

# Proyecto **FPMAD***digital*

*Recursos digitales y multimedia para Formación Profesional*



**Comunidad  
de Madrid**

Dirección General  
de Educación Secundaria,  
Formación Profesional  
y Régimen Especial

CONSEJERÍA DE EDUCACIÓN,  
UNIVERSIDADES, CIENCIA  
Y PORTAVOCÍA



Unión Europea

Fondo Social Europeo

*“El FSE invierte en tu futuro”*

Financiado como parte de la respuesta  
de la Unión a la pandemia de COVID-19

# CFGS Desarrollo de Aplicaciones Multiplataforma

módulo profesional

0486 - Acceso a Datos

unidad didáctica

01 Ficheros, colecciones y data frames

resultados de aprendizaje

01 Desarrolla aplicaciones que gestionan información almacenada en ficheros identificando el campo de aplicación de los mismos y utilizando clases específicas.



Comunidad  
de Madrid

Dirección General  
de Educación Secundaria,  
Formación Profesional  
y Régimen Especial

CONSEJERÍA DE EDUCACIÓN,  
UNIVERSIDADES, CIENCIA  
Y PORTAVOCÍA



Unión Europea

Fondo Social Europeo

“El FSE invierte en tu futuro”

Financiado como parte de la respuesta  
de la Unión a la pandemia de COVID-19

# **Resultados de aprendizaje y unidades didácticas**

# Resultados de aprendizaje y unidades didácticas

RESULTADOS DE APRENDIZAJE						UNIDAD DIDÁCTICA
1	2	3	4	5	6	
X						1.- Ficheros, colecciones y data frames
				X		2.- Manejo de XML
	X	X				3.- Procesamiento de BBDD Relacionales
			X			4.- Uso de BBDD NoSQL
		X	X		X	5.- Programación de componentes de acceso a datos

# **Unidades didácticas y materiales asociados**

# Unidades didácticas y materiales multimedia

RRAA						UDD	Material Multimedia
1	2	3	4	5	6		
X						1.- Ficheros, colecciones y data frames	1.1 Contenidos básicos 1.2 Ejemplos aplicados
				X		2.- Manejo de XML	2.1 Contenidos básicos 2.2 Ejemplos aplicados
	X	X				3.- Procesamiento de BBDD Relacionales	3.1 Contenidos básicos 3.2 Ejemplos aplicados
			X			4.- Uso de BBDD NoSQL	4.1 Contenidos básicos 4.2 Ejemplos aplicados
		X	X		X	5.- Programación de componentes de acceso a datos	5.1 Contenidos básicos 5.2 Ejemplos aplicados

# **Repositorios de materiales y prácticas**

# Repositorio de materiales y prácticas

**Todos los proyectos mostrados, así como otros materiales utilizados en las unidades didácticas los podrás encontrar completos en:**

**<https://github.com/joseluisgs/FP-NextGen-AccesoDatos>**

Cualquier error o propuestas de mejora se publicarán en el repositorio indicado.  
Gracias por tu colaboración.



# Contenidos

1. **Ficheros y directorios**
2. **Expresiones regulares**
3. **Clases Genéricas**
4. **Colecciones**
5. **Operaciones sobre colecciones**

# Ficheros y directorios

# Ficheros y directorios

**Ficheros de texto:** cuando el contenido del fichero contenga exclusivamente caracteres de texto (podemos leerlo con un simple editor de texto)

**Ficheros binarios:** son los ficheros que no estén compuestos exclusivamente de texto. Pueden contener imágenes, videos, ficheros, o combinaciones de cualquier fuente de información.

**Flujos:** son un canal de comunicación de las operaciones de entrada salida. Este esquema nos da independencia para poder trabajar igual tanto si estamos escribiendo en un fichero, como en consola, o si estamos leyendo de teclado, o de una conexión de red.

**Ficheros de acceso aleatorio:** podemos acceder de forma aleatoria a un archivo o fichero. Además, los archivos de este tipo, se pueden leer, o bien leer y escribir a la vez.

# Ficheros y directorios

## Clase File

Nombre	Uso
isDirectory	Devuelve true si el File es un directorio
isFile	Devuelve true si el File es un fichero
createNewFile	Crea un nuevo fichero, si aun no existe.
createTempFile	Crea un nuevo fichero temporal
delete	Elimina el fichero o directorio
getName	Devuelve el nombre del fichero o directorio
getAbsolutePath	Devuelve la ruta absoluta del File
getCanonicalPath	Devuelve la ruta canónica del File
list, listFiles	Devuelve el contenido de un directorio

# Ficheros y directorios

**NIO2.** es la nueva API para el manejo de rutas, ficheros y operaciones de entrada/salida. Es compatible con las interfaces funcionales.

Las principales clases son las siguientes:

- **Path:** es una abstracción sobre una ruta de un sistema de ficheros. Puede usarse como reemplazo completo de `java.io.File`, pero con los métodos `File.toPath()` y `Path.toFile()` se ofrece compatibilidad entre ambas representaciones.
- **Files:** es una clase de utilidad con operaciones básicas sobre ficheros.
- **FileSystems:** otra clase de utilidad para obtener referencias a sistemas de archivos



```
1 private void loadData() {
2     List<Accidente> accidentes;
3     String dataPath = "data" + File.separator + "accidentes.csv";
4     String appPath = System.getProperty("user.dir");
5     Path filePath = Paths.get(appPath + File.separator + dataPath);
6     System.out.println("Loading data from: " + filePath);
7     // Existe usando Paths
8     if (Files.exists(filePath)) {
9         System.out.println("File data exists");
10    } else {
11        System.out.println("File data does not exist");
12    }
13
14    // Leemos los datos...
15    try {
16        accidentes = Files.lines(filePath)
17            .skip(1)
18            .map(this::getAccidente)
19            .collect(Collectors.toList());
20    } catch (Exception e) {
21        System.out.println("Error reading file: " + e.getMessage());
22    }
23 }
```

# Expresiones regulares



# Expresiones regulares

**Una expresión regular** define un patrón de búsqueda para cadenas de caracteres.

Algunos ejemplos de uso de expresiones regulares pueden ser:

- comprobar que la fecha leída cumple el patrón dd/mm/aaaa.
- comprobar que un NIF está formado por 8 cifras, un guión y una letra.
- comprobar que una dirección de correo electrónico es una dirección válida.
- comprobar que una contraseña cumple unas determinadas condiciones.
- comprobar que una URL es válida.
- comprobar cuántas veces se repite dentro de la cadena una secuencia de caracteres determinada.



# Expresiones regulares

## Meta caracteres

- \d: Dígito. Equivale a [0-9]
- \D: No dígito. Equivale a [^0-9]
- \s: Espacio en blanco. Equivale a [ \t\n\x0b\r\f]
- \S: No espacio en blanco. Equivale a [^\s]
- \w: Una letra mayúscula o minúscula, un dígito o el carácter \_
- \_: Equivale a [a-zA-Z0-9\_]
- \W: Equivale a [^\w]
- \b: Límite de una palabra.

## Cuantificadores

- {X}: Indica que lo que va justo antes de las llaves se repite X veces
- {X,Y}: Indica que lo que va justo antes de las llaves se repite mínimo X veces y máximo Y veces. También podemos poner {X,} indicando que se repite un mínimo de X veces sin límite máximo.
- \*: Indica 0 ó más veces. Equivale a {0,}
- +: Indica 1 ó más veces. Equivale a {1,}
- ?: Indica 0 ó 1 veces. Equivale a {0,1}

# Expresiones regulares

**Clase Pattern:** Un objeto de esta clase representa la expresión regular. Contiene el método `compile(String regex)` que recibe como parámetro la expresión regular y devuelve un objeto de la clase `Pattern`.

**La clase Matcher:** Esta clase compara el `String` y la expresión regular. Contienen el método `matches(CharSequence input)` que recibe como parámetro el `String` a validar y devuelve `true` si coincide con el patrón. El método `find()` indica si el `String` contienen el patrón.


## Utilidades:

<https://regex101.com/>

<https://regexr.com/>

<https://www.regextester.com/>

<https://www.freeformatter.com/regex-tester.html>



```
1 private void esEmailValido(String email) {
2     Pattern pat = Pattern.compile("^([\\w-]+(\\.[\\w-]+)*@[A-Za-z0-9]+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})$");
3     Matcher mat = pat.matcher(email);
4     if(mat.find()){
5         System.out.println("Correo Válido");
6     }else{
7         System.out.println("Correo No Válido");
8     }
9 }
```

# Clases Genéricas



# Clases Genéricas

Se trata de una clase parametrizada sobre uno o más tipos. El tipo se asigna en tiempo de compilación

```
1 // Ejemplo de de Interfaz de operaciones CRUD
2 // T es el tipo de objeto a salvar, consultar, ...
3 // ID es el tipo de indentificador del objeto
4 public interface CRUDRepository<T, ID> {
5
6     List<T> findAll() throws SQLException;
7
8     Optional<T> findById(ID id) throws SQLException;
9
10    T save(T entity) throws SQLException;
11
12    T update(ID id, T entity) throws SQLException;
13
14    T delete(ID id) throws SQLException;
15 }
16
```

Los nombres de tipos de parámetros más usados son:

- E (element, elemento)
- K (key, clave)
- N (number, número)
- T (type, tipo)
- V (value, valor)
- S, U, V, ... (2º, 3º, 4º, ... tipo)

# Colecciones



# Colecciones

Las **colecciones** nos permiten trabajar con una serie de objetos o datos de una manera eficiente. Se clasifican en:

- **Listas:** son lineales con posibilidad de orden y permiten elementos repetidos.
- **Conjuntos:** no soporta duplicados, tienen posibilidad de orden de elementos.
- **Mapas:** estructuras de clave-valor con posibilidad de orden de los elementos.

```
1 // Listas de personas
2 var personasAula = new ArrayList<Persona>();
3 // Conjunto de DNI no repetidos
4 var dniConjuntos = new HashSet<String>();
5 // Mapa de pais-capital: clave-valor
6 var paisCapital = new HashMap<String, String>();
```

# Operaciones sobre colecciones



# Operaciones con colecciones

Se recomienda utilizar la **programación funcional** para poder operar con ellos. Muy similar en casi todos los lenguajes.

**Predicate<T>**: comprueba si se cumple o no una condición. Se utiliza mucho junto a expresiones lambda a la hora de filtrar: `.filter(p -> p.getEdad() >= 35)`

**Function<T, R>**: sirve para aplicar una transformación a un objeto. El ejemplo más claro es el mapeo de objetos en otros: `.map(p -> p.getNombre())`

**Consumer<T>**: sirve para consumir objetos. Uno de los ejemplos más claros es imprimir: `.forEach(p-> System.out.println(p.toString()))`

# Operaciones con colecciones

Podemos enfocarlo como una analogía al SQL para enfocarlo con más facilidad.

SQL	API Stream
from	stream()
select	map()
where	filter() (antes de un collecting)
order by	sorted()
distinct	distinct()
having	filter() (después de un collecting)
join	flatMap()
union	concat().distinct()
offset	skip()
limit	limit()
group by	collect(groupingBy())
count	count()

# Operaciones con colecciones

**Procesamiento horizontal (colecciones):** Menos memoria, pero más pasadas. Su comportamiento es paso por paso como una cadena de montaje, se dice que su comportamiento es impaciente o “eager”: *Kotlin Collections*.

**Procesamiento vertical (secuencias):** Más memoria, pero menos pasadas. Su comportamiento es completo elemento por elemento, se dice que es perezosa o “lazy”: *Kotin Sequences y Java Streams*.

**Data Frame:** Estructuras y operaciones optimizadas para procesar masivamente colecciones de datos complejos y estructurados de mucha cantidad de elementos. Usadas para Big Data o ciencias de los datos. *Kotlin Data Frames*.

# Operaciones con colecciones

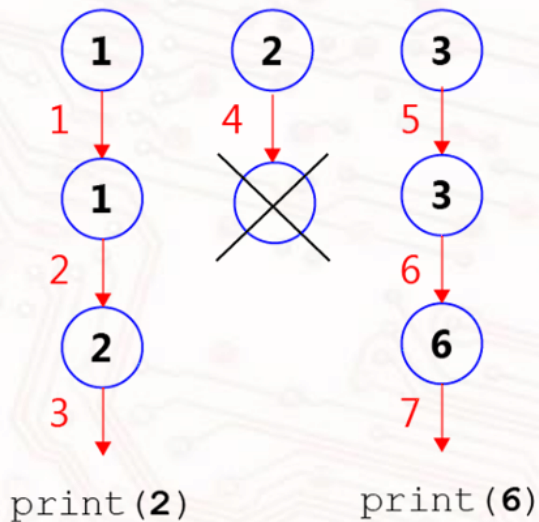
secuencias

colecciones

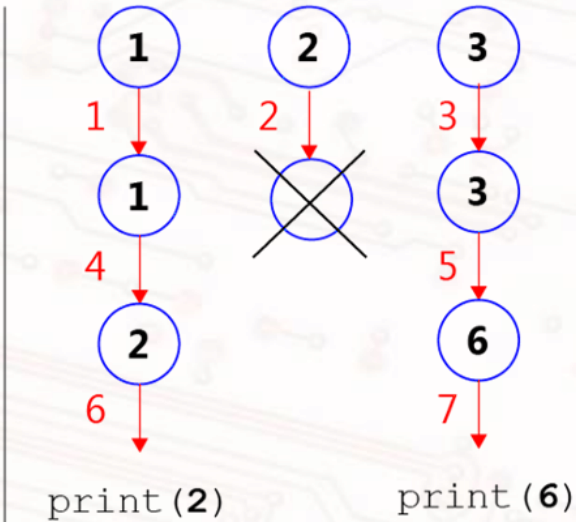
```
filter { it % 2 == 1 }
```

```
map { it * 2 }
```

```
forEach { print(it) }
```



Lazy: elemento por  
elemento



Eager: paso por paso

# Conclusiones



# ¡Vamos con la práctica!

"Menos del 10% del código tienen que ver directamente con el propósito del sistema; el resto tiene que ver con la entrada y salida, validación de datos, mantenimiento de estructuras de datos y otras labores domésticas"  
- Mary shaw



**Comunidad  
de Madrid**

Dirección General  
de Educación Secundaria,  
Formación Profesional  
y Régimen Especial

CONSEJERÍA DE EDUCACIÓN,  
UNIVERSIDADES, CIENCIA  
Y PORTAVOCÍA



**Unión Europea**

Fondo Social Europeo

*“El FSE invierte en tu futuro”*

**Financiado como parte de la respuesta  
de la Unión a la pandemia de COVID-19**