelvan datos, los pasos serán:

1.- Establecer la conexión con la base de datos, como vimos previamente.

2.- Construir la consulta SELECT sobre un String:

```
String sql = "SELECT nombre, precio FROM productos";
```

3.- Ejecutar la consulta con *PreparedStatement* "recoger" los datos devueltos en un *ResultSet*.

```
PreparedStatement ps = conexion.prepareStatement(sql);
ResultSet rs = ps.executeQuery();
```

4.- Procesar los datos devueltos en el *ResultSet* según nuestros requerimientos.

```
while (rs.next()) {
    System.out.println("Fila número: " + rs.getRow());
    System.out.println("\t nombre: " + rs.getString(1));
    System.out.println("\t precio: " + rs.getFloat(2));
}
```

Como ya comentamos, **ResultSet** será una especie de tabla con los resultados de la consulta. Dispone de diversos métodos para su gestión.

- **next()**: pasa a la siguiente fila del resultado devolviendo *false* si no hay más filas. Ojo: inicialmente no está en la primera fila, hay que hacer un primer *next()* para llegar a la primera fila. Cuando no hay más filas devuelve *false*, por eso es típica la estructura del *while* anterior.
- **getXXXX(n)**: Son diversos métodos para obtener, de la fila actual, el dato que hay en la columna 'n'. La primera columna no es la 0, es la 1. Son varios estos métodos: *getInt*, *getFloat*, *getString*, *getDate*, *getBoolean*, etc.
- En vez del número de columna, también se puede poner el nombre por ejemplo: *rs.getFloat("precio")*;
- *getDate(n)* devuelve *java.sql.Date*, se puede convertir a *LocalDate*: *rs.getDate(n).toLocalDate()*,
- **getRow()**: Devuelve el número de fila actual del *ResultSet*.
- Otros métodos para navegación por el *ResultSet*. Además de **next()** disponemos de estos otros métodos para desplazarnos por el conjunto de datos:
 - **previous()**: pasa a la posición anterior a la actual.
 - **beforeFirst()**: pasa a la posición previa a la primera del *ResultSet*.
 - **last()**: pasa a la última posición del *ResultSet*.
 - **absolute(int f)**: pasa a la fila 'f' del *ResultSet*.
 - **relative(int f)**: desplaza 'f' filas la posición actual ('f' puede ser positivo o negativo)

Importante: Por defecto, *ResultSet* es de solo lectura y se puede recorrer solo hacia adelante, sin volver atrás, por lo que no se podría hacer un *previous()* o un *beforeFirst()*. Si hiciésemos un *last()* no podríamos volver atrás. Este comportamiento se puede cambiar.

```
PreparedStatement ps = conexion.prepareStatement(sql,
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

Así podríamos recorrer hacia adelante y detrás el *ResultSet* como deseemos.

5.- Finalmente, deberíamos cerraremos el *PreparedStatement*, *ResultSet* y la conexión. Esto no es necesario si usamos *try with resources*, como comentamos previamente.

```
rs.close();
ps.close();
conexion.close();
```

Estos procesos deberán gestionar las diferentes excepciones que se puedan producir, quedando finalmente el código completo así:

```
String sql = "SELECT nombre, precio FROM productos";
try ( Connection conexion = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/nombreBD", "usuario", "contraseña");
      PreparedStatement ps = conexion.prepareStatement(sql)) {
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
        System.out.println("Fila número: " + rs.getRow());
        System.out.println("\t nombre: " + rs.getString(1));
        System.out.println("\t precio: " + rs.getFloat(2));
    }
} catch (SQLException e) {
    System.out.println("Código de Error: " + e.getErrorCode() + "\n" +
        "SQLState: " + e.getSQLState() + "\n" +
        "Mensaje: " + e.getMessage() + "\n");
}
```

Parametrización de consultas: También se pueden parametrizar las consultas, tal y como se muestra en el siguiente ejemplo, donde se mostrará la información de los productos de un determinado, seleccionado en tiempo de ejecución:

```

float p1 = 100;
float p2 = 300;
String sql = "SELECT nombre, precio FROM productos " +
    "WHERE precio > ? and precio < ?";
PreparedStatement ps = conexion.prepareStatement(sql);

ps.setFloat(1, p1);
ps.setFloat(2, p2);
ResultSet rs = ps.executeQuery();

```

Las interrogaciones del String con la sentencia SQL son sustituidas por los métodos *setXXX*, en los que se indica el número de orden de interrogación y el valor que tomará. En este caso, ejecutaría la consulta:

```
SELECT nombre, precio FROM productos WHERE precio > 100 and precio < 300;
```

A tener en cuenta:

- Aunque los parámetros sean textos o fechas, no tenemos que preocuparnos de las comillas en la consulta, al sustituir la interrogación por el valor, JDBC se encarga de ello.
- Si el parámetro es una fecha, el valor debe ser de tipo `java.sql.Date`, no `LocalDate`, por lo que si tuviésemos una variable llamada 'fec' de tipo `LocalDate`, la llamada debería ser:

```
ps.setDate(1, java.sql.Date.valueOf(fec));
```

Funciones agregadas: En el caso de las funciones agregadas, podremos tener en cuenta que sólo van a devolver un valor, por lo que no será necesario preparar el código para recorrer el *ResultSet*. Podremos acceder directamente a su primera fila. Ejemplo:

```

String sql = "SELECT count(*) FROM productos "
PreparedStatement ps = conexion.prepareStatement(sql);
ResultSet rs = ps.executeQuery();
rs.next();           //pasa al primer registro directamente
System.out.println("Cantidad de productos: " + rs.getInt(1));

```

Ojo: Si dejas un espacio entre 'count' y '(*)' MySQL produce error de sintaxis.

Insertar, modificar y borrar datos

Estas operaciones son similares a la anterior de lectura, con las siguientes características:

- Serán sentencias SQL de tipo INSERT, UPDATE o DELETE respectivamente.
- El método de *PreparedStatement* que usaremos será *executeUpdate()* en vez de *executeQuery*.
- Podemos usar parámetros mediante las interrogaciones en la consulta y los métodos *setXXX* del *PreparedStatement*.
- La operación no devuelve un *ResultSet*, sino un entero con el número de filas afectadas.

```

String sql = "INSERT INTO productos (nombre, precio) VALUES (?,?)";
String prod = "Producto1";
float precio = 30f;

try ( Connection conexion = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/nombreBD", "usuario", "contraseña");
      PreparedStatement ps = conexion.prepareStatement(sql)) {
    ps.setString (1, prod);
    ps.setFloat (2, precio);

    int cantFilas = ps.executeUpdate();
    System.out.println( cantFilas + " fila/s insertadas");

} catch (SQLException e) {
    System.out.println("Código de Error: " + e.getErrorCode() + "\n" +
                       "SQLState: " + e.getSQLState() + "\n" +
                       "Mensaje: " + e.getMessage() + "\n");
}

```

Modificaciones a través del ResultSet

ResultSet no es la estructura idónea para modificar los datos contenidos en nuestra base de datos, siempre es preferible el método `executeUpdate()` de `PreparedStatement` pero para pequeños proyectos puede ser cómodo. Por otra parte, **estas operaciones que vamos a describir puede que no funcionen en todos los gestores de bases de datos ni con todos los drivers**. Si queremos explotar esta técnica, disponemos de una evolución de `ResultSet` optimizada llamada **RowSet**.

Vimos previamente que un ResultSet por defecto solo se podía recorrer hacia adelante, pero también vimos que podíamos cambiar esta condición en el momento de invocar a `prepareStatement()`.

Este método nos permite modificar también la propiedad de solo lectura que tiene por defecto el ResultSet, de forma que podamos modificarlo, y por tanto actualizar la base de datos.

Así pues, la estructura de `prepareStatement()` sería así:

```
prepareStatement(String SQL, tipoResultSet, concurrenciaResultSet);
```

Los distintos tipos de `ResultSet` son:

- **ResultSet.TYPE_FORWARD_ONLY**. Sólo movimiento hacia delante (por defecto).
- **ResultSet.TYPE_SCROLL_INSENSITIVE**. Puede hacer cualquier movimiento, pero no refleja los cambios en la base de datos.
- **ResultSet.TYPE_SCROLL_SENSITIVE**. Puede hacer cualquier movimiento y además refleja los cambios en la base de datos.

Y en cuanto a la concurrencia:

- **ResultSet.CONCUR_READ_ONLY**. Sólo lectura (por defecto).
- **ResultSet.CONCUR_UPDATABLE**. Actualizable.

Podríamos entonces definir un ResultSet cuyos cambios tuviesen reflejo en la Base de Datos.

```
PreparedStatement ps = conexion.prepareStatement(sql,
                                              ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

Actualizaciones: Para hacer actualizaciones, utilizaremos los métodos del `ResultSet`, **updateXXXX()**, análogos a `getXXXX()` que en vez de obtener el valor en una determinada columna, la actualizarían. Por ejemplo:

`rs.updateFloat(2, 100f);` actualizaría la segunda columna de la fila actual del `ResultSet` a 100 (debería ser float).

Además de actualizar la columna o columnas que queramos, debemos hacer a continuación `updateRow()` para hacer efectiva la actualización.

El siguiente ejemplo, subiría el precio un 10% a todos los productos de la tabla y luego los mostraría:

```
String sql = "SELECT nombre, precio FROM productos";
try (Connection conexion = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/prog", "root", ""));
    PreparedStatement ps = conexion.prepareStatement(sql,
        ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
        rs.updateFloat(2, rs.getFloat(2)*1.10f);           //actualiza precio
        rs.updateRow();                                     //actualiza la fila
    }
    rs.beforeFirst();                                    //vuelve al principio
    while (rs.next()) {
        System.out.println("Fila número: " + rs.getRow());
        System.out.println("\t nombre: " + rs.getString(1));
        System.out.println("\t precio: " + rs.getFloat(2));
    }
} catch (SQLException e) {
    System.out.println("Código de Error: " + e.getErrorCode() + "\n"
        + "SQLState: " + e.getSQLState() + "\n"
        + "Mensaje: " + e.getMessage() + "\n");
}
```

(*) Para que funcione, la tabla debe tener definida clave primaria e incluir el campo clave en el `ResultSet`

Borrados: Para borrar simplemente, llamaremos al método `deleteRow()` y borrará la fila actual. El siguiente ejemplo, borraría los productos de menos de 200 euros.

```
while (rs.next()) {
    if (rs.getFloat(2) < 200) {
        System.out.println("Borrando producto: " + rs.getString(1));
        rs.deleteRow();
    }
}
```

Inserciones: Para hacer inserciones, primero invocaremos al método `moveToInsertRow()`, que vendría a ser como mover la posición actual del `ResultSet` a una posición vacía. Luego llamaremos a los métodos `updateXXXX()` para cada uno de los campos del `ResultSet`, para finalmente invocar a `insertRow()` para hacer efectiva la inserción.

Una vez insertado, el `ResultSet` queda situado en la fila insertada, si queremos volver a la fila en la que estábamos haremos `moveToCurrentRow()`, y si queremos avanzar haremos `next()`.

El siguiente ejemplo, crea para cada producto, un producto nuevo con el mismo nombre pero añadiéndole "LowCost" al final y con la mitad de precio del producto original.

```

String sql = "SELECT nombre, precio FROM productos";
try( Connection conexion = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/prog", "root", ""));
    PreparedStatement ps = conexion.prepareStatement(sql,
    ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)) {
    ResultSet rs = ps.executeQuery();

    while (rs.next()) {
        String n = rs.getString(1) + " LowCost";
        float p = rs.getFloat(2) / 2f;
        rs.moveToInsertRow();
        rs.updateString(1, n);
        rs.updateFloat(2, p);
        rs.insertRow();
        rs.next();
    }
    rs.beforeFirst();
    while (rs.next()) {
        System.out.println("Fila número: " + rs.getRow());
        System.out.println("\t nombre: " + rs.getString(1));
        System.out.println("\t precio: " + rs.getFloat(2));
    }
} catch (SQLException e) {
    System.out.println("Código de Error: " + e.getErrorCode() + "\n"
        + "SQLState: " + e.getSQLState() + "\n"
        + "Mensaje: " + e.getMessage() + "\n");
}
}

```

Patrón DAO / Repository

El patrón Arquitectónico “Data Access Object” (DAO) y su más reciente evolución, el patrón “Repository” permiten separar la lógica de acceso a datos de la lógica de negocio, de tal forma que una sola clase encapsule y aísle el proceso de acceso a datos del resto de la aplicación. Hay algún matiz entre uno y otro patrón, pero no vamos a profundizar en ello.

Si recordáis, cuando hablamos de interfaz gráfico, tratábamos de separar la parte de interacción con el usuario de la lógica del programa, para poder utilizar la parte lógica en otros entornos. Esto es algo similar, pero en relación con el acceso a la base de datos. Así, sería muy fácil cambiar de gestor e incluso de tipo de base de datos (BD orientada a objetos, BD NoSQL, etc...) ya que el resto de la aplicación no se vería afectada.

Por otra parte, en las actualizaciones y correcciones referentes al acceso a la base datos, es más fácil localizar el código afectado, así como su testing.

La idea sería entonces tener una clase que agrupase todas estas operaciones. Supongamos que tenemos una aplicación que gestiona una clase llamada Empleado:

```

public class Empleado {
    private int id;
    private String nombre;
    private LocalDate fechaNacimiento;
    private String categoria;
    private float salario;

    // + constructor con todos los atributos, getters y setters
}

```

y que tenemos los datos de esa clase en una base datos, en una tabla llamada *Empleado*. Podríamos hacer una clase que englobase todos los accesos.

1) Primero deberíamos crear la clase y definir como vamos a establecer la conexión con la BD, una única vez al comienzo de la aplicación o bien establecer la conexión en cada operación. Si elegimos la primera opción, incluiríamos un método así:

```
public class EmpleadoRepository {
    private Connection con = null;

    public void conectar() throws SQLException {
        String JDBC_URL = "jdbc:mysql://localhost:3306/prog";
        con = DriverManager.getConnection(JDBC_URL, "usuario", "contraseña");
    }
}
```

De una forma más sofisticada, podemos establecer esa conexión en el constructor de la clase, empleando el patrón Singleton visto previamente y tomando los datos de usuario y contraseña de un fichero de tipo *Properties*, para dotar de mayor seguridad a la aplicación.

```
public class EmpleadoRepository {
    private static EmpleadoRepository instance = null;
    private Connection con = null;

    private EmpleadoRepository () throws SQLException , IOException {
        String JDBC_URL = "jdbc:mysql://localhost:3306/prog";
        if (con == null) {
            Properties props = new Properties();
            props.load(new FileInputStream("login.props"));
            con = DriverManager.getConnection(JDBC_URL, props);
        }
    }
    public static EmpleadoRepository getInstance() throws SQLException {
        if (instance == null)
            instance = new EmpleadoRepository ();
        return instance;
    }
}
```

La segunda opción es no definir un método de conexión y abrir y cerrar la conexión en cada operación tal y como hemos visto en capítulos anteriores. A nivel de seguridad y de rendimiento, probablemente esta sea la mejor opción.

2) Ahora podríamos hacer métodos para la inserción, búsqueda, o cualquier otra funcionalidad que precisase nuestra aplicación, estableciendo la conexión o no, dependiendo de la elección tomada en el punto anterior:

```
public class EmpleadoRepository {
    private String JDBC_URL = "jdbc:mysql://localhost:3306/prog";

    public int insert(Empleado emp) throws SQLException {
        int cantFilas;
        try (Connection conexion = DriverManager.getConnection("JDBC_URL", "root", "")) {
            PreparedStatement ps = con.prepareStatement(
                "INSERT INTO empleado (id, nombre, fechaNacimiento, categoria, salario)
                 VALUES (?, ?, ?, ?, ?)");
            ps.setInt(1, emp.getId());
            ps.setString(2, emp.getNombre());
            ps.setDate(3, Date.valueOf(emp.getFechaNacimiento()));
            ps.setString(4, emp.getCategoría());
            ps.setFloat(5, emp.getSalario());
            cantFilas = ps.executeUpdate();
            ps.close();
        }
        return cantFilas;
    }
}
```

Ojo: si se produce una excepción, no hay 'catch' y se propaga al método llamante para que sea éste el que la trate, así dotamos de mayor independencia esta clase.

```

public List<Empleado> findAll() throws SQLException {
    try (Connection conexion = DriverManager.getConnection("JDBC_URL", "root", "")) {
        PreparedStatement ps = con.prepareStatement("SELECT * FROM empleado");
        ResultSet rs = ps.executeQuery();

        List<Empleado> result = new ArrayList<>();
        while (rs.next()) {
            result.add( new Empleado(rs.getInt("id"), rs.getString("nombre"),
                                     rs.getDate("fechaNacimiento").toLocalDate(),
                                     rs.getString("categoria"),
                                     rs.getFloat("salario")));
        }
        rs.close();
        ps.close();
    }
    return result;
}

```

- 3) Ahora, desde nuestra aplicación, primero debemos crear la instancia de la clase. Se nos presentan tres opciones:

- Si hemos optado por crear un método para establecer la conexión, primeo lo llamaremos:

```

EmpleadoRepository empleadoRepository = EmpleadoRepository();
try {
    empleadoRepository.conectar();
    //o bien, si hemos hecho la conexión en el constructor con Singleton:
} catch (SQLException e) { e.printStackTrace(); }

```

- Si hemos desarrollado la conexión con Singleton, en vez de llamar a conectar haríamos:

```
EmpleadoRepository empleadoRepository = EmpleadoRepository.getInstance();
```

- Si hemos optado por establecer la conexión en cada operación simplemente instanciaríamos la clase:

```
EmpleadoRepository empleadoRepository = EmpleadoRepository();
```

- 4) Ahora ya podremos llamar a los métodos de la clase `EmpleadoRepository` para realizar las operaciones CRUD sobre la base de datos: insertar, borrar, consultar, modificar...

```

try {
    empleadoRepository.insert
        (new Empleado(12, "Ana Pérez", "31/12/1999", "Analista", 2000f));
} catch (SQLException e) { e.printStackTrace(); }

```

Decíamos que los métodos de la clase `EmpleadoRepository` podían lanzar las excepciones, por lo que los programas y clases que los llamen deberán capturarlas.

- 5) Una última optimización que podemos incorporar a este esquema es añadir una interfaz que defina los métodos de nuestro patrón *Repository*:

```

public interface EmpleadoRepository {
    void conectar() throws SQLException;
    int insert(Empleado emp) throws SQLException;
    List<Empleado> findAll() throws SQLException;
    ...
}

```

Y la clase anterior la implementaría:

```

public class EmpleadoRepositoryImpl implements EmpleadoRepository {
    ...
}

```

En la aplicación usaríamos una referencia a la interfaz instanciada con el constructor de la clase. ¿Qué ocurre si el día de mañana cambiamos a un sistema gestor de base de datos totalmente diferente o la forma de realizar las consultas? Simplemente creariamos una nueva clase que implementase la interfaz. Como nuestra aplicación usa los métodos de la interfaz, no se vería afectada (solo en la instancia inicial, que debería invocar al constructor de la nueva clase).

SQLite

SQLite es una librería gratuita que implementa un motor de base de datos sencillo, autocontenido, sin necesidad de servidor alguno y ni de configuración.

Toda la información de una base de datos (tablas, índices, triggers, etc.) queda almacenada en un solo archivo, que es fácilmente **portable** entre distintas plataformas, simplemente copiando un único archivo. Como inconvenientes, cabe citar que no tiene gestión de usuarios ni privilegios y que tiene pocos tipos de datos (dispone de tipos Integer, Real, Text y Blob pero no Boolean o Date/Time).

Todas estas características la hacen idónea en muchas situaciones: en pequeños proyectos o proyectos con poco tratamiento de base de datos, en fase de prueba, cuando vamos a presentar un prototipo previo a nuestros clientes, **aplicaciones móviles**, etc.

Para incluir una base de datos SQLite en nuestro proyecto seguiremos los siguientes pasos:

- 1.- Nuevo proyecto *Maven* (podría ser de otro tipo, pero con *Maven* es más sencillo).
- 2.- Vamos al repositorio *Maven* y buscamos la librería SQLite, para obtener la dependencia.

The screenshot shows the Maven Repository interface. In the top navigation bar, there is a search bar with the placeholder "Search for groups, artifacts, categories". Below the search bar, a chart titled "Indexed Artifacts (17.0M)" shows the growth of indexed projects from 2008 to 2018, starting at 0 million and reaching approximately 12 million by 2018. On the left sidebar, there is a "Popular Categories" section listing various software components like Aspect Oriented, Actor Frameworks, Application Metrics, Build Tools, Bytecode Libraries, Command Line Parsers, Cache Implementations, Cloud Computing, and Code Analyzers. The main content area displays the details for the "SQLite JDBC" artifact. The artifact page title is "SQLite JDBC > 3.31.1" and it is described as the "SQLite JDBC library". Key metadata shown includes: License (Apache 2.0), Date (May 05, 2020), Files (jar (6.8 MB) with a "View All" link), Repositories (Central), and Used By (648 artifacts). Below the artifact details, there is a list of build tools that can use this dependency: Maven, Gradle, SBT, Ivy, Grape, Leiningen, and Buildr. At the bottom of the page, there is a snippet of XML code representing the dependency declaration:

```
<!-- https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc -->
<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.31.1</version>
</dependency>
```

3.- Incluimos en el archivo de configuración del proyecto: *pom.xml* la dependencia:

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc -->
    <dependency>
        <groupId>org.xerial</groupId>
        <artifactId>sqlite-jdbc</artifactId>
        <version>3.31.1</version>
    </dependency>
</dependencies>
```

4.- En el programa, para la conexión con la base de datos, usamos la cadena:

```
Connection conexion = DriverManager.getConnection("jdbc:sqlite:archivo.db");
```

Siendo *archivo.db* el fichero donde está almacenado la base de datos. Puede estar en la raíz del proyecto o en una subcarpeta. En este último caso incluiríamos la ruta en la cadena de conexión:

```
"jdbc:sqlite:data/archivo.db"
```

Previamente deberíamos tener creado ese archivo .db con las tablas y demás elementos de la base de datos. Podemos hacerlo con programas como SQLiteManager, DBBrowser, Navicat, etc. y en nuestro caso será muy cómodo usar la extensión de Chrome: *SQLiteManager*.

5.- El resto de operaciones con la base de datos es exactamente igual a lo visto con MySQL a lo largo de todo este tema.

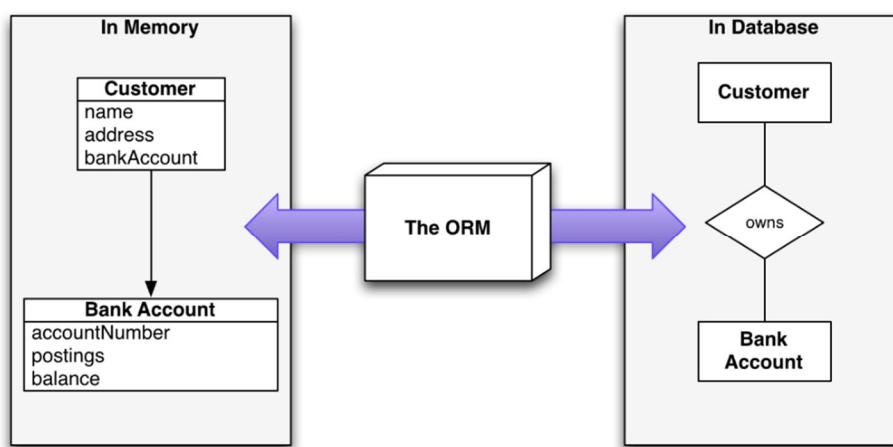
Otros Temas Interesantes

El acceso a base de datos daría para hacer un solo curso en sí mismo, en este apartado queremos dejar unas ideas sobre aspectos de los que puede ser interesante ampliar conocimientos.

Mapeo Objeto-Relacional. Hibernate

El *desfase objeto-relacional* surge cuando en el desarrollo de una aplicación con un lenguaje orientado a objetos se hace uso de una base de datos relacional. En cuanto al desfase, ocurre que en nuestra aplicación Java tendremos la definición de clases con sus atributos y métodos mientras que en la base de datos tendremos una tabla cuyos campos se tendrán que corresponder con los atributos que hayamos definido anteriormente en esa clase. Puesto que son estructuras que no tienen nada que ver entre ellas, tenemos que hacer el mapeo manualmente, haciendo coincidir (a través de los getters o setters) cada uno de los atributos con cada uno de los campos (y viceversa) cada vez que queramos leer o escribir un objeto desde y hacia la base de datos, respectivamente.

Eso hace que tengamos que estar continuamente descomponiendo los objetos para escribir la sentencia SQL para insertar, modificar o eliminar, o bien recomponer todos los atributos para formar el objeto cuando leamos algo de la base de datos, como hemos visto previamente.



Mapeo Objeto-Relacional (ORM)

Si contamos con un framework como **Hibernate**, esta misma operación se traduce en unas pocas líneas de código en las que podemos trabajar directamente con el objeto Java, puesto que el framework realiza el mapeo en función de las anotaciones que hemos implementado a la hora de definir la clase, que le indican a éste con qué tabla y campos de la misma se corresponde la clase y sus atributos, respectivamente.

```
@Entity
@Table(name = "actor", catalog = "db_peliculas")
public class Actor {
    private Integer id;
    private String nombre;
    private Date fechaNacimiento;

    // Constructor/es
    public Actor() { . . . }

    . . .

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "id")
    public Integer getId() { return this.id; }
    public void setId(Integer id) { this.id = id; }

    @Column(name = "nombre")
    public String getNombre() { return this.nombre; }

    . . .
```

Luego, las operaciones de base de datos, la haremos a través de métodos *hibernate* sin ver el SQL que hay por detrás. Ejemplo:

```
Session sesion = HibernateUtil.getCurrentSession();
sesion.beginTransaction();
sesion.save(unObjeto);
sesion.getTransaction().commit();
sesion.close();
```

Más información en: <https://datos.codeandcoke.com/apuntes:hibernate>

Transacciones

Una transacción es un conjunto de operaciones sobre una base de datos que se deben ejecutar como una unidad. Hay ocasiones en las que es necesario que varias operaciones sobre la base de datos se realicen en bloque, es decir, que se ejecuten o todas o ninguna, pero no que se realicen unas sí y otras no. Si se ejecutan parcialmente hasta que una da error, el estado de la base de datos puede quedar inconsistente. En este caso necesitaríamos un mecanismo para devolverla a su estado anterior, pudiendo deshacer todas las operaciones realizadas.

El objeto *Connection* por defecto realiza automáticamente cada operación sobre la base de datos. Esto significa que cada vez que se ejecuta una instrucción, se refleja en la base de datos y no puede ser deshecha. Por defecto está habilitado el modo auto-commit en la conexión.

Los siguientes métodos en la interfaz *Connection* son utilizados para gestionar las transacciones en la base de datos:

```
void setAutoCommit(boolean valor)
void commit()
void rollback()
```

Para iniciar una transacción deshabilitamos el modo auto-commit mediante el método `setAutoCommit(false)`. Esto nos da el control sobre lo que se realiza y cuándo se realiza. Una llamada al método `commit()` realizará todas las instrucciones emitidas desde la última vez que se invocó el método `commit()`.

Una llamada a `rollback()` deshará todos los cambios realizados desde el último `commit()`. Una vez se ha emitido una instrucción `commit()`, esas transacciones no pueden deshacerse con `rollback()`. En muchas ocasiones, deberemos hacer `rollback()` de operaciones en la sección `catch` de los métodos, si se produce una excepción en una operación previa.

RowSet

Como vimos previamente, `ResultSet` no funciona para hacer actualizaciones con todos los gestores de base de datos. `RowSet` es una evolución de `ResultSet`, que soluciona este problema y ofrece otras ventajas, como las siguientes:

- `ResultSet` mantiene la conexión con la base de datos permanentemente, y `RowSet` puede ser conectado o no.
- `RowSet` es *Serializable*, puede ser transmitido por la red y puede ser tratado como un JavaBean. `ResultSet` no tiene ninguna de estas características.

Instalación y uso de MySQL

Como ya hemos comentado, necesitamos instalar un sistema gestor de base de datos que contenga las tablas sobre la que lanzaremos las consultas SQL. En este caso instalaremos un servidor MySQL.

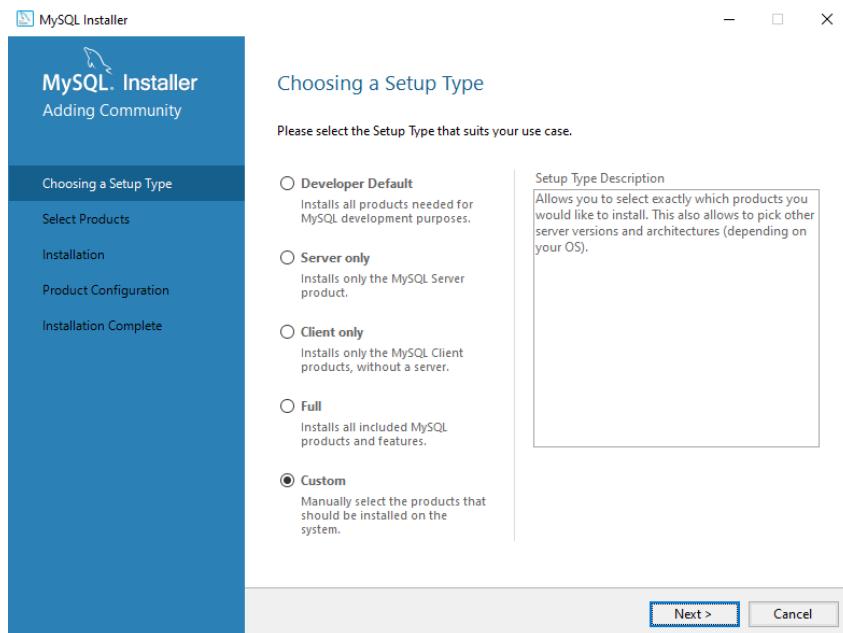
Instalación

Lo primero que debemos hacer es descargar el archivo de instalación del servidor (MySQL versión Community) desde: <https://dev.mysql.com/downloads/mysql/>

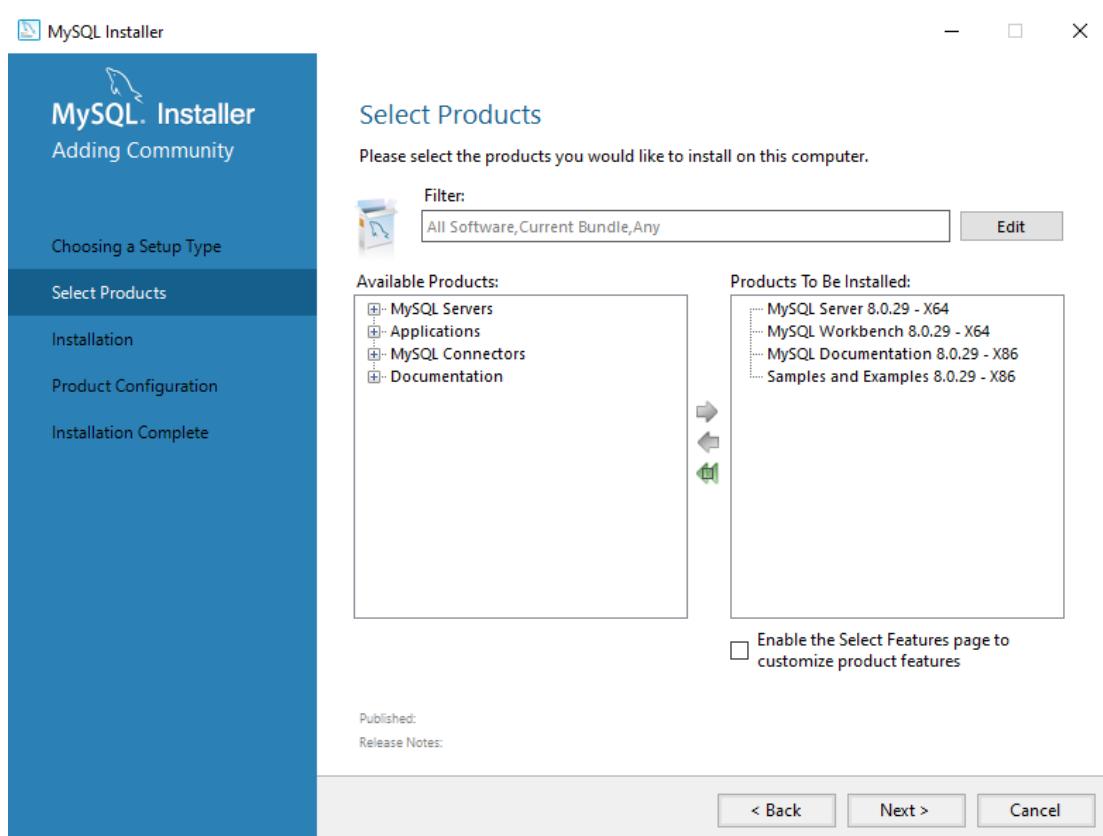
The screenshot shows the MySQL Community Downloads page. At the top, there's a navigation bar with links for "General Availability (GA) Releases" (which is selected), "Archives", and "RSS". Below the navigation, the title "MySQL Community Server 8.0.29" is displayed. A dropdown menu for "Select Operating System" is open, showing "Microsoft Windows" as the current selection. To the right of the dropdown, there's a link "Looking for previous GA versions?". Under the heading "Recommended Download:", there's a section for "MySQL Installer for Windows" with the subtext "All MySQL Products. For All Windows Platforms. In One Package.". It includes a thumbnail image of the MySQL logo and a "Go to Download Page >" button. Below this, under "Other Downloads:", there are two rows of download links. The first row contains "Windows (x86, 32 & 64-bit), MySQL Installer MSI" with a file size of 214.0M and a "Download" button. The second row contains "Windows (x86, 64-bit), ZIP Archive" (file mysql-8.0.29-winx64.zip) and "Debug Binaries & Test Suite" (file mysql-8.0.29-winx64-debug-test.zip), both with file sizes of 538.6M and "Download" buttons. The bottom of the page has a footer with links to "MySQL Home", "MySQL Support", "MySQL Documentation", "MySQL News", and "MySQL Events".

Elegimos "MySQL Installer for Windows" (All MySQL Products), que incluirá también MySQL Workbench, herramienta que usaremos para conectarnos al servidor para tareas administrativas como arrancar y parar el servidor y desde la que podremos realizar consultas a la base de datos.

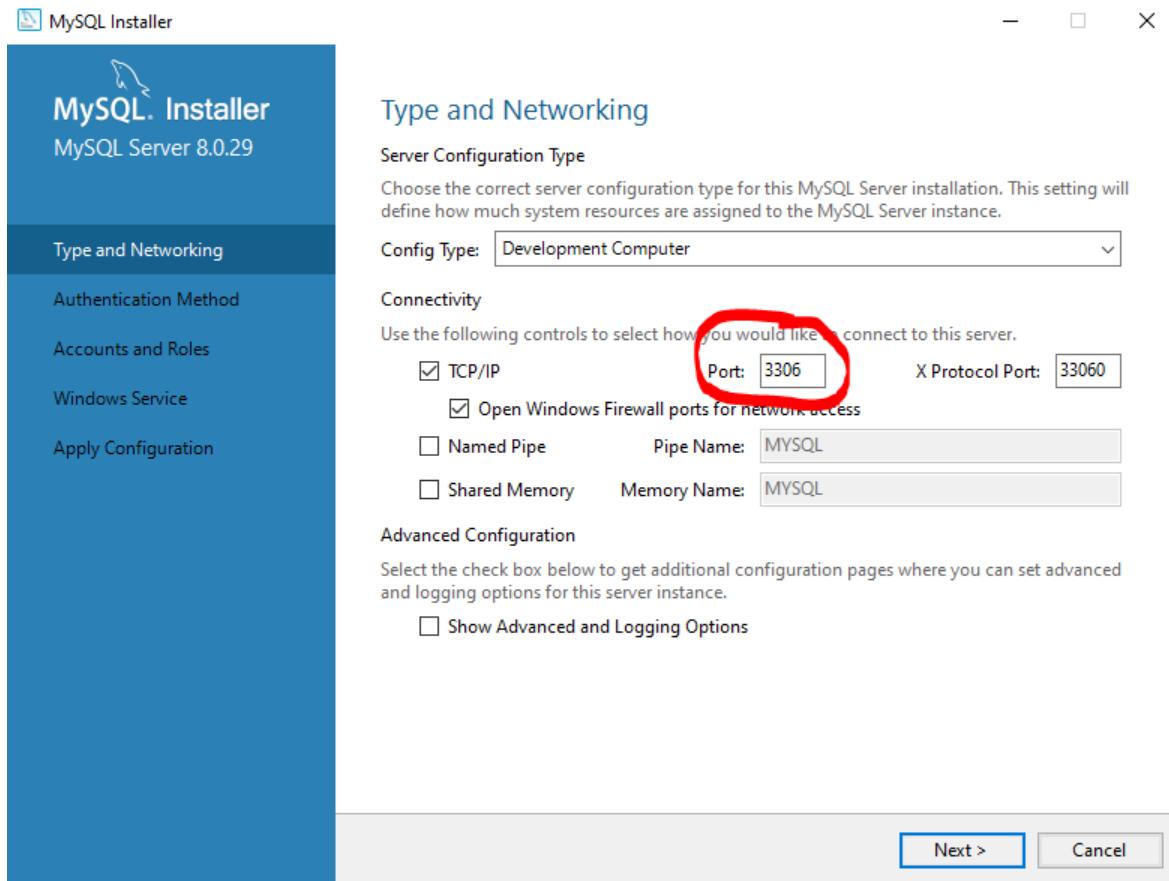
Hacemos doble click sobre el instalador. En el proceso de instalación, seleccionamos el tipo de instalación "Custom".



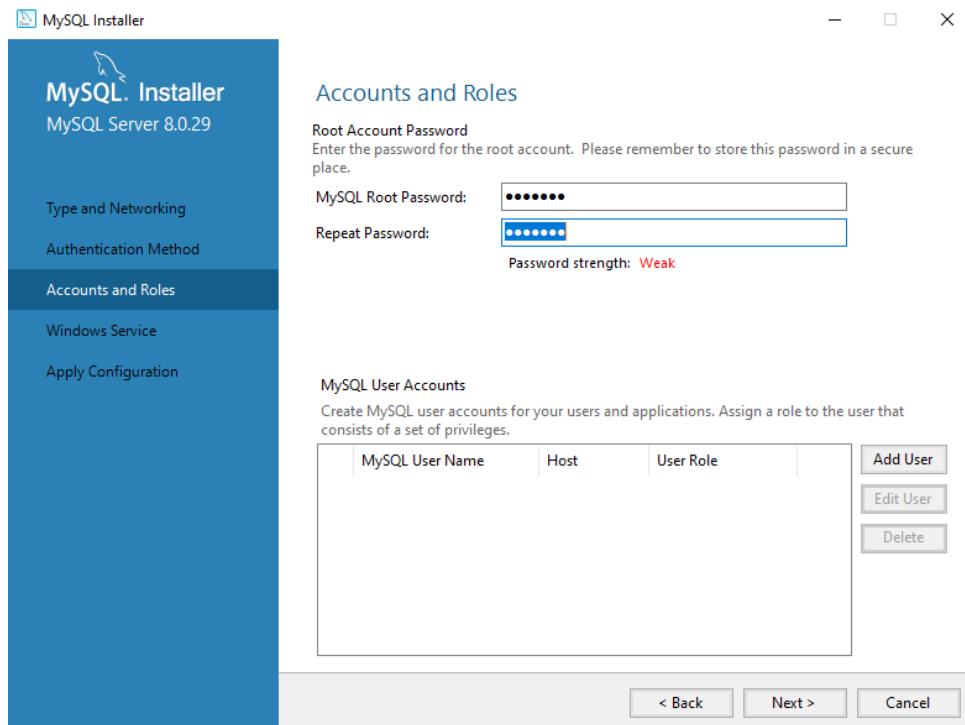
Y en los productos a instalar, seleccionamos solo el MySQL Server, MySQL WorkBench y documentación y ejemplos.



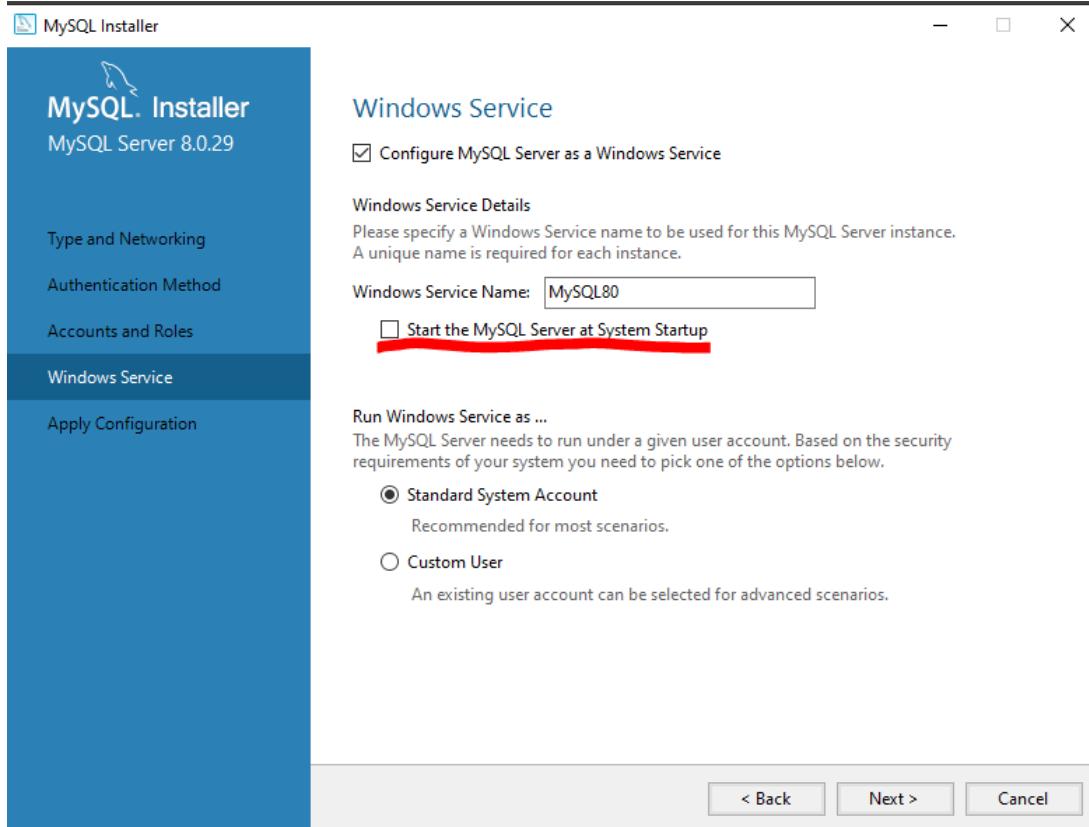
Pasaremos entonces a la configuración. No cambiamos nada:



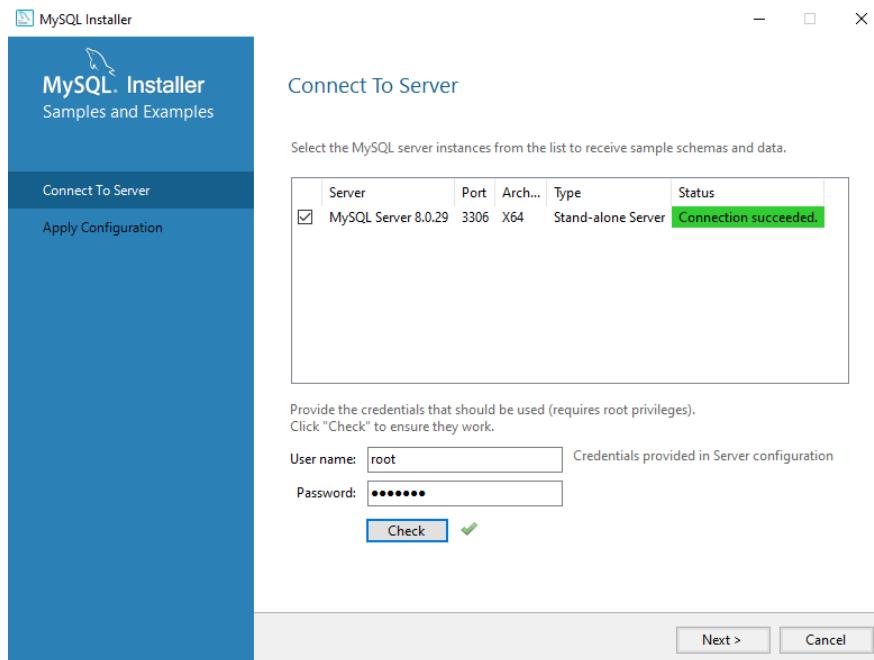
Seguimos en la configuración, asignando contraseña al usuario root, en nuestro caso *abc123*.



Desmarcamos que arranque al iniciar el sistema:

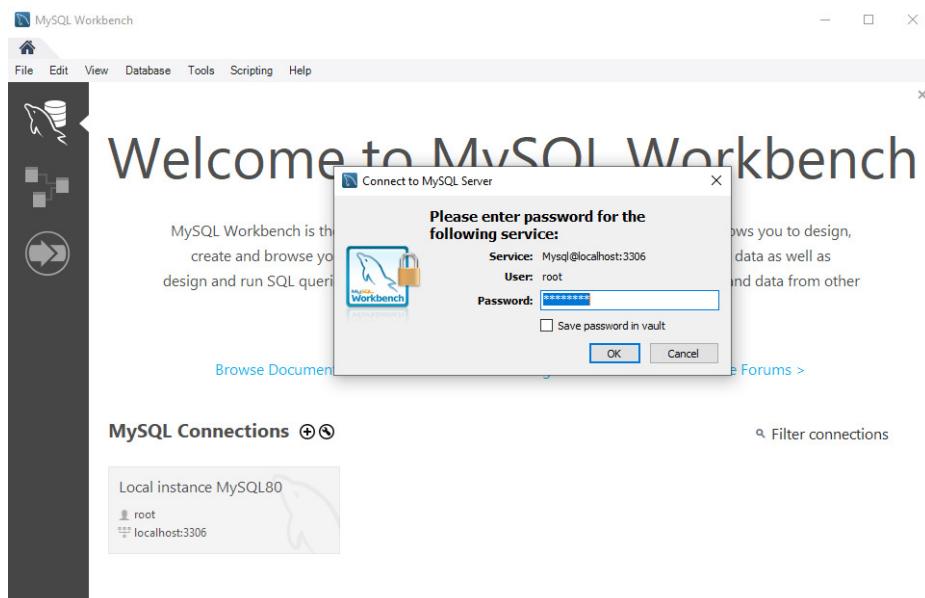


Instalamos los ejemplos, conectándonos como root:

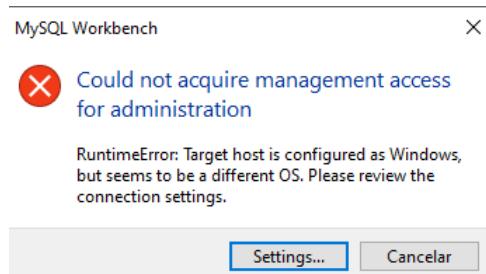


Conexión al servidor

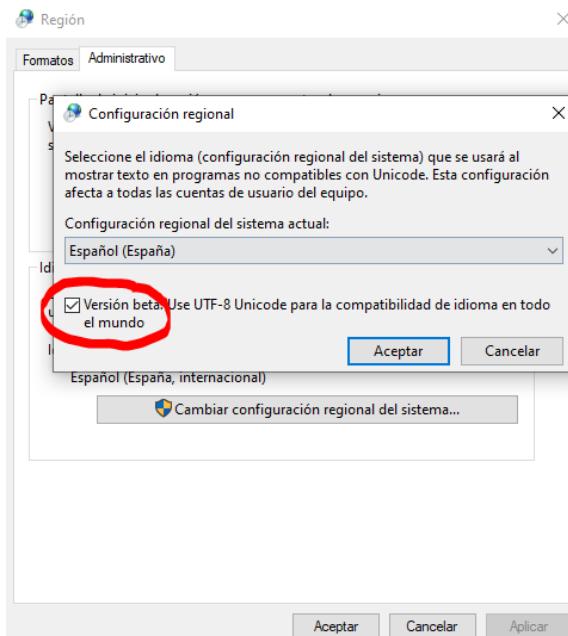
Para arrancar el servidor iremos a los servicios de Windows, buscaremos el servicio MySQL80 (o como le hayamos llamado en la instalación) y lo iniciamos. Ahora usaremos MySQL Workbench para todas nuestras operaciones. Lo primero que debemos hacer es seleccionar nuestro servidor para conectarnos a él (nos pedirá la contraseña de root):



La siguiente operación a realizar es comprobar el estado del servidor: Menú superior *Server > Server Status*. Podemos recibir este error:



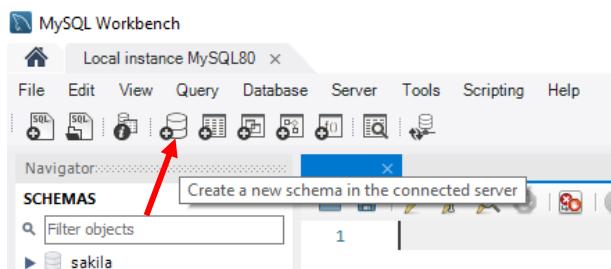
En ese caso, nos dirigimos al Panel de Control de Windows > Región. Pestaña *Administrativo*. Botón [*Cambiar configuración regional del sistema...*] y marcamos el check: *Versión beta. Use UTF-8 Unicode*. Reiniciamos el sistema y reiniciamos el servicio MySQL80.



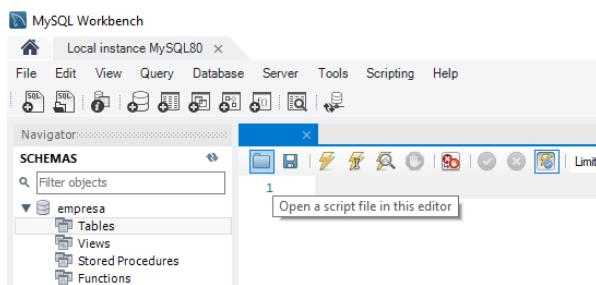
Creación de la Base de Datos

Los pasos para crear una base de datos y llenarla con datos serán los siguientes.

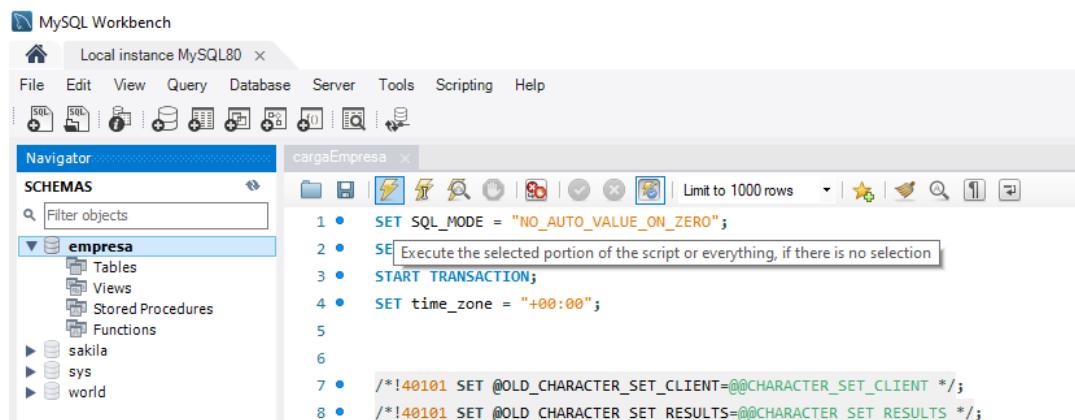
1. Abrir MySQL Workbench y conectarse al servidor.
2. Crear una nueva base de datos (un nuevo esquema) desde el siguiente icono y asignándole un nombre. En el caso de nuestras prácticas será '*empresa*'.



3. Ahora podríamos crear las tablas y llenar los datos mediante el lenguaje SQL, por ejemplo: "*create table nombreTabla...*", "*insert into nombreTabla values ()*", etc. pero también lo podemos hacer mediante un archivo que ya tiene todas esas instrucciones SQL preparadas en forma de script, para que las ejecute, según se indica en el siguiente punto.
4. En el editor de SQL, seleccionamos el icono de abrir un script y seleccionamos el archivo .sql proporcionado por el profesor.



5. Hacemos doble clic sobre la base de datos '*empresa*' en el panel lateral de forma que quede en negrita, y por tanto seleccionada como la base de datos sobre la que ejecutaremos el script y pulsamos en el icono del "rayo" para ejecutarlo.



6. Podemos consultar ahora la tabla '*empleadd*' y veremos las filas insertadas.

Parar el servidor

Para finalizar la sesión, de nuevo desde los Servicios de Windows, buscamos el servicio MySQL, botón derecho > Detener.

Docker

Otra forma de instalar MySQL sería mediante un contenedor Docker. Puedes consultar estos videos:

Usar Docker, hay que instalar Docker Desktop:

- Instalación Docker en Windows: <https://www.youtube.com/watch?v=et7H0EQ8fY>
- Instalación de MySQL: <https://www.youtube.com/watch?v=kphq2TsVRIs>
- Persistencia de Bases de datos en MySQL: <https://www.youtube.com/watch?v=-pzptvcJNh0>

19. Programación Funcional

La forma de programar que hemos seguido a lo largo de este manual se denomina “programación imperativa” ya que se basa en detallar todos los pasos ordenados para la resolución de un problema. En ciertas ocasiones podremos emplear una forma de programar más orientada a lo que queremos obtener y no al detalle de pasos para obtenerlo, ese estilo de programación se llama “declarativa”.

Formalmente, la programación declarativa se define, en contraposición a la programación imperativa, como un paradigma de programación basado en el desarrollo de programas especificando o “declarando” un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución. La solución es obtenida mediante mecanismos internos de control, sin especificar exactamente cómo encontrarla. Un ejemplo de lenguaje puramente declarativo que conocéis bien es SQL.

La **programación funcional** es un tipo de programación declarativa, basada en el uso de funciones, dotándolas de mayor versatilidad. Java, desde su versión 8, permite la programación funcional empleando funciones Lambda (a través de las Interfaces Funcionales) y el API Stream.

Como ejemplo de lo que vamos a aprender en este tema, supongamos el siguiente enunciado: *Crea un conjunto (Set) a partir de una lista llamada ‘números’, que contenga el resultado de elevar al cuadrado todos sus elementos que sean pares*. En programación imperativa necesitaríamos un “for” para recorrer cada elemento de la colección, dentro del bucle un “if” para verificar la condición de si el número es par o no, y en caso afirmativo calcular el cuadrado y a continuación guardar dicho valor en un conjunto. Todo ello, con programación funcional se reducirá a:

```
Set cuadradosPares = numeros.stream()
    .filter(x -> x%2==0)
    .map(x->x*x)
    .collect(Collectors.toSet());
```

Pero para llegar entender este código basado en *Streams*, tenemos que ver previamente lo que son las interfaces funcionales y las funciones Lambda.

Interfaces Funcionales

Las interfaces definían métodos que suponían un “un compromiso” que debían cumplir las clases que la implementaban. Adicionalmente podían tener métodos estáticos, métodos por defecto y métodos privados.

Una interfaz funcional solo puede tener un método abstracto, pudiendo tener métodos por defecto, privados o estáticos. En realidad, se podría matizar esta definición; una interfaz funcional puede tener más de un método abstracto, pero todos menos uno deben ser sobrescritura de métodos de la clases Object (por ejemplo: *toString()*, *equals()*, etc.).

La anotación **@FunctionalInterface** no es obligatoria, pero comprueba en tiempo de compilación si se cumplen las condiciones que comentamos. Este sería un ejemplo:

```
@FunctionalInterface
interface ICalculadora {
    public double calcular (int a, int b);
}
```

Como ya sabemos de capítulos anteriores, necesitaremos crear una clase que implemente la interfaz y por tanto los métodos definidos en ella (salvo los métodos default, estáticos o privados):

```
class Sumador implements ICalculadora {
    @Override
    public double calcular(int i, int j) { return (double) i+j; }
}
```

Podríamos crear más clases que implementasen esa interfaz, como un *Multiplicador* que desarrollaría el código del método de forma distinta.

Finalmente, deberemos crear instancias de la definida e invocar a su método:

```
ICalculadora s = new Sumador();           //también: Sumador s = new Sumador();
System.out.printf("%f%n", s.calcular(3,5));
```

También vimos que las interfaces se podían implementar con una **clase anónima**, que en un solo paso y sin crear la clase explícitamente, crea la instancia de la clase y sobrescribe el método abstracto definido en la interfaz:

```
ICalculadora s = new ICalculadora () {      //no creamos la clase Sumador
    @Override
    public double calcular(int i, int j) {
        return (double) i+j;
    }
}
System.out.printf("%f%n", s.calcular(3,5));
```

Un ejemplo de interfaz funcional es *Comparator*, vista también en capítulos anteriores. Tiene un solo método abstracto: *compare (Object o1, Object o2)*. Esta interfaz la usábamos para ordenar colecciones por distintos atributos. El método *Collections.sort ()* recibe como primer parámetro la lista a ordenar y como segundo parámetro una instancia de una clase que implemente *Comparator*.

Teníamos una clase *Peli*:

```
class Peli {
    public String nombre;
    public int año;

    public Peli(String n, int a) {this.nombre = n; this.año = a; }
}
```

Y una lista de *Peli*:

```
List<Peli> lista = new ArrayList<>();
lista.add(new Peli("Episode 7: The Force Awakens", 2015));
lista.add(new Peli("Episode 4: A New Hope", 1977));
lista.add(new Peli("Episode 1: The Phantom Menace", 1999));
```

Podíamos ordenar la lista por el atributo que deseásemos. Para ello creábamos una clase que implementase *Comparatory* su método *compare ()*. Este método tenía que devolver entero positivo si el primer parámetro era mayor que el segundo según el criterio de ordenación deseado, entero negativo si el segundo parámetro era mayor, o cero si ambos parámetros eran iguales.

Versión con clase anónima:

```
Collections.sort(lista, new Comparator () {
    @Override
    public int compare(Object o1, Object o2) {
        Peli p1 = (Peli) o1; Peli p2 = (Peli) o2;
        return p1.año - p2.año;
    }
});
```

Versión sin clase anónima:

```
class ComparaAño implements Comparator {
    public int compare(Object o1, Object o2) {
        Peli p1 = (Peli) o1; Peli p2 = (Peli) o2;
        return p1.año - p2.año;
    }
}

ComparaAño compAño = new ComparaAño ();
Collections.sort(lista, compAño);
```

Comprobamos como el usar clases anónimas reduce la cantidad de código.

Otro aspecto interesante de las interfaces (no solo de las funcionales) era que podíamos definirlas sobre tipos genéricos, con lo que aún podemos reducir más el código, omitiendo los castings:

```
Collections.sort(lista, new Comparator <Peli> () {
    @Override
    public int compare(Peli p1, Peli p2) {
        return p1.año - p2.año;
    }
);
for (Peli p : lista) System.out.println(p.nombre + ":" + p.año);
```

Es importante entender todo lo anterior para seguir avanzando en este tema.

Predicate, Consumer, Function y Supplier

Existen de forma predefinida una serie de interfaces funcionales que permiten realizar un gran abanico de operaciones y que serán fundamentales para trabajar con Streams como veremos más adelante y encajarán perfectamente con el uso de expresiones Lambda.

Esas interfaces funcionales son las siguientes: Predicados, Consumidores, Funciones y Proveedores.

Int. Funcional	Método abstracto	Descripción
Predicate	boolean test (T t)	Devuelve true como resultado de evaluar el parámetro pasado. Tiene otros métodos (<i>default</i>) para combinar con éste: <i>or()</i> , <i>and()</i> , <i>negate()</i> .
Consumer	void accept (T t)	Sirve para “consumir” los datos recibidos, por ejemplo, mostrarlos por pantalla. Tiene el método <i>default: andThen()</i> para encadenar con otro consumer.
Function (*)	R apply (T t)	Sirve para transformar un objeto. Recibe un parámetro de un tipo y devuelve otro de un tipo diferente. Tiene los métodos <i>default: andThen(), compose() y identity()</i> .
Supplier	T get()	No recibe parámetros y sirve para obtener objetos. Hay interfaces especializados como <i>IntSupplier, LongSupplier, DoubleSupplier, BooleanSupplier</i> .

(*) Existe BiFunction, para usar si necesitamos recibir dos parámetros

Vamos a ver unos ejemplos. Para su uso es necesario incorporar los siguientes *import*:

```
import java.util.function.Predicate;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;
```

Aunque ahora no le veamos mucha utilidad, luego, con las expresiones Lambda y la API Stream veremos cómo el uso de estas interfaces funcionales simplifica mucho el código.

Ejemplo Predicate:

```
Predicate <String> cadLarga = new Predicate <>() {
    @Override
    public boolean test(String s) {
        if (s.length() > 8) return true;
        return false;
    }
};

if (cadLarga.test("123456789")) System.out.println("Es larga");
else System.out.println("No Es larga");
```

Ejemplo Consumer:

```
Consumer <String> cadena = new Consumer <>() {
    @Override
    public void accept (String s) {
        System.out.println(s);
    }
};

cadena.accept ("123456789");
```

Ejemplo Function:

```
Function <Integer, Long> cuadrado = new Function <>() {
    @Override
    public Long apply (Integer a) {return (long)a * a; }
};

System.out.println(cuadrado.apply(10));
```

Ejemplo Supplier:

```
Random random = new Random();
Supplier <Integer> aleatorio = new Supplier <> () {
    Random random = new Random();
    @Override
    public Integer get () {
        Random random = new Random();
        int n = random.nextInt(10);
        return n; }
};

System.out.println(aleatorio.get());
```

Funciones Lambda

A continuación, veremos lo que son las funciones Lambda, pero ya podemos adelantar que allá donde haya una interfaz funcional, **podremos emplear una expresión Lambda para definir su método abstracto**, de una forma más intuitiva y con menos código que con la programación imperativa.

Las expresiones lambda son funciones anónimas, sin nombre cuya sintaxis básica se detalla a continuación:

`(parámetros) -> { cuerpo de la función }`

El operador lambda (`->`) separa la declaración de parámetros de la declaración del cuerpo de la función.

Parámetros:

- Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
- Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.
- Los argumentos de una función Lambda pueden ser declarados explícitamente o a su vez pueden ser inferidos por el compilador de acuerdo al contexto. Veremos más adelante como hace esto. Ejemplos:
 - `(int x) -> {cuerpo}`
 - `(int x, int y....) -> {cuerpo}.`
 - `x -> {cuerpo}`
 - `()-> {cuerpo}.`

Cuerpo de lambda:

- Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la cláusula `return` en el caso de que deban devolver valores. Se hace un `return` implícito del valor calculado en esa única línea.
- Cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la cláusula `return` en el caso de que la función deba devolver un valor. Ejemplos:
 - `z -> z + 2`
 - `() -> System.out.println("Mensaje 1")`
 - `n -> {System.out.print(n + " ");}`
 - `(int longitud, int altura) -> { return altura * longitud; }`
 - `(String x) -> {
 String retorno = x;
 retorno = retorno.concat(" ***");
 return retorno; }`
 - `(int a, int b) -> { return a + b; }.`
 - `() -> { return 3.1415 }.`

Usando la función Lambda:

Las funciones Lambda no se pueden invocar directamente en el código, sino que **precisan estar definidas como implementación del método abstracto de una interfaz funcional**. Así pues, en el ejemplo creado en el apartado anterior:

```
@FunctionalInterface
interface ICalculadora {
    public double calcular (int a, int b);
}
```

Podríamos implementar (sobrescribir) el método *calcular()* mediante una Lambda. Así lo habíamos hecho sin Lambda, con una clase anónima:

```
ICalculadora s = new ICalculadora () { //no creamos la clase Sumador
    @Override
    public double calcular(int i, int j) {
        return (double) i+j;
    }
};
```

Y así lo haríamos con la función Lambda en clase anónima:

```
ICalculadora s = (x,y) -> x+y; //return implícito
```

En cualquiera de los dos casos, hay que invocar al método:

```
System.out.printf("%f%n", s.calcular(3, 5));
```

Vemos como no es necesario hacer *new* ni especificar el método que sobrescribe, ya que solo hay uno. Tampoco es necesario indicar los parámetros porque se infieren de la definición de la interfaz.

El otro ejemplo visto de interfaz funcional, de Comparator:

```
Collections.sort(lista, new Comparator <Peli> () {
    @Override
    public int compare(Peli p1, Peli p2) {
        return p1.año - p2.año;
    }
});
```

Quedaría así con expresiones Lambda:

```
Collections.sort(lista, (p1, p2) -> p1.año - p2.año);
```

Predicate, Consumer, Function y Supplier

Como acabamos de comentar, las funciones Lambda están vinculadas a una interfaz funcional y vimos que Java tiene predefinidas una serie de ellas (*Predicate*, *Consumer*, *Function* y *Supplier*) que nos permiten realizar muchas tareas.

Vamos a reescribir los ejemplos de estas interfaces vistas en el apartado anterior, ahora empleando funciones Lambda.

Predicate sin Lambda vs. con Lambda

<pre>Predicate <String> cadLarga = new Predicate <>() { @Override public boolean test(String s) { if (s.length() > 8) return true; return false; } };</pre>
<pre>Predicate<String> cadLarga = (s) -> s.length() > 8;</pre>

Consumer sin Lambda vs. con Lambda

```
Consumer <String> cadena = new Consumer <>() {
    @Override
    public void accept (String s) {
        System.out.println(s);
    }
};

Consumer <String> cadena = (s) -> System.out.println(s);
```

Function sin Lambda vs. con Lambda

```
Function <Integer, Long> cuadrado = new Function <>() {
    @Override
    public Long apply (Integer a) {return (long)a * a; }

};

Function<Integer, Long> cuadrado = a -> (long) a * a;
```

Supplier sin Lambda vs. con Lambda

```
Random random = new Random();
Supplier <Integer> aleatorio = new Supplier <> () {
    Random random = new Random();
    @Override
    public Integer get () {
        Random random = new Random();
        int n = random.nextInt(10);
        return n;
    }
};

Random random = new Random();
Supplier <Integer> aleatorio = () -> random.nextInt(10);
```

Referencias a métodos

Con las **referencias a métodos** no sólo se puede utilizar expresiones lambda para implementar la interfaz funcional sino que se puede hacer referencia a los métodos del objeto utilizando el operador *double colon*, dos puntos dobles, :: y sustituyen a una expresión lambda.

Nr	Method Reference Type	Method Reference	Lambda expression
1	Static method	String::valueOf	(int i) -> String.valueOf(i)
2	Instance method of a particular object	s::substring	(int beg, int end) -> s.substring(beg, end)
3	Instance method of an arbitrary object	String::equals	(String s1, String s2) -> s1.equals(s2)
		JLabel::getIcon	(JLabel lb) -> lb.getIcon()
4	Constructor	String::new	() -> new String()

Es muy típico ver la referencia de métodos en las colecciones, en su método **forEach**, que recibe una interfaz funcional representada mediante esta referencia de métodos.

No confundir con el bucle **for each** que llevamos utilizando todo el curso

```
ArrayList1.forEach(System.out::println);
```

También resulta muy visual la referencia al método estático *comparing()* de la interfaz funcional *Comparator* que devuelve un comparador implementando el método *compare()* sobre el atributo pasado.

```
class Peli {
    private String nombre;
    private int año;
    public Peli (String n, int a) {this.nombre = n; this.año = a; }
    public int getAño () {return año;}
    public String getNombre () {return nombre;}
    @Override
    public String toString () {return nombre+ " (" +año+"); }
}

List<Peli> lista = new ArrayList<>();
lista.add(new Peli("Episode 7: The Force Awakens", 2015));
lista.add(new Peli("Episode 4: A New Hope", 1977));
lista.add(new Peli("Episode 1: The Phantom Menace", 1999));
Collections.sort(lista, Comparator.comparing(Peli::getAño));
lista.forEach(System.out::println);
```

API Stream

A través del API Stream podemos trabajar sobre colecciones de una manera limpia y clara, evitando bucles y algoritmos que ralentizan los programas y hacen complejo entender su funcionalidad.

Existen 3 partes que componen un Stream que de manera general serían:

1.- Un Stream funciona a partir de una lista o colección, que también se la conoce como la fuente de donde obtienen información. El método **stream()** convertirá la colección en *Stream* para comenzar su tratamiento.

2.- Operaciones intermedias que actúan sobre el Stream. Estas son solo algunas:

- **map**: Obtiene un nuevo *Stream* resultado de aplicarle una función a cada elemento (en general una expresión Lambda que implementa la interfaz funcional *Function*).
- **filter**: selecciona elementos según la expresión pasada como parámetro que será una implementación de la interfaz funcional *Predicate*, generalmente mediante una expresión Lambda.
- **sorted**: ordena el *Stream*. Lo hace por el criterio por defecto de la clase (*Comparable*) o con el comparador pasado como parámetro, generalmente como expresión Lambda.
- **skip (n)**: elimina de *Stream* los 'n' primeros de elementos del *Stream*.
- **limit (n)**: se queda solo con los 'n' primeros elementos del *Stream*, eliminando los sobrantes.

3.- Operaciones terminales. Son la última operación que se hace con el *Stream*. Ejemplos:

- **collect**: Obtienen una colección con el resultado de los procesos previos aplicados al Stream.
- **forEach**: Itera sobre cada elemento de Stream (el método *peek()* valdría para iterar sobre el Stream como operación intermedia, no terminal).

- **reduce:** Permite realizar un cálculo sobre los elementos del Stream, utilizando un operador binario para definir la operación. Tiene muchas posibilidades, por ejemplo:
 - `.reduce(0, (a,b) -> a+b)` acumula, sería como $a+=b$, empezando con $a=0$.
 - `.reduce(Integer::sum)` hace lo mismo que la anterior, pero devuelve un *Optional*. Esta es una clase que almacena un valor, gestionando los valores nulos. Para obtener su valor hay que ejecutar su método `get` (luego lo veremos en los ejemplos)
 - `.reduce(Integer::min)` Análogo al anterior, pero con el mínimo.
 - `.reduce(Integer::max)` Análogo al anterior, pero con el máximo.
- **count:** Obtiene un *long* con la cantidad de elementos del Stream una vez procesado.
- **sum:** Obtiene la suma de los valores, previamente debemos hacer un `mapToInt(Integer::intValue)`

Para entender todas estas operaciones lo mejor es ver ejemplos en funcionamiento. Partiremos en todos los casos de una lista con los 10 primeros enteros.

```
List <Integer> numeros = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
```

Ejemplo 1: Mostrar los elementos del ArrayList inicial elevados al cuadrado.

```
numeros.stream()
    .map(x->x*x)
    .forEach(y->System.out.println(y));
```

O también:

```
numeros.stream()
    .map(x->x*x)
    .forEach(System.out::println);
```

A veces, se presenta todo en una sola línea, pero se entiende peor:

```
numeros.stream().map(x->x*x).forEach(System.out::println);
```

Ejemplo 2: Obtener una nueva lista con elementos de la lista inicial elevados al cuadrado.

```
List cuadrados = numeros.stream()
    .map(x->x*x)
    .collect(Collectors.toList());
```

Ejemplo 3: Obtener un Set con elementos de la lista inicial que sean pares elevados al cuadrado:

```
Set cuadradosPares = numeros.stream()
    .filter(x -> x%2==0)
    .map(x->x*x)
    .collect(Collectors.toSet());
```

Ejemplo 4: Mostrar el cuadrado de los elementos de las posiciones 4 a 8, ambas incluidas (empezando en cero).

```
numeros.stream()
    .skip(4)
    .limit(5)
    .map(x->x*x)
    .forEach(System.out::println);
```

Ejemplo 5: Obtener la suma de los elementos impares:

Este ejercicio podemos hacerlo de varias formas:

```
int res1 = numeros.stream()
    .filter(x -> x%2!=0)
    .reduce(0, (a, b) -> a + b);
System.out.println(res1);
```

O también:

```
int res2 = lista.stream()
    .filter(x -> x%2!=0)
    .mapToInt(Integer::intValue)
    .sum();
System.out.println(res2);
```

Y también con esta última versión:

```
Optional<Integer> res3 = lista.stream()
    .reduce(Integer::sum);
System.out.println(res3.get());
```

Ejemplo 6: A partir de un ArrayList de String, hacer una lista con los que empiecen por "A":

```
List <String> textos = Arrays.asList("Alfa", "Bravo", "Charlie", "Aback");
textos.stream()
    .filter(s->s.startsWith("A"))
    .forEach(System.out::println);
```

Ejemplo 7: A partir de una lista de "Peli" mostrar los títulos de las películas posteriores al año 1998, ordenadas alfabéticamente.

```
lista.stream()
    .filter(x -> x.getAño() > 1998)
    .map(x -> x.getNombre())
    .sorted()
    .forEach(System.out::println);
```

Ejemplo 8: A partir de una lista de "Peli" mostrar el título de la película más antigua:

```
Optional<Peli> var = lista.stream()
    .min( (a,b)->a.getAño()-b.getAño());
System.out.println(var.get().getNombre());
```

Tratar Ficheros como Streams

La programación funcional se puede aplicar también al tratamiento de ficheros. Podemos convertir el contenido de un fichero en un Stream y luego aplicarle todas las operaciones que acabamos de ver.

El siguiente ejemplo, leería un fichero de texto línea a línea y mostraría por pantalla en mayúsculas aquellas líneas de menos de 10 caracteres:

```
String fichero = "fichero.txt";
try (Stream<String> stream = Files.lines(Paths.get(fichero))) {
    stream.filter (x -> x.length() < 10)
        .map(String::toUpperCase)
        .forEach(System.out::println);
} catch (IOException e) {e.printStackTrace();}
```

El mundo de los Streams y la programación funcional es muy amplio, con muchas más posibilidades de las vistas en este manual.

Últimas Consideraciones

Para terminar este manual deberíamos señalar que a lo largo de toda la materia y las prácticas hemos programado de una forma más o menos intuitiva y organizada. Si bien es cierto que comentamos ligeramente el patrón MVC y siempre procuramos separar la lógica de la presentación (bien por consola o bien mediante interfaz gráfica Swing) no hemos seguido patrones de diseño ni técnicas sofisticadas de programación que estábamos aprendiendo conceptos nuevos partiendo dese cero y no era el momento apropiado.

Ahora, una vez afianzado todo este conocimiento, sí es momento de empezar a trabajar esos aspectos de diseño que nos van a permitir que nuestro software sea más escalable, más fácil de entender, más fácil de probar y más fácil de mantener.

Sería interesante entonces, estudiar asuntos como:

- Patrones de diseño: (*ver <https://refactoring.guru/es/design-patterns>*)
- Uso de estándares y convenciones: nomenclatura de elementos (variables, métodos, programas, etc.), arquitectura de los proyectos, estructura y apariencia del código, etc.
- Refactorización.
- Principios SOLID: (*ver <https://devexperto.com/principios-solid/>*)
- Evitar acoplamiento de código.
- Diseñar orientado a interfaces.

En cuanto a la plataforma, el curso se ha basado en Java SE para aplicaciones de escritorio. El siguiente paso sería dar el salto a aplicaciones de dispositivos (Java ME) y de aplicaciones de servidor (Java EE). El desarrollo web probablemente el mercado más amplio hacia el que se dirija cualquier desarrollador en Java. Sería interesante entonces aprender *frameworks* de desarrollo orientados sobre todo al mundo web. Actualmente *Spring* figuran entre los más populares.

Además de lo que es la programación en sí, todo programador debe conocer tecnologías complementarias para desarrollar las aplicaciones. Destacamos:

- Gestión de dependencias (Maven)
 - Testing (JUnit, Mockito) y documentación (JavaDoc)
 - Gestión de versiones (Git, Github)
 - Contenedores (Docker, Kubernetes)
 - Herramientas DevOps
 - Planificación de proyectos (Scrum)
- ...y por supuesto, profundos conocimientos en seguridad y en bases de datos relacionales y NoSQL.

Anexos

Instalación de JDK y NetBeans

Instalación

En este curso instalaremos Netbeans en su versión 14. Como este IDE está escrito en Java, necesitaremos el entorno de ejecución de Java aunque quisiésemos desarrollar aplicaciones en otro lenguaje. En nuestro caso, que además vamos a programar en Java debemos instalar el JDK (*Java Development Kit*). Emplearemos en este caso la versión **jdk 17 de Oracle**.

JDK es un conjunto de herramientas para desarrollar aplicaciones Java e incluye el entorno de ejecución JRE y el compilador Java. JRE (*Java Runtime Environment*) es el entorno necesario para ejecutar las aplicaciones Java e incluye entre otros la JVM (*Java Virtual Machine*) encargada de interpretar el bytecode Java. Hay una versión oficial de Oracle, con restricciones de uso, y otras versiones libres como puede ser OpenJDK.

Dispones de enlaces de descarga tanto de Netbeans como jdk desde la URL mostrada en la primera página de este manual con su código QR.

Por lo tanto, la instalación constará de tres pasos:

- 1) Instalación de JDK: consiste en la descompresión del zip descargado en cualquier carpeta o bien la ejecución del archivo, en el caso de que descargásemos un ejecutable.
- 2) La instalación de Netbeans consiste en la descompresión del archivo archivo .zip descargado. En Windows podemos descomprimirlo en *c:\Archivos de Programa*. Se creará una carpeta llamada *netbeans*, que si queremos podemos renombrar a *netbeans14* o a lo que queramos.
- 3) Antes de la primera ejecución hay que informar a Netbeans de dónde está instalado Java. Para ello debemos **editar el archivo *netbeans.conf* situado en la carpeta *etc***, en la ruta donde esté instalado el jdk de Java (*otra opción sería configurar la variable PATH dentro de las variables de entorno de nuestro sistema Windows para incorporar la ruta de los archivos de Java: *jdk/bin**).

```

C:\Program Files\NetBeans 14\etc\netbeans.conf - Notepad++
Archivo Editar Buscar Vista Codificación Lenguaje Configuración Herramientas Macro Ejecutar Plugins Ventana ?
netbeans.conf
72 # Be careful when changing jdkhome.
73 # There are two NetBeans launchers for Windows (32-bit and 64-bit) and
74 # installer points to one of those in the NetBeans application shortcut
75 # based on the Java version selected at installation time.
76 #
77 netbeans_jdkhome="C:\Program Files\Java\jdk-17"
78 #
79 # Additional module clusters:
80 # using ${path.separator} (';' on Windows or ':' on Unix):
81 #

```

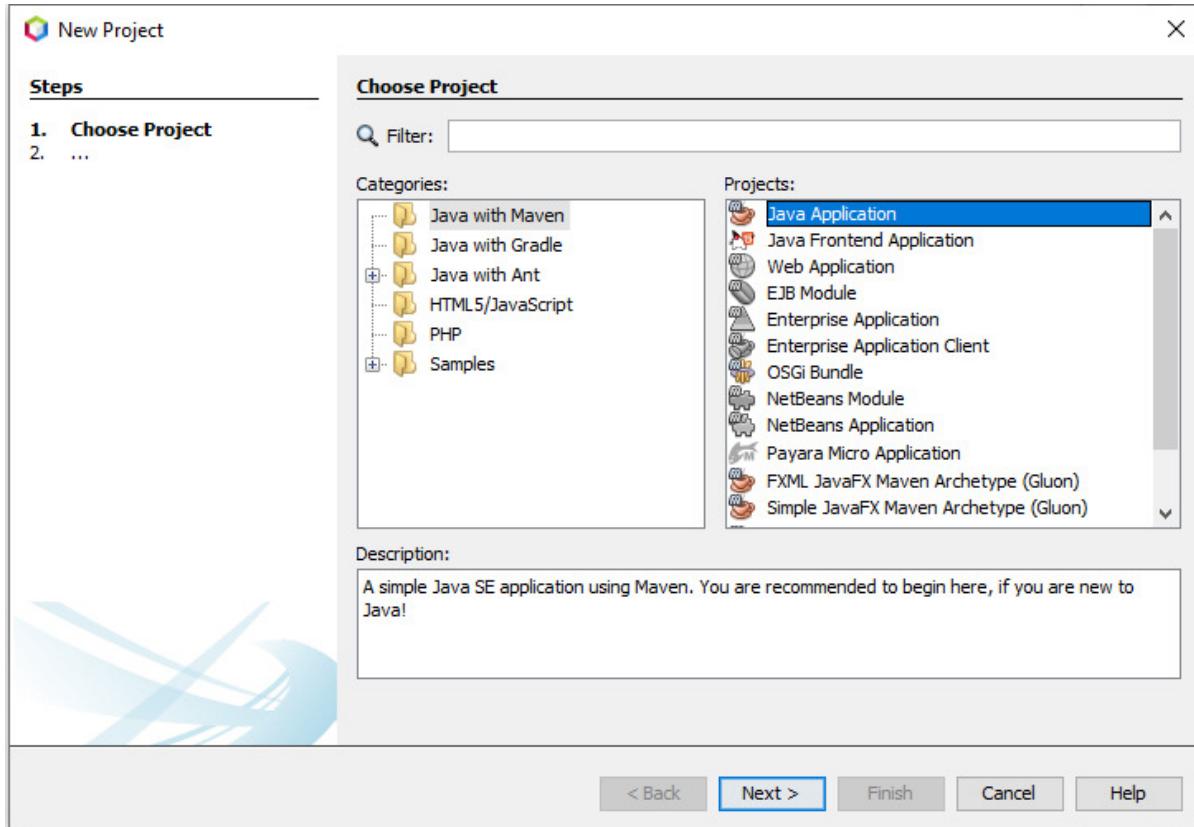
Como último paso, podemos crear en el escritorio un ícono del ejecutable situado en *c:\Archivos de Programa\NetBeans\bin* para acceder al IDE de una forma más cómoda.

Aunque no es necesario, es aconsejable añadir a la variable de entorno PATH la ruta de la carpeta */bin* del JDK (por ejemplo: "*C:\Program Files\Java\jdk-17\bin*") y crear una variable de entorno llamada JAVA_HOME con la ruta en la que está instalado el JDK (por ejemplo: "*C:\Program Files\Java\jdk-17*").

Crear un programa

NetBeans permite editar un archivo independiente, pero donde se saca todo el provecho al IDE es utilizando un proyecto, es decir, una carpeta con una estructura controlada por NetBeans que puede tener archivos de diferentes tipos relacionados que darán lugar a una aplicación completa.

Para crear un proyecto nuevo: menú *File> New Project* indicando el tipo de proyecto, en nuestro caso usaremos *Java with Maven > Java application* aunque también podríamos hacer proyectos bajo Gradle o Ant.



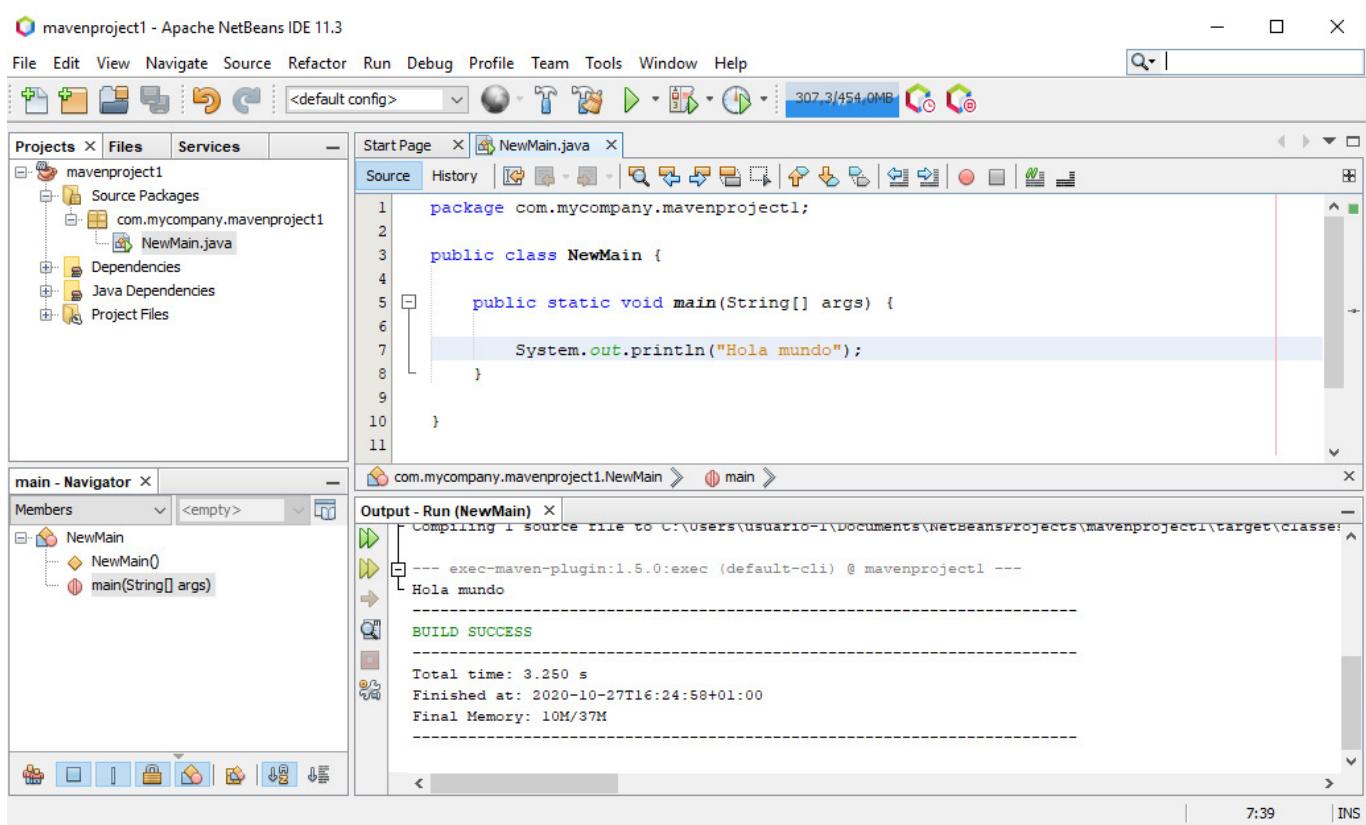
En la siguiente ventana indicaremos el nombre del proyecto, dejando el resto de parámetros con su configuración por defecto en estos primeros pasos.

Ahora en la ventana de proyectos, sobre *Source Packages*, en el nombre del paquete, haremos botón derecho *New > Java Main Class*. La primera vez descargará e instalará los plugins necesarios.

En el menú File disponemos de las funciones para guardar nuestro proyecto, exportarlo a zip, etc.

Ejecutar el programa

Para ejecutar nuestro programa, menú *Run > Run Project*, o simplemente pulsando F6. Esta operación compilará nuestro código (archivos *.java*) generando los bytecode (archivos *.class*). Si se produce algún error nos lo comunicará y en caso de que todo esté correcto ejecutará el programa.



En la ventana *Output* comprobaremos el resultado de la ejecución.

En este curso haremos muchos programas (esto es, clases con método *main* que se pueden ejecutar) y tener un proyecto para cada programa puede ser tedioso ya que generaría multitud de carpetas y archivos. Para evitar esto, podemos crear varios programas en un mismo proyecto, incluso en el mismo paquete.

El único problema será que a la hora de ejecutar el proyecto mediante *Run* (ó *F6*) solo puede haber una clase “principal” y se ejecutaría siempre el mismo programa. Para solucionar esto, Netbeans nos ofrece la opción *Run File* (*May + F6*) con la que se ejecutaría el programa que tuviésemos activo en pantalla en ese momento.

Funciones interesantes

Netbeans dispone de multitud de funcionalidades para hacernos más sencillo y cómodo la edición y prueba de nuestros programas, y poco a poco las irás descubriendo. A modo de resumen vamos a destacar las siguientes:

Menú File

- **Open, Close, Save, etc...**
- **Import Eclipse Project**
- **Export e Import Project to ZIP**
- **Project Properties:** Carpetas del proyecto, clase main, empaquetado, etc.

Menú Edit

- **Buscar y reemplazar** texto en todo un proyecto
- **Grabación de macros**

Menú View

- **Editors > History:** ver las distintas versiones por las que ha pasado el código solo en la sesión actual. Se genera una versión cada vez cada vez que se guarda el archivo.
- **Split:** parte la ventana en dos para ver distintas partes del mismo programa/clase.
- **Full Screen:** Cntrl+May+ENTER, o Alt+May+ENTER.
- **Barras de herramientas.** Personalizar.

Menú Source

- **Alt + Ins:** generar código: constructor, getters y setters...
- **Cntrl + P:** muestra los parámetros de un método (hay que estar en los paréntesis del método)
- **Cntrl + SPACE:** Completa código
- **Rename Cntrl+R:** renombrar todas las ocurrencias de una variable, etc.
- **Copy Cntrl+C:** Copia una clase en otro paquete

Menú Run

- **May + F6:** compila y ejecuta archivo actual
- **F6:** compila y ejecuta proyecto

Menú Tools

- **Options:** Preferencias de código....
- **Plugins:** Nuevas funcionalidades o comportamiento.

Menú Debug

- **F8 (Step Over)** Ejecuta paso a paso el programa. Si hay una llamada a un método lo ejecuta completamente y vuelve a la línea siguiente a la llamada.
- **F7 (Step Into)** Ejecuta paso a paso el programa. Si hay una llamada a un método, entra en el y lo ejecuta línea a línea.
- **F4 (Run to cursor)** Ejecuta hasta donde tengas el cursor.
- **F5 (Continue)** Continua la ejecución (hasta el final o siguiente breakpoint si lo hay)
- **Clic en el número de línea:** pone/quita un breakpoint en esa línea

Y además:

- En la ventana de código, si tenemos dos o más códigos abiertos, podemos arrastrar unos sobre otros para verlos en paralelo, o en vertical o en horizontal.
- **Zoom in/out:** con ALT+Rueda Ratón
- **Cntr + May + W :** Cierra todas las pestañas de código
- **Cntr + May + I** (o botón derecho: *Fix Imports*) Sobre una clase, mete import necesario.
- **Alt + May + F** (o botón derecho: *Format code*) Indenta el código
- **Alt + F7:** (Search usages) Busca dónde se usa una determinada clase
- **Cntr + May + C:** Pone/quita comentarios a las líneas seleccionadas
- **May + Alt + Flecha arriba/abajo:** Mover línea (o varias si seleccionadas)
- **May + Cntrl + Flecha arriba/abajo:** Clonar línea (o varias si seleccionadas)
- **Cntr + E:** Eliminar línea
- Selección rectangular (en vez de por línea)

Diferencias Tipos primitivos vs. Objetos

Operación	Tipo Primitivo	Objeto
Crear variable	int i, j;	cuenta c1, c2;
Inicializar	i=7; j=3;	c1=new cuenta(1, 100.0);
Crear e inicializar	int i=7;	cuenta c2 = new cuenta(2, 200.0);
Modificar	i++; i=j*2;	c1.ingresar (50d); c2.setSaldo (150d); c1.saldo=100d; //si saldo fuese public
Copiar	i = j;	c1.setId(c2.getId());c1.setSaldo(c2.getSaldo()); ver método clone(); //Error:c1 = c2;
Comparar	i==7 o bien 7 ==i	c1.equals(c2) o bien c2.equals(c1)
Mostrar por pantalla	System.out.print(i);	System.out.print(c1.getSaldo()); System.out.print(c1); //solo ok si toString();
Definir Array	int [] arr = new int[3];	cuenta [] arr = new cuenta[3];
Inicializar Array	int[] arr = new int[]{1,2,3};	cuenta[] arr = new cuenta[] { new cuenta(1,100.0), new cuenta(2,50.5), new cuenta(3,10d) };
Añadir valor en Array	arr[0]=7; arr[1]=arr[0];	arr[0]=new cuenta (2, 20d); //Error: arr[1]=arr[0]; arr[1].setId(arr[0].getId()); arr[1].setSaldo(arr[0].getSaldo());
Devolver valor	public int método (){ int i;... return i; o bien return 7; }	public cuenta método () { cuenta c1; ... return c1; o bien return new cuenta(3,0d); }
Pasar como parámetro	metodo (int x) {...}	metodo (cuenta x) {...}
Miembro de clase	class alumno { String nombre; int [] notas = new int[3]; }	class cliente { String nombre; cuenta [] finanzas = new cuenta[10]; }
Asignar a un miembro	alumno al=new alumno(); al.notas[0]=7; for(int i=0;i<al.notas.length;i++) { al.notas[i]=5; }	cliente cli = new cliente(); cli.finanzas[0]=new cuenta(4,30d); for(int i=0;i<cli.finanzas.length;i++) { cli.finanzas[i]=new cuenta(i,0d); }

```

public class cuenta {
    private int id;
    private double saldo;

    cuenta (int id,double saldo){this.id=id; this.saldo=saldo; }
    public void setId (int id) {this.id = id;}
    public void setSaldo (double saldo) {this.saldo = saldo;}
    public int getId () {return this.id;}
    public double getSaldo () {return this.saldo;}
    public void ingresar(double imp) {this.saldo+=imp;}
    public void retirar(double imp) {if (imp<this.saldo) this.saldo-=imp;}
    public boolean equals (cuenta z) {if (this.id==z.id) return true;return false;}
    public String toString (){return String.format("%d>>%2f",this.id,this.saldo);}
}

```

Conversiones entre tipos/clases

De	A:	Código
String	StringBuilder	<pre>String str = "Hola"; StringBuilder sb = new StringBuilder(str);</pre>
StringBuilder	String	<pre>StringBuilder sb= new StringBuilder("Hola"); String str= sb.toString();</pre>
String	char	<pre>char ch = str.charAt(0);</pre>
char	String	<pre>String str=String.valueOf('a');</pre>
String	int	<pre>int i = Integer.parseInt(str);</pre>
int	String	<pre>String str = Integer.toString(i); O también: String str = String.valueOf(i);</pre>
int	Integer	<pre>Integer I = new Integer(i);</pre>
Integer	int	<pre>int i = I.intValue();</pre>
Array	ArrayList	<pre>Integer [] arr = {1,2,3}; List aL = Arrays.asList(arr);</pre>
ArrayList	Array	<pre>ArrayList <Integer> aL = new ArrayList <>(); aL.add(1);aL.add(2);aL.add(3); Integer[] arr = new Integer[aL.size()]; arr = aL.toArray(arr);</pre>

En los casos anteriores 'int' es extrapolable a 'long', 'float', 'double', etc. y por otra parte 'Integer' es extrapolable a 'Long', 'Float', 'Double', etc.

Proyecto Lombok

Lombok es una librería que, a través de anotaciones, reduce el código común que tenemos que codificar ahorrándonos tiempo y mejorando la legibilidad del mismo. Con esas anotaciones se pueden generar de forma automática getters, setters, constructores, etc. y esas transformaciones en el código se realizan en tiempo de compilación.

Tiene multitud de anotaciones, que se pueden emplear a nivel atributo, método, clase, etc. Estas serían algunas de las más utilizadas:

- **@Getter** : genera getter público para el atributo. Si lo podemos antes de la clase, lo genera para todos sus atributos. Los getters comienzan por 'get' salvo para atributos de tipo boolean que comienzan por 'is'
- **@Setter**: Análogo a getter, pero generando setters.
- **@EqualsAndHashCode**: Genera los métodos equals y hashCode de la clase. Por defecto, ambos métodos se basarán en todos los atributos de la clase, aunque se puede modificar este comportamiento como veremos a continuación.
- **@ToString**: Genera una cadena con el nombre de la clase y con cada atributo y su valor separado por comas.
- **@NoArgsConstructor**: genera el constructor por defecto.
- **@RequiredArgsConstructor**: genera el constructor con parámetros finales o marcados con **@NonNull**
- **@AllArgsConstructor**: genera el constructor con parámetros para todos los atributos.
- **@Data**: es de los más utilizados y agrupa las anotaciones: Getter, Setter, ToString, EqualsAndHashCode y RequiredArgsConstructor.
- **@Builder**: genera un método para instanciar la clase de una forma más legible que con un constructor y desacopla dicha instantiación, de forma que, aunque en un futuro cambien los constructores de la clase, la instantiación con builder seguirá funcionando.

Estas anotaciones y muchas otras son parametrizables, de forma que podemos adaptar su comportamiento a nuestras necesidades. Por ejemplo, si queremos que los métodos equals y hashCode solo tengan en cuenta un atributo para la comparación, podemos parametrizarlo así:

```
@EqualsAndHashCode(of="atributo")
```

Para emplear lombok en un proyecto Maven solo debemos buscar su dependencia en el repositorio oficial: <https://mvnrepository.com/> e incluir su dependencia en el pom.xml de nuestro proyecto.

Project Lombok » 1.18.24

Spice up your java: Automatic Resource Management, automatic generation of getters, setters, equals, hashCode and toString, and more!

License	MIT
Categories	Code Generators
HomePage	https://projectlombok.org
Date	(Apr 18, 2022)
Files	pom (1 KB) jar (1.9 MB) View All
Repositories	Central
Used By	15,754 artifacts

Maven Gradle Gradle (Short) Gradle (Kotlin) SBT Ivy Grape Leiningen
Buildr

```
<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.24</version>
    <scope>provided</scope>
</dependency>
```

Include comment with link to declaration

Ahora, en la clase que deseemos, añadiremos las anotaciones que generarán el código, por ejemplo:

The screenshot shows the NetBeans IDE interface. On the left, the 'Navigator' window is open, showing the class structure of 'Alumno'. It lists various methods and fields, such as 'Alumno(String nombre, int edad, boolean graduado)', 'Alumno()', 'builder() : AlumnoBuilder', and several equals and hashCode methods. On the right, the 'Source' window displays the generated Java code for the 'Alumno' class, which includes imports for Lombok annotations, the class definition with private fields and their getters, and the 'AlumnoBuilder' inner class.

```

package com.mycompany.mavenproject4;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode(of = {"nombre", "edad"})
@Builder
public class Alumno {
    private String nombre;
    private int edad;
    private boolean graduado;
}

class AlumnoBuilder {
    ...
}

```

En este caso se generarían getters, setters, `toString`, el constructor con todos los parámetros, el constructor sin parámetros, y los métodos `equals` y `hashCode` basados en el atributo `nombre`. También el método `builder` que comentamos antes.

Si trabajamos en Netbeans, en la ventana *Navigator* podemos ver la firma de todos los métodos generados.

Ahora podemos emplear todo el código generado como si lo hubiésemos escrito nosotros mismos:

```

Alumno a1 = new Alumno("Pepe", 13, false);
Alumno a2 = Alumno.builder()
    .nombre("Pepe")
    .edad(12)
    .build();
a2.setGraduado(true);
if (a1.equals(a2)) System.out.println(a1.toString());

```

Enlaces de Interés

- Documentación oficial. Especificación API con las clases, atributos, métodos, etc.
<https://docs.oracle.com/en/java/javase/14/docs/api/index.html>
- JDK:
<https://www.oracle.com/es/java/technologies/javase-downloads.html>
- Dudas frecuentes:
<https://stackoverflow.com/>
- Manuales web:
<https://javiergarciaescobedo.es/programacion-en-java>
<https://www.javatpoint.com/>
<https://guru99.es/java-tutorial/>
<https://www.discoduroderoer.es/curso-java/>
- Gestor de paquetes (dependencias Maven):
<https://mvnrepository.com/>
- IDE Netbeans y documentación
<https://apache.netbeans.org/>
- Patrones De Diseño:
<https://www.geeksforgeeks.org/design-patterns-set-1-introduction/>
- MVC
<http://aalmiray.github.io/griffon-patterns/>
- Frameworks
<https://curiotek.com/2017/06/16/java-que-es-spring/>