Working with text files in Java involves reading data from text files, writing data to text files, and manipulating the contents of text files. Here's an example that demonstrates basic operations with text files in Java:

import java.io.\*;

public class TextFileExample {

public static void main(String[] args) {

String fileName = "example.txt";

// Writing data to a text file

try (BufferedWriter writer = new BufferedWriter(new FileWriter(fileName))) {

writer.write("Hello, World!");

writer.newLine();

writer.write("This is an example of writing to a text file.");

writer.newLine();

writer.write("Goodbye!");

} catch (IOException e) {

System.err.println("Error writing to the file: " + e.getMessage());

}

// Reading data from a text file

try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {

String line;

while ((line = reader.readLine()) != null) {

System.out.println(line);

}

} catch (IOException e) {

System.err.println("Error reading the file: " + e.getMessage());

}

}

}

Working with binary files in Java involves reading and writing raw binary data to files. Binary files contain data in a format that is not directly human-readable, such as images, audio files, or serialized objects. Here's an example that demonstrates basic operations with binary files in Java:

import java.io.\*;

public class BinaryFileExample {

public static void main(String[] args) {

String fileName = "example.bin";

// Writing data to a binary file

try (DataOutputStream outputStream = new DataOutputStream(new FileOutputStream(fileName))) {

outputStream.writeInt(42);

outputStream.writeDouble(3.14);

outputStream.writeBoolean(true);

} catch (IOException e) {

System.err.println("Error writing to the file: " + e.getMessage());

}

// Reading data from a binary file

try (DataInputStream inputStream = new DataInputStream(new FileInputStream(fileName))) {

int intValue = inputStream.readInt();

double doubleValue = inputStream.readDouble();

boolean booleanValue = inputStream.readBoolean();

System.out.println("Int Value: " + intValue);

System.out.println("Double Value: " + doubleValue);

System.out.println("Boolean Value: " + booleanValue);

} catch (IOException e) {

System.err.println("Error reading the file: " + e.getMessage());

}

}

}

Serialization in Java is the process of converting objects into a stream of bytes to store them in memory, transfer them over a network, or persist them to disk. Deserialization is the reverse process, where the byte stream is used to recreate the object. The serialization and deserialization process allows objects to be easily stored, transmitted, and reconstructed when needed. Java provides built-in support for serialization through the **java.io.Serializable** interface and the **ObjectInputStream** and **ObjectOutputStream** classes.

Here's an example that demonstrates serialization and deserialization in Java:

import java.io.\*;

public class SerializationExample {

public static void main(String[] args) {

String fileName = "serialized\_object.ser";

// Serialization

try (ObjectOutputStream outputStream = new ObjectOutputStream(new FileOutputStream(fileName))) {

MyClass myObject = new MyClass("John Doe", 42);

outputStream.writeObject(myObject);

System.out.println("Object serialized and saved to file.");

} catch (IOException e) {

System.err.println("Error during serialization: " + e.getMessage());

}

// Deserialization

try (ObjectInputStream inputStream = new ObjectInputStream(new FileInputStream(fileName))) {

MyClass myObject = (MyClass) inputStream.readObject();

System.out.println("Object deserialized from file:");

System.out.println(myObject);

} catch (IOException | ClassNotFoundException e) {

System.err.println("Error during deserialization: " + e.getMessage());

}

}

}

class MyClass implements Serializable {

private String name;

private int age;

public MyClass(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return "MyClass{name='" + name + "', age=" + age + "}";

}

}

Java properties files provide a convenient way to store configuration settings or simple key-value pairs in a text-based format. They are commonly used to externalize configuration from code, making it easier to modify settings without recompiling the application. Java provides the **java.util.Properties** class to work with properties files. Here's an example that demonstrates how to read and write properties files in Java:

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.OutputStream;

import java.util.Properties;

public class PropertiesWriterExample {

public static void main(String[] args) {

Properties properties = new Properties();

properties.setProperty("database.url", "jdbc:mysql://localhost:3306/mydb");

properties.setProperty("database.username", "myusername");

properties.setProperty("database.password", "mypassword");

try (OutputStream output = new FileOutputStream("config.properties")) {

properties.store(output, "My Configuration");

System.out.println("Properties saved to config.properties file.");

} catch (IOException e) {

System.err.println("Error writing properties file: " + e.getMessage());

}

}

}