



# Java y Herramientas de Desarrollo

## Sesión 6: Flujos de E/S y Red



## Puntos a tratar

- Flujos de E/S
- Entrada y salida estándar
- Acceso a ficheros
- Codificación de datos
- Serialización de objetos
- Acceso a la red
- Conexión con URLs
- Petición y respuesta



# Flujos de E/S

- Las aplicaciones muchas veces necesitan enviar datos a un determinado destino o leerlos de una determinada fuente
  - Ficheros en disco, red, memoria, otras aplicaciones, etc
  - Esto es lo que se conoce como E/S
- Esta E/S en Java se hace mediante flujos (*streams*)
  - Los datos se envían en serie a través del flujo
  - Se puede trabajar de la misma forma con todos los flujos, independientemente de su fuente o destino
    - Todos derivan de las mismas clases



# Tipos de flujos según el tipo de datos

- Según el tipo de datos que transportan, distinguimos
  - Flujos de bytes (con sufijos `InputStream` y `OutputStream`)
  - Flujos de caracteres (con sufijos `Reader` y `Writer`)
- Superclases

	Entrada	Salida
Bytes	<code>InputStream</code>	<code>OutputStream</code>
Caracteres	<code>Reader</code>	<code>Writer</code>



# Tipos de flujos según su propósito

- Distinguimos:

- Canales de datos

Simplemente llevan datos de una fuente a un destino

Ficheros: `FileInputStream`, `FileReader`,  
`FileOutputStream`, `FileWriter`

Memoria: `ByteArrayInputStream`, `CharArrayReader`, ...

Tuberías: `PipedInputStream`, `PipedReader`, `PipedWriter`, ...

- Flujos de procesamiento

Realizan algún procesamiento con los datos

Impresión: `PrintWriter`, `PrintStream`

Conversores de datos: `DataOutputStream`, `DataInputStream`

Bufferes: `BufferedReader`, `BufferedInputStream`, ...



# Acceso a los flujos

- Todos los flujos tienen una serie de métodos básicos

Flujos	Métodos
<code>InputStream, Reader</code>	<code>read, reset, close</code>
<code>OutputStream, Writer</code>	<code>write, flush, close</code>

- Los flujos de procesamiento
  - Se construyen a partir de flujos canales de datos
  - Los extienden proporcionando métodos de más alto nivel, p.ej:

Flujos	Métodos
<code>BufferedReader</code>	<code>readLine</code>
<code>DataOutputStream</code>	<code>writeInt, writeUTF, ...</code>
<code>PrintStream, PrintWriter</code>	<code>print, println</code>



## Objetos de la E/S estándar

- En Java también podemos acceder a la entrada, salida y salida de error estándar
- Accedemos a esta E/S mediante flujos
- Estos flujos se encuentran como propiedades estáticas de la clase `System`

	Tipo de flujo	Propiedad
Entrada	<code>InputStream</code>	<code>System.in</code>
Salida	<code>PrintStream</code>	<code>System.out</code>
Salida de error	<code>PrintStream</code>	<code>System.err</code>



# Salida estándar

- La salida estándar se ofrece como flujo de procesamiento `PrintStream`
  - Con un `OutputStream` a bajo nivel sería demasiado incómoda la escritura
- Este flujo ofrece los métodos `print` y `println` que permiten imprimir cualquier tipo de datos básico

- En la salida estándar

```
System.out.println("Hola mundo");
```

- En la salida de error

```
System.err.println("Error");
```





# Flujos de ficheros

- Canales de datos para acceder a ficheros

	Entrada	Salida
Caracteres	<code>FileReader</code>	<code>FileWriter</code>
Binarios	<code>FileInputStream</code>	<code>FileOutputStream</code>

- Se puede acceder a bajo nivel directamente de la misma forma que para cualquier flujo
- Podemos construir sobre ellos flujos de procesamiento para facilitar el acceso de estos flujos



# Lectura y escritura de ficheros

```
public void copia_fichero() {
    int c;
    try {
        FileReader in = new FileReader("fuente.txt");
        FileWriter out = new FileWriter("destino.txt");
        while( (c = in.read()) != -1)
        {
            out.write(c);
        }
        in.close();
        out.close();
    } catch(FileNotFoundException e1) {
        System.err.println("Error: No se encuentra el fichero");
    } catch(IOException e2) {
        System.err.println("Error leyendo/escribiendo fichero");
    }
}
```



# Uso de flujos de procesamiento

```
public void escribe_fichero() {
    FileWriter out = null;
    PrintWriter p_out = null;
    try {
        out = new FileWriter("result.txt");
        p_out = new PrintWriter(out);
        p_out.println("Este texto será escrito en el fichero");
    } catch(IOException e) {
        System.err.println("Error al escribir en el fichero");
    } finally {
        p_out.close();
    }
}
```



# Sistema de ficheros

- La clase `File` contiene utilidades para trabajar con el sistema de ficheros
  - Constantes para indicar los separadores de directorios ('/' ó '\\')
  - Hace las aplicaciones independientes de la plataforma
  - Crear, borrar o renombrar ficheros y directorios
  - Listar los ficheros de un directorio
  - Comprobar y establecer los permisos sobre ficheros
  - Obtener la ruta de un fichero
  - Obtener datos sobre ficheros (tamaño, fecha, etc)
  - Etc...



## Acceso a recursos

- Los recursos incluidos en un JAR no se encuentran directamente en el sistema de ficheros
  - No podremos utilizar los objetos anteriores para acceder a ellos
- Accedemos a un recurso en el JAR con

```
getClass().getResourceAsStream("/datos.txt");
```

- Anteponiendo '/' se busca de forma relativa al raíz del JAR
- Si no, buscará de forma relativa al directorio correspondiente al paquete de la clase actual



# Codificación

- Podemos codificar de forma sencilla los datos para enviarlos a través de un flujo de bytes (en serie)
- Utilizaremos un flujo `DataOutputStream`

```
String nombre = "Jose";
int edad = 25;

ByteArrayOutputStream baos = new ByteArrayOutputStream();

DataOutputStream dos = new OutputStream(baos);
dos.writeUTF(nombre);
dos.writeInt(edad);

dos.close();
baos.close();

byte [] datos = baos.toByteArray();
```



# Descodificación

- Para descodificar estos datos del flujo realizaremos el proceso inverso
- Utilizamos un flujo `DataInputStream`

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);
DataInputStream dis = new DataInputStream(bais);

String nombre = dis.readUTF();
int edad = dis.readInt();

dis.close();
bais.close();
```



# Entrada/Salida de objetos

- Si queremos enviar un objeto a través de un flujo deberemos convertirlo a una secuencia de bytes
- Esto es lo que se conoce como *serialización*
- Java serializa automáticamente los objetos
  - Obtiene una codificación del objeto en forma de array de bytes
  - En este array se almacenarán los valores actuales de todos los campos del objeto serializado





# Objetos serializables

- Para que un objeto sea serializable debe cumplir:

1. Implementar la interfaz `Serializable`

```
public MiClase implements Serializable {  
    ...  
}
```

Esta interfaz no obliga a definir ningún método, sólo marca el objeto como serializable

2. Todos los campos deben ser

Datos elementales o  
Objetos serializables



# Flujos de objetos

- Para enviar o recibir objetos tendremos los flujos de procesamiento

`ObjectInputStream`

`ObjectOutputStream`

- Estos flujos proporcionan respectivamente los métodos

`readObject`

`writeObject`

- Con los que escribir o leer objetos del flujo
  - Utilizan la serialización de Java para codificarlos y descodificarlos



# Métodos de acceso a la red

- En Java podemos comunicarnos con máquinas remotas de diferentes formas
  - Mediante sockets
    - Bajo nivel
    - Problemas con firewalls intermedios
    - No adecuado para aplicaciones web
  - Acceso a URLs
    - Intercambia contenido utilizando protocolos estándar (p.ej. HTTP)
    - Java ofrece facilidades para trabajar con estos protocolos
    - No es necesario implementar los protocolos manualmente
    - Amigable con firewalls



# Acceso a alto nivel

- Encontramos también métodos de acceso remoto de alto nivel
  - Objetos distribuidos RMI/CORBA
    - Invocación de métodos remotos
    - Problemas con firewalls
    - RMI sólo accede a objetos Java
  - Servicios web
    - Permite invocar operaciones remotas
    - Protocolos web estándar, amigable con firewalls
    - Independiente del lenguaje y de la plataforma



# URLs

- URL = *Uniform Resource Locator*
  - Cadena para localizar los recursos en Internet
- Se compone de

```
protocolo://servidor[:puerto]/recurso
```

- P.ej. `http://www.ua.es/es/index.html`
  - Se conecta al servidor `www.ua.es`
  - A través del puerto por defecto (puerto 80)
  - Utilizando protocolo `HTTP` para comunicarse
  - Solicita el recurso `/es/index.html`



# URLs en Java

- Se encapsulan en la clase `URL`

```
URL url = new URL("http://www.ua.es/es/index.html");
```

- Es obligatorio especificar el protocolo
  - P.ej. `www.ua.es` es una URL mal formada
- Si la URL está mal formada se producirá una excepción `MalformedURLException`

```
try {  
    URL url = new URL("http://www.ua.es/es/index.html");  
} catch(MalformedURLException e) {  
    System.err.println("Error: URL mal construida");  
}
```



## Lectura del contenido

- Podemos leer el contenido de la URL abriendo un flujo de entrada con

```
InputStream in = url.openStream();
```

- Leeremos de este flujo de la misma forma que con cualquier otro flujo
  - Con los métodos a bajo nivel (byte a byte)
  - O utilizando un flujo de procesamiento
- P.ej, si la URL corresponde a un documento HTML obtendremos el código fuente de este documento



## Conexión con una URL

- Para poder tanto enviar como recibir datos debemos abrir una conexión con la URL

```
URLConnection con = url.openConnection();
```

- Creará un tipo de conexión adecuado para la URL a la que accedemos
  - P.ej, si la URL es `http://www.ua.es` creará una conexión de tipo `HttpURLConnection`
- Si vamos a enviar datos, activaremos la salida

```
con.setDoOutput(true);
```





# Estados de la conexión

- Configuración
  - Se encuentra en este estado al crearla
  - No se ha establecido la conexión
  - Podemos configurar los parámetros de la conexión
- Conectado
  - Se ha establecido la conexión
  - Podemos interactuar con el recurso al que accedemos
  - Se pasa a este estado cuando intentamos acceder a información sobre el recurso
- Cerrado
  - Se ha cerrado la conexión



# Configuración

- Podemos establecer una serie de propiedades
- Estas propiedades se enviarán al servidor al realizar la conexión
- Son parejas *<clave, valor>*

```
con.setRequestProperty(nombre, valor);
```

- Por ejemplo

```
con.setRequestProperty("IF-Modified-Since",  
                        "22 Sep 2002 08:00:00 GMT");  
con.setRequestProperty("Content-Language", "es-ES");
```



# Leer y enviar contenido

- Podemos abrir un flujo de salida para enviar contenido al servidor (si hemos activado la salida)

```
OutputStream out = con.getOutputStream();
```

- Podemos abrir un flujo de entrada para leer el contenido devuelto

```
InputStream in = con.getInputStream();
```

- Al abrir estos flujos se pasa automáticamente a estado conectado



# Cabeceras de la respuesta

- Además del contenido podemos obtener cabeceras con información sobre el recurso

```
String valor = con.getHeaderField(nombre);
```

- Cabeceras estándar:

<code>getLength</code>	Tamaño del contenido
<code>getType</code>	Tipo MIME del contenido
<code>getEncoding</code>	Codificación del contenido
<code>getExpiration</code>	Fecha de caducidad
<code>getDate</code>	Fecha del envío
<code>getLastModified</code>	Fecha de última modificación

- Leer estas cabeceras provoca el paso a estado conectado



# Ejemplo

```
// Creamos la URL y la conexión activando la salida
URL url = new URL("http://j2ee.ua.es/chat/enviar");
URLConnection con = url.openConnection();
con.setDoOutput(true);

// Escribimos los datos en un buffer en memoria
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);
dos.writeUTF(nick);
dos.writeUTF(msg);
dos.close();

// Establecemos las propiedades de tipo y tamaño del contenido
con.setRequestProperty("Content-Length", String.valueOf(baos.size()));
con.setRequestProperty("Content-Type", "application/octet-stream");

// Abrimos el flujo de salida para enviar los datos al servidor
OutputStream out = con.getOutputStream();
baos.writeTo(out);

// Abrimos el flujo de entrada para leer la respuesta obtenida
InputStream in = con.getInputStream();
```



# JavaMail

- API Java con las clases abstractas que definen los objetos e interfaces necesarias para implementar un sistema de e-mail.
- Proveedores JavaMail
  - Proporcionan la funcionalidad necesaria para establecer la comunicación a través de protocolos concretos.
- Protocolos soportados por la implementación de Sun:
  - SMTP (*Simple Mail Transfer Protocol*)
  - IMAP (*Internet Message Access Protocol*)
  - MIME (*Multipurpose Internet Mail Extensions*)
  - POP3 (*Post Office Protocol 3*)



# Elementos de JavaMail

- Elementos de JavaMail:
  - Gestión de sesión: Protocolo de transferencia.
  - Manipulación de mensajes: Creación y manipulación.
  - Almacenamiento: Jerarquías de almacenamiento.
  - Transporte: Mecanismos de entrega del mensaje.
- Conceptos JavaMail:
  - `Transport`: Servicio con capacidad para enviar mensajes a un destino (usualmente con SMTP)
  - `Store`: Servicio con el que hay que conectar para descargar mensajes que han sido enviados a nuestro buzón (p.e. POP3 o IMAP)



# Configuración de la sesión

- **Parámetros de sesión:** `java.util.Properties`
  - Definen los parámetros de sesión. Pueden ser impuestos por nosotros o bien extraídos del sistema:
  - Parámetros: protocolo, host, usuario, etc..

```
Properties props = new Properties();
props.put("mail.transport.protocol", "smtp");
props.put("mail.smtp.host", "mail.alu.ua.es");
props.put("mail.smtp.port", "25");

props.put("mail.smtp.starttls.enable", "true");
Session session = Session.getInstance(props);
```





# Mensaje

- Nos ocupamos ahora de la “construcción del mensaje”.
- Clase abstracta: `Message`
  - Proporciona un contenedor para la representación del mensaje de correo.
- Estructura del mensaje:
  - Cabecera: Asunto, receptor/es remitente, fecha de envío, etc.
  - Contenido: Datos propios del mensaje.
- Subclase para tratamiento de tipos MIME.
  - `javax.mail.internet.MimeMessage`



# Composición del mensaje

- Constructor
  - A partir de los datos de la sesión o bien como copia de otra instancia (constructor de copia).

```
Message msg = new MimeMessage(session);
```

- Métodos *setters*: (existen los correspondiente *getters*)
  - `setFrom()`: Usando `javax.mail.internet.InternetAddress`
  - `setRecipients()`: Especificar *to*, *cc*, *bcc*
  - `setSubject()`: Asunto del mensaje
  - `setText()`: Si se trata de una mensaje de texto
  - `setContent()`: Si se trata de un mensaje MIME
  - `setSentDate()`: Fecha de envío.



## Envío del mensaje

- Especificamos destinatarios en el mensaje

```
msg.setRecipients(  
    Message.RecipientType.TO,  
    InternetAddress.parse(  
        "malozano@dccia.ua.es", false));
```

- Enviamos el mensaje usando Transport

```
Transport.send(msg);
```



# Mensaje con adjuntos

```
// Parte de texto
BodyPart messageBodyPart = new MimeBodyPart();
messageBodyPart.setText("Aqui te envio el fichero");

// Parte adjunta
BodyPart attachBodyPart = new MimeBodyPart();
DataSource source = new FileDataSource(filename);
attachBodyPart.setDataHandler(new DataHandler(source));
attachBodyPart.setFileName(filename);

// Componemos el mensaje
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);
multipart.addBodyPart(attachBodyPart);
message.setContent(multipart);
Transport.send(message);
```



# Autenticación

- Especificamos como obtener *login* y *password*

```
public class Autenticador extends Authenticator {
    public PasswordAuthentication getPasswordAuthentication()
    {
        String username, password;
        Scanner scan = new Scanner(System.in);
        username = scan.nextLine();
        password = scan.nextLine();
        return new PasswordAuthentication(username, password);
    }
}
```

- Creamos la sesión

```
Authenticator aut = new Autenticador();
props.put("mail.smtp.auth", "true");
Session session = Session.getDefaultInstance(props, aut);
```



# ¿Preguntas...?