Apuntes-de-Acceso-a-datos.pdf

hugoomazariio_

Acceso a datos

1º Desarrollo de Aplicaciones Multiplataforma

Estudios España

Apuntes de Teoría: Manejo de Archivos en Java

El manejo de archivos en Java permite leer, escribir, modificar y eliminar archivos en el sistema. Se pueden manejar archivos de texto y archivos binarios.

1. Tipos de Archivos

En programación, los archivos pueden clasificarse en dos grandes categorías:

1.1 Archivos de Texto

- Contienen caracteres legibles por humanos.
- Se pueden editar con cualquier editor de texto.
- Ejemplos: `.txt`, `.csv`, `.json`, `.xml`.
- Utilizan codificación como UTF-8 o ASCII.

1.2 Archivos Binarios

- Contienen datos en formato binario, no legibles directamente por humanos.
- Se requieren programas específicos para interpretarlos.
- Ejemplos: `.jpg`, `.png`, `.exe`, `.mp3`, `.dat`.
- Se leen y escriben en bloques de bytes.

2. Clases Principales para Manejo de Archivos en Java

Java proporciona diversas clases para la manipulación de archivos en los paquetes `java.io` y `java.nio`.

2.1 Clases del paquete `java.io`

- **File**: Representa un archivo o directorio en el sistema de archivos.
- **FileReader y BufferedReader**: Para leer archivos de texto línea por línea.
- **FileWriter y BufferedWriter**: Para escribir en archivos de texto.
- **FileInputStream y FileOutputStream**: Para leer y escribir archivos binarios.
- **DataInputStream y DataOutputStream**: Para leer y escribir datos primitivos en archivos.
- **RandomAccessFile**: Permite acceso aleatorio a un archivo.

2.2 Clases del paquete `java.nio`

- **Files**: Proporciona métodos estáticos para operar con archivos y directorios.
- **Paths**: Permite trabajar con rutas de archivos de manera más sencilla.
- **Path**: Representa una ruta de archivo o directorio.
- **FileChannel**: Para leer y escribir datos en archivos de manera eficiente.

3. Operaciones Básicas con Archivos en Java

3.1 Creación de Archivos

```
import java.io.File;
import java.io.IOException;

public class CrearArchivo {
    public static void main(String[] args) {
        try {
            File archivo = new File("archivo.txt");
            if (archivo.createNewFile()) {
                System.out.println("Archivo creado: " +
archivo.getName());
            } else {
                System.out.println("El archivo ya existe.");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3.2 Escritura en Archivos

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class EscribirArchivo {
    public static void main(String[] args) {
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter("archivo.txt"))) {
            writer.write("Hola, mundo!");
            writer.newLine();
            writer.write("Otra línea de texto.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3.3 Lectura de Archivos

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class LeerArchivo {
    public static void main(String[] args) {
```

```
        try (BufferedReader reader = new BufferedReader(new
FileReader("archivo.txt"))) {
            String linea;
            while ((linea = reader.readLine()) != null) {
                System.out.println(linea);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3.4 Eliminación de Archivos

```
import java.io.File;

public class EliminarArchivo {
    public static void main(String[] args) {
        File archivo = new File("archivo.txt");
        if (archivo.delete()) {
            System.out.println("Archivo eliminado: " +
archivo.getName());
        } else {
            System.out.println("No se pudo eliminar el archivo.");
        }
    }
}
```

4. Manejo de Directorios

Además de archivos, Java permite trabajar con directorios.

4.1 Crear un Directorio

```
import java.io.File;

public class CrearDirectorio {
    public static void main(String[] args) {
        File directorio = new File("miDirectorio");
        if (directorio.mkdir()) {
            System.out.println("Directorio creado correctamente.");
        } else {
            System.out.println("No se pudo crear el directorio.");
        }
    }
}
```

4.2 Listar Archivos en un Directorio

```
import java.io.File;

public class ListarArchivos {
    public static void main(String[] args) {
        File directorio = new File("."); // Directorio actual
        String[] archivos = directorio.list();
        if (archivos != null) {
            for (String archivo : archivos) {
                System.out.println(archivo);
            }
        }
```

```
    }
}
```

4.3 Verificar Si un Archivo o Directorio Existe

```
import java.io.File;

public class VerificarArchivo {
    public static void main(String[] args) {
        File archivo = new File("archivo.txt");
        if (archivo.exists()) {
            System.out.println("El archivo existe.");
        } else {
            System.out.println("El archivo no existe.");
        }
    }
}
```

Apuntes de Teoría: JDBC (Java Database Connectivity)

JDBC es una API que permite a Java interactuar con bases de datos relacionales como MySQL, PostgreSQL, SQL Server y Oracle. Proporciona una forma estándar de conectar, consultar y manipular bases de datos desde aplicaciones Java.

1. Arquitectura de JDBC

JDBC sigue un modelo de cuatro capas:

1. **Application Layer**: Código Java que usa JDBC.
2. **JDBC API**: Conjunto de clases e interfaces que permiten interactuar con la base de datos.
3. **JDBC Driver Manager**: Carga y gestiona los drivers de la base de datos.
4. **Database Driver**: Driver específico para la base de datos usada.

2. Dependencias para JDBC en Maven

Para utilizar JDBC con MySQL en un proyecto Maven, se debe agregar la siguiente dependencia en el archivo `pom.xml`:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>
```

Para PostgreSQL:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.5.1</version>
</dependency>
```

3. Establecer una Conexión

Para conectar con una base de datos, se usa la clase `DriverManager`.

```
import java.sql.*;

public class ConexionJDBC {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base";
        String usuario = "root";
        String clave = "password";

        try (Connection conexion = DriverManager.getConnection(url,
usuario, clave)) {
            System.out.println("Conexión exitosa");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

4. Operaciones CRUD con JDBC

4.1 Creación de Tablas

Para crear tablas se usa `Statement`.

```
try (Connection conexion = DriverManager.getConnection(url, usuario,
clave);
     Statement stmt = conexion.createStatement()) {

    String sql = "CREATE TABLE usuarios (
                    id INT AUTO_INCREMENT PRIMARY KEY,
                    nombre VARCHAR(50),
                    email VARCHAR(50)
                )";
    stmt.executeUpdate(sql);
    System.out.println("Tabla creada exitosamente");
} catch (SQLException e) {
    e.printStackTrace();
}
```

4.2 Inserción de Datos

```
String sql = "INSERT INTO usuarios (nombre, email) VALUES (?, ?)";
try (PreparedStatement pstmt = conexion.prepareStatement(sql)) {
    pstmt.setString(1, "Juan Pérez");
    pstmt.setString(2, "juan@example.com");
    pstmt.executeUpdate();
    System.out.println("Registro insertado");
} catch (SQLException e) {
    e.printStackTrace();
```

Google Gemini: Plan Pro a 0€ durante 1 año.

Tu ventaja por ser estudiante

Entra en wlh.es/estudiacongemini

Consigue la oferta

Sintetiza horas de investigación en minutos.

Oferta válida hasta el 9 de diciembre de 2025

```
}
```

4.3 Lectura de Datos

```
String sql = "SELECT * FROM usuarios";
try (Statement stmt = conexion.createStatement();
     ResultSet rs = stmt.executeQuery(sql)) {

    while (rs.next()) {
        int id = rs.getInt("id");
        String nombre = rs.getString("nombre");
        String email = rs.getString("email");
        System.out.println(id + " - " + nombre + " - " + email);
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

4.4 Actualización de Datos

```
String sql = "UPDATE usuarios SET email = ? WHERE id = ?";
try (PreparedStatement pstmt = conexion.prepareStatement(sql)) {
    pstmt.setString(1, "nuevo@example.com");
    pstmt.setInt(2, 1);
    pstmt.executeUpdate();
    System.out.println("Registro actualizado");
} catch (SQLException e) {
    e.printStackTrace();
}
```

4.5 Eliminación de Datos

```
String sql = "DELETE FROM usuarios WHERE id = ?";
try (PreparedStatement pstmt = conexion.prepareStatement(sql)) {
    pstmt.setInt(1, 1);
    pstmt.executeUpdate();
    System.out.println("Registro eliminado");
} catch (SQLException e) {
    e.printStackTrace();
}
```

5. Transacciones en JDBC

JDBC permite manejar transacciones con `setAutoCommit(false)` y `commit()`.

```
try (Connection conexion = DriverManager.getConnection(url, usuario,
clave)) {
    conexion.setAutoCommit(false);

    try (PreparedStatement pstmt = conexion.prepareStatement("INSERT
INTO usuarios (nombre, email) VALUES (?, ?)");) {
        pstmt.setString(1, "Carlos");
        pstmt.setString(2, "carlos@example.com");
        pstmt.executeUpdate();

        conexion.commit();
        System.out.println("Transacción completada");
    } catch (SQLException e) {
        conexion.rollback();
        System.out.println("Transacción revertida");
```

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

6. Conexión a Bases de Datos desde un Archivo `properties`

Es una buena práctica almacenar credenciales en un archivo `.properties`.

```
jdbc.url=jdbc:mysql://localhost:3306/mi_base
jdbc.user=root
jdbc.password=password
```

Código para leer el archivo `.properties`:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.sql.*;
import java.util.Properties;

public class ConexionDesdeArchivo {
    public static void main(String[] args) {
        Properties props = new Properties();
        try (FileInputStream fis = new
FileInputStream("config.properties")) {
            props.load(fis);
            String url = props.getProperty("jdbc.url");
            String usuario = props.getProperty("jdbc.user");
            String clave = props.getProperty("jdbc.password");

            try (Connection conexion =
DriverManager.getConnection(url, usuario, clave)) {
                System.out.println("Conexión establecida desde
archivo");
            }
        } catch (IOException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Apuntes de Teoría: JDBC (Java Database Connectivity)

JDBC es una API que permite a Java interactuar con bases de datos relacionales como MySQL, PostgreSQL, SQL Server y Oracle. Proporciona una forma estándar de conectar, consultar y manipular bases de datos desde aplicaciones Java.

1. Arquitectura de JDBC

JDBC sigue un modelo de cuatro capas:

1. **Application Layer**: Código Java que usa JDBC.
2. **JDBC API**: Conjunto de clases e interfaces que permiten interactuar con la base de datos.
3. **JDBC Driver Manager**: Carga y gestiona los drivers de la base de datos.
4. **Database Driver**: Driver específico para la base de datos usada.

2. Dependencias para JDBC en Maven

Para utilizar JDBC con MySQL en un proyecto Maven, se debe agregar la siguiente dependencia en el archivo `pom.xml`:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>
```

Para PostgreSQL:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.5.1</version>
</dependency>
```

3. Establecer una Conexión

Para conectar con una base de datos, se usa la clase `DriverManager`.

```
import java.sql.*;

public class ConexionJDBC {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base";
        String usuario = "root";
        String clave = "password";

        try (Connection conexion = DriverManager.getConnection(url,
usuario, clave)) {
            System.out.println("Conexión exitosa");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

4. Operaciones CRUD con JDBC

4.1 Creación de Tablas

Para crear tablas se usa `Statement`.

```
try (Connection conexion = DriverManager.getConnection(url, usuario,
clave);
     Statement stmt = conexion.createStatement()) {

    String sql = "CREATE TABLE usuarios (
                    id INT AUTO_INCREMENT PRIMARY KEY,
                    nombre VARCHAR(50),
                    email VARCHAR(50)
                )";
    stmt.executeUpdate(sql);
    System.out.println("Tabla creada exitosamente");
} catch (SQLException e) {
    e.printStackTrace();
}
```

4.2 Inserción de Datos

```
String sql = "INSERT INTO usuarios (nombre, email) VALUES (?, ?)";
try (PreparedStatement pstmt = conexion.prepareStatement(sql)) {
    pstmt.setString(1, "Juan Pérez");
    pstmt.setString(2, "juan@example.com");
    pstmt.executeUpdate();
    System.out.println("Registro insertado");
} catch (SQLException e) {
    e.printStackTrace();
}
```

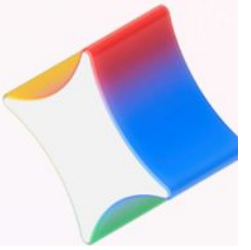
4.3 Lectura de Datos

```
String sql = "SELECT * FROM usuarios";
try (Statement stmt = conexion.createStatement();
     ResultSet rs = stmt.executeQuery(sql)) {

    while (rs.next()) {
        int id = rs.getInt("id");
        String nombre = rs.getString("nombre");
        String email = rs.getString("email");
        System.out.println(id + " - " + nombre + " - " + email);
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

4.4 Actualización de Datos

```
String sql = "UPDATE usuarios SET email = ? WHERE id = ?";
try (PreparedStatement pstmt = conexion.prepareStatement(sql)) {
    pstmt.setString(1, "nuevo@example.com");
    pstmt.setInt(2, 1);
    pstmt.executeUpdate();
    System.out.println("Registro actualizado");
} catch (SQLException e) {
    e.printStackTrace();
}
```



4.5 Eliminación de Datos

```
String sql = "DELETE FROM usuarios WHERE id = ?";
try (PreparedStatement pstmt = conexion.prepareStatement(sql)) {
    pstmt.setInt(1, 1);
    pstmt.executeUpdate();
    System.out.println("Registro eliminado");
} catch (SQLException e) {
    e.printStackTrace();
}
```

5. Transacciones en JDBC

JDBC permite manejar transacciones con `setAutoCommit(false)` y `commit()`.

```
try (Connection conexion = DriverManager.getConnection(url, usuario,
clave)) {
    conexion.setAutoCommit(false);

    try (PreparedStatement pstmt = conexion.prepareStatement("INSERT
INTO usuarios (nombre, email) VALUES (?, ?)");) {
        pstmt.setString(1, "Carlos");
        pstmt.setString(2, "carlos@example.com");
        pstmt.executeUpdate();

        conexion.commit();
        System.out.println("Transacción completada");
    } catch (SQLException e) {
        conexion.rollback();
        System.out.println("Transacción revertida");
        e.printStackTrace();
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

6. Conexión a Bases de Datos desde un Archivo `properties`

Es una buena práctica almacenar credenciales en un archivo `.properties`.

```
jdbc.url=jdbc:mysql://localhost:3306/mi_base
jdbc.user=root
jdbc.password=password
```

Código para leer el archivo `.properties`:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.sql.*;
import java.util.Properties;

public class ConexionDesdeArchivo {
    public static void main(String[] args) {
        Properties props = new Properties();
        try (FileInputStream fis = new
FileInputStream("config.properties")) {
            props.load(fis);
            String url = props.getProperty("jdbc.url");
```

```
            String usuario = props.getProperty("jdbc.user");
            String clave = props.getProperty("jdbc.password");

            try (Connection conexion =
DriverManager.getConnection(url, usuario, clave)) {
                System.out.println("Conexión establecida desde
archivo");
            }
        } catch (IOException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Apuntes de Teoría: Spring Boot en Java con Maven

Spring Boot es un framework basado en Spring que permite crear aplicaciones Java de manera rápida y sencilla, minimizando la configuración manual y proporcionando herramientas para desarrollo eficiente.

1. Características de Spring Boot

- **Configuración Automática**: Usa `Spring Boot Starter` para configuración predeterminada.
- **Servidor Integrado**: Incorpora servidores como Tomcat, Jetty o Undertow.
- **Arquitectura Basada en Microservicios**: Ideal para aplicaciones escalables.
- **Manejo de Dependencias con Spring Boot Starters**.
- **Spring Boot Actuator**: Monitorización y métricas integradas.

2. Configuración de un Proyecto Spring Boot con Maven

Para crear un proyecto Spring Boot, se puede usar Spring Initializr o definir manualmente el `pom.xml`.

2.1 Dependencias en `pom.xml`

```
<dependencies>
    <!-- Starter para aplicaciones web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Starter para JPA y Hibernate -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
```

```
    <!-- Driver para MySQL -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>

    <!-- Spring Boot Starter Test -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
    </dependency>
</dependencies>
```

3. Estructura de un Proyecto Spring Boot

```
mi-proyecto/
├── src/main/java/com/ejemplo/
│   ├── MiProyectoApplication.java
│   ├── controlador/
│   │   ├── UsuarioController.java
│   ├── modelo/
│   │   ├── Usuario.java
│   ├── repositorio/
│   │   ├── UsuarioRepository.java
├── src/main/resources/
│   ├── application.properties
```

4. Creación de la Aplicación Principal

Spring Boot usa la anotación `@SpringBootApplication` para iniciar la aplicación.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MiProyectoApplication {
    public static void main(String[] args) {
        SpringApplication.run(MiProyectoApplication.class, args);
    }
}
```

5. Configuración con `application.properties`

```
server.port=8081
spring.datasource.url=jdbc:mysql://localhost:3306/mi_base
spring.datasource.username=root
spring.datasource.password=1234
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

6. Creación de un Controlador REST

```
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api")
public class UsuarioController {
    @GetMapping("/saludo")
    public String saludo() {
        return "¡Hola desde Spring Boot!";
```

```
    }
}
```

7. Creación de un Modelo y Repositorio con JPA

7.1 Modelo (Entidad)

```
import jakarta.persistence.*;

@Entity
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    private String email;

    // Getters y Setters
}
```

7.2 Repositorio

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface UsuarioRepository extends JpaRepository<Usuario,
Long> {
}
```

8. Inyección de Dependencias en Spring Boot

Spring Boot permite la inyección de dependencias con `@Autowired`.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class UsuarioService {
    @Autowired
    private UsuarioRepository usuarioRepository;

    public List<Usuario> listarUsuarios() {
        return usuarioRepository.findAll();
    }
}
```

9. Manejo de Errores con `@ExceptionHandler`

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;

@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(Exception.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public String manejarExcepcion(Exception e) {
        return "Error: " + e.getMessage();
    }
}
```

10. Pruebas en Spring Boot

Spring Boot permite realizar pruebas unitarias y de integración con `@SpringBootTest`.

```
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class MiProyectoApplicationTests {
    @Test
    void contextLoads() {
    }
}
```

11. Spring Boot Actuator para Monitorización

Spring Boot Actuator proporciona métricas y monitorización.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Activar en `application.properties`:

```
management.endpoints.web.exposure.include=*
```

12. Despliegue de una Aplicación Spring Boot

- **Ejecutar la aplicación:** `mvn spring-boot:run`
- **Empaquetar la aplicación en un `.jar`:** `mvn clean package`
- **Ejecutar el `.jar`:** `java -jar target/mi-proyecto-0.0.1-SNAPSHOT.jar`