

Archivos

UNRN

Universidad Nacional
de Río Negro

VII

2024





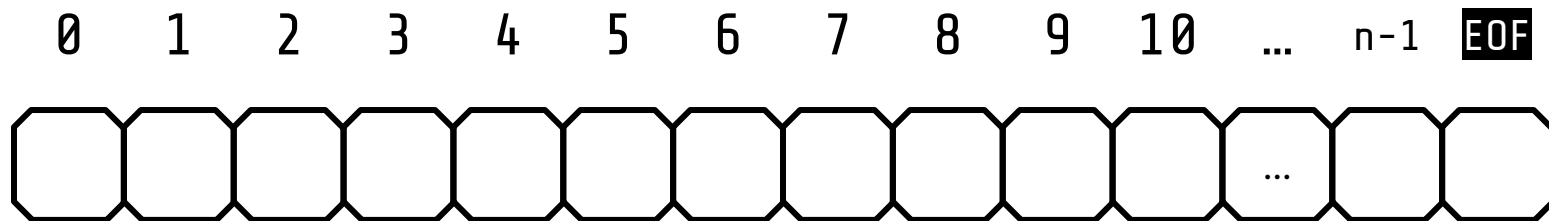
¿Preguntas?

Archivos

Files and Streams

Como almacenamiento más allá del cierre del programa

Como ve Java a un archivo



Casi como un arreglo

Modos

binario

texto

Cada posición se puede tomar de
a byte o de a char

texto

La interpretación de la información es entendible* por una persona, pero menos eficiente.

binario

La interpretación de la información es más eficiente pero accesible solo por el programa.

Diferencias guardando números

texto

Un número de 10 dígitos ocupa
160-bytes, un carácter por dígito

binario

Un número de 10 dígitos ocupa
~4-bytes



**Igual solo vamos a
tratar con archivos
de texto**

**El comportamiento
es *casi* como un
arreglo**

**Porque trabajamos
sin tener una idea
clara del tamaño**

Se procesan como un “flujo”



stream

Como
System.in
System.out
System.err

**Tienen una
dirección y una
posición**

Vamos a usar Scanner y Formatter

**Se manipulan con
clases ubicadas en
java.io
java.nio**

Primero ubiquemos los archivos

rutas

java.nio.Path / java.nio.Paths



Una ubicación dentro de la estructura de directorios.

Puede apuntar a un archivo o directorio.

Que varían entre Windows y GNU/Linux

absoluta

C:\users\usuario\

/home/usuario

relativa

Descargas\archivo

documentos/subdirectorio

File.separator

Contiene el separador correcto al sistema operativo para armar cadenas con rutas.

java.nio.file.Paths

```
Path <- Paths.getPath(String primero, ...)
```

java.nio.file.Path

```
Path actual = Paths.getPath(".");  
actual.toAbsolutePath();
```

Archivos

java.nio.file.Files

Consultas

`Files.exist(path)`

`Files.notExists(path)`

`Files.isRegularFile(path)`

`Files.isReadable(path)`

`Files.isWritable(path)`

`Files.isSameFile(path1, path2)`


`Files.size(path)`


Contenido en una ubicación

```
DirectoryStream<Path> stream =  
    Files.newDirectoryStream(path);
```

Largo pero simple

```
Path pwd = Paths.get(".");
try {
    DirectoryStream<Path> stream
        = Files.newDirectoryStream(pwd);
    for (Path contenido : stream) {
        System.out.println(contenido.toString());
    }
    stream.close();
} catch (IOException exc) {
    exc.printStackTrace();
}
```





**Cuando el archivo
entra entero en
memoria (~2gb)**

Leer el archivo

```
String <- readString(Path path)
```

Escribir a un archivo

```
Files.writeString(path, cadena, modo);
```

```
StandardOpenOption.APPEND
```

```
StandardOpenOption.WRITE
```

```
StandardOpenOption.CREATE
```

Escritura y lectura todo junto

```
Path ruta = Paths.get(".", "test.txt");
try {
    String cadena = "Hola Mundo!";
    Files.writeString(ruta, cadena, StandardOpenOption.APPEND);
    String contenido = Files.readString(ruta);
    System.out.println(contenido);
} catch (IOException exc) {
    exc.printStackTrace();
}
```


Todas las operaciones de archivos lanzan IOException



Formas de crear una cadena

java.util.Formatter

```
StringBuilder builder = new StringBuilder();  
Formatter salida = new Formatter(builder);  
  
salida.format(...igual que printf...);  
  
salida.close();
```

Se puede usar con un Path

```
Path ruta = Paths.get(".", "test.txt");  
try (Formatter salida = new Formatter(rutaToFile())) {  
    salida.format(...igual que printf...);  
} catch (IOException exc ) {  
    gestionamos errores  
}
```

Try con recurso

Al usar algo que requiera 'cierre' y pueda fallar

```
try ( construcción y asignación del recurso ) {  
    uso del recurso  
} catch (excepciones) {  
    gestion de errores  
}
```

Llama **close** automáticamente al finalizar

Alternativa

Excepciones con `try/catch/finally`

try - catch - finally

```
try {
```

Código que puede fallar

```
} catch (excepción) {
```

Gestión de errores

```
} finally {
```

Esto se ejecuta en cualquiera de los dos casos

```
}
```

Con tantos **catch** como sea necesario

Usado con archivos también

try/finally con Formatter

```
Path ruta = Paths.get(".", "test.txt");
Formatter salida = null;
try {
    salida = new Formatter(ruta.toFile());
    salida.format(...igual que printf...);
} catch (IOException exc ) {
    gestion de errores
} finally {
    if (salida != null) {
        salida.close();
    }
}
```

Aunque es menos compacto que el try/resources

try/finally con Scanner

```
Path ruta = Paths.get(".", "test.txt");
Scanner scanner = null;
try {
    scanner = new Scanner(ruta.toFile());
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (IOException e) {
    gestion de errores
} finally {
    if (scanner != null) {
        scanner.close();
    }
}
```

Pero no tan usado

**Se van a encontrar que
donde hay una
IOException están
obligados a hacer algo***

Excepciones II

Repaso excepciones

Ejemplo I

```
public class FailApp{

    public static void main(String[] args) {
        a();
    }
    static void a() {
        b();
    }
    static void b() {
        throw new RuntimeException();
    }
}
```

Ejemplo II

```
public class FailApp{

    public static void main(String[] args) {
        a();
    }
    static void a() {
        try{
            b();
        } catch (RuntimeException exc){
            System.out.println("Ouch");
        }
    }
    static void b() {
        throw new RuntimeException();
    }
}
```


Ejemplos III

```
public class FailApp{

    public static void main(String[] args) {
        funcion();
    }

    static void funcion() {
        try {
            System.out.println("Tambores");
            throw new RuntimeException();
            System.out.println("OK");
        } catch (RuntimeException exc) {
            System.out.println("Ouch");
        }
    }
}
```



¿Preguntas?

Tipos de excepciones

RuntimeException

Excepciones sin tipo

Características generales

- Simplemente, se lanzan (¡o se reciben!)
- Se usan para situaciones inesperadas pero recuperables
- Por lo general, las podemos evitar con un `if`.
- Y encontrarnos con una es generalmente un bug en el programa.

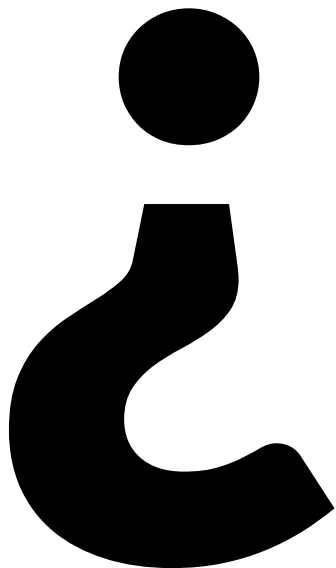
```
if (numero < 0){  
    throw new PreconditionException("no puede ser negativo");  
}
```

Exception

Excepciones con tipo

Características generales

- O se atajan o se delegan, alguien se tiene que hacer cargo.
- Su lanzamiento se declara explícitamente en la función.
- Se usan para situaciones que si o si tenemos que considerar.



**Cuando usar una
u otra**



1

**Si quien llama a la
función lo puede
evitar con un if
RuntimeException**

2

Si tienen dudas; Exception.

**Obligar el tratamiento de una
excepción tampoco es tan malo.**

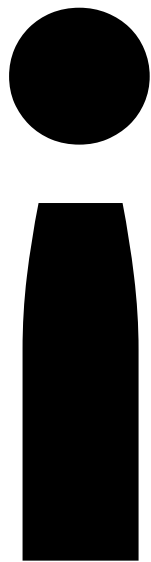
**Dejen documentada
la razón por la cual
usan una u otra.**



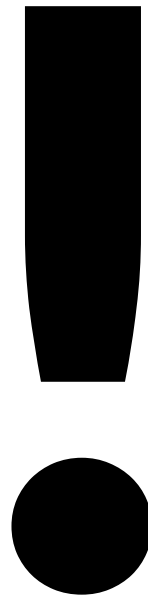
**Aunque es
técnicamente
posible**

Que main emita una excepción

```
public class FailApp{  
  
    public static void main(String[] args) throws UnaExcepcion{  
        metodoFail();  
    }  
    static void metodoFail() {  
        throw new UnaExcepcion();  
    }  
}
```



**Alguien se tiene
que hacer cargo**



¡Hay que atajar todas las excepciones con tipo!

```
public class FailApp{

    public static void main(String[] args){
        try{
            metodoFail();
        } catch (UnaExcepcion exc){
            exc.printStackTrace();
        }
    }
    static void metodoFail() {
        throw new UnaExcepcion();
    }
}
```


¡Su código **no puede
dejar excepciones
*con tipo sin atajar!***

—

**Pero algo
netamente
*erróneo***

Para 'suavizar' una excepción

```
try{  
    //código que puede fallar  
catch (IOException exc){  
    throw new RuntimeException(exc);  
}
```



¿Preguntas?

**Pero en el momento
apropiado.**

**No es cuestión de hacer
try/catch por todos
lados**

**Si solo hacen un
print/printStackTrace**

**Dejen que la
excepción siga su
camino**

**Ya que solo la
“silencian”**

**Con archivos es
particularmente
notorio**

¿Esta función puede cumplir con su objetivo siempre?

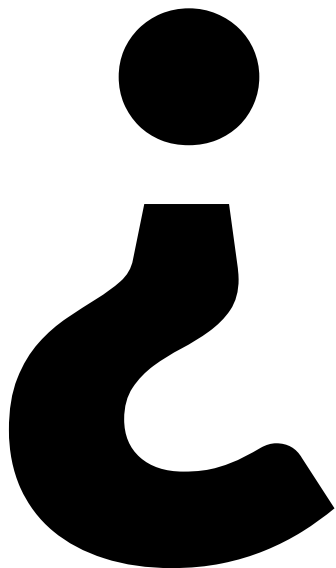
```
public static File crearArchivo(String nombre) {  
    File archivo = new File(nombre);  
    try {  
        archivo.createNewFile();  
    } catch (IOException exc) {  
        exc.printStackTrace();  
    }  
    return archivo;  
}
```

¿Esta función puede cumplir con su objetivo siempre?

```
public static File crearArchivo(String nombre) {  
    File archivo = new File(nombre);  
    try {  
        archivo.createNewFile();  
    } catch (IOException exc) {  
        exc.printStackTrace();  
    }  
    return archivo;  
}
```

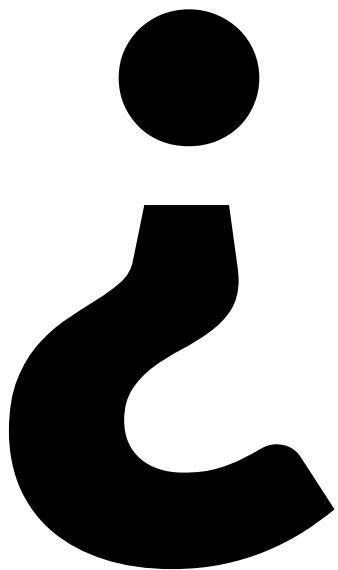
¿Y acá? ¿Que pasa si 'archivo' ya existía?

```
public static void escribirInforme(String archivo, String[] informe){  
    File destino = crearArchivo(archivo);  
    escribir(destino, informe);  
}
```



**¿Y si el archivo ya
existía?**





**¿Y si el archivo no
se puede
escribir?**



**Es importante
pensar en el
“usuario” de la
función**

uno mismo

Necesito saber **que** falló y **como** fallo para tomar la decisión correcta

No apuren la captura de la excepción

Da una falsa sensación de seguridad

```
public static File crearArchivo(String nombre) {  
    File archivo = new File(nombre);  
    try {  
        archivo.createNewFile();  
    } catch (IOException exc) {  
        exc.printStackTrace();  
    }  
    return archivo;  
}
```

**Simplemente, no
hay nada que hacer**

¿Esta función puede cumplir con su objetivo siempre?

```
public static File crearArchivo(String nombre)
                                throws IOException{
    File archivo = new File(nombre);
    archivo.createNewFile();
    return archivo;
}
```

De esta manera cuando lleguen a escribir...

```
public static void escribirInforme(String archivo, String[] informe)
    throws IOException{
    File destino;
    try{
        destino = crearArchivo(archivo);
    }catch (FileNotFoundException exc){
        throw new ArchivoNoEncontrado(exc);
    }
    escribir(destino, informe);
}
```

**Encontrar el equilibrio
requiere práctica**



¿Preguntas?

Creación de Excepciones

En un archivo separado

```
public static class NoMasIntentosException extends Exception{  
    public NoMasIntentosException(){  
        super();  
    }  
}
```

El TP3 tiene un ejemplo más completo.

Pero, es necesario declarar su uso

```
/**  
 *Pide un número entero, con un mensaje personalizado  
 * y una cantidad limitada de intentos  
 * @throws NoMasIntentosException cuando agotamos intentos  
 */  
public static int pideInt(String mensaje, int intentos)  
    throws NoMasIntentosException {  
    código que resuelve el ejercicio  
}
```



¿Preguntas?



Una observación extra

printStackTrace

es un `print` con pasos adicionales
Y como tales, *técnicamente* no van dentro de las funciones



¿Preguntas?

extends Exception
Vamos a ver sobre
este tema la
próxima clase.

TP4

Archivos

unrn.edu.ar

