Java NIO.2

desde java 1.4 actualizado en java 1.7

https://docs.oracle.com/javase/8/docs/technotes/guides/io/index.html
https://docs.oracle.com/javase/tutorial/essential/io/fileio.html

https://www.baeldung.com/java-nio-vs-nio-2
https://www.developer.com/java/an-introduction-to-java-nio-and-nio-
https://docs.oracle.com/en/java/javase/19/core/java-nio.html 2/

Path

- El path es la ruta/camino de localización de ficheros o directorios.
- Ruta absoluta: empieza en la raíz del sistema de ficheros (SF)
- Ruta relativa: empieza en un punto diferente a la raíz.
- Path permite gestionar rutas de acceso en el SF. Permite manejar diferentes SF (ext3, ntfs, fat,…) en diferentes SO
- Tenemos la interfaz Path y la clase Paths

Clase Paths

- Dispone de métodos estáticos para crear paths
- `public static Path get(String first, String... more)`
  Crea un path concatenando los diferentes Strings recibidos como parámetros.
- El path se consigue llamando al método getPath del SF por defecto.

Clase Path: Ejemplo

```
Path p1=Paths.get("f.txt");
Path p1=Paths.get("f.txt");
Path p2=Paths.get("dir","f1.txt");
Path p3=FileSystems.getDefault().getPath("dir","dir2","f2.txt");

System.out.println(p1);
System.out.println(p2.toAbsolutePath().toString());
System.out.println(p3.getFileName());
for (int i=0; i<p3.getNameCount(); i++)
  System.out.print(p3.getName(i)+" ->");
System.out.println();
System.out.println(p3.subpath(0, 1));
System.out.println(p3.getNameCount());
System.out.println(p3.getParent());
System.out.println(p3.getRoot());
Path p4=Paths.get("dir","..","..", "dir", "f1.txt");
Path normalizado= p4.normalize();
System.out.println("No normalizado:"+p4);
System.out.println("Normalizado:"+ normalizado);
```

SALIDA:

f.txt

C:\Users\josema\workspace\Mars\PrJavanio\dir\f1.txt

f2.txt

dir ->dir2 ->f2.txt ->

dir

3

dir\dir2

null

No normalizado:dir\..\..\dir\f1.txt

Normalizado:..\dir\f1.txt

Clase Files:

- **Tiene numerosos métodos estáticos para comprobar características de ficheros/ directorios, moverlos, copiarlos, crearlos, etc**. :
- *static long copy(InputStream in, Path target, CopyOption... options)*
- *static Path createDirectories(Path dir, FileAttribute<?>... attrs)*
- *static Path createFile(Path path, FileAttribute<?>... attrs)*
- *static void delete(Path path)*
- *static FileTime getLastModifiedTime(Path path, LinkOption... options)*
- *static UserPrincipal getOwner(Path path, LinkOption... options)*
- *static boolean isExecutable(Path path)*
- *static Path move(Path source, Path target, CopyOption... options)*
- *static DirectoryStream<Path> newDirectoryStream(Path dir, String glob)* Para ver el contenido de un directorio
- *static List<String> readAllLines(Path path, Charset cs)* para leer un fichero (de texto)

Clase Files: Ejemplo

```
Path p1=Paths.get("fich.txt");
if (Files.notExists(p1))
  Files.createFile(p1);
System.out.println("El fichero existe:"+Files.exists(p1));
Path p2=Paths.get("copia.txt");
Files.copy(p1, p2,  StandardCopyOption.REPLACE_EXISTING);
System.out.println("Es ejecutable:"+Files.isExecutable(p2));
System.out.println("Propietario:"+Files.getOwner(p1));
```

Clase Files: Ejemplo

```
BufferedWriter bw= Files.newBufferedWriter(p1,
StandardOpenOption.APPEND);
bw.write("Datos añadidos");
bw.close();

bw=Files.newBufferedWriter(p2, Charset.forName("UTF-8"));
bw.write("datos" );
bw.write("xxx 2");
bw.close();
```

Clase Files: Ejemplo

```
System.out.println("Contenido fichero");
Path src=Paths.get(".", "fich.txt");
System.out.println("existe:"+Files.exists(src));
List<String> fichs= Files.readAllLines(src);
for (String s: fichs){
  System.out.println(s);
}
src=Paths.get("src" );
System.out.println("Contenido directorio src");
DirectoryStream<Path> dirStr=Files.newDirectoryStream(src);
for (Path f: dirStr){
   System.out.println(f.getFileName());
}
```

Lecturas de ficheros- Stream

Es posible procesar ficheros de texto utilizando la clase Stream o utilizando listas. El procesamiento suele hacerse con expresiones lambda tratando cada línea por separado.

- public static List<String> **readAllLines**(Path path, Charset cs) throws IOException : Lee todas las líneas de un fichero y las devuelve en una lista de Strings.
- public static Stream<String> **lines**(Path path, Charset cs) throws IOException y devuelve un Stream. Se puede leer las líneas del fichero, según se van necesitando (lazy) y utilizar los métodos de Stream para su procesamiento posterior.

También se pueden utilizer métodos para ficheros Binarios como:

public byte[] **readAllBytes**() throws IOException

Ejem 1a JavaNIO

```
public class StreamFichConLambda {

  static void listarFichTxt() throws IOException {
  Path path = Paths.get("datos/fichero.txt");
  Stream<String> stream = Files.lines(path);

 // stream.forEach(System.out::println);
  //también se puede poner:
  stream.forEach((s)->System.out.println(s));

  stream.close();
  }
```

Ejem 1b JavaNIO

```
 static void listarOrdenadoFichTxt() throws IOException {
  Path path = Paths.get("datos/fichero.txt");
  Stream<String> stream = Files.lines(path);

  stream.sorted().forEach(System.out::println);

  stream.close();
  }

  public static void main (String arg[]) throws IOException {
  listarFichTxt();
  listarOrdenadoFichTxt();
  }
}
```

Ejem 2a JavaNIO

```
class Alumno {
  String nombre;
  int []notas;
  Alumno(String n, int []notas){
    nombre=n;
    this.notas=notas;
  }
  public String toString(){
    int media=0;
    for (int i=0; i<notas.length; i++)
      media+=notas[i];
    media=media/notas.length;
    return "nombre:"+nombre+" media:"+media;
  }
}
```

Ejem 2b JavaNIO

En el tratamiento utilizamos expresiones Lambda.
No es posible utilizar variables externas dentro de la expresión lambda a menos que estas sean final.

```
public class LecturaFicheroProcesado {
//Leemos un fichero que tiene nombre;nota1;nota2;nota3 en cada línea
public static void main(String[] args) throws IOException {
Path fich=Paths.get("fnotas.txt");
List<Alumno> lisAl=null;
if (!Files.exists(fich))
  System.out.println("Fich no existe");
else{
  try (Stream<String> flujo= Files.lines(fich,  Charset.forName("Cp1252“))){
    lisAl=flujo.peek(dato->System.out.println("línea:"+dato))
   //peek procesa la línea pero  la pasa igual que la recibe al map
      .map(linea->linea.split(";"))
      .map(plin->{  //toma la línea separada por el split
        String nombre=plin[0];
        int  notas[]={Integer.parseInt(plin[1]),Integer.parseInt(plin[2])
Integer.parseInt(plin[3])};
        System.out.println("Las notas son "+notas[0]+","+notas[1] +","+ notas[2]);
        System.out.println("La suma es "+(notas[0]+notas[1]+notas[2]));
        Alumno a=new Alumno(nombre,notas);
        return a;
      }
     ).collect(Collectors.toList());
   }catch (Exception e){
   System.out.println("Error");
   e.printStackTrace();
  }
  for (Alumno a:lisAl)
    System.out.println(a);
 }//else
}//main
}//class
```

https://es.stackoverflow.com/questions/288782/diferencia-entre-cp-1252-y-utf-8

Último fichero modificado

```
public static Path findUsingNIOApi(String sdir) throws IOException {
    Path dir = Paths.get(sdir);
    if (Files.isDirectory(dir)) {
        Optional<Path> opPath = Files.list(dir)
          .filter(p -> !Files.isDirectory(p))
          .sorted((p1, p2)-> Long.valueOf(p2.toFile().lastModified())
            .compareTo(p1.toFile().lastModified()))
          .findFirst();

        if (opPath.isPresent()){
            return opPath.get();
        }
    }

    return null;
}
```

https://www.baeldung.com/java-last-modified-file

Files con streams-list

- Files tiene varios métodos que devuelven streams
- static Stream<Path> **list**(Path dir) devuelve todas las rutas de un directorio dado. Ejem para listar el contenido del directorio ejemplo (sin incluir los ocultos en unix)

```
try (Stream<Path> stream =
Files.list(Paths.get(System.getProperty("user.home"),"ejemplo")))
{
stream.map(String::valueOf) .filter(path -> !path.startsWith("."))
.sorted().forEach(System.out::println);
}
```

Files con streams- find

- static Stream<Path> **find**(Path start, int maxDepth, BiPredicate<Path,BasicFileAttributes> matcher, FileVisitOption... options) : Devuelve todas las rutas a partir de un directorio dado que cumplen una condición. Se le puede indicar una profundidad máxima. Ejem para listar los fichs txt hasta una profundidad de 5 directorios desde el dir ejemplo

```
Path start = Paths.get(System.getProperty("user.home"),"ejemplo");
int maxDepth = 5;
try (Stream<Path> stream = Files.find(start, maxDepth, (path, attr) ->
String.valueOf(path).endsWith(".txt"))) {
stream.sorted().map(String::valueOf).forEach(System.out::println);
}
```

- https://www.baeldung.com/java-list-directory-files

Files con streams -walk

- static Stream<Path> **walk**(Path start, int maxDepth, FileVisitOption... options) : Devuelve todas las rutas a partir de un directorio dado. Se le puede indicar una profundidad máxima. Ejem. lista todos los ficheros de un directorio

```
Path inicio = Paths.get(System.getProperty("user.home"), "ejemplo");
int maxDepth = 5;
List<Path> result;
try (Stream<Path> walk = Files.walk(inicio)) {
    result = walk.filter(Files::isRegularFile).collect(Collectors.toList());
}
for (Path p:result)
   System.out.println(p);
```

Files con streams -lines

public static Stream<String> **lines**(Path path, Charset cs) throws IOException : Devuelve las líneas de un fichero de texto en un Stream.

```
Path inicio = Paths.get(System.getProperty("user.home"),"ejemplo/ej1.txt");
Optional<List<String[]>> lista;
try (Stream<String> stream = Files.lines(inicio, Charset.forName("Cp1252"))) {
  lista= Optional.of(stream.map(s -> s.split("[; ]")).collect(Collectors.toList()));
}
if (!lista.isPresent())
        for (String[]l : lista.get()) {
          for (String s: l)
                  System.out.print(s+"#");
          System.out.println();
        }
else    System.out.println("fich vacío");
```

Asynchronous File Channel

```
Path path = Paths.get( URI.create(
this.getClass().getResource("/file.txt").toString()));

AsynchronousFileChannel fileChannel =
  AsynchronousFileChannel.open( path, StandardOpenOption.READ);
ByteBuffer buffer = ByteBuffer.allocate(1024);
Future<Integer> operation = fileChannel.read(buffer, 0);
// Aquí puedes ejecutar otras cosas porque el read se ejecuta
en background
operation.get();
String fileContent = new String(buffer.array()).trim();
buffer.clear();
```

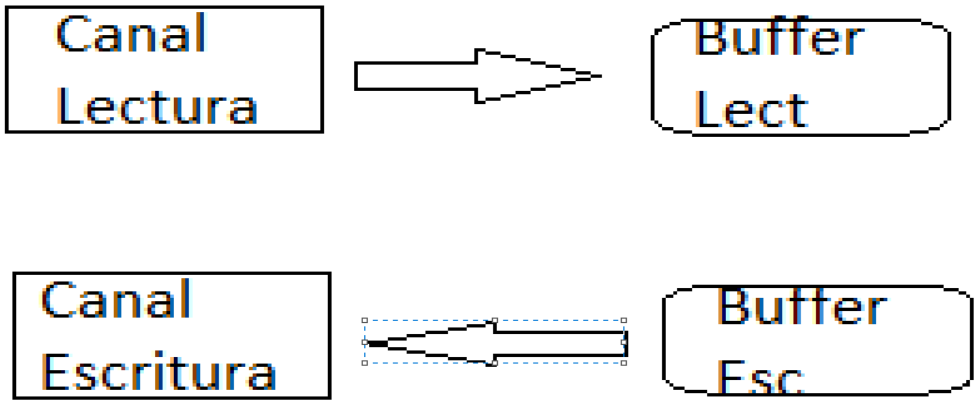
Lenguaje Java

Java NIO

http://tutorials.jenkov.com/java-nio/overview.html
https://howtodoinjava.com/java-nio-tutorials/
https://www.baeldung.com/java-filechannel

Java NIO

- Originalmente era un sistema no bloqueante de gestión de entrada/salida aunque ahora hay formas bloqueantes.
- Se utilizan **canales** para realizar las lecturas y escrituras desde y hacia **bufferes**.
- Se utilizan **selectores**, objetos que pueden vigilar varios canales para ver si ocurren eventos en los mismos (llegada de datos, apertura del canal,..).
- Es posible leer de un canal en varios bufferes (scattering reads) o escribir desde diferentes bufferes en un solo canal (gatherning writes)



Canales

Los canales se utilizan para leer hacia bufferes y escribir desde bufferes

- **FileChannel**: Lee y escribe datos en ficheros.
- **DatagramChannel**: Lee y escribe datos en sockets UDP
- **SocketChannel**: Lee y escribe datos en sockets TCP
- **ServerSocketChannel:** Permite esperar conexiones TCP. Para cada conexión se crea un SocketChannel
- **AsynchronousSocketChannel.** Socket channel asíncrono, sin bloqueo

Bufferes

Los bufferes sirven como almacenamiento intermedio entre los canales y el programa.

El **proceso básico de lectura** es :

1. Escribir datos del canal al buffer.
2. Usar buffer.flip() para cambiar el tipo de uso del buffer de escritura a lectura
3. Pasar los datos del buffer a una variable del programa por ejemplo con *var= buffer.get()*
4. Vaciar completamente el buffer con buffer.clear(). También se puede usar buffer.compact() para eliminar del buffer los datos leídos del mismo en el punto 3.

El **proceso básico de escritura** es :

1. Escribir en el buffer los datos con el método put(datos).
2. Usar buffer.flip() para cambiar el tipo de uso del buffer de escritura a lectura
3. Pasar los datos del buffer a un canal : canal.write(buffer)
4. Vaciar completamente el buffer con buffer.clear(). También se puede usar buffer.compact() para eliminar del buffer los datos leídos del mismo en el punto 3.

Bufferes

Bufferes básicos

- ByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

Buffer

Reserva de memoria en un buffer:

public static TipoBuffer allocate(int capacidad)

Propiedades de un buffer:

Capacidad: cuántos datos admite. Si se llena hay que vaciarlo

Posición: La posición inicial del buffer es 0. Cuando escribes X datos en el mismo la posición cambia a X+posición anterior. La máxima posición es la capacidad -1. Cuando lees datos de un buffer la posición aumenta en el número de datos leídos.

Limite. En modo escritura el límite es igual a la capacidad del buffer. En modo lectura del buffer el límite es la cantidad de datos que puedes leer del buffer, es decir los datos que se escribieron en el buffer**.**

Ejemplo de lectura de fichero I

```
public class CanalBasico1 {
public static void main(String[] args) throws IOException{
    RandomAccessFile aFile = new RandomAccessFile("datos/prueba.txt", "rw");
    FileChannel inChannel = aFile.getChannel();
    ByteBuffer buf = ByteBuffer.allocate(18);

    int bytesRead = inChannel.read(buf); //buffer en modo escritura (de
fichero a buffer)
    while (bytesRead != -1) {
      System.out.println("Tamaño buffer " + bytesRead+ " bytes");
      buf.flip(); //cambio el buffer a lectura. (de buffer a pantalla con el
get posterior
      while(buf.hasRemaining()){ //mientras el buffer no esté vacío
          System.out.print((char) buf.get());
      }
      buf.clear();//vacio el buffer para poder rellenarlo con la nueva
lectura
          System.out.println("\nLectura en buffer");
      bytesRead = inChannel.read(buf);
    }
    aFile.close();
}
}} // fin del bucle for
 System.out.println ("Fin del prog.");
}//main
}
```

Salida:

```
Tamaño buffer 18 bytes
uno
dos tres
cua
Lectura en buffer
Tamaño buffer 16 bytes
tro
5 6 7
ocho
Lectura en buffer
```

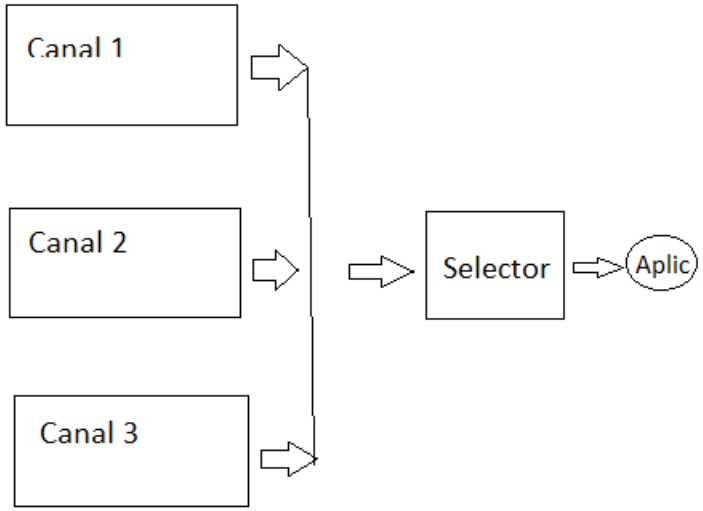
Contenido Fichero:

uno
dos tres
cuatro
5 6 7
ocho

Lectura en múltiples canales

Los canales son realmente útiles cuando se utilizan, no para procesar datos de un fichero, sino para la comunicación entre diferentes procesos que se comunican mediante canales (sockets).

En este caso se puede realizar la espera de llegada de datos en varios canales simultáneamente sin necesidad de polling en cada uno de ellos utilizando **selectores**.



Ejemplo de selector

```
public class Selectores1 {
public static void main(String[] args) throws IOException {
Selector selector = Selector.open();
SocketChannel canal1=……;          SocketChannel canal2=……;
//si el canal se usa en un selector no puede estar bloqueado.
//Por tanto, no se pueden usar selectores con ficheros.
canal1.configureBlocking(false);    canal2.configureBlocking(false);
//Añado canales al selector
SelectionKey key1 = canal1.register(selector, SelectionKey.OP_READ);
SelectionKey key2 = canal2.register(selector, SelectionKey.OP_READ);
//Vemos si hay datos para leer en algún canal (devuelve en cuantos) durante 2 segs.
int numsCanalesPreparados=selector.select(2000);
Set<SelectionKey> selectedKeys;
if (numsCanalesPreparados != 0) {
  selectedKeys= selector.selectedKeys();
//Buscamos el canal con datos disponibles como si hubieramos seleccionado distintas
//operaciones, no solo lectura.
  Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
  while(keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if(key.isAcceptable() key.isReadable()) {
        //El canal esta listo para ser leído
    }
    keyIterator.remove(); //eliminamos la clave del iterador
  }
}
canal1.close();      canal2.close();
}
}
```

Socket

- Los sockets comunican procesos de diferentes (o el mismo) ordenador.
- **Apertura**

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(new InetSocketAddress("http://miequipo.com", 80));
```

- **Cierre**

```
socketChannel.close();
```

- **Lectura**

```
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = socketChannel.read(buf);
```

- **Escritura**

```
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();
while(buf.hasRemaining()) {
  socketChannel.write(buf);
}
```

Socket- no bloqueantes

- Los sockets pueden utilizarse en modo no bloqueante para poder realizar otras tareas sin necesidad de esperar los datos. En este modo es necesario utilizar las siguientes operaciones
- Conexión:

```
static void conexionAlSocket() {  //Modo no bloqueante
SocketChannel socketChannel = SocketChannel.open();
socketChannel.configureBlocking(false);
socketChannel.connect(new InetSocketAddress("http://miequipo.com", 80));

ByteBuffer buf = ByteBuffer.allocate(48);
while(! socketChannel.finishConnect() ){
  int bytesRead = socketChannel.read(buf);
  if (bytesRead >0)
  //Procesar datos leídos
}
}
```

ServerSockets

- Los ServerSockets permiten estar escuchando en un puerto para aceptar conexiones.

```
ServerSocketChannel serverSocketChannel =
ServerSocketChannel.open();

serverSocketChannel.socket().bind(new InetSocketAddress(9999));
serverSocketChannel.configureBlocking(false);
while(true){
    SocketChannel socketChannel = serverSocketChannel.accept();
    //si configuramos el socket como no bloqueante la instrucción
    //  anterior no es bloqueante
    // y hay que preguntar si hay conexión o no:
    if (socketChannel != null)
      //Utilizar el socketChannel
}
```

DatagramChannels

- DatagramChannels (para sockets UDP)
  - Creación

    DatagramChannel channel = DatagramChannel.open();
    channel.socket().bind(new InetSocketAddress(9999));
  - Lectura

    ByteBuffer buf = ByteBuffer.allocate(48);
    buf.clear();
    channel.receive(buf);
  - Escritura

    String newData = "Tiempo"+ System.currentTimeMillis();
    ByteBuffer buf = ByteBuffer.allocate(48);
    buf.clear();
    buf.put(newData.getBytes());
    buf.flip();
    int bytesSent = channel.send(buf, new InetSocketAddress("jenkov.com", 80));

Pipes

- Pipes para comunicar procesos del mismo ordenador
  - Creación

    Pipe pipe = Pipe.open();
  - Lectura

    Pipe.SourceChannel sourceChannel = pipe.source(); //acceso
    ByteBuffer buf = ByteBuffer.allocate(48);
    int bytesRead = inChannel.read(buf);
  - Escritura

    Pipe.SinkChannel sinkChannel = pipe.sink(); //acceso
    String newData = "Dato Escrito"+ System.currentTimeMillis();
    ByteBuffer buf = ByteBuffer.allocate(48);
    buf.clear();
    buf.put(newData.getBytes());
    buf.flip();
    while(buf.hasRemaining()) {
    sinkChannel.write(buf);
    }

Diferencia JavaNIO vs IO

- *IO está orientado a los streams mientras que **JavaNIO está orientado** a los **bufferes**.*
- *IO es bloqueante mientras que **JavaNIO puede ser no bloqueante**.*
- *Con **javaNIO puedes utilizar selectores** pero con IO no puedes*