UD4 - 03. Utilización de objetos predefinidos

Introducción

En Java, los objetos predefinidos facilitan el desarrollo de aplicaciones de red al ofrecer clases listas para manejar tareas comunes, como gestionar flujos de entrada y salida, realizar conexiones a través de sockets, o incluso consumir APIs externas. Estos objetos forman parte de los paquetes estándar de Java, como `java.net`, `java.io`, y `java.util`.

Clases y Objetos Predefinidos Relevantes

1. `InputStream` y `OutputStream` (Flujos de Datos)

- **Función**: Permiten leer y escribir datos entre una aplicación y una fuente o destino (archivo, socket, etc.).
- **Principales métodos**:
  - `read()`: Lee bytes de datos.
  - `write()`: Escribe bytes de datos.

Ejemplo de uso con `Socket`:

```
import java.io.*;
import java.net.Socket;

public class ClienteSimple {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 5000);
             OutputStream out = socket.getOutputStream();
             InputStream in = socket.getInputStream()) {

            String mensaje = "Hola desde el cliente";
            out.write(mensaje.getBytes());

            byte[] buffer = new byte[1024];
            int bytesLeidos = in.read(buffer);
```

```
            System.out.println("Respuesta del servidor: " + new
String(buffer, 0, bytesLeidos));

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. `BufferedReader` y `BufferedWriter` (Lectura y Escritura Eficiente)

- **Función**: Proporcionan una forma eficiente de leer y escribir líneas de texto.
- **Principales métodos**:
    - `readLine()`: Lee una línea completa de texto.
    - `write(String str)`: Escribe una línea de texto.
    - `flush()`: Fuerza el envío del texto.

Ejemplo de servidor usando `BufferedReader` y `BufferedWriter`:

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class ServidorSimple {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(5000)) {
            System.out.println("Servidor escuchando en el puerto
5000...");

            while (true) {
                Socket cliente = serverSocket.accept();
                BufferedReader reader = new BufferedReader(new
InputStreamReader(cliente.getInputStream()));
                BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(cliente.getOutputStream()));

                String mensaje = reader.readLine();
                System.out.println("Mensaje del cliente: " +
mensaje);
```

```
                writer.write("Mensaje recibido: " + mensaje + "\n");
                writer.flush();
                cliente.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3. `PrintWriter`

- **Función**: Permite escribir datos de manera más sencilla, especialmente para manejar cadenas de texto.
- **Métodos comunes**:
  - `println(String s)`: Escribe una línea de texto seguida de un salto de línea.
  - `close()`: Cierra el flujo.

Ejemplo con `PrintWriter`:

```
import java.io.PrintWriter;
import java.net.Socket;

public class ClientePrintWriter {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 5000);
             PrintWriter writer = new
PrintWriter(socket.getOutputStream(), true)) {

            writer.println("¡Hola desde PrintWriter!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

4. `ObjectInputStream` y `ObjectOutputStream` (Serialización de Objetos)

- **Función**: Permiten enviar y recibir objetos completos a través de un flujo.

- **Principales métodos**:
    - `writeObject(Object o)`: Envía un objeto serializado.
    - `readObject()`: Recibe un objeto deserializado.

Ejemplo de envío de un objeto:

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

class Persona implements Serializable {
    String nombre;
    int edad;

    Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    @Override
    public String toString() {
        return "Nombre: " + nombre + ", Edad: " + edad;
    }
}

public class ServidorObjetos {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(5000)) {
            System.out.println("Servidor esperando envío de
objetos...");
            Socket cliente = serverSocket.accept();
            ObjectInputStream objectInput = new
ObjectInputStream(cliente.getInputStream());

            Persona persona = (Persona) objectInput.readObject();
            System.out.println("Objeto recibido: " + persona);
            cliente.close();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

5. Scanner

- **Función**: Facilita la lectura de entradas desde diversos orígenes, como `System.in` (teclado) o flujos de entrada de red.

Ejemplo básico de lectura de datos desde la consola:
java
Copiar código

```
import java.util.Scanner;

public class EntradaConsola {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Introduce tu nombre: ");
        String nombre = scanner.nextLine();
        System.out.println("Hola, " + nombre);
        scanner.close();
    }
}
```

Uso de Objetos Predefinidos en Servicios REST

Cuando trabajamos con servicios web, frameworks como **Spring Boot** utilizan objetos predefinidos como `HttpEntity`, `ResponseEntity`, `RestTemplate`, etc., para gestionar peticiones y respuestas.

Ejemplo:

```
import org.springframework.web.client.RestTemplate;

public class ConsumidorRest {
    public static void main(String[] args) {
        RestTemplate restTemplate = new RestTemplate();
        String respuesta =
restTemplate.getForObject("https://jsonplaceholder.typicode.com/post
s/1", String.class);
        System.out.println("Respuesta del servicio: " + respuesta);
    }
```

```
}
```

-

Conclusión

Los objetos predefinidos de Java simplifican el manejo de la comunicación en red, la lectura y escritura de datos, y la serialización de objetos. Desde clases como `BufferedReader` y `PrintWriter` hasta flujos más avanzados como `ObjectInputStream`, estos componentes permiten crear aplicaciones de red más robustas y eficientes. Además, herramientas como `RestTemplate` facilitan el consumo de APIs REST de manera sencilla.