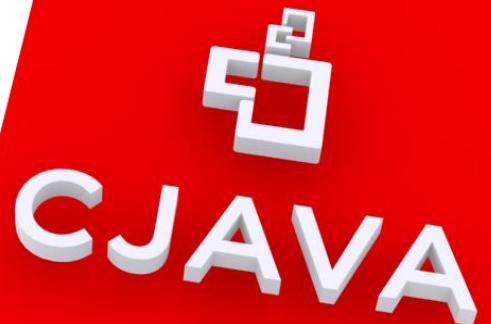


Edwin Maraví
emaravi@cjavaperu.com





Misión

Nuestro equipo trabaja para integrar la tecnología Java en la sociedad como solución a todas sus necesidades.





CJAVA
siempre para apoyarte

010101010101010101010101010101
001010101010101010101010101010101
010101010101010101010101010100101
01010101010101010101
1010101010101010
0101010101010101010101010101010101
101010101010101010101001010010101010101
01010101010101

Visión

Poder aportar al desarrollo del País usando tecnología Java.



Servicios Académicos

Programer (80 horas - Certificación Java 11)

[Certificado: Java Programer]

Developer (80 horas - Spring FrameWork y Angular)

[Certificado: Java Developer]

Expert (80 horas – Microservicios y DevOps)

[Certificado: Java Expert]

Architect (80 horas)

[Certificado: Java Arquitect]

Carrera (12 meses)

[Diploma: Carrera Java]

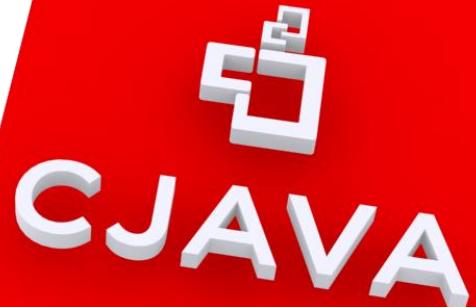
Programmer



Quienes Somos

Somos una organización orientada a **desarrollar, capacitar e investigar tecnología JAVA** a través de un prestigioso staff de profesionales a nivel nacional.





Contáctenos

- Av. Arenales 395 oficina 405
Santa Beatriz - Lima 01 - Perú
- Teléfono: 433-6948
- RPC / WhatsApp: 932 656 459
- Email: info@cjavaperu.com

Síguenos

- [/cjava.peru.1](https://www.facebook.com/cjava.peru.1)
- [@cjava_peru](https://twitter.com/cjava_peru)
- [/cjavaperu](https://www.linkedin.com/company/cjavaperu/)



Exceptions y Assertions



Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Definir el objetivo de las excepciones de Java
- Utilizar las sentencias try y throw
- Utilizar las cláusulas catch, multi-catch y finally
- Cerrar automáticamente recursos con una sentencia try-with-resources
- Reconocer categorías y clases de excepciones comunes
- Crear excepciones personalizadas y recursos que se puedan cerrar automáticamente
- Probar invariantes a través de afirmaciones.



Manejo de errores

Las aplicaciones encontrarán errores durante su ejecución.
Una aplicación fiable debe manejar los errores lo mejor posible.
Los errores:

- Deben ser la “excepción” y no el comportamiento esperado
- Deben poder manejarse para crear aplicaciones fiables
- Se pueden producir como resultado de bugs en las aplicaciones
- Se pueden producir debido a factores más allá del control de la aplicación
 - Bases de datos inaccesibles
 - Fallo del disco duro



Manejo de excepciones en Java

Cuando se usan bibliotecas de Java que se basan en recursos externos, el compilador le exigirá que “maneje o declare” las excepciones que se puedan producir.

- Manejar una excepción significa que hay que agregar un bloque de código para manejar el error.
- Declarar una excepción significa que se declara que un método puede fallar y no ejecutarse correctamente.



La sentencia try-catch

La sentencia try-catch se utiliza para manejar excepciones.

```
try {  
    System.out.println("About to open a file");  
    InputStream in =  
        new FileInputStream("missingfile.txt");  
    System.out.println("File open");  
} catch (Exception e) {  
    System.out.println("Something went wrong!");  
}
```

Se omite la línea si la línea anterior no pudo abrir el archivo.

Esta línea solo se ejecuta
si se produjo algún error
en el bloque try.



Objetos Exception

A las cláusulas catch se les pasan referencias a objetos java.lang.Exception. La clase java.lang.Throwable es la clase principal para Exception, y describe varios métodos utilizables.

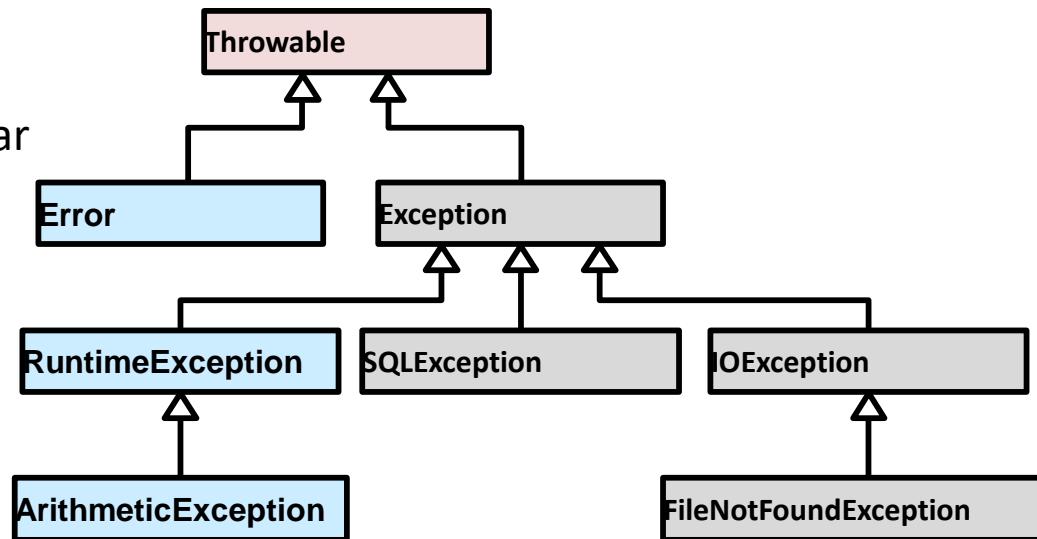
```
try{
    //...
} catch (Exception e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
}
```



Categorías de excepciones

La clase `java.lang.Throwable` conforma la base de la jerarquía de clases de excepciones. Hay dos categorías de excepciones principales:

- Excepciones comprobadas, las cuales se deben “manejar o declarar”.
- Excepciones no comprobadas, que normalmente no se “manejan o declaran”.





Manejo de excepciones

Debe siempre capturar el tipo de excepción más específico. Se pueden asociar varios bloques catch con una única sentencia try.

```
try {  
    System.out.println("About to open a file");  
    InputStream in = new FileInputStream("missingfile.txt");  
    System.out.println("File open");  
    int data = in.read();  
    in.close();  
} catch (FileNotFoundException e) {  
    System.out.println(e.getClass().getName());  
    System.out.println("Quitting");  
} catch (IOException e) {  
    System.out.println(e.getClass().getName());  
    System.out.println("Quitting");  
}
```

El orden es importante. Primero es necesario capturar las excepciones más específicas (es decir, las clases secundarias antes que las clases principales).



La cláusula finally

```
InputStream in = null;
try {
    System.out.println("About to open a file");
    in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int data = in.read();
} catch (IOException e) {
    System.out.println(e.getMessage());
} finally {
    try {
        if(in != null) in.close();
    } catch(IOException e) {
        System.out.println("Failed to close file");
    }
}
```

Las cláusulas finally se ejecutan con independencia de si se ha generado o no un objeto Exception.

Siempre hay que cerrar los recursos abiertos



La sentencia try-with-resources

Java SE 7 proporciona una nueva sentencia try-withresources para cerrar automáticamente recursos.

```
System.out.println("About to open a file");
try (InputStream in =
      new FileInputStream("missingfile.txt")) {
    System.out.println("File open");
    int data = in.read();
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```



Excepciones suprimidas

Si se produce una excepción en el bloque try de una sentencia try-with-resources y se produce una excepción mientras se cierran los recursos, las excepciones resultantes se suprimirán.

```
} catch (Exception e) {  
    System.out.println(e.getMessage());  
    for (Throwable t : e.getSuppressed()) {  
        System.out.println(t.getMessage());  
    }  
}
```



La interfaz de AutoCloseable

Un recurso en una sentencia try-with-resources debe implantar una de las siguientes opciones:

- `java.lang.AutoCloseable`
 - Nueva en JDK 7
 - Puede devolver un objeto `Exception`
- `java.io.Closeable`
 - Refactorizada en JDK7 para ampliar `AutoCloseable`
 - Puede devolver un objeto `IOException`

```
public interface AutoCloseable {  
    void close() throws Exception;  
}
```



Captura de varias excepciones

Java SE 7 proporciona una nueva cláusula multi-catch.

```
ShoppingCart cart = null;
try (InputStream is = new FileInputStream(cartFile);
     ObjectInputStream in = new ObjectInputStream(is)) {
    cart = (ShoppingCart)in.readObject();
} catch (ClassNotFoundException | IOException e) {
    System.out.println("Exception deserializing " + cartFile);
    System.out.println(e);
    System.exit(-1);
}
```

Cuando hay varios tipos de excepciones, se separan con una barra vertical.



Declaración de excepciones

Puede declarar que un método devuelve una excepción en lugar de manejarla.

```
public static int readByteFromFile() throws IOException {  
    try (InputStream in = new FileInputStream("a.txt")) {  
        System.out.println("File open");  
        return in.read();  
    }  
}
```



Observe la falta de cláusulas catch. La sentencia try withresources se está utilizando solo para cerrar recursos.



Manejo de excepciones declaradas

Las excepciones que pueden devolver los métodos deben manejarse. Al declarar una excepción simplemente se especifica que es otra persona la que tiene que manejarla.

```
public static void main(String[] args) {  
    try {  
        int data = readByteFromFile();  
    } catch (IOException e) {  
        System.out.println(e.getMessage());  
    }  
}
```

Método que ha declarado una excepción



Devolución de excepciones

Puede volver a emitir una excepción capturada previamente. Tenga en cuenta que hay una cláusula throws y una sentencia throw.

```
public static int readByteFromFile() throws IOException {
    try (InputStream in = new FileInputStream("a.txt")) {
        System.out.println("File open");
        return in.read();
    } catch (IOException e) {
        e.printStackTrace();
        throw e;
    }
}
```



Excepciones personalizadas

Puede crear clases de excepciones personalizadas extendiendo Exception o una de sus subclases.

```
public class DAOException extends Exception {  
  
    public DAOException() {  
        super();  
    }  
  
    public DAOException(String message) {  
        super(message);  
    }  
}
```



Excepciones de envoltorio

Utilice una excepción de envoltorio para ocultar el tipo de excepción que se esté generando sin simplemente ocultar la excepción.

```
public class DAOException extends Exception {  
    public DAOException(Throwable cause) {  
        super(cause);  
    }  
  
    public DAOException(String message, Throwable cause) {  
        super(message, cause);  
    }  
}
```



Revisión del patrón DAO

El patrón DAO utiliza la abstracción (una interfaz) para permitir sustituir la implantación. Un DAO de base de datos o archivo debe gestionar excepciones. Una implantación de DAO puede usar una excepción de envoltorio para preservar la abstracción y evitar ocultar excepciones.

```
public Employee findById(int id) throws DAOException {  
    try {  
        return employeeArray[id];  
    } catch (ArrayIndexOutOfBoundsException e) {  
        throw new DAOException("Error finding employee in DAO", e);  
    }  
}
```



Afirmaciones

- Utilice afirmaciones para documentar y verificar las suposiciones y la lógica interna de un único método:
 - Invariantes internas
 - Invariantes de flujo de control
 - Condiciones posteriores e invariantes de clases
- Usos de afirmaciones no adecuados
 - Las afirmaciones se pueden desactivar en el tiempo de ejecución; por ello:
 - No utilice afirmaciones para comprobar los parámetros de un método público.
 - No utilice métodos que puedan generar efectos secundarios en la comprobación d



Sintaxis de las afirmaciones

- La sintaxis de una afirmación es la siguiente:
`assert <expresión_booleana>;`
`assert <expresión_booleana> : <expresión_detalle>;`
- Si `<expresión_booleana>` se evalúa en `false`, entonces se devuelve `AssertionError`.
- El segundo argumento se convierte a una cadena y se utiliza como texto descriptivo en el mensaje `AssertionError`.



Invariantes internas

- El problema es:

```
1  if (x > 0) {  
2      // do this  
3  } else {  
4      // do that  
5  }
```

- La solución es:

```
1  if (x > 0) {  
2      // do this  
3  } else {  
4      assert ( x == 0 );  
5      // do that, unless x is negative  
6  }
```



Invariantes de flujo de control

Ejemplo:

```
1 switch (suit) {  
2     case Suit.CLUBS: // ...  
3         break;  
4     case Suit.DIAMONDS: // ...  
5         break;  
6     case Suit.HEARTS: // ...  
7         break;  
8     case Suit.SPADES: // ...  
9         break;  
10    default: assert false : "Unknown playing card suit";  
11        break;  
12 }
```

Condiciones posteriores e invariantes de clases

Ejemplo:

```
1  public Object pop() {  
2      int size = this.getElementCount();  
3      if (size == 0) {  
4          throw new RuntimeException("Attempt to pop from empty stack");  
5      }  
6  
7      Object result = /* code to retrieve the popped element */ ;  
8  
9      // test the postcondition  
10     assert (this.getElementCount() == size - 1);  
11  
12     return result;  
13 }
```



Control de evaluación de tiempo de ejecución de afirmaciones

- Si se ha desactivado la comprobación de afirmaciones, el código se ejecuta igual de rápido que si las comprobaciones no hubieran estado allí nunca.
- La comprobación de afirmaciones está desactivada por defecto. Active las afirmaciones con cualquiera de los siguientes comandos:

```
java -enableassertions MyProgram
```

```
java -ea MyProgram
```

- La comprobación de las afirmaciones se puede controlar por clases, paquetes y jerarquías de paquetes. Consulte:
<http://download.oracle.com/javase/7/docs/technotes/guides/language/assert.html>



Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Definir el objetivo de las excepciones de Java
- Utilizar las sentencias try y throw
- Utilizar las cláusulas catch, multi-catch y finally
- Cerrar automáticamente recursos con una sentencia try-with-resources
- Reconocer categorías y clases de excepciones comunes
- Crear excepciones personalizadas y recursos que se puedan cerrar automáticamente
- Probar invariantes a través de afirmaciones



Fundamentos Java IO



Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Describir los aspectos básicos de entrada y salida en Java.
- Leer datos de la consola y escribir datos en ella.
- Utilizar flujos para leer y escribir datos.
- Leer y escribir objetos mediante serialización.

Conceptos básicos de E/S en Java

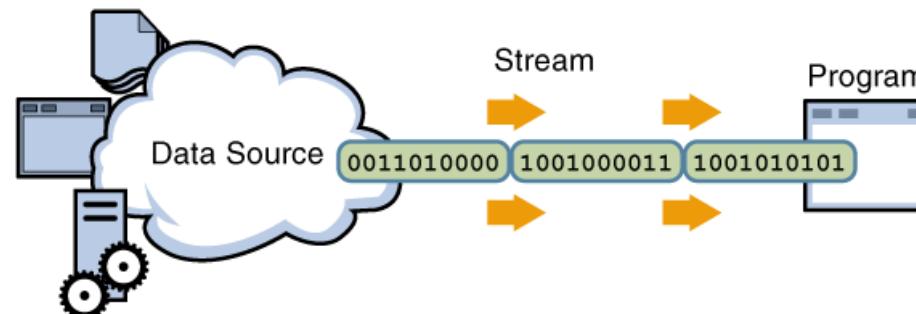
El lenguaje de programación Java proporciona un completo juego de bibliotecas para realizar funciones de entrada/salida (E/S).

- Java define los canales de E/S como flujos.
- Un flujo de E/S representa un origen de entrada o un destino de salida.
- Los flujos pueden representar muchos tipos de orígenes y destinos diferentes, como archivos de disco, dispositivos, otros programas y matrices de memoria.
- Los flujos admiten muchos tipos de datos diferentes, como bytes simples, tipos de datos primitivos, caracteres localizados y objetos.

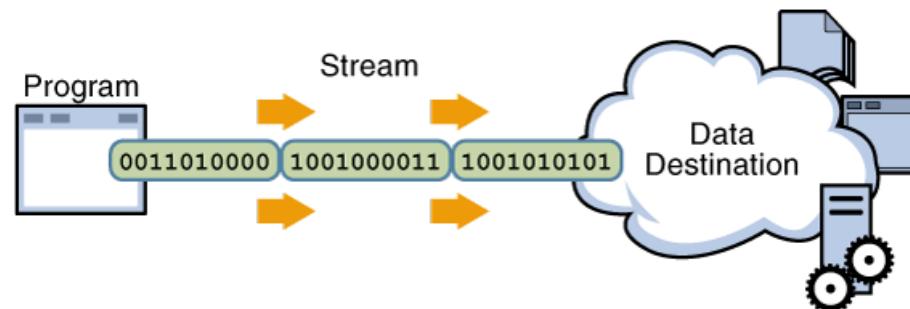


Flujos de E/S

- Los programas utilizan flujos de entrada para leer datos desde un origen un elemento cada vez.



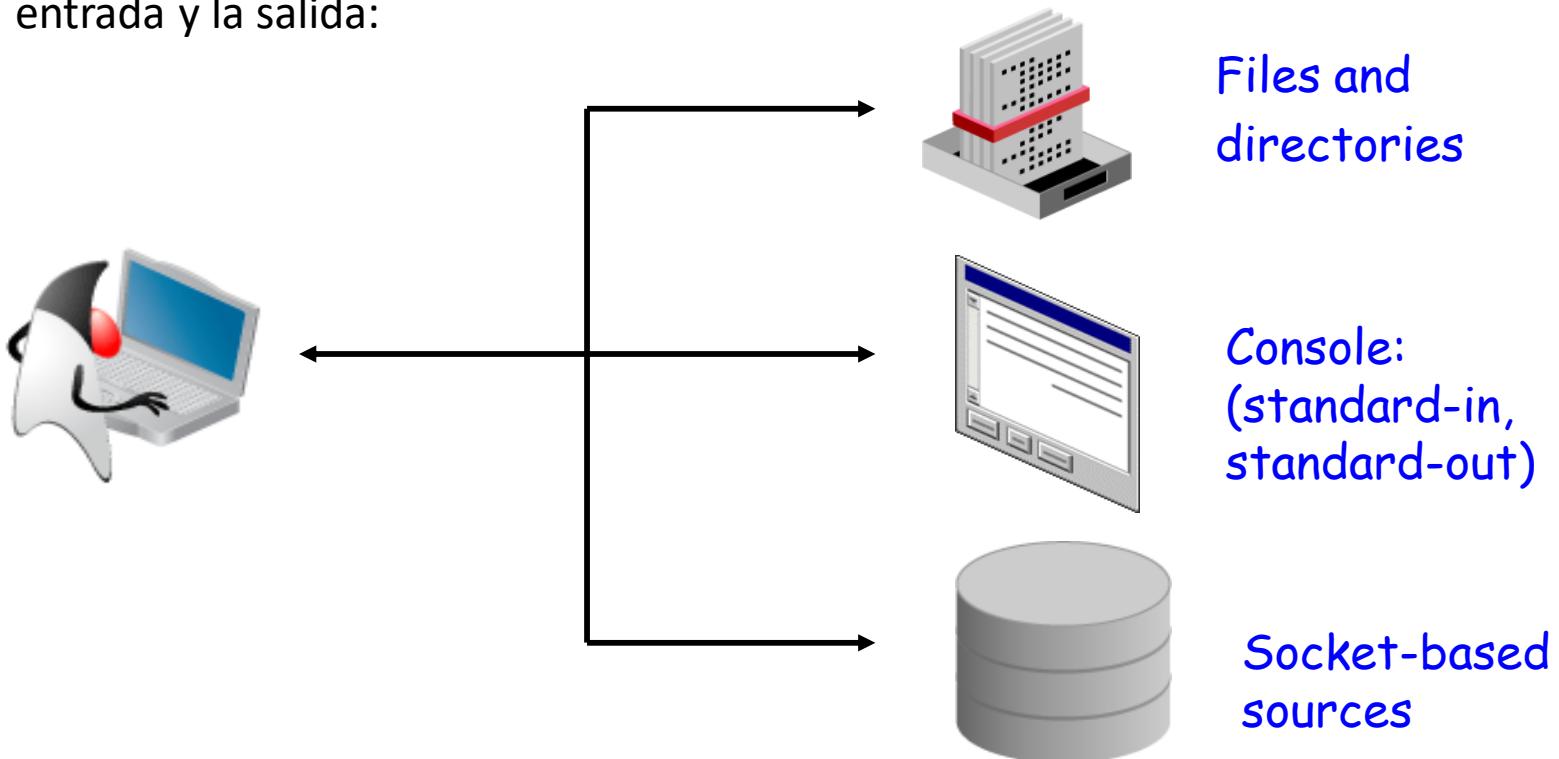
- Los programas utilizan flujos de salida para escribir datos en un destino (receptor) un elemento cada vez.





Aplicación de E/S

Generalmente existen tres maneras en las que un desarrollador puede usar la entrada y la salida:





Datos dentro de flujos

- La tecnología Java soporta dos tipos de flujos: de caracteres y de bytes.
- La entrada y la salida de datos de caracteres se maneja a través de lectores y escritores.
- La entrada y la salida de datos de bytes se maneja a través de flujos de entrada y flujos de salida:
 - Normalmente, el término *flujo* hace referencia a un flujo de bytes.
 - Los términos *lector* y *escritor* hacen referencia a flujos de caracteres.

Flujo	Flujos de bytes	Flujos de caracteres
Flujos de origen	InputStream	Reader
Flujos de receptor	OutputStream	Writer



Métodos InputStream de flujos de bytes

- Los tres métodos básicos read son:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

- Otros métodos incluyen:

```
void close();           // Close an open stream
int available();       // Number of bytes available
long skip(long n);    // Discard n bytes from stream

boolean markSupported(); // 
//void mark(int readlimit); // Push-back operations
void reset();
```

Métodos outputStream de flujos de bytes

- Los tres métodos básicos write son:

```
void write(int c)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

- Otros métodos incluyen:

```
void close(); // Automatically closed in try-with-resources
void flush(); // Force a write to the stream
```



Ejemplo de flujo de bytes

```
1 import java.io.FileInputStream; import java.io.FileOutputStream;
2 import java.io.FileNotFoundException; import java.io.IOException;
3
4 public class ByteStreamCopyTest {
5     public static void main(String[] args) {
6         byte[] b = new byte[128]; int bLen = b.length;
7         // Example use of InputStream methods
8         try (FileInputStream fis = new FileInputStream (args[0]));
9             FileOutputStream fos = new FileOutputStream (args[1])) {
10             System.out.println ("Bytes available: " + fis.available());
11             int count = 0; int read = 0;
12             while ((read = fis.read(b)) != -1) {
13                 if (read < bLen) fos.write(b, 0, read);
14                 else fos.write(b);
15                 count += read;
16             }
17             System.out.println ("Wrote: " + count);
18         } catch (FileNotFoundException f) {
19             System.out.println ("File not found: " + f);
20         } catch (IOException e) {
21             System.out.println ("IOException: " + e);
22         }
23     }
24 }
```

Tenga en cuenta que es necesario saber cuántos bytes se leen cada vez en la matriz de bytes.

Métodos Reader de flujos de caracteres

- Los tres métodos básicos read son:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

- Otros métodos incluyen:

```
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```

Métodos Writer de flujos de caracteres

- Los métodos básicos write son:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

- Otros métodos incluyen:

```
void close()
void flush()
```



Ejemplo de flujo de caracteres

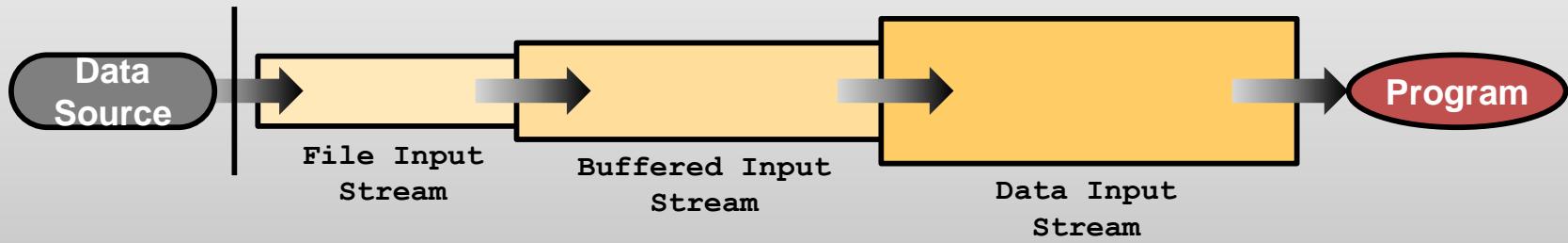
```
1 import java.io.FileReader; import java.io.FileWriter;
2 import java.io.IOException; import java.io.FileNotFoundException;
3
4 public class CharStreamCopyTest {
5     public static void main(String[] args) {
6         char[] c = new char[128]; int cLen = c.length;
7         // Example use of InputStream methods
8         try (FileReader fr = new FileReader(args[0]));
9             FileWriter fw = new FileWriter(args[1])) {
10             int count = 0;
11             int read = 0;
12             while ((read = fr.read(c)) != -1) {
13                 if (read < cLen) fw.write(c, 0, read);
14                 else fw.write(c);
15                 count += read;
16             }
17             System.out.println("Wrote: " + count + " characters.");
18         } catch (FileNotFoundException f) {
19             System.out.println("File " + args[0] + " not found.");
20         } catch (IOException e) {
21             System.out.println("IOException: " + e);
22         }
23     }
24 }
```

Ahora, en lugar de una matriz de bytes, esta versión utiliza una matriz de caracteres.

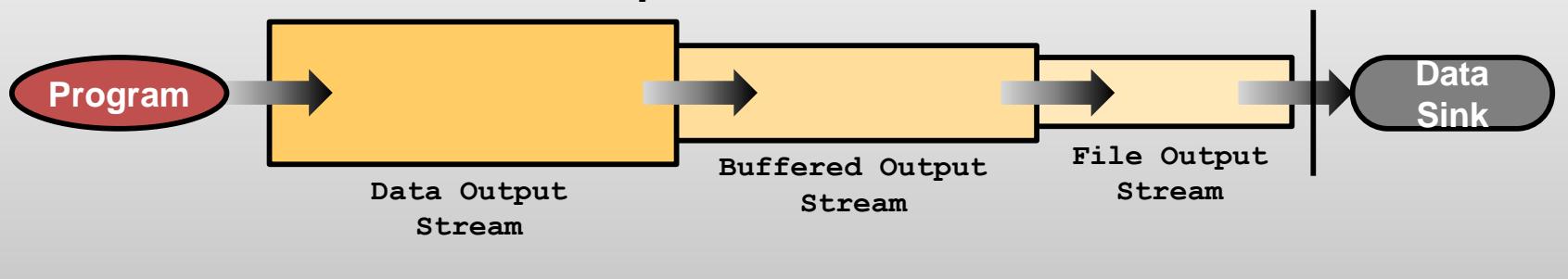


Cadenas de flujos de E/S

Input Stream Chain



Output Stream Chain





Ejemplo de flujos de cadena

```
1 import java.io.BufferedReader; import java.io.BufferedWriter;
2 import java.io.FileReader; import java.io.FileWriter;
3 import java.io.FileNotFoundException; import java.io.IOException;
4
5 public class BufferedStreamCopyTest {
6     public static void main(String[] args) {
7         try (BufferedReader bufInput
8              = new BufferedReader(new FileReader(args[0])));
9             BufferedWriter bufOutput
10            = new BufferedWriter(new FileWriter(args[1]))) {
11             String line = "";
12             while ((line = bufInput.readLine()) != null) {
13                 bufOutput.write(line);
14                 bufOutput.newLine();
15             }
16         } catch (FileNotFoundException f) {
17             System.out.println("File not found: " + f)
18         } catch (IOException e) {
19             System.out.println("Exception: " + e);
20         }
21     }
22 }
```

Una clase FileReader encadenada a una clase BufferedReader: esto permite usar un método que lea una cadena.

El buffer de caracteres se sustituye por String. Observe cómo readLine() utiliza el carácter de nueva línea como terminador. Así pues, deberá volver a agregarlo al archivo de salida.



Flujos de procesamiento

Funcionalidad	Flujos de caracteres	Flujos de bytes
Almacenamiento en buffer (cadenas)	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtrado	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Conversión (de bytes a caracteres)	InputStreamReader OutputStreamWriter	
Serialización de objetos		ObjectInputStream ObjectOutputStream
Conversión de datos		DataInputStream DataOutputStream
Recuento	LineNumberReader	LineNumberInputStream
Consulta hacia	PushbackReader	PushbackInputStream
Impresión	PrintWriter	PrintStream



E/S de la consola

La clase System del paquete java.lang tiene tres campos de instancias estáticas: out, in y err.

- El campo System.out es una instancia estática de un objeto PrintStream que le permite escribir en una *salida estándar*.
- El campo System.in es una instancia estática de un objeto InputStream que le permite leer a partir de una *entrada estándar*.
- El campo System.err es una instancia estática de un objeto PrintStream que le permite escribir en un *error estándar*.



Java.io.Console

Además de los objetos PrintStream, System también puede acceder a una instancia del objeto java.io.Console:

```
10  Console cons = System.console();
11  if (cons != null) {
12      String userTyped; String pwdTyped;
13      do {
14          userTyped = cons.readLine("%s", "User name:");
15          pwdTyped = new String(cons.readPassword("%s", "Password: "));
16          if (userTyped.equals("oracle") && pwdTyped.equals("tiger")) {
17              userValid = true;
18          } else {
19              System.out.println("Wrong user name/password. Try again.\n");
20          }
21      } while (!userValid);
22  }
```

readPassword no hace eco de los caracteres introducidos en la consola.

- Tenga en cuenta que deberá pasar el nombre de usuario y la contraseña para el proceso de autenticación.



Escritura en una salida estándar

- Los métodos `println` y `print` forman parte de la clase `java.io.PrintStream`.
- Los métodos `println` imprimen el argumento y un carácter de línea nueva (`\n`).
- Los métodos `print` imprimen el argumento sin un carácter de línea nueva.
- Los métodos `print` y `println` se sobrecargan para la mayor parte de los tipos primitivos (`boolean`, `char`, `int`, `long`, `float` y `double`) y para `char[]`, `Object` y `String`.
- Los métodos `print(Object)` y `println(Object)` llaman al método `toString` en el argumento.



Lectura a partir de una entrada estándar

```
1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 public class KeyboardInput {
5     public static void main(String[] args) {
6         try (BufferedReader in =
7             new BufferedReader (new InputStreamReader (System.in))) {
8             String s = "";
9             // Read each input line and echo it to the screen.
10            while (s != null) {
11                System.out.print("Type xyz to exit: ");
12                s = in.readLine();
13                if (s != null) s = s.trim();
14                System.out.println("Read: " + s);
15                if (s.equals ("xyz")) System.exit(0);
16            }
17        } catch (IOException e) {
18            System.out.println ("Exception: " + e);
19        }
20    }
```

Encadenar un lector en buffer a un flujo de entrada que toma la entrada de la consola.



E/S de canal

Introducidos en JDK 1.4, los canales leen bytes y caracteres en bloques, en lugar de un byte o un carácter cada vez.

```
1 import java.io.FileInputStream; import java.io.FileOutputStream;
2 import java.nio.channels.FileChannel; import java.nio ByteBuffer;
3 import java.io.FileNotFoundException; import java.io.IOException;
4
5 public class ByteChannelCopyTest {
6     public static void main(String[] args) {
7         try (FileChannel fcIn = new FileInputStream(args[0]).getChannel();
8              FileChannel fcOut = new FileOutputStream(args[1]).getChannel()) {
9             ByteBuffer buff = ByteBuffer.allocate((int) fcIn.size());
10            fcIn.read(buff);
11            buff.position(0);
12            fcOut.write(buff);
13        } catch (FileNotFoundException f) {
14            System.out.println("File not found: " + f);
15        } catch (IOException e) {
16            System.out.println("IOException: " + e);
17        }
18    }
19}
```

Crear un buffer con el mismo tamaño que el tamaño del archivo y después leer y escribir el archivo en una sola operación.



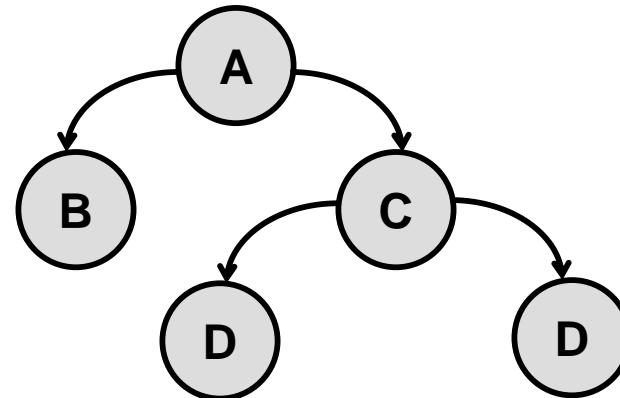
Persistencia

Guardar datos en algún tipo de almacenamiento permanente se conoce como "persistencia". Los objetos que pueden ser persistentes se pueden almacenar en un disco (o en otro dispositivo de almacenamiento), o se pueden enviar a otra máquina para almacenarse allí.

- Los objetos no persistentes solo existen mientras se esté ejecutando la máquina Java Virtual Machine.
- La serialización de Java es el mecanismo estándar para guardar un objeto como secuencia de bytes que después se puede reconstruir en una copia del objeto.
- Para serializar un objeto de una clase específica, la clase debe implantar la interfaz `java.io.Serializable`.

Serialización y gráficos de objetos

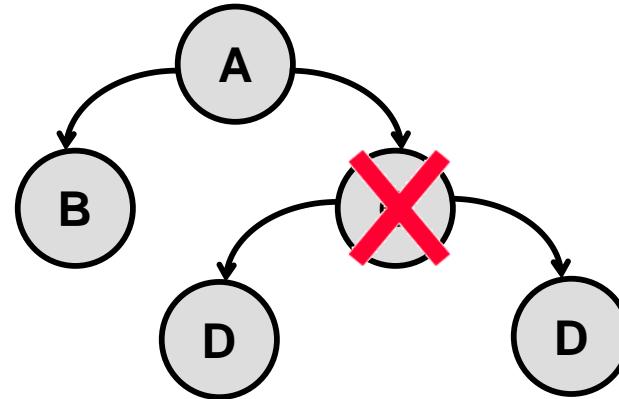
- Cuando se serializa un objeto, solo se conservan los campos del objeto.
- Cuando un campo hace referencia a un objeto, los campos del objeto al que se hace referencia se serializan también si la clase del objeto es igualmente serializable.
- El árbol de los campos de un objeto constituye el *gráfico del objeto*.





Campos y objetos transitorios

- Algunas clases de objetos no son serializables, ya que representan información transitoria específica del sistema operativo.
- Si el gráfico del objeto contiene una referencia no serializable, se devuelve NotSerializableException y la operación de serialización falla.
- Los campos que no deben serializarse o que no necesitan hacerlo, se pueden marcar con la palabra clave transient.





Transient: ejemplo

```
public class Portfolio implements Serializable {  
    public transient FileInputStream inputFile;  
    public static int BASE = 100;  
    private transient int totalValue = 10;  
    protected Stock[] stocks;  
}
```

Los campos static no se serializan.

La serialización incluirá todos los miembros de la matriz stocks.

- El modificador de acceso del campo no afecta a los datos que se vayan a serializar.
- Los valores almacenados en campos estáticos no se serializan.
- Cuando se anula la serialización del objeto, los valores de los campos estáticos se establecen en los valores declarados en la clase. El valor de los campos transitorios no estáticos se establece en el valor por defecto para el tipo.

UID de versión de serialización

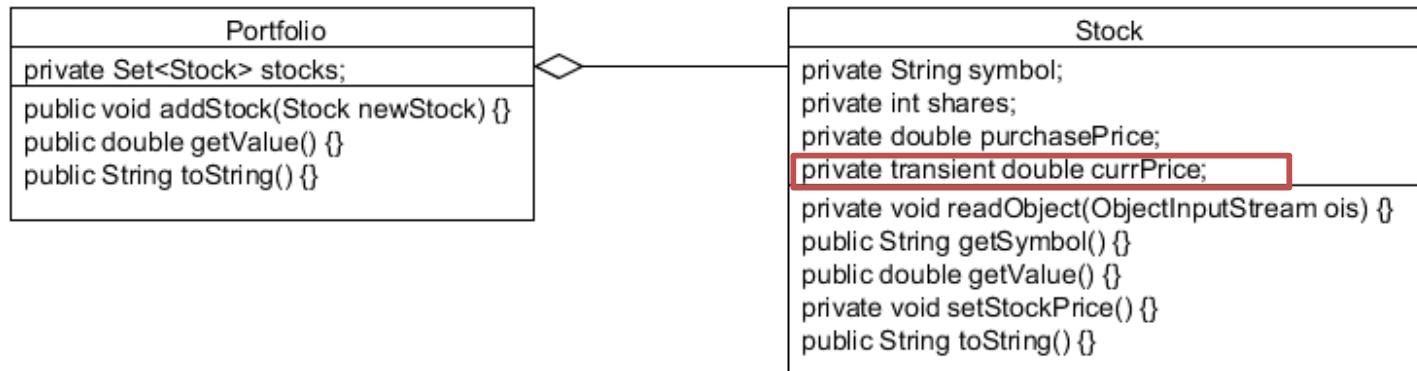
- Durante el proceso de serialización se usa un número de versión, serialVersionUID, para asociar la salida serializada con la clase empleada en el proceso de serialización.
- Al anular la serialización, se comprueba el valor de serialVersionUID para verificar que las clases cargadas son compatibles con el objeto cuya serialización se está anulando.
- Si el receptor de un objeto serializado ha cargado clases para ese objeto con serialVersionUID diferentes, la anulación de la serialización resultará en InvalidClassException.
- Una clase serializable puede declarar su propio serialVersionUID declarando de forma explícita un campo llamado serialVersionUID como final estático y tipo long:

```
private static long serialVersionUID = 42L;
```

Ejemplo de serialización

En este ejemplo hay una cartera de valores constituida por un juego de acciones.

- Durante la serialización, el precio actual no se serializa, por lo que se marca como transient.
- No obstante, queremos que el valor actual de las acciones se establezca en el valor de mercado actual al anular la serialización.





Escritura y lectura de un flujo de objetos

```
1  public static void main(String[] args) {  
2      Stock s1 = new Stock("ORCL", 100, 32.50);  
3      Stock s2 = new Stock("APPL", 100, 245);  
4      Stock s3 = new Stock("GOGL", 100, 54.67);  
5      Portfolio p = new Portfolio(s1, s2, s3);  
6      try (FileOutputStream fos = new FileOutputStream(args[0]));  
7          ObjectOutputStream out = new ObjectOutputStream(fos)) {  
8              out.writeObject(p);  
9      } catch (IOException i) {  
10          System.out.println("Exception writing out Portfolio: " + i);  
11      }  
12      try (FileInputStream fis = new FileInputStream(args[0]);  
13          ObjectInputStream in = new ObjectInputStream(fis)) {  
14          Portfolio newP = (Portfolio)in.readObject();  
15      } catch (ClassNotFoundException | IOException i) {  
16          System.out.println("Exception reading in Portfolio: " + i);  
17      }  
}
```

Portfolio es el objeto raíz.

El método writeObject escribe el gráfico de objeto de p en el flujo de archivo.

El método readObject restaura el objeto desde el flujo de archivo.



Métodos de serialización

Los objetos que se están serializando (o se está anulando su serialización) pueden controlar la serialización de sus propios campos.

```
public class MyClass implements Serializable {  
    // Fields  
    private void writeObject(ObjectOutputStream oos) throws IOException {  
        oos.defaultWriteObject();  
        // Write/save additional fields  
        oos.writeObject(new java.util.Date());  
    }  
}
```

Llamada a defaultWriteObject para serializar los campos de estas clases.

- Por ejemplo, en esta clase la hora actual se escribe en el gráfico del objeto.
- Durante la anulación de la serialización se invoca a un método similar:

```
private void readObject(ObjectInputStream ois) throws  
    ClassNotFoundException, IOException {}
```



Ejemplo de readObject

```
1 public class Stock implements Serializable {  
2     private static final long serialVersionUID = 100L;  
3     private String symbol;  
4     private int shares;  
5     private double purchasePrice;  
6     private transient double currPrice;  
7  
8     public Stock(String symbol, int shares, double purchasePrice) {  
9         this.symbol = symbol;  
10        this.shares = shares;  
11        this.purchasePrice = purchasePrice;  
12        setStockPrice();  
13    }  
14  
15    // This method is called post-serialization  
16    private void readObject(ObjectInputStream ois)  
17        throws IOException, ClassNotFoundException {  
18        ois.defaultReadObject();  
19        // perform other initialization  
20        setStockPrice();  
21    }  
22 }
```

El valor currPrice de las acciones lo establece el método setStockPrice al crear el objeto Stock, pero no se llama al constructor durante la anulación de la serialización.

El valor currPrice de las acciones se establece después de que se anule la serialización de los otros campos.



Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Describir los aspectos básicos de entrada y salida en Java
- Leer datos de la consola y escribir datos en ella
- Utilizar flujos para leer y escribir datos
- Leer y escribir objetos mediante serialización



Java File I/O (NIO.2)

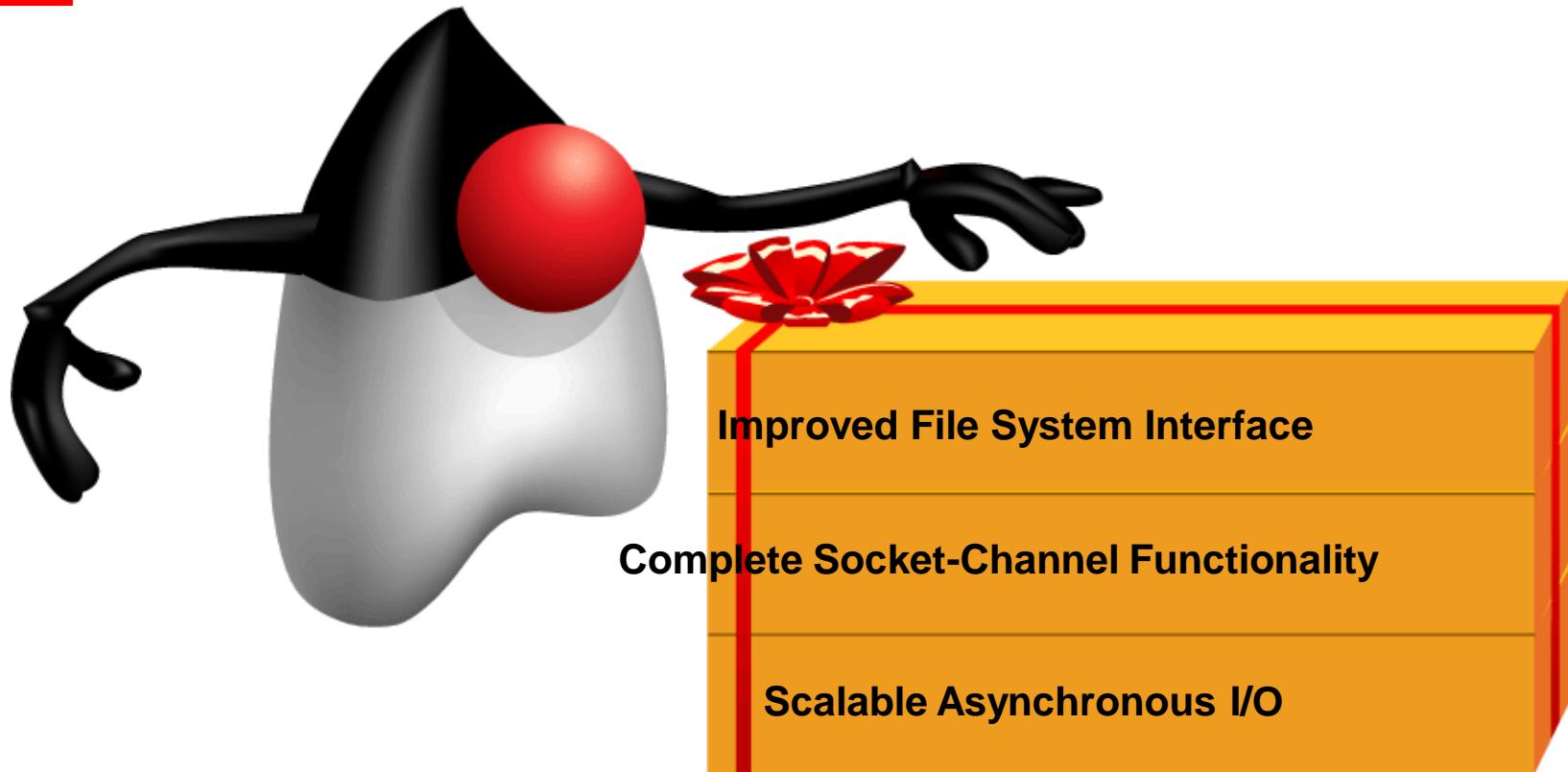


Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Utilizar la interfaz Path para realizar operaciones en archivos y en rutas de acceso a directorios
- Utilizar la clase Files para comprobar, suprimir, copiar o mover un archivo o un directorio
- Utilizar métodos de la clase Files para leer y escribir archivos mediante E/S de canales o E/S de flujos
- Leer y cambiar atributos de archivos y de directorios
- Acceder de forma recurrente al árbol de un directorio
- Localizar un archivo a través de la clase PathMatcher

Nueva API de E/S de archivos Java (NIO.2)





Limitaciones de java.io.File

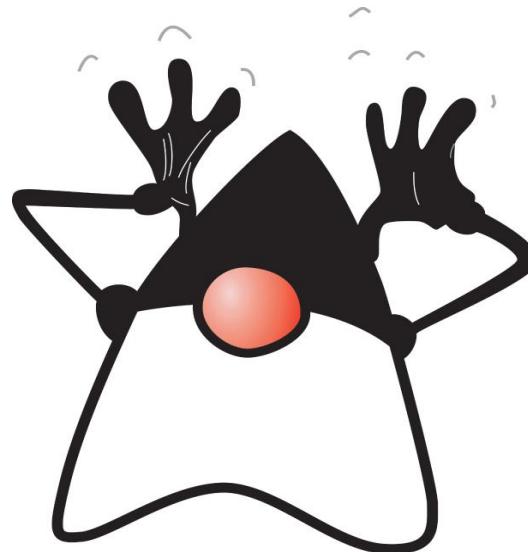
Does not work well with symbolic links



Scalability issues



Performance issues



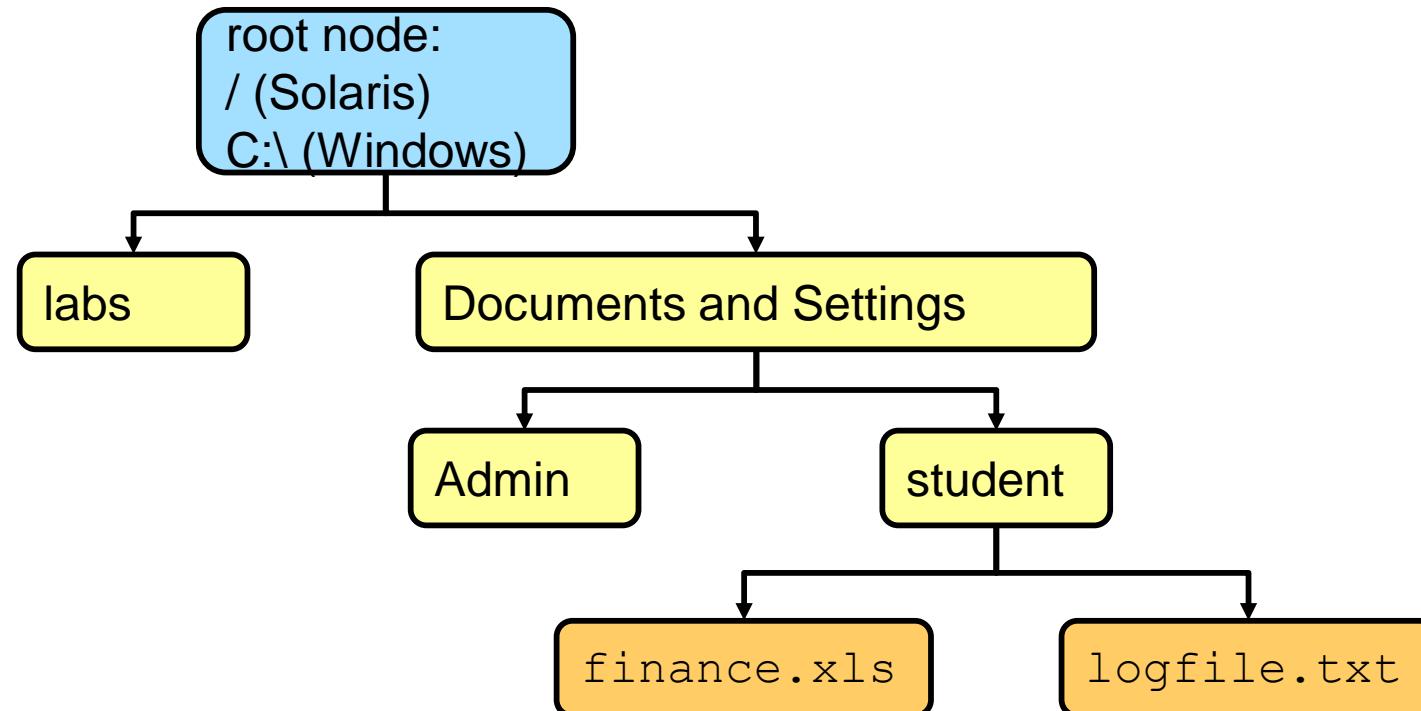
Very limited set of
file attributes



Very basic file system access functionality

Sistemas de archivos, rutas y archivos

En NIO.2, los archivos y los directorios se representan a través de una ruta, que es la ubicación relativa o absoluta del archivo o del directorio.





Ruta de acceso relativa frente a ruta de acceso absoluta

- Las rutas de acceso pueden ser *relativas* o *absolutas*.
- Las rutas de acceso absolutas contienen el elemento raíz y la lista completa del directorio para localizar el archivo.
- Ejemplo:

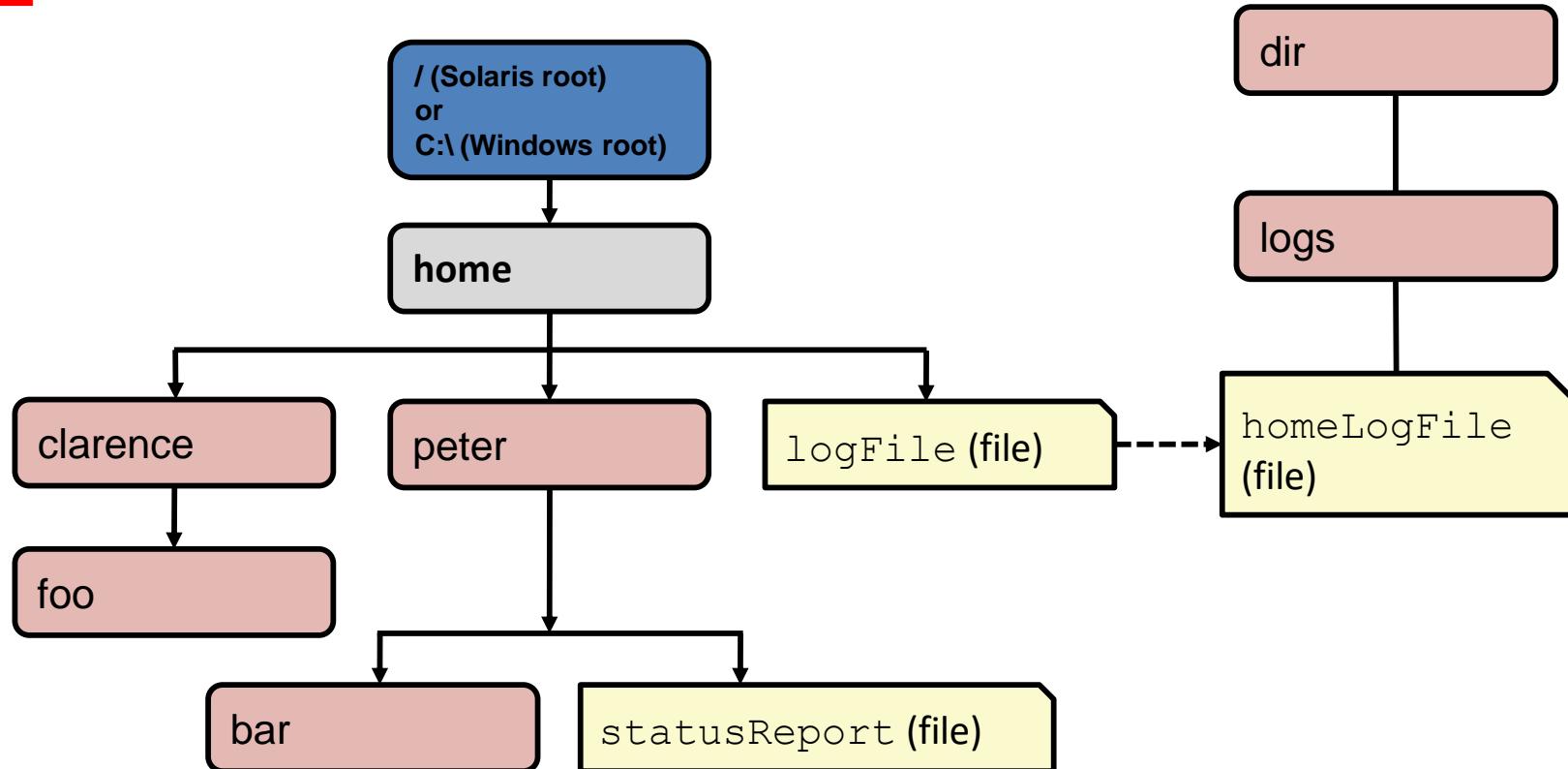
```
...
/home/peter/statusReport
...
```

- Una ruta de acceso relativa debe combinarse con otra ruta para poder acceder a un archivo.
- Ejemplo:

```
...
clarence/foo
...
```



Enlaces simbólicos





Conceptos de Java NIO.2

Antes de JDK 7, la clase `java.io.File` era el punto de entrada para todas las operaciones de archivo o de directorio. NIO.2 introduce un nuevo paquete y nuevas clases:

- `java.nio.file.Path`: localiza un archivo o un directorio mediante una ruta de acceso dependiente del sistema.
- `java.nio.file.Files`: realiza operaciones en archivos y en directorios a través de una interfaz `Path`.
- `java.nio.file.FileSystem`: proporciona una interfaz hacia un sistema de archivos y una fábrica para crear un objeto `Path` y otros objetos que acceden a un sistema de archivos.
- Todos los métodos que acceden al sistema de archivos devuelven `IOException` o una subclase.



Interfaz Path

La interfaz `java.nio.file.Path` proporciona el punto de entrada para manipular archivos y directorios en NIO.2.

- Para obtener un objeto `Path`, deberá obtener una instancia del sistema de archivos por defecto y después invocar al método `getPath`:

```
FileSystem fs = FileSystems.getDefault();
Path p1 = fs.getPath ("D:\\\\labs\\\\resources\\\\myFile.txt");
```

Barra diagonal inversa con escape

- El paquete `java.nio.file` proporciona también una clase helper final estática llamada `Paths` para realizar la acción `getDefault`:

```
Path p1 = Paths.get ("D:\\\\labs\\\\resources\\\\myFile.txt");
Path p2 = Paths.get ("D:", "labs", "resources", "myFile.txt");
Path p3 = Paths.get ("/temp/foo");
Path p4 = Paths.get (URI.create ("file:///~/somefile"));
```

Características de la interfaz Path

La interfaz Path define los métodos utilizados para localizar un archivo o un directorio en un sistema de archivos. Estos métodos incluyen:

- Para acceder a los componentes de una ruta:
 - `getFileName`, `getParent`, `getRoot` y `getNameCount`
- Para realizar operaciones en una ruta:
 - `normalize`, `toUri`, `toAbsolutePath`, `subpath`, `resolve` y `relativize`
- Para comparar rutas:
 - `startsWith`, `endsWith` y `equals`



Path: ejemplo

```
1 public class PathTest
2     public static void main(String[] args) {
3         Path p1 = Paths.get(args[0]);
4         System.out.format("getFileName: %s%n", p1.getFileName());
5         System.out.format("getParent: %s%n", p1.getParent());
6         System.out.format("getNameCount: %d%n", p1.getNameCount());
7         System.out.format("getRoot: %s%n", p1.getRoot());
8         System.out.format("isAbsolute: %b%n", p1.isAbsolute());
9         System.out.format("toAbsolutePath: %s%n", p1.toAbsolutePath());
10        System.out.format("toURI: %s%n", p1.toUri());
11    }
12 }
```

```
java PathTest D:/Temp/Foo/file1.txt
getFileName: file1.txt
getParent: D:\Temp\Foo
getNameCount: 3
getRoot: D:\
isAbsolute: true
toAbsolutePath: D:\Temp\Foo\file1.txt
toURI: file:///D:/Temp/Foo/file1.txt
```

Ejecutado en un equipo Windows.
Observe que, excepto en un shell cmd,
se pueden usar barras diagonales
normales e inversas.

Eliminación de redundancias de Path

- Muchos sistemas de archivos utilizan una notación “.” para denotar el directorio actual y “..” para denotar el directorio principal.
- Los dos ejemplos siguientes incluyen redundancias:

```
/home/./clarence/foo  
/home/peter/../clarence/foo
```

- El método `normalize` elimina los elementos redundantes, entre ellos todas las incidencias de “.” o de “directory/..”.
- Ejemplo:

```
Path p = Paths.get("/home/peter/../clarence/foo");  
Path normalizedPath = p.normalize();
```

```
/home/clarence/foo
```



Creación de una subruta

- Se puede obtener una parte de una ruta creando una subruta con el método subpath:

```
Path subpath(int beginIndex, int endIndex);
```

- El elemento que devuelve endIndex es uno menos que el valor endIndex.
- Ejemplo:

Temp = 0
foo = 1
bar = 2

```
Path p1 = Paths.get ("D:/Temp/foo/bar");  
Path p2 = p1.subpath (1, 3);
```

foo\bar

Incluir el elemento en el índice 2.



Unión de dos rutas

- El método resolve se utiliza para combinar dos rutas.
- Ejemplo:

```
Path p1 = Paths.get("/home/clarence/foo");
p1.resolve("bar");      // Returns /home/clarence/foo/bar
```

- Al pasar una ruta absoluta al método resolve se devuelve la ruta de acceso transferida.

```
Paths.get("foo").resolve("/home/clarence"); // Returns /home/clarence
```

Creación de una ruta entre dos rutas

- El método relativize le permite construir una ruta desde una ubicación del sistema de archivos a otra ubicación.
- El método construye una ruta que empieza en la ruta original y termina en la ubicación especificada por la ruta transferida.
- La nueva ruta es relativa a la ruta original.
- Ejemplo:

```
Path p1 = Paths.get("peter");
Path p2 = Paths.get("clarence");

Path p1Top2 = p1.relativize(p2);      // Result is ../clarence
Path p2Top1 = p2.relativize(p1);      // Result is ../peter
```



Trabajo con enlaces

- La interfaz Path reconoce enlaces.
- Todos los métodos Path:
 - detectan qué hacer al encontrarse un enlace simbólico; o
 - proporcionan una opción que permite configurar el comportamiento en caso de encontrarse un enlace simbólico.

```
createSymbolicLink(Path, Path, FileAttribute<?>)
```

Creating a symbolic link

```
createLink(Path, Path)
```

```
isSymbolicLink(Path)
```

```
readSymbolicLink(Path)
```

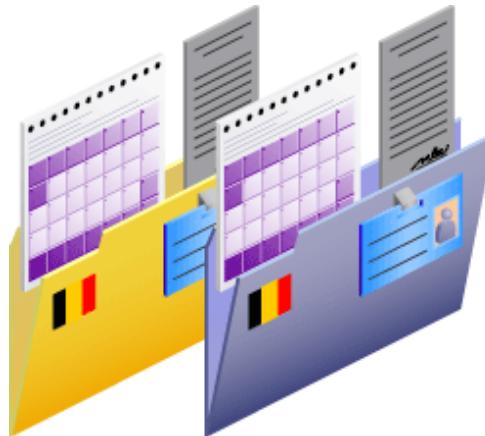
Creating a hard link

Detecting a symbolic link

Finding the target of a link



Operaciones File



Checking a File or Directory

Deleting a File or Directory

Copying a File or Directory

Moving a File or Directory

Managing Metadata

Reading, Writing, and Creating Files

Random Access Files

Creating and Reading Directories



Comprobación de un directorio o un archivo

Los objetos Path representan el concepto de una ubicación de archivo o de directorio. Para poder acceder al archivo o al directorio es necesario acceder primero al sistema de archivos y determinar si existe o no con los siguientes métodos Files:

- `exists(Path p, LinkOption... option)` Realiza pruebas para ver si existe un archivo. Por defecto, le siguen enlaces simbólicos.
- `notExists(Path p, LinkOption... option)` Realiza pruebas para ver si no existe un archivo. Por defecto, le siguen enlaces simbólicos.
- Ejemplo:

Argumento opcional

```
Path p = Paths.get(args[0]);
System.out.format("Path %s exists: %b%n", p,
                  Files.exists(p, LinkOption.NOFOLLOW_LINKS));
```



Comprobación de un directorio

Para verificar que se puede acceder a un archivo, la clase Files proporciona los siguientes métodos boolean.

- isReadable (Path)
- isWriteable (Path)
- isExecutable (Path)

Tenga en cuenta que estas pruebas no son atómicas con respecto al resto de las operaciones del sistema de archivos. Por lo tanto, es posible que los resultados de estas pruebas no resulten fiables una vez terminen los métodos.

- El método isSameFile (Path, Path) realiza pruebas para comprobar si hay dos rutas que apuntan al mismo archivo. Esto resulta especialmente útil en sistemas de archivos que soportan enlaces simbólicos.

Creación de archivos y directorios

Se pueden crear archivos y directorios con uno de los siguientes métodos:

```
Files.createFile (Path dir);  
Files.createDirectory (Path dir);
```

- El método `createDirectories` se puede utilizar para crear directorios que no existen, de arriba abajo:

```
Files.createDirectories(Paths.get("D:/Temp/foo/bar/example"));
```



Supresión de un directorio o un archivo

Puede suprimir archivos, directorios o enlaces. La clase Files proporciona dos métodos:

- delete(Path)
- deleteIfExists(Path)

```
//...  
Files.delete(path);  
//...
```

Throws a NoSuchFileException,
DirectoryNotEmptyException,
or
IOException

```
//...  
Files.deleteIfExists(Path);  
//...
```

No exception thrown

Copie de un directorio o un archivo

- Puede copiar un archivo o un directorio mediante el método copy(Path, Path, CopyOption...).
- Cuando se copian directorios, los archivos que incluyen no se copian.

StandardCopyOption parameters

```
//...
copy(Path, Path, CopyOption...)
//...
```

REPLACE_EXISTING
COPY_ATTRIBUTES
NOFOLLOW_LINKS

- Ejemplo:

```
import static java.nio.file.StandardCopyOption.*;
//...
Files.copy(source, target, REPLACE_EXISTING, NOFOLLOW_LINKS);
```

Copia entre un flujo y una ruta

También puede interesarle poder copiar (o escribir) desde un objeto Stream a un archivo o desde un archivo a un objeto Stream. La clase Files proporciona dos métodos que facilitan esta tarea:

```
copy(InputStream source, Path target, CopyOption... options)
copy(Path source, OutputStream out)
```

- Un uso interesante de este primer método es copiar desde una página web y guardar en un archivo:

```
Path path = Paths.get("D:/Temp/oracle.html");
URI u = URI.create("http://www.oracle.com/");
try (InputStream in = u.toURL().openStream()) {
    Files.copy(in, path, StandardCopyOption.REPLACE_EXISTING);
} catch (final MalformedURLException | IOException e) {
    System.out.println("Exception: " + e);
}
```

Desplazamiento de un directorio o un archivo

- Puede desplazar un archivo o un directorio mediante el método move(Path, Path, CopyOption...).
- Al desplazar un directorio, no se desplazará su contenido.

StandardCopyOption parameters

```
//...  
move(Path, Path, CopyOption...)  
//...
```

REPLACE_EXISTING
ATOMIC_MOVE

- Ejemplo:

```
import static java.nio.file.StandardCopyOption.*;  
//...  
Files.move(source, target, REPLACE_EXISTING);
```

Listado del contenido de un directorio

La clase `DirectoryStream` proporciona un mecanismo para iterar sobre todas las entradas de un directorio.

```
1 Path dir = Paths.get("D:/Temp");
2 // DirectoryStream is a stream, so use try-with-resources
3 // or explicitly close it when finished
4 try (DirectoryStream<Path> stream =
5         Files.newDirectoryStream(dir, "*.zip")) {
6     for (Path file : stream) {
7         System.out.println(file.getFileName());
8     }
9 } catch (PatternSyntaxException | DirectoryIteratorException |
10        IOException x) {
11     System.err.println(x);
12 }
```

- `DirectoryStream` escala para soportar directorios de gran tamaño.



Lectura o escritura de todos los bytes o líneas de un archivo

- Los métodos readAllBytes o readAllLines leen el contenido completo de un archivo en una sola transferencia.
- Ejemplo:

```
Path source = ...;
List<String> lines;
Charset cs = Charset.defaultCharset();
lines = Files.readAllLines(file, cs);
```

- Utilice métodos write para escribir bytes o líneas en un archivo.

```
Path target = ...;
Files.write(target, lines, cs, CREATE, TRUNCATE_EXISTING, WRITE);
```

Enumeraciones StandardOpenOption.

Canales y ByteBuffers

- La E/S de flujo lee un carácter cada vez, mientras que la E/S de canal lee un buffer cada vez.
- La interfaz ByteChannel proporciona una funcionalidad de lectura y escritura básica.
- Un objeto SeekableByteChannel es un objeto ByteChannel que tiene la capacidad de mantener una posición en el canal y cambiarla.
- Los dos métodos para escribir y leer E/S de canal son:

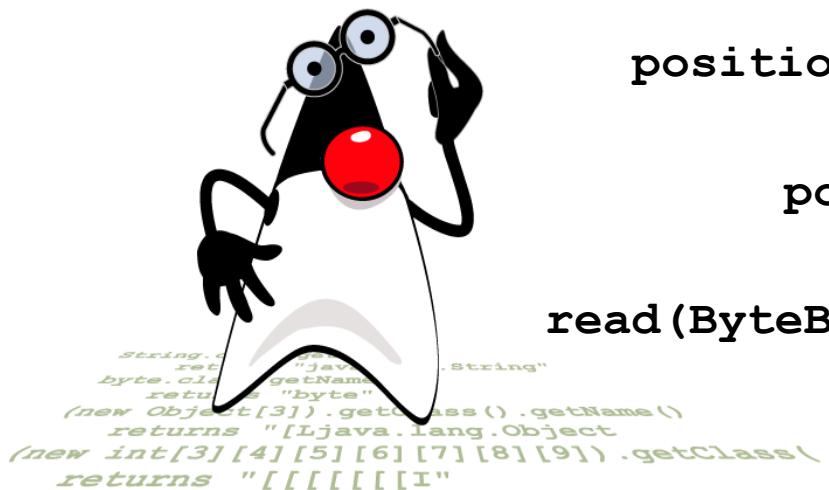
```
newByteChannel(Path, OpenOption...)
newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?>...)
```

- La capacidad de realizar desplazamientos a diferentes puntos en un archivo y después leer o escribir desde esa ubicación hace posible un acceso aleatorio a un archivo.



Archivos de acceso aleatorio

- Los archivos de acceso aleatorio permiten acceder aleatoriamente y no secuencialmente al contenido de un archivo.
- Para acceder a un archivo aleatoriamente, abra el archivo, busque una ubicación concreta y lea desde el archivo o escriba en él.
- La funcionalidad de acceso aleatorio se activa mediante la interfaz SeekableByteChannel.



position ()

write (ByteBuffer)

position (long)

read (ByteBuffer)

truncate (long)

Métodos de E/S en buffer para archivos de texto

- El método newBufferedReader abre un archivo para su lectura.

```
//...
BufferedReader reader = Files.newBufferedReader(file, charset);
line = reader.readLine();
```

- El método newBufferedWriter escribe en un archivo a través de un objeto BufferedWriter.

```
//...
BufferedWriter writer = Files.newBufferedWriter(file, charset);
writer.write(s, 0, s.length());
```



Flujos de bytes

- NIO.2 soporta también métodos para abrir flujos de bytes.

```
InputStream in = Files.newInputStream(file);
BufferedReader reader = new BufferedReader(new InputStreamReader(in));
line = reader.readLine();
```

- Para crear, agregar a un archivo o escribir en él, utilice el método newOutputStream.

```
import static java.nio.file.StandardOpenOption.*;
//...
Path logfile = ...;
String s = ...;
byte data[] = s.getBytes();
OutputStream out =
        new BufferedOutputStream(file.newOutputStream(CREATE, APPEND));
out.write(data, 0, data.length);
```



Gestión de metadatos

Método	Explicación
size	Devuelve el tamaño del archivo especificado en bytes.
isDirectory	Devuelve true si el objeto Path especificado localiza un archivo que es un directorio.
isRegularFile	Devuelve true si el objeto Path especificado localiza un archivo que es un archivo normal.
isSymbolicLink	Devuelve true si el objeto Path especificado localiza un archivo que es un enlace simbólico.
isHidden	Devuelve true si el objeto Path especificado localiza un archivo considerado oculto por el sistema de archivos.
getLastModifiedTime	Devuelve o establece la fecha de última modificación setLastModifiedTime del archivo especificado.
setLastModifiedTime	
getAttribute	Devuelve o establece el valor de un atributo de archivo.
setAttribute	

Atributos de archivo (DOS)

- Los atributos de archivo se pueden leer desde un archivo o un directorio en una sola llamada:

```
DosFileAttributes attrs =  
    Files.readAttributes (path, DosFileAttributes.class);
```

- Los sistemas de archivos DOS pueden modificar atributos después de crearse los archivos:

```
Files.createFile (file);  
Files.setAttribute (file, "dos:hidden", true);
```



Atributos de archivo DOS: ejemplo

```
DosFileAttributes attrs = null;
Path file = ....;
try { attrs =
            Files.readAttributes(file, DosFileAttributes.class);
} catch (IOException e) { //... }
FileTime creation = attrs.creationTime();
FileTime modified = attrs.lastModifiedTime();
FileTime lastAccess = attrs.lastAccessTime();
if (!attrs.isDirectory()) {
    long size = attrs.size();
}
// DosFileAttributes adds these to BasicFileAttributes
boolean archive = attrs.isArchive();
boolean hidden = attrs.isHidden();
boolean readOnly = attrs.isReadOnly();
boolean systemFile = attrs.isSystem();
```



Permisos de POSIX

NIO.2 permite crear archivos y directorios en sistemas de archivos POSIX con sus primeros permisos establecidos.

```
1 Path p = Paths.get(args[0]);  
2 Set<PosixFilePermission> perms =  
3     PosixFilePermissions.fromString("rwxr-x---");  
4 FileAttribute<Set<PosixFilePermission>> attrs =  
5     PosixFilePermissions.asFileAttribute(perms);  
6 try {  
7     Files.createFile(p, attrs);  
8 } catch (FileAlreadyExistsException f) {  
9     System.out.println("FileAlreadyExists" + f);  
10 } catch (IOException i) {  
11     System.out.println("IOException:" + i);  
12 }
```

Crear un archivo en Path p con atributos opcionales.



Operaciones recursivas

La clase Files ofrece un método para recorrer el árbol de archivos en busca de operaciones recursivas, como de copia y de supresión.

- `walkFileTree (Path start, FileVisitor<T>)`
- Ejemplo:

```
public class PrintTree implements FileVisitor<Path> {  
    public FileVisitResult preVisitDirectory(Path, BasicFileAttributes) {}  
    public FileVisitResult postVisitDirectory(Path, BasicFileAttributes) {}  
    public FileVisitResult visitFile(Path, BasicFileAttributes) {}  
    public FileVisitResult visitFileFailed(Path, BasicFileAttributes) {}  
}
```

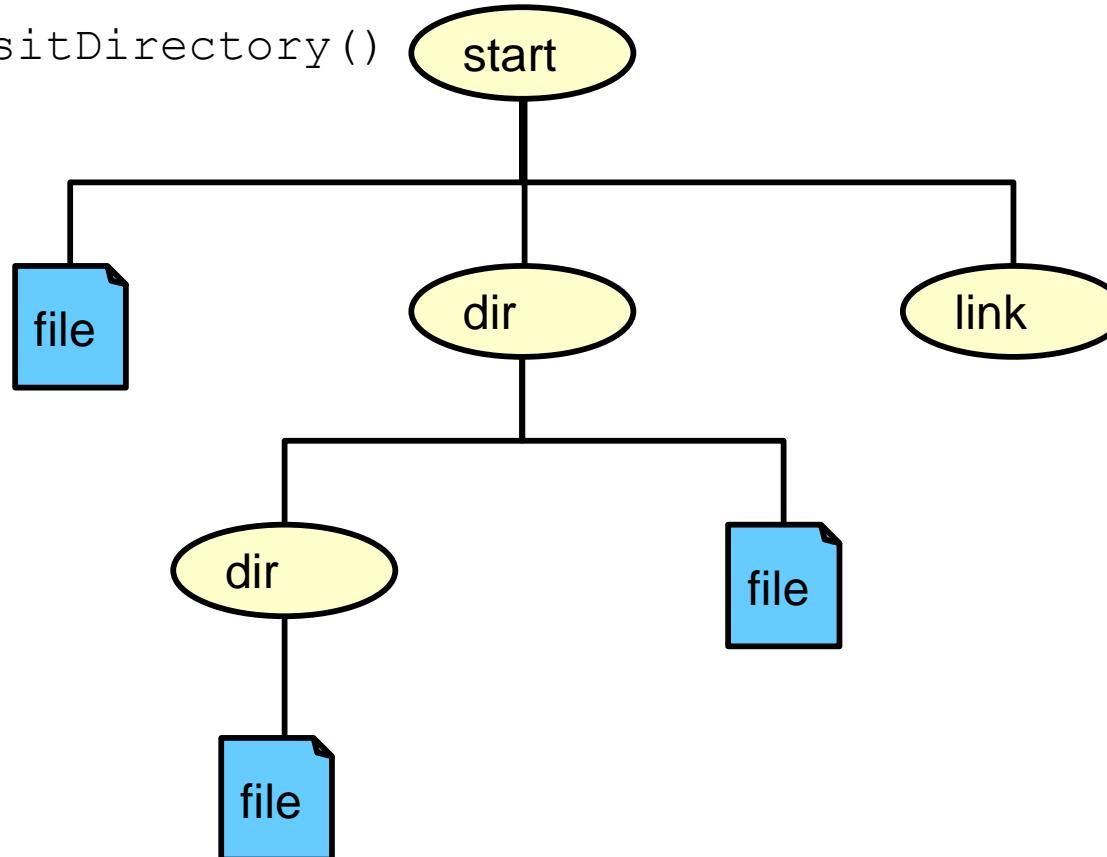
```
public class WalkFileTreeExample {  
    public printFileTree(Path p) {  
        Files.walkFileTree(p, new PrintTree());  
    }  
}
```

El árbol de archivos se explora de forma recurrente. Se llama como directorios a los métodos definidos por PrintTree y se llega a los archivos en el árbol. Cada método se transfiere a la ruta actual como el primer argumento del método.



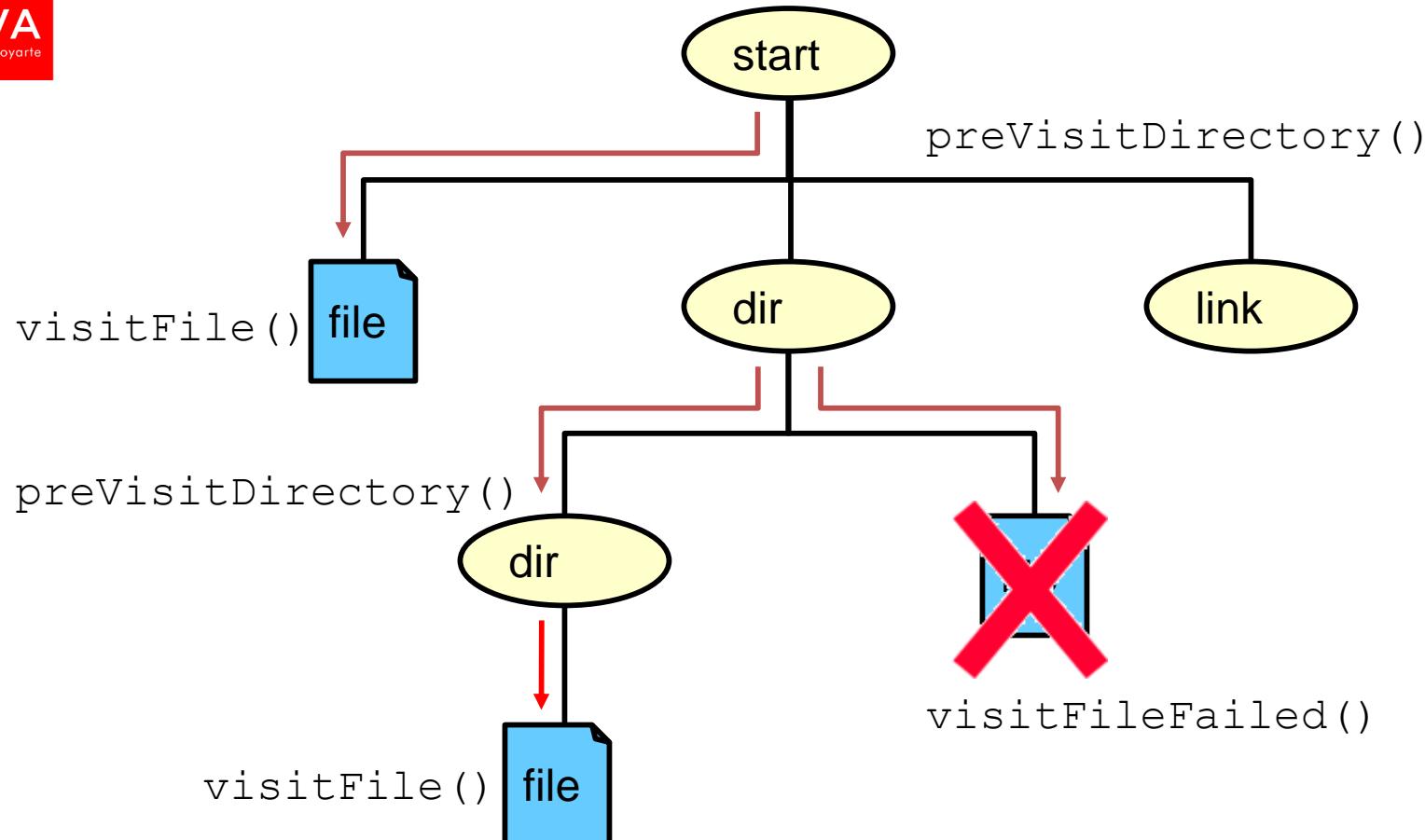
Orden del método FileVisitor

preVisitDirectory()



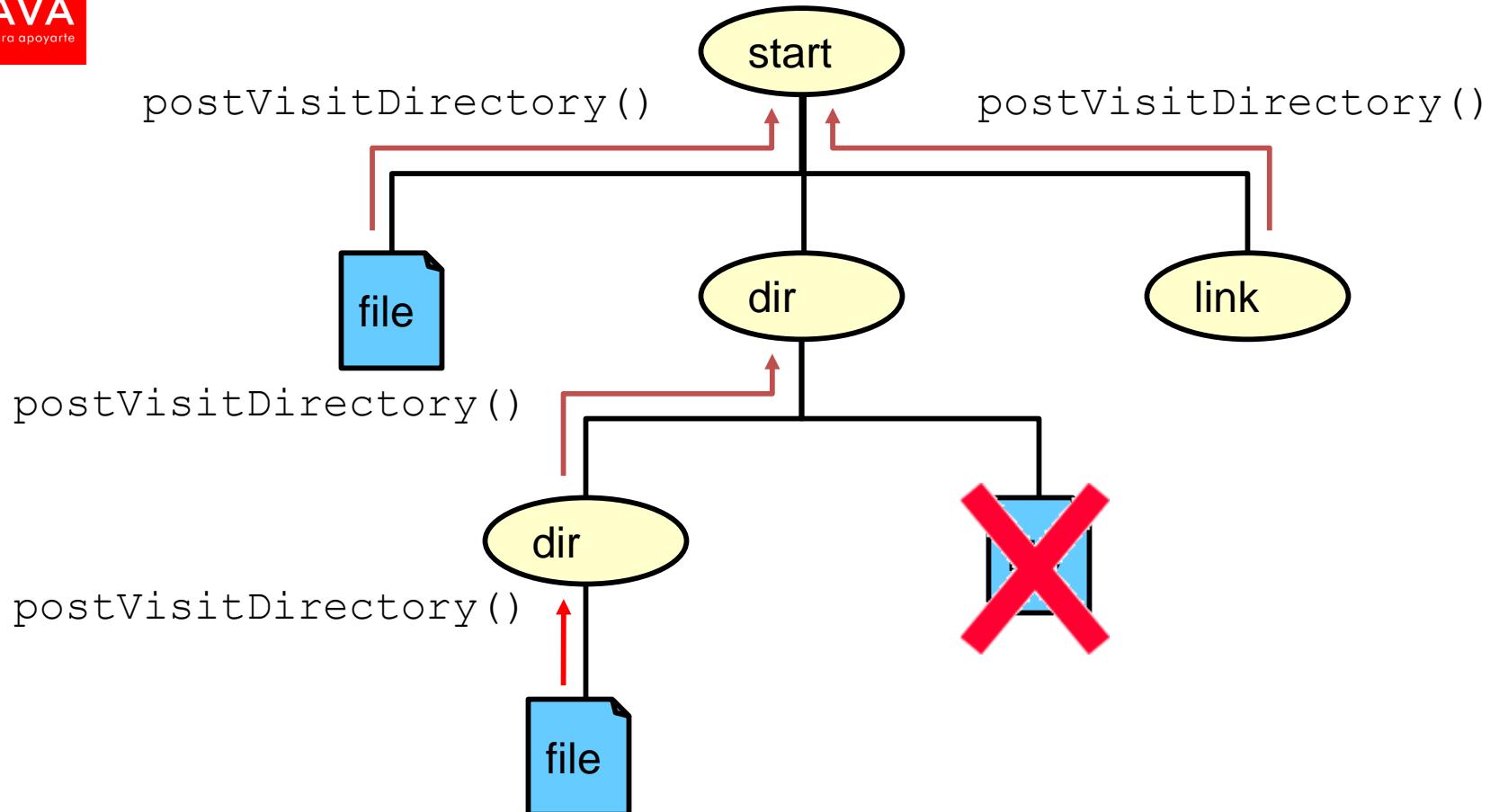


Orden del método FileVisitor





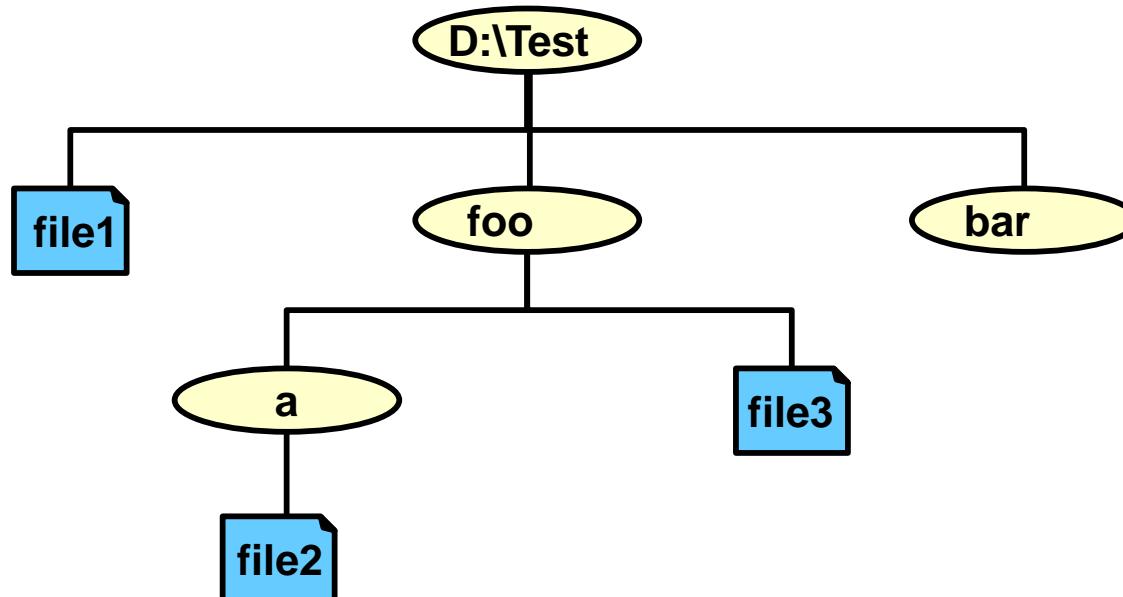
Orden del método FileVisitor





Ejemplo: WalkFile TreeExample

```
Path path = Paths.get("D:/Test");
try {
    Files.walkFileTree(path, new PrintTree());
} catch (IOException e) {
    System.out.println("Exception: " + e);
}
```





Búsqueda de archivos

Para localizar un archivo, generalmente se examina un directorio. Se podría usar una herramienta de búsqueda, o un comando como:

```
dir /s *.java
```

- Este comando examinará de forma recurrente el árbol del directorio, empezando por donde se encuentre y mirando todos los archivos que contengan la extensión .java. La interfaz java.nio.file.PathMatcher incluye un método de coincidencia para determinar si un objeto Path coincide con una cadena de búsqueda especificada.
- Cada implantación de sistema de archivos proporciona un objeto PathMatcher recuperable mediante la fábrica FileSystems:

```
PathMatcher matcher = FileSystems.getDefault().getPathMatcher  
(String syntaxAndPattern);
```



Patrón y sintaxis de PathMatcher

- La cadena syntaxAndPattern presenta la siguiente forma: *sintaxis:patrón*. Donde *sintaxis* puede ser “glob” y “regex”.
- La sintaxis glob es parecida a las expresiones regulares, pero más simple:

Ejemplo de patrón	Coincidencias
* .java	Una ruta que representa el nombre de archivo que termina en .java.
* .*	Encuentra coincidencias de nombres de archivos que tienen un punto.
* .{java, class}	Encuentra coincidencias de nombres de archivos que terminan en .java o en .class.
foo .?	Encuentra coincidencias de nombres de archivos que empiezan por foo. y tienen una extensión de un único carácter.
C: \\ *	Encuentra coincidencias de C:\foo y C:\bar en la plataforma de Windows (observe cómo la barra diagonal inversa tiene carácter de escape. Como un literal de cadena en el lenguaje Java, el patrón sería C:*.).



PathMatcher: ejemplo

```
1 public static void main(String[] args) {  
2     // ... check for two arguments  
3     Path root = Paths.get(args[0]);  
4     // ... check that the first argument is a directory  
5     PathMatcher matcher =  
6         FileSystems.getDefault().getPathMatcher("glob:" +  
args[1]);  
7     // Finder is class that implements FileVisitor  
8     Finder finder = new Finder(root, matcher);  
9     try {  
10         Files.walkFileTree(root, finder);  
11     } catch (IOException e) {  
12         System.out.println("Exception: " + e);  
13     }  
14     finder.done();  
15 }
```



Clase Finder

```
1 public class Finder extends SimpleFileVisitor<Path> {  
2     private Path file;  
3     private PathMatcher matcher;  
4     private int numMatches;  
5     // ... constructor stores Path and PathMatcher objects  
6     private void find(Path file) {  
7         Path name = file.getFileName();  
8         if (name != null && matcher.matches(name)) {  
9             numMatches++;  
10            System.out.println(file);  
11        }  
12    }  
13    @Override  
14    public FileVisitResult visitFile(Path file,  
15                                     BasicFileAttributes attrs) {  
16        find(file);  
17        return CONTINUE;  
18    }  
19    //...  
20 }
```

Otras clases útiles de NIO.2

- La clase FileStore es útil para proporcionar información de uso sobre el sistema de archivos, como el total de espacio en disco utilizable y asignado.

Filesystem	kbytes	used	avail
System (C:)	209748988	72247420	137501568
Data (D:)	81847292	429488	81417804

- Se puede usar una instancia de la interfaz WatchService para informar sobre cambios en los objetos Path registrados. WatchService se puede usar para identificar el momento de adición de los archivos a un directorio, así como su supresión o modificación.

```
ENTRY_CREATE: D:\test\New Text Document.txt
ENTRY_CREATE: D:\test\Foo.txt
ENTRY MODIFY: D:\test\Foo.txt
ENTRY MODIFY: D:\test\Foo.txt
ENTRY_DELETE: D:\test\Foo.txt
```



Cambio a NIO.2

Se ha agregado un método a la clase `java.io.File` para que JDK 7 pueda proporcionar compatibilidad con versiones posteriores de NIO.2.

```
Path path = file.toPath();
```

- Esto le permite aprovechar NIO.2 sin tener que reescribir una gran cantidad de código.
- Además, también puede sustituir el código existente para mejorar el mantenimiento futuro. Por ejemplo, puede sustituir `file.delete()`; por:

```
Path path = file.toPath();
Files.delete (path);
```

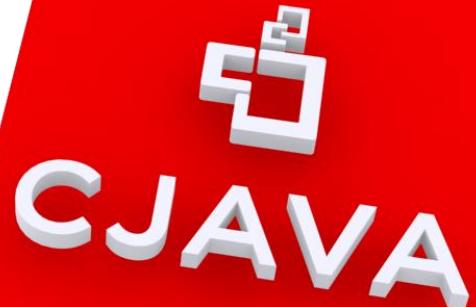
- Por el contrario, la interfaz `Path` proporciona un método para construir un objeto `java.io.File`:

```
File file = path.toFile();
```



Resumen

- En esta lección, debe haber aprendido a hacer lo siguiente:
- Utilizar la interfaz Path para realizar operaciones en archivos y en rutas de acceso a directorios
- Utilizar la clase Files para comprobar, suprimir, copiar o mover un archivo o un directorio
- Utilizar métodos de la clase Files para leer y escribir archivos mediante E/S de canales o E/S de flujos
- Leer y cambiar atributos de archivos y de directorios
- Acceder de forma recurrente al árbol de un directorio
- Localizar un archivo a través de la clase PathMatcher



Contáctenos

- Av. Arenales 395 oficina 405
Santa Beatriz - Lima 01 - Perú
- Teléfono: 433-6948
- RPC / WhatsApp: 932 656 459
- Email: info@cjavaperu.com

Síguenos

- [/cjava.peru.1](https://www.facebook.com/cjava.peru.1)
- [@cjava_peru](https://twitter.com/cjava_peru)
- [/cjavaperu](https://www.linkedin.com/company/cjavaperu/)



CJAVA

siempre para apoyarte

#CJavaNoPara

Gracias