Binary Files

Storing data in its binary format is a lot **more efficient than storing it as text**. Handling binary files in Java involves working with raw byte data rather than characters or text. This is essential for processing images, videos, audio, or custom data structures efficiently.

- **Byte-Oriented**: Binary operations handle data as 8-bit bytes.
- **Non-Human Readable**: Unlike text files, **binary files appear as gibberish if opened in a standard text editor**.

1. Basic Byte Streams

The foundation for **binary I/O** consists of the **abstract classes InputStream** and **OutputStream**.

- **FileInputStream:** Reads raw bytes from a file.
- **FileOutputStream:** Writes raw bytes to a file.

2. High-Level Data Streams

To work with **primitive data types** (like int, double, or boolean) instead of raw bytes, you wrap basic streams with **Data Streams**.

- **DataOutputStream:** Provides methods like **writeInt()**, **writeDouble()**, and **writeUTF()** for strings.
- **DataInputStream**: Provides corresponding methods like **readInt()** and **readDouble()**.

Important: You must read data in the **exact same order** it was written.

In Java, **FileOutputStream** and **DataOutputStream** are **used together to write data to files**, but they serve different purposes. While the **former connects to the actual file on your disk**, the **latter provides the tools to write complex data types like integers or booleans easily**.

DataOutputStream is wrapped around a **FileOutputStream** object to write data to a binary file:

```
FileOutputStream fstream = new FileOutputStream("MyInfo.dat");
DataOutputStream outputFile = new DataOutputStream(fstream);
```

To simplify the code, this can be combined onto one line:

DataOutputStream outputFile = new DataOutputStream(new FileOutputStream("MyInfo.dat"));

The **DataOutputStream** has methods for **writing** all the **primitives and strings**:

– writeInt();
– writeChar();
– writeDouble();
● The string method is
– writeUTF()
● This uses a string encoding format called **UTF-8**

Hints

writeInt (and similar I/O methods) requires a **throws IOException clause** in Java because it performs checked operations that can fail—such as **disk full**, **file locked**, or **network disconnection**—which the compiler forces the programmer to handle. This structure ensures safe error handling and forces accountability on the developer for potential runtime issues.

Writing a file

To write the **number 5** to a **file**:

```
FileOutputStream fstream = new FileOutputStream("MyInfo.dat");
DataOutputStream outputFile = new DataOutputStream(fstream);
outputFile.writeInt(5);
outputFile.close();
```

Example

```
import java.io.*;
import java.io.IOException;

public class BinaryF1 {
  public static void main(String[] args) {
// Use try-with-resources to automatically close the streams
try {
   DataOutputStream dos = new DataOutputStream(new
FileOutputStream("data.bin"));
```

```
    dos.writeInt(123);          // Writes 4 bytes representing the integer
    dos.writeDouble(45.67);     // Writes 8 bytes representing the double
    dos.writeBoolean(true);     // Writes 1 byte (1 for true, 0 for false)
    dos.writeUTF("Hello Java");  // Writes a portable UTF-encoded string
    dos.close();

    System.out.println("Data successfully written to file."); }
  catch (IOException e) {
// Prints the error details to the console
          e.printStackTrace(); } } }
```

Output

Data successfully written to file.

Reading from a binary file

- Just being able to read data isn't very useful.
- To read data back out of the written file, you can use the complementary classes:
  - **FileInputStream**
  - **DataInputStream**

- They are used the same way as writing only for reading:

Example: Reading from a binary file

```
import java.io.*;
import java.io.IOException;

public class ReadBinary {
    public static void main(String[] args) throws IOException {

  try{
    DataOutputStream dataOutputStream = new DataOutputStream(
                        new FileOutputStream("data.bin"));

    dataOutputStream.writeInt(123);
    dataOutputStream.writeFloat(123.45F);
    dataOutputStream.writeLong(789);

   dataOutputStream.close();
```

```
  DataInputStream dataInputStream = new DataInputStream(
        new FileInputStream("data.bin"));

        int   int123     = dataInputStream.readInt();
        float float12345 = dataInputStream.readFloat();
        long  long789    = dataInputStream.readLong();

        dataInputStream.close();

        System.out.println("int123     = " + int123);
        System.out.println("float12345 = " + float12345);
        System.out.println("long789    = " + long789); }

catch(IOException e) {
 // Prints the error details to the console
   e.printStackTrace();  }}}
```

Output

```
int123     = 123
float12345 = 123.45
long789    = 789
```

Bitwise operators

Bitwise operators in Java are used to **perform operations directly on the binary representation (bits) of integer values**. **Instead of working on whole numbers**, these operators **manipulate data bit by bit**, enabling fast and efficient low-level operations.

Bitwise operators work only with integer data types such as byte, short, int, long, and char

- **& AND**
- **| OR**
- **~ NOT**

Bitwise operators work like mathematical operators:
They take **two operands and return a number**

- 1 + 2 = 3

- 5 * 3 = 15
- 5 / 2 = 2.5
- 4 – 3 = 1

1. Bitwise AND (&)

This operator is a **binary operator**, denoted by **'&'**. It returns **bit by bit AND** of input values, i.e., if **both bits are 1, it gives 1, else it shows 0**.

Example:

a = **5** = 0101 (In Binary)
b = **7** = 0111 (In Binary)

Bitwise AND Operation of 5 and 7

```
  0101
& 0111
_______
  0101 = 5 (In decimal)
```

2. Bitwise OR (|)

This operator is a binary operator, denoted by **'|'**. It returns bit by bit **OR of input** values, i.e., if either of the bits is 1, it gives 1, else it shows 0.

Example:

a = **5** = 0101 (In Binary)
b = **7** = 0111 (In Binary)

Bitwise OR Operation of 5 and 7

```
  0101
| 0111
_______
  0111 = 7 (In decimal)
```

3. Bitwise XOR (^)

This operator is a binary operator, denoted by **'^'**. It returns bit by bit **XOR** of input values, i.e., if corresponding **bits are different**, it **gives 1**, else it **shows 0**.

Example:

a = 5 = 0101 (In Binary)
b = 7 = 0111 (In Binary)

Bitwise XOR Operation of 5 and 7
  0101
^ 0111

_____
  0010 = 2 (In decimal)

4. Bitwise Complement (~)
This operator is a **unary operator**, denoted by **'~'**. It inverts all the bits of the given number (**0 becomes 1 and 1 becomes 0**).

Important: In Java, int is a 32-bit signed integer.

Example:
a = 5

32-bit binary representation of 5:

00000000 00000000 00000000 00000101
After applying **~a** (bitwise complement):
11111111 11111111 11111111 11111010
In decimals, this value is **-6**.
This is because in Java:
~N = -(N + 1)
So
~5 = -(5 + 1) = -6

Explanation: The bitwise complement operator ~ in Java inverts all 32 bits of an integer.

For any integer N, the result of ~N is equal to:
~N = -(N + 1)
Therefore, ~5 = -(5 + 1) = -6.

Here is a Java program demonstrating all bitwise operators.

```
public class Ge {
    public static void main(String[] args) {
        // Initial values
```

```
        int a = 5;
        int b = 7;

        // bitwise and
        // 0101 & 0111=0101 = 5
        System.out.println("a&b = " + (a & b));

        // bitwise or
        // 0101 | 0111=0111 = 7
        System.out.println("a|b = " + (a | b));

        // bitwise xor
        // 0101 ^ 0111=0010 = 2
        System.out.println("a^b = " + (a ^ b));

        // bitwise not
        // ~00000000 00000000 00000000 00000101=11111111 11111111 11111111
11111010
        // will give 2's complement (32 bit) of 5 = -6
        System.out.println("~a = " + ~a);

        // can also be combined with assignment operator to provide shorthand
        // assignment a=a&b
        a &= b;
        System.out.println("a= " + a);}}
```

Output

```
a&b = 5
a|b = 7
a^b = 2
~a = -6
a= 5
```

Reducing repeated code

At its simplest level, breaking software into smaller pieces can reduce repetition. For example:

```java
public static void drawShape() {
    System.out.println("+----+");
    System.out.println("|    |");
    System.out.println("|    |");
    System.out.println("+----+");
}

public static void main(String[] args) {
    drawShape();
    drawShape();
}
```

```java
public static void main(String[] args) {
    System.out.println("+----+");
    System.out.println("|    |");
    System.out.println("|    |");
    System.out.println("+----+");

    System.out.println("+----+");
    System.out.println("|    |");
    System.out.println("|    |");
    System.out.println("+----+");
}
```

Repeated code is bad because it is difficult to make changes too. If you want to change the code, you must change it in multiple places.

```java
public static void drawShape() {
        System.out.println("  /\\ ");
        System.out.println(" /  \\ ");
        System.out.println("/    \\");
        System.out.println("+----+");

}

public static void main(String[] args) {
    drawShape();
    drawShape();
}
```

```java
public static void main(String[] args) {
        System.out.println("  /\\ ");
        System.out.println(" /  \\ ");
        System.out.println("/    \\");
        System.out.println("+----+");

        System.out.println("  /\\ ");
        System.out.println(" /  \\ ");
        System.out.println("/    \\");
        System.out.println("+----+");
}
```

Model–view–controller (MVC)

The MVC (Model–View–Controller) design pattern **divides an application into three** separate components: **Model**, **View**, and **Controller**. This separation of concerns **improves code organization**, **maintainability**, and **scalability**. Each component handles a specific responsibility, making the application easier to modify and extend.

- **Model:** Manages application data and business logic.
- **View:** Handles the user interface and presentation of data.
- **Controller:** Processes user input and coordinates between Model and View.
- **Promotes modularity, reusability, and easier maintenance of the codebase**.

Here's a simple example of the MVC pattern in Java:

```
// Model
class Person {
    private String name;

    public String getName() {
        return name;     }

    public void setName(String name) {
        this.name = name;    } }

// View
class PersonView {
    public void printPersonDetails(String personName) {
        System.out.println("Person: " + personName);     } }

// Controller
class PersonController {
    private Person model;
    private PersonView view;

    public PersonController(Person model, PersonView view) {
        this.model = model;
        this.view = view;     }

    public void setPersonName(String name) {
        model.setName(name);     }
    public String getPersonName() {
        return model.getName();     }

    public void updateView() {
        view.printPersonDetails(model.getName());     } }

// Client code
public class MVCPatternDemo {
    public static void main(String[] args) {
        // Create model and view
        Person model = new Person();
        PersonView view = new PersonView();
```

```
// Create controller and bind model and view
PersonController controller = new PersonController(model, view);

// Set person name and update the view
controller.setPersonName("John Doe");
controller.updateView();    } }
```

In this example, the **Person class** represents the **model**, which stores the person's name. The **PersonView class** represents the **view**, responsible for **displaying** the person's details. The **PersonController class** is the controller that **handles user input**, **updates the model**, and refreshes the view.

Example

Design and implement an application using the MVC (Model–View–Controller) design pattern to **manage student information**. The application should store student details such as name and roll number, display these details to the user, and allow updates to the student data.

The Model should handle student data, the View should be responsible for displaying the student information, and the Controller should act as an intermediary between the Model and the View by updating data and refreshing the display when changes occur.