Acceso a Datos

Tema 1 Streams, Ficheros y Expresiones Regulares

José Luis González Sánchez

Contenidos

¿Qué voy a aprender?

Contenidos

1. TDA, Clases Genéricas y Colecciones
2. API Stream
3. Ficheros y Directorios
4. Java NIO2
5. Expresiones Regulares

Clases Genéricas y Colecciones

Generalizando nuestra información

Clase Genérica

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

```java
public class Box {
  private Object object;

  public void set(Object object) {
    this.object = object;
  }
  public Object get() {
    return object;
  }
}
```

Código propenso a producir errores

```java
public class Box<T> {
  private T object;

  public void set(T object) {
    this.object = object;
  }

  public T get() {
    return object;
  }
}
```

Se asigna el tipo a tiempo de compilación

Clase Genérica

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

```java
public class Par<T, S> {
  private T obj1;
  private S obj2;

  // Resto de la clase

}
```

Los nombres de tipos de párametros 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)

Clase Genérica

Para instanciar un objeto genérico, tenemos que indicar los tipos dos veces.

```
Par<String, String> pareja2 = new Par<String, String>("Hola", "Mundo");
```

- Este estilo es muy verboso. Desde Java SE 7 tenemos el operador <>

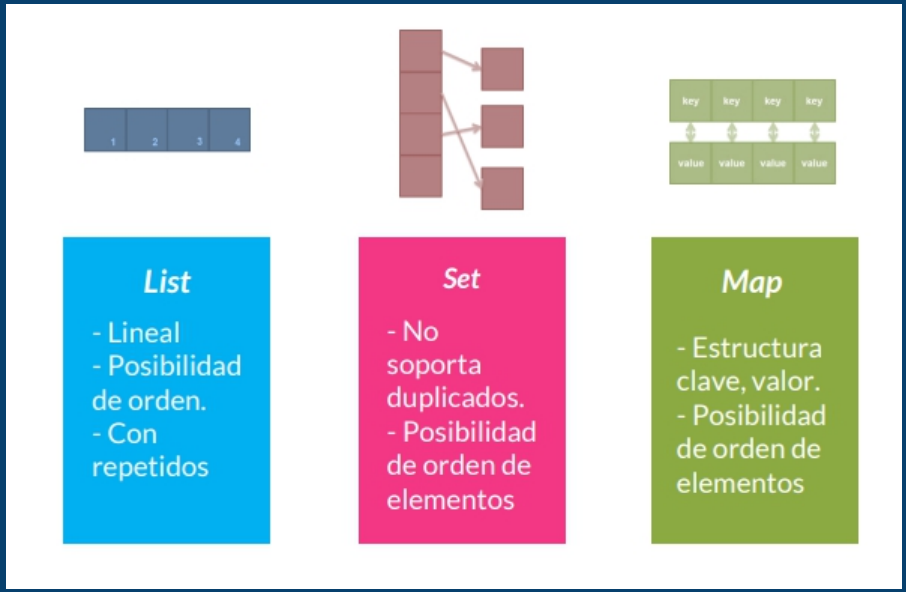
```
Par<String, String> pareja2 = new Par<>("Hola", "Mundo");
```

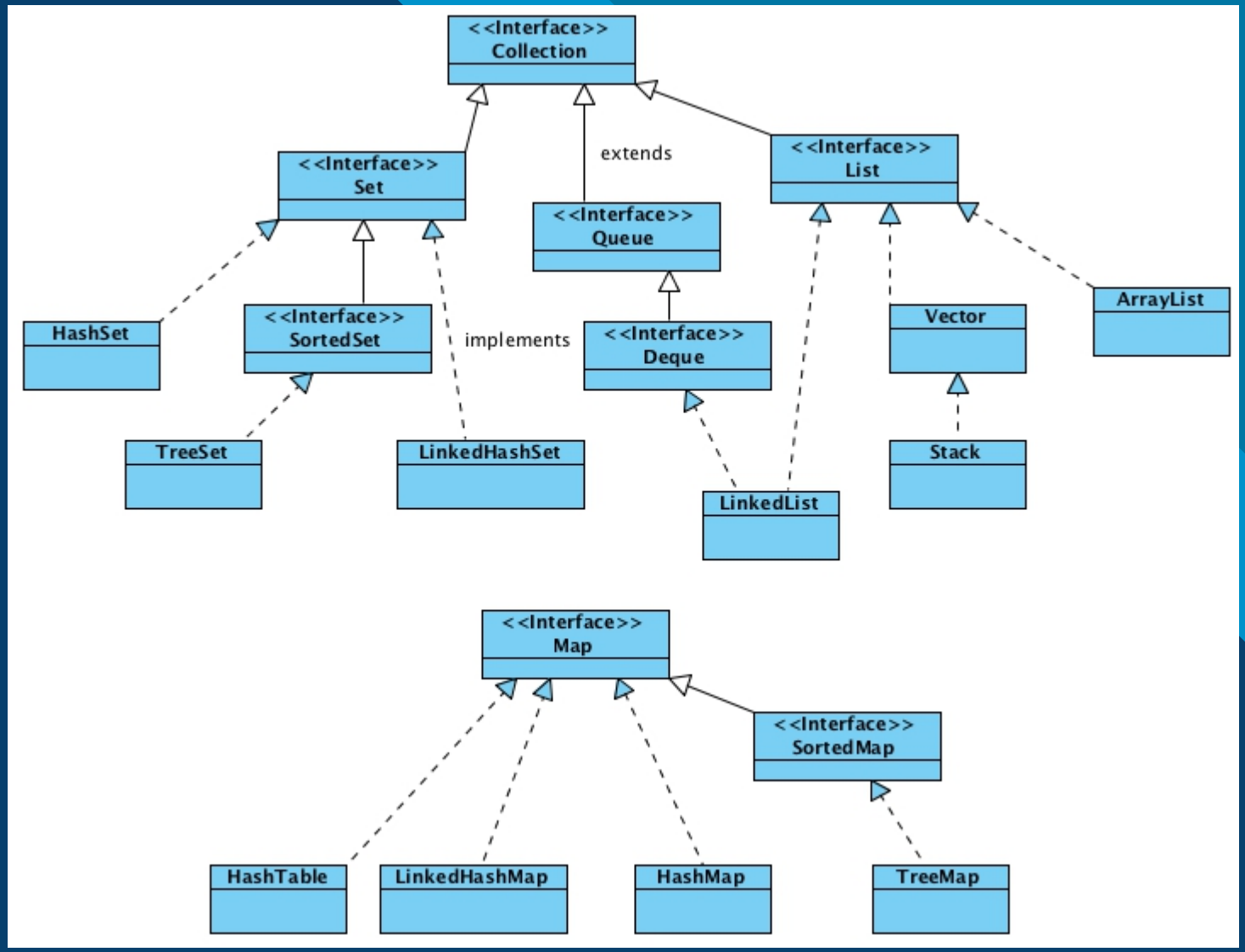
- Podemos indicar que el tipo parametrizado sea uno en particular (o sus derivados).

```
public class NumericBox<T extends Number> ...
public class StrangeBox <T extends A & B>
```

- Podemos indicar más de un tipo
  - Solo uno de ellos puede ser una clase.
  - El resto deben ser interfaces
  - La clase a extender debe ser la primera de la lista.

Colecciones





Comparable y Comparator

Muchas operaciones entre objetos nos obligan a compararlos: buscar, ordenar, …Los tipos primitivos y algunas clases ya implementan su orden (natural, lexicográfico). Para nuestras clases (modelo) tenemos que especificar el orden con el que las vamos atratar.

Comparable: Se trata de un interfaz sencillo. Recibe un objeto del mismo tipo. Devuelve 0 si son iguales, un valor negativo si es menor, y uno positivo si es mayor. **Nos sirve para indicar el orden principal de una clase**.

```
public interface Compararable<T> {
        // 0 si es igual, 1 si es mayor, -1 si es menor.
        public int compareTo(T o);
}
```

Comparator: Se trata de un interfaz sencillo. Recibe dos argumentos. Devuelve 0 si son iguales, un valor negativo si es menor, y uno positivo si es mayor. **Nos sirve para indicar un orden puntual, diferente al orden principal de una clase**.

```
public interface Comparator<T> {
        public int compare(T o1, T o2);
}
```

API Stream

Trabajando funcionalmente y de manera fluída

API Stream

- Antes de nada debemos introducir la **programación funcional en Java**
- **En la programación Estructurada nos centramos en en el QUÉ y el CÓMO, en la funcional nos centramos en el QUÉ**
- **Expresiones Lambda:** Las expresiones lambda son funciones anónimas, es decir, funciones que no necesitan una clase. Su sintáxis básica se detalla a continuación: (parámetros) -> { cuerpo-lambda}
- 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.
- 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 clausula return en el caso de que deban devolver valores.
  - Cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la clausula return en el caso de que la función deba devolver un valor .

API Stream

- **Predicate<T>**: Comprueba si se cumple o no una condición. Se utiliza mucho junto a expresiones lambada a la hora de filtrar:

```
.filter((p) -> p.getEdad() >= 35)
```

- **Consumer<T>**: Sirve para consumir objetos. Uno de los ejemplos más claros es imprimir.

```
.forEach(System.out::println)
```

  - Adicionalmente, tiene el método andThen, que permite componer consumidores, para encadenar una secuencia de operaciones.

- **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())
```

  - Adicionalmente, tiene otros métodos:
    - andThen, que permite componer funciones.
    - compose, que compone dos funciones, a la inversa de la anterior.
    - identity, una función que siempre devuelve el argumento que recibe

API Stream

- **Supplier<T>**: Sirve para devolver un valor.
  - Tiene algunos interfaces especializados para tipos básicos:
    - IntSupplier
    - LongSupplier
    - DoubleSupplier
    - BooleanSupplier

API Stream - Métodos de Búsqueda

- Son un tipo de operaciones terminales sobre un stream, que nos permiten:

- Identificar si hay elementos que cumplen una determinada condición
- Obtener (si el stream contiene alguno) determinados elementos en particular.
- Algunos de los métodos de búsqueda son:

- allMatch(Predicate<T>): verifica si todos los elementos de un stream satisfacen un predicado.
- anyMatch(Predicate<T>): verifica si algún elemento de un stream satisface un predicado.
- noneMatch(Predicate<T>): opuesto de allMatch(…)
- findAny(): devuelve en un Optional<T> un elemento (cualquiera) del stream. Recomendado en streams paralelos.
- findFirst() devuelve en un Optional<T> el primer elemento del stream. NO RECOMENDADO en streams paralelos.

API Stream - Métodos de datos, cálculo y ordenación

- **Métodos de datos y cálculo:** Los streams nos ofrecen varios tipos de métodos terminales para realizar operaciones y cálculos con los datos. Durante el curso trabajaremos con tres tipos:
  - Reducción y resumen (en esta lección)
  - Agrupamiento
  - Particionamiento
- **Métodos de reducción:** Son métodos que reducen el stream hasta dejarlo en un solo valor.
  - reduce(BinaryOperator<T>):Optional<T> realiza la reducción del Stream usando una función asociativa. Devuevle un Optional
  - reduce(T, BinaryOperator<T>):T realiza la reducción usando un valor inicial y una función asociativa. Se devuelve un valor como resultado.
- **Métodos de resumen:** Son métodos que resumen todos los elementos de un stream en uno solo:
  - count: devuelve el número de elementos del stream.
  - min(...), max(...): devuelven el máximo o mínimo (se puede utilizar un Comparator para modificar el orden natural).
- **Métodos de ordenación:** Son operaciones intermedias, que devuelven un stream con sus elementos ordenados.
  - sorted() el stream se ordena según el orden natural.
  - sorted(Comparator<T>) el stream se ordena según el orden indicado por la instancia de Comparator.

API Stream - Map y FlapMap

- **Uso de map**: map es una de las operaciones intermedias más usadas, ya que permite la transformación de un objeto en otro, a través de un Function<T, R>. Se invoca sobre un Stream<T> y retorna un Stream<R>. Además, es muy habitual realizar transformaciones sucesivas.

```
lista.stream().map(Persona::getNombre).map(String::toUpperCase).forEach(System.out::println);
```

- **Uso de flatMap:** Los streams sobre colecciones de un nivel permiten transformaciones a través de map pero, ¿qué sucede si tenemos una colección de dos niveles (o una dentro de objetos de otro tipo)?:

```
public class Persona {

        private String nombre;

        private List<Viaje> viajes = new ArrayList<>();

}
```

- Para poder trabajar con la colección interna, necesitamos un método que nos unifique un Stream<Stream<T>> en un solo Stream<T>. Ese es el cometido de flatMap.

```
lista.stream().map((Persona p) -> p.getViajes()).flatMap(viajes -> viajes.stream()).map((Viaje v) ->
v.getPais()).forEach(System.out::println);
```

API Stream - Collectors

- **Collectors:** Los collectors nos van a permitir, en una operación terminal, construir una collección mutable, el resultado de las operaciones sobre un stream.

- **Colectores "básicos":** Nos permiten operaciones que recolectan todos los valores en uno solo. Se solapan con alguans operacinoes finales ya estudiadas, pero están presentes porque se pueden combinar con otros colectores más potentes.
  - counting: cuenta el número de elementos.
  - minBy(…), maxBy(…): obtiene el mínimo o máximo según un comparador.
  - summingInt, summingLong, summingDouble: la suma de los elementos (según el tipo).
  - averagingInt, averagingLong, averagingDouble: la media (según el tipo).
  - summarizingInt, summarizingLong, summarizingDouble: los valores anteriores, agrupados en un objeto (según el tipo).
  - joinning: unión de los elementos en una cadena.
- **Colectores "grouping by":** Hacen una función similar a la cláusula GROUP BY de SQL, permitiendo agrupar los elementos de un stream por uno o varios valores. Retornan un Map.

API Stream - Filters

- **filter** es una operación intermedia, que nos permite eliminar del stream aquellos elementos que no cumplen con una determinada condición, marcada por un Predicate<T>.

```
personas.stream().filter(p -> p.getEdad() >= 18 && p.getEdad() <= 65).forEach(persona -> System.out.printf("%s (%d años)%n", persona.getNombre(), persona.getEdad()));
```

- Es muy combinable con algunos métodos como findAny o findFirst:

```
Persona p1 = personas.stream().filter(p -> p.getNombre().equalsIgnoreCase("Pepe")).findAny().orElse(new Persona());
```

API Stream - Referencia a métodos

- Las referencias a métodos son una forma de hacer nuestro código aun más conciso:
  - Clase::metodoEstatico: referencia a un método estático.
  - objeto::metodoInstancia: referencia a un método de instancia de un objeto concreto.
  - Tipo::nombreMetodo: referencia a un método de instancia de un objeto arbitrario de un tipo en particular.
  - Clase::new: referencia a un constructor.

API Stream - Similitudes a SQL

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()

API Stream - Resumen

- **Stream** Representa un flujo de elementos y podemos aplicar métodos tales como
  - filter para filtrar los elementos. Éste método recibe un predicado como argumento.
  - map para devolver solo el atributo indicado. Es como el select de sql.
  - sorted para ordenar nuestros elementos. Recibe un Comparator.
  - min, max, count que permiten obtener el minimo, máximo y conteo de elementos respectivamente.
  - collect aquí es donde definiremos nuestros funciones de agregado, principalmente agrupados, particiones, etc.
  - Comparator nos proporciona métodos útiles para realizar ordenamientos. Lo usaremos en conjunto con el método sorted
  - Collectors Nos proporciona métodos útiles para agrupar, sumar, promediar, obtener estadísticas. Lo usaremos en conjunto con el método collect

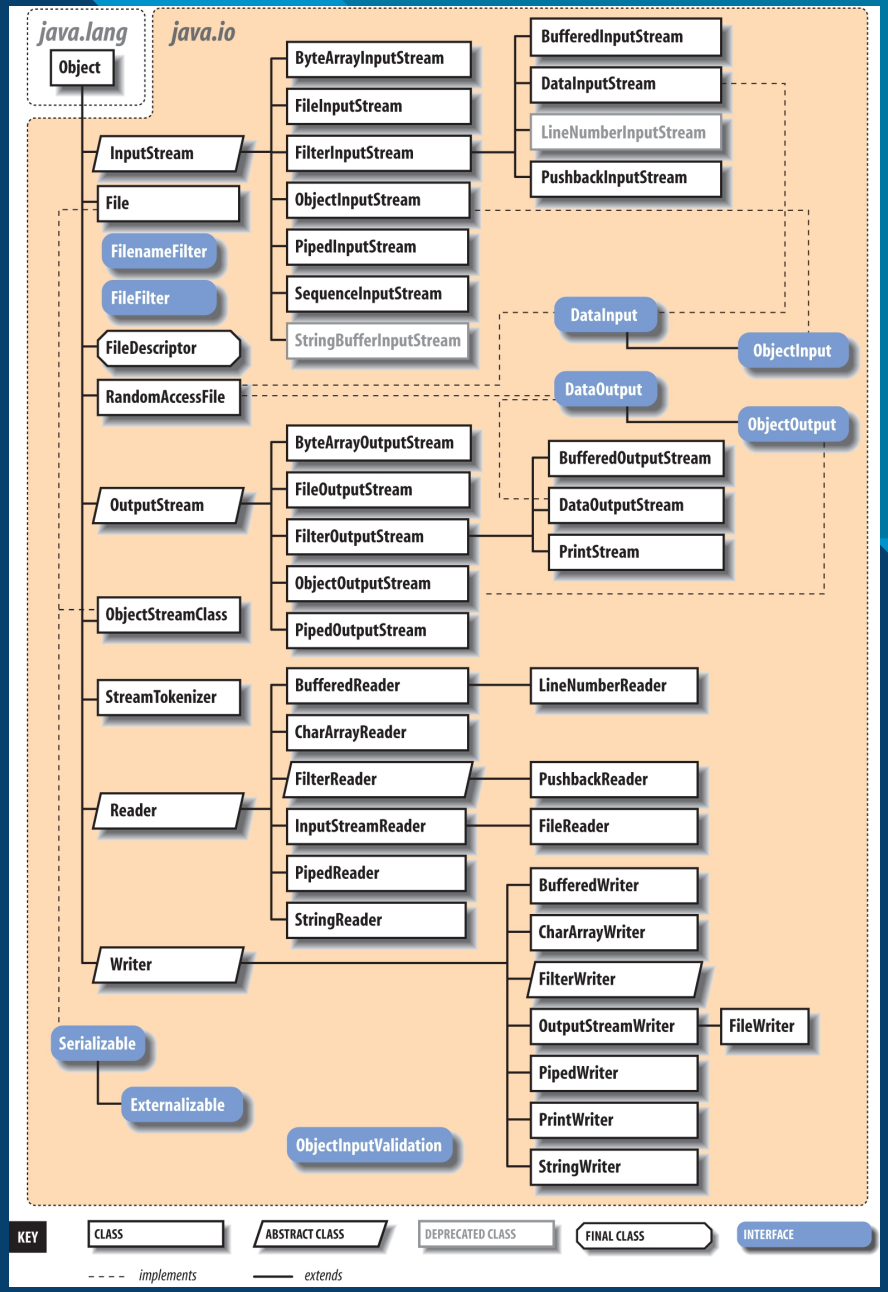
Ficheros y Directorios

Almacenamiento de la información en disco

Ficheros: Tipos y Flujos

- **Ficheros de texto** cuando el contenido del fichero contenga exclusivamente caracteres de texto (podemos leerlo con un simple editor de texto)
- **Ficheros binarios** cuando no estén compuestos exclusivamente de texto. Pueden contener imágenes, videos, ficheros, . . . aunque también podemos considerar un fichero binario a un fichero de Microsoft Word en el que sólo hayamos escrito algún texto puesto que, al almacenarse el fichero, el procesador de texto incluye alguna información binaria
- **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.
  - Flujos de entrada: sirven para introducir datos en la aplicación.
  - Flujos de salida: sirven para sacar datos de la aplicación.
  - Flujos de bytes: manejan datos en crudo.
  - Flujos de caracteres: manejan caracteres o cadenas.

Ficheros: Flujos



Ficheros: Flujos de Salida

- **Flujos de salida de bytes**. Algunas de las clases que podemos usar son:
  - OutputStream: clase abstracta, padre de la mayoría de los flujos de bytes.
  - FileOutputStream: flujo que permite escribir en un fichero, byte a byte.
  - BufferedOutputStream: flujo que permite escribir grupos (buffers) de bytes.
  - ByteArrayOutputStream: flujo que permite escribir en memoria, obteniendo lo escrito en un array de bytes.
- **Flujos hacia otros flujos**. Solo FileOutputStream tiene un constructor que acepta una ruta (entre otras opciones). El resto reciben en sus constructores un tipo de OutputStream. ¿Por qué? Porque podemos construir flujos que escriben en flujos (encadenados).
- **Flujos de salida de caracteres**
  - Writer: clase abstracta, padre de la mayoría de los flujos de caracteres.
  - FileWriter: flujo que permite escribir en un fichero, caracter a caracter.
  - BufferedWriter: flujo que permite escribir líneas de texto.
  - StringWriter: flujo que permite escribir en memoria, obteniendo lo escrito en un String
  - OutputStreamWriter: flujo que permite transformar un OutputStream en un Writer.
  - PrintWriter: flujo que permite escribir tipos básicos Java.

Ficheros: Flujos de Entrada

- **Flujos de entrada de bytes.** Algunas de las clases que podemos usar son:
  - InputStream: clase abstracta, padre de la mayoría de los flujos de bytes.
  - FileInputStream: flujo que permite leer de un fichero, byte a byte.
  - BufferedInputStream: flujo que permite leer grupos (buffers) de bytes.
  - ByteArrayInputStream: flujo que permite leer de memoria (de un array de bytes).
- **Flujos de entrada de caracteres**. Algunas de las clases que podemos usar son:
  - Reader: clase abstracta, padre de la mayoría de los flujos de caracteres.
  - FileReader: flujo que permite leer de un fichero, caracter a caracter.
  - BufferedReader: flujo que permite leer líneas de texto.
  - StringReader: flujo que permite leer desde la memoria.
  - InputStreamReader: flujo que permite transformar un InputStream en un Reader.

Ficheros: 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.
- Por otra parte, estos archivos no son stream y se numeran por un índice que empieza por cero. Este índice, se llama puntero o cursor de lectura/escritura e indica la posición a partir de la cual se empezará a leer o escribir en el archivo. Es importante indicar que la información almacenada en este tipo de archivos, se guarda en forma de bytes. Esto quiere decir que este la clase RandomAccessFile trata del archivo como un array de bytes.
- Para poder determinar cuanto ocupa cada registro en un archivo de acceso aleatorio, debemos tener en cuenta los bytes de cada uno de los datos. De esta forma, dependiendo del tipo de dato que queramos almacenar, cada registro ocupará un tamaño u otro.
- Hay que tener en cuenta que los datos tipo String (tipo texto), en java son considerados como objetos. Por este motivo, un dato tipo String, se considera un array de caracteres tipo char. Esto quiere decir que la información de tipo String, ocupará dos bytes por cada carácter tipo char. A esta información, se le deben sumar los bytes ocupados por los espacios en blanco y los saltos de línea.

Ficheros: Acceso Aleatorio

Tipo de Dato	Tamaño en Bytes
Char	2 bytes
Byte	1 byte
Short	2 bytes
Int	4 bytes
Long	8 bytes
Float	4 bytes
Double	8 bytes
Boolean	1 byte
Espacio en blanco (un char)	1 byte
Salto de línea (enter)	1 byte
String	2 bytes por cada char

Ficheros: 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

Java Nio.2

Más potencia y más sencillez

Java NIO.2

- En la versión 1.4 de Java se añadió un nuevo sistema de entrada/salida llamado NIO para suplir algunas de sus deficiencias que posteriormente en Java 7 se mejoró aún más con NIO.2. Entre las mejoras se incluyen permitir navegación de directorios sencillo, soporte para reconocer enlaces simbólicos, leer atributos de ficheros como permisos e información como última fecha de modificación, soporte de entrada/salida asíncrona y soporte para operaciones básicas sobre ficheros como copiar y mover ficheros.
- Las clases principales de esta nueva API para el manejo de rutas, ficheros y operaciones de entrada/salida son las siguientes:
  - **Path**: es una abstracción sobre una ruta de un sistema de ficheros. No tiene porque existir en el sistema de ficheros pero si si cuando se hacen algunas operaciones como la lectura del fichero que representa. Puede usarse como reemplazo completo de java.io.File pero si fuera necesario 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 como punto de entrada para obtener referencias a sistemas de archivos

Java NIO.2: Files

- **COMPROBACIONES**
  - Existencia (exists)
  - Acceso (isReadable, isWritable,isExecutable)
  - Son el mismo fichero (isSameFile)
- **COPIAR, BORRAR Y MOVER**
  - Borrar (delete, deleteIfExists)
  - Copiar (copy)
  - Mover (move)
- **CREAR, ESCRIBIR Y LEER**
  - Para crear ficheros
    - Ficheros regulares (createFile)
    - Temporales (createTempFile)
  - Buffered
    - Leer (newBufferedReader)
    - Escribir (newBufferedWriter)

Java NIO.2: Files

- **CREAR, ESCRIBIR Y LEER**
  - Unbuffered
    - Leer (newInputStream)
    - Escribir (newOutputStream)
- **TRABAJO CON DIRECTORIOS**
  - Listar
    - Raíz del sistema (FileSystem.getRootDirectories)
    - Contenido de directorios (newDirectoryStream)
  - Crear
    - Crear un directorio (createDirectory, createDirectories)
    - Temporal (createTempDirectory)

Java NIO.2: Path

- Con la clase **Path** se pueden hacer operaciones sobre rutas como obtener la ruta absoluta de un Path relativo o el Path relativo de una ruta absoluta, de cuanto elementos se compone la ruta, obtener el Path padre o una parte de una ruta.
- Otros métodos interesantes son relativize(), normalize(), toAbsolutePath(), resolve(), startsWith() y endsWith().

Expresiones Regulares

Encontrar patrones en nuestros textos

Expresiones Regulares

- Una expresión regular define un patrón de búsqueda para cadenas de caracteres.
- La podemos utilizar para comprobar si una cadena contiene o coincide con el patrón. El contenido de la cadena de caracteres puede coincidir con el patrón 0, 1 o más veces.
- Algunos ejemplos de uso de expresiones regulares pueden ser:
  - para comprobar que la fecha leída cumple el patrón dd/mm/aaaa
  - para comprobar que un NIF está formado por 8 cifras, un guión y una letra
  - para comprobar que una dirección de correo electrónico es una dirección válida.
  - para comprobar que una contraseña cumple unas determinadas condiciones.
  - Para comprobar que una URL es válida.
  - Para comprobar cuántas veces se repite dentro de la cadena una secuencia de caracteres determinada.
- El patrón se busca en el String de izquierda a derecha. Cuando se determina que un carácter cumple con el patrón este carácter ya no vuelve a intervenir en la comprobación.

Expresiones Regulares

- Cuando queremos comparar una expresión regular con una cadena de texto, podemos querer dos posibles cosas:
  - Contiene una parte de la cadena de texto que cumpla esa expresión regular. Por ejemplo, podemos querer buscar una fecha en formato "dd/mm/yyyy" en la cadena "Hoy es 12/02/2017" y efectivamente, la cadena contiene una fecha en ese formato.
  - La cadena Coincide totalmente con la expresión regular. En el ejemplo anterior, la cadena "Hoy es 12/02/2017" sí contiene una fecha en formato "dd/mm/yyyy" dentro, pero NO coincide con ese formato, puesto que le sobra el trozo de texto "Hoy es " para tener la coincidencia exacta con una fecha.
- **Símbolos comunes en expresiones regulares**
  - .: Un punto indica cualquier carácter
  - ^expresión: El símbolo ^ indica el principio del String. En este caso el String debe contener la expresión al principio.
  - expresión$: El símbolo $ indica el final del String. En este caso el String debe contener la expresión al final.
  - [abc]: Los corchetes representan una definición de conjunto. En este ejemplo el String debe contener las letras a ó b ó c.
  - [abc][12]: El String debe contener las letras a ó b ó c seguidas de 1 ó 2
  - [^abc]: El símbolo ^ dentro de los corchetes indica negación. En este caso el String debe contener cualquier carácter excepto a ó b ó c.
  - [a-z1-9]: Rango. Indica las letras minúsculas desde la a hasta la z (ambas incluidas) y los dígitos desde el 1 hasta el 9 (ambos incluidos)
  - A|B: El carácter | es un OR. A ó B
  - AB: Concatenación. A seguida de B

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,}
  - +: ndica 1 ó más veces. Equivale a {1,}
  - ?: Indica 0 ó 1 veces. Equivale a {0,1}

Expresiones Regulares en JAVA

- **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**

"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

Recursos

- Twitter: https://twitter.com/joseluisgonsan
- GitHub: https://github.com/joseluisgs
- Web: https://joseluisgs.github.io
- Discord: https://discord.gg/eFMKxfPY
- Aula Virtual: https://aulavirtual33.educa.madrid.org/ies.luisvives.leganes/course/view.php?id=247

Gracias

José Luis González Sánchez