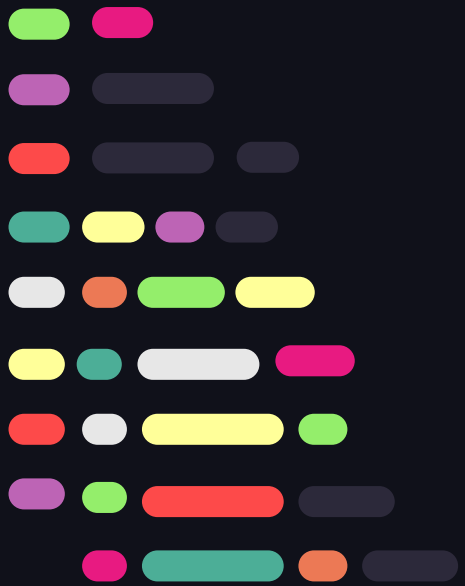
Java Programming Week-06

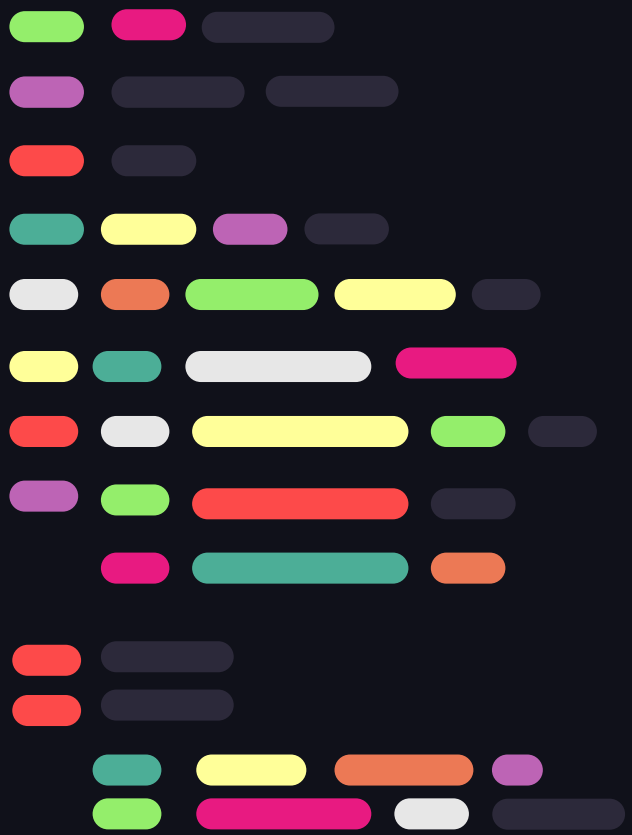
<Mohan De Zoysa –IIT JAVA CERTIFICATION>

Outline



Java file handling

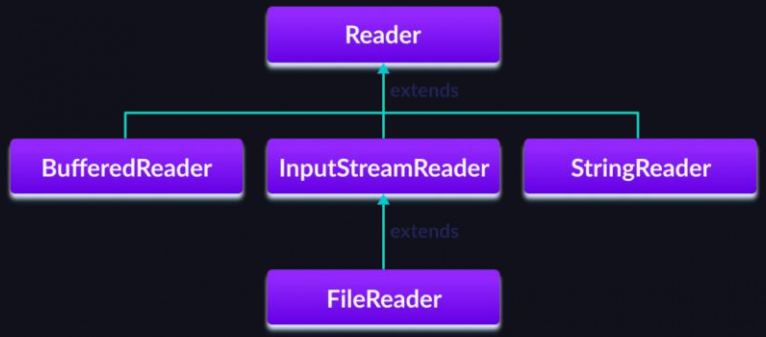
- Reading from and writing to files
- Byte streams and character streams
- File and directory operations
- Serialization and deserialization



File Handling in Java

File handling means performing various functions on a file, like reading, writing, editing, etc. Java provides us with a provision to perform these operations through programming, without having to open the file. In this article, we will look at all those properties of java programming and how to perform these operations using Java

In Java, there are several classes and libraries available for reading from and writing to files. Each option has its own advantages and disadvantages depending on the specific requirements of your application



Java Reader Class

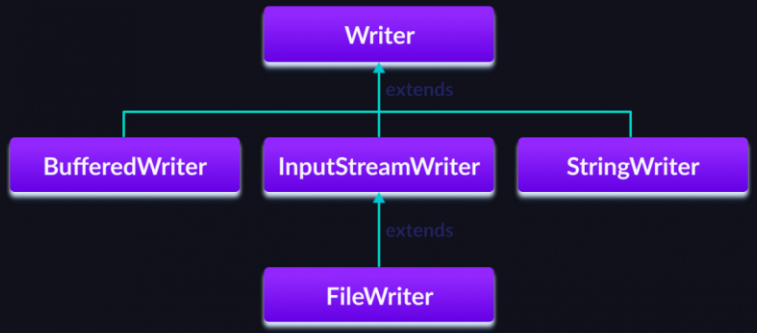
The Reader class of the java.io package is an abstract superclass that represents a stream of characters.

Since Reader is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.

Subclasses of Reader

In order to use the functionality of Reader, we can use its subclasses. Some of them are:

- InputStreamReader
- FileReader
- BufferedReader
- StringReader



Java Writer Class

The Writer class of the java.io package is an abstract superclass that represents a stream of characters.

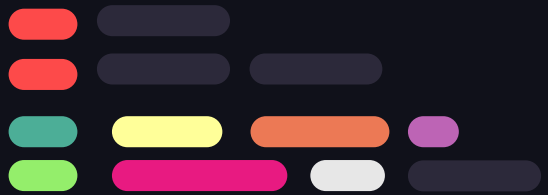
Since Writer is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.

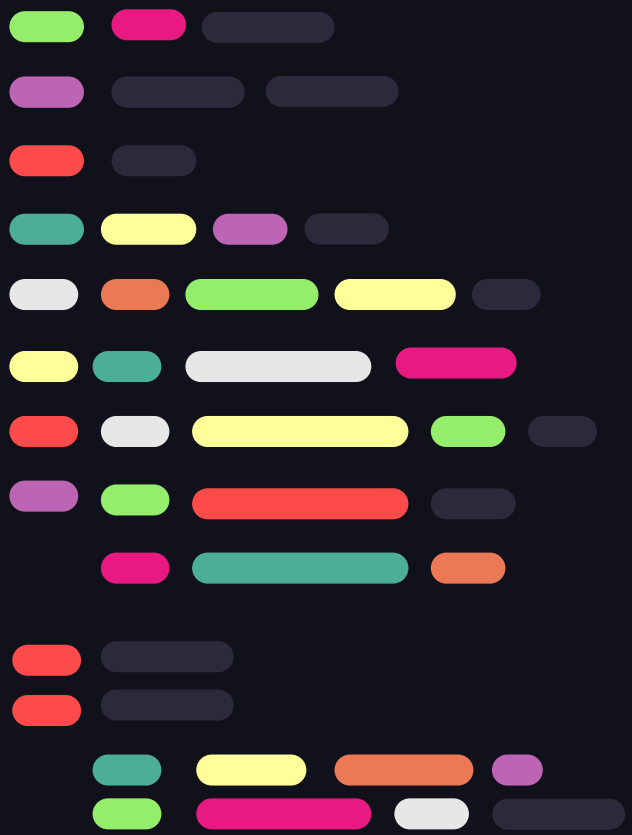
Subclasses of Writer

In order to use the functionality of the Writer, we can use its subclasses. Some of them are:

OutputStreamWriter
FileWriter
BufferedWriter
StringWriter

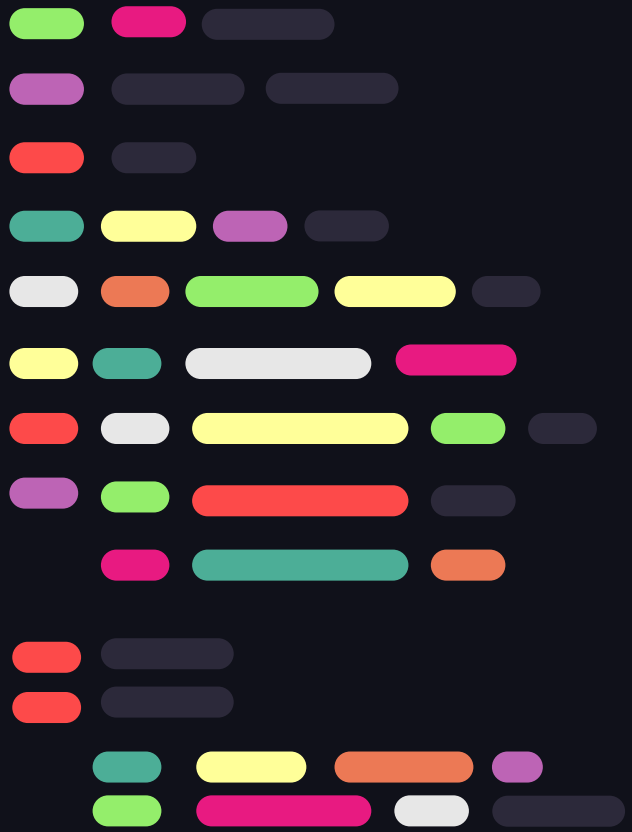
Java File Handling





Java File Handling

Java provides us with library classes and various methods to perform file handling easily. All these methods are present in the File Class of the java.io package.



Java File Handling

Stream is a concept of java that pipelines a sequence of objects to obtain the desired result.

A stream can not be called to be a data structure, rather it just takes input from the collection of I/O.

A stream can be classified into two types: Byte Stream and Character Stream.

Java Stream Class

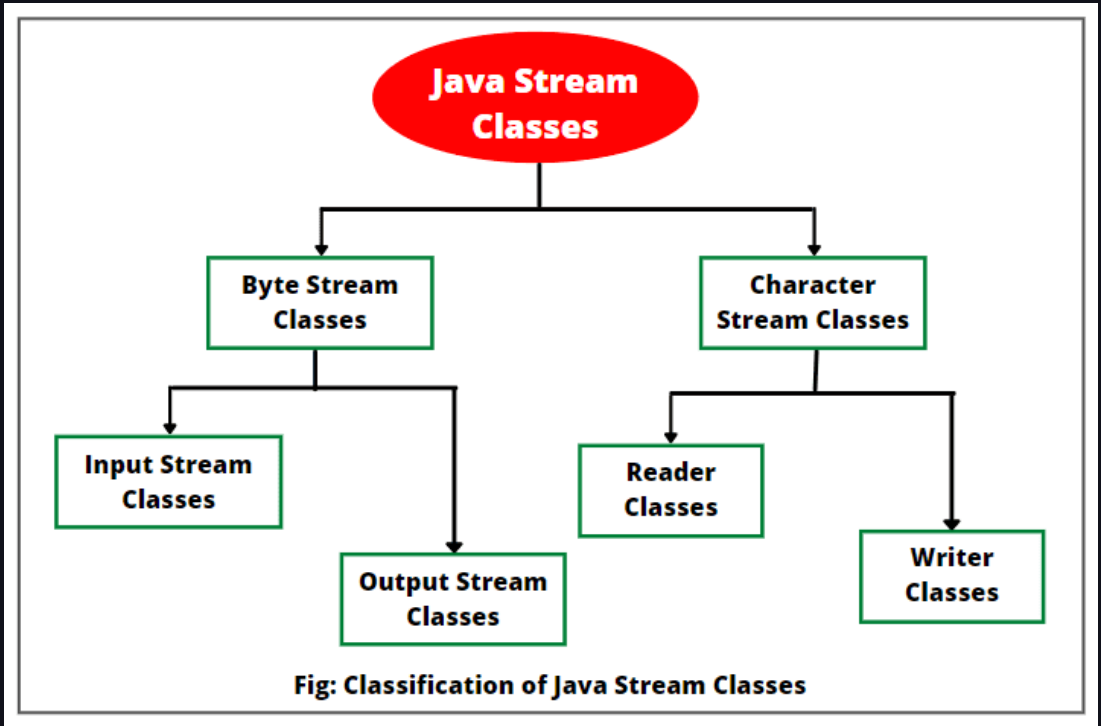
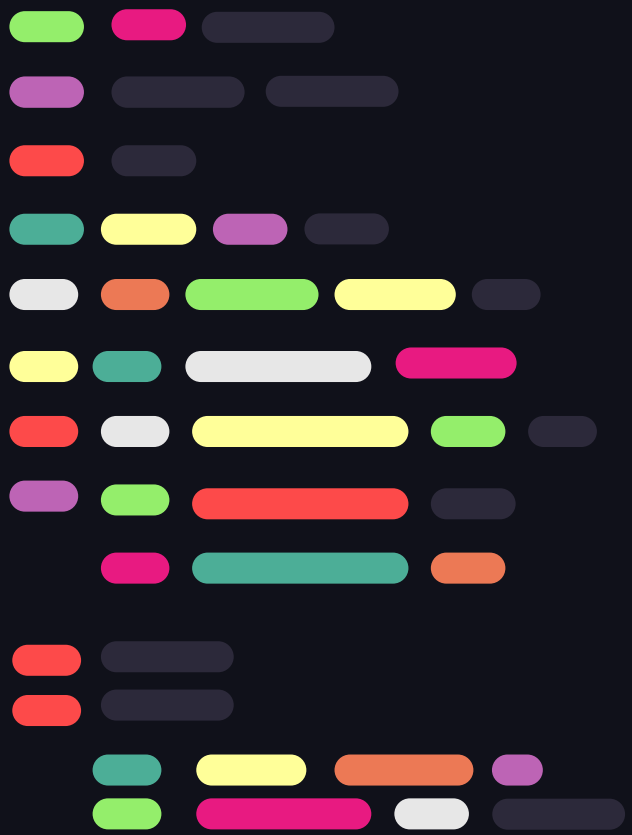


Fig: Classification of Java Stream Classes



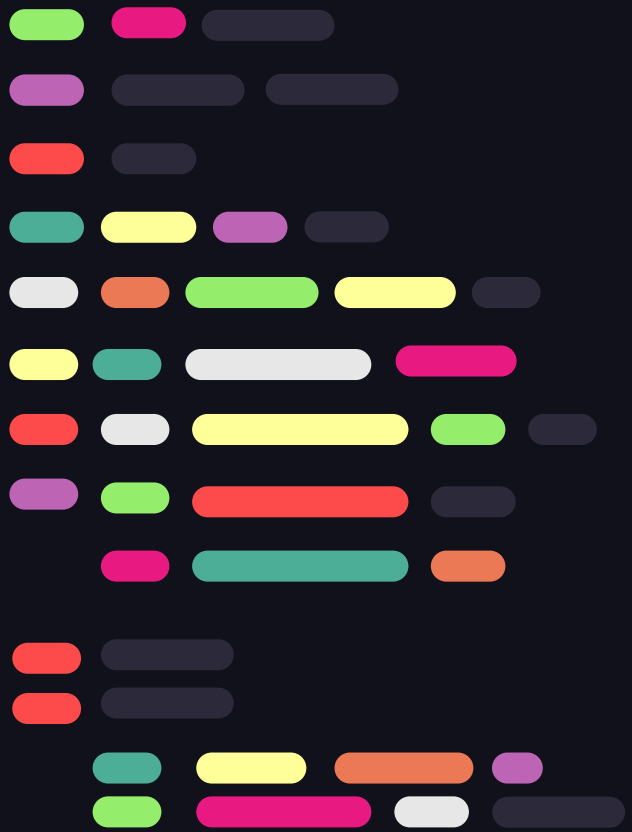
Byte Stream

The byte stream deals with mainly byte data. We know that one byte is equal to eight-bit. Thus, this stream mainly deals with 8bit of data. This stream performs an Input-output operation per 8bit of data. The byte stream contains two stream classes, Input Stream classes and Output Stream Classes.

Input Stream Classes: This stream helps take input(Read Data) from the collection in I/O File.

Output Stream Classes: This stream helps to give output(Write Data) into the collection in I/O File

The most commonly used Input and Output Stream Classes are FileInputStream and FileOutputStream.



Character Stream

There is also Character Stream which allows I/O operation on 16bit of Unicode data at a time. Character Stream takes 16 bits of data at a time. It is faster as it can take double the intake as compared to a byte stream. Character streams usually use Byte Stream classes to implement operations.

The two main classes used in Character Stream are FileReader and FileWriter.

The methods present in File class(java.io.File)

	Method Name	Return	Description of the method
1.	canRead()	Boolean	This method checks whether the file is readable or not.
2.	createNewFile()	Boolean	This method creates a new file in the desired path. The file created is generally empty.
3.	canWrite()	Boolean	This method checks whether the file is writable or not,i.e, not a read-only file.
4.	exists()	Boolean	This method verifies if the file asked for is present or not in the directory.
5.	delete()	Boolean	This method is used to delete a file from the directory.
6.	getName()	String	This method helps us find the name of a particular file from the directory.
7.	getAbsolutePath()	String	This method returns the absolute path of the given file.
8.	length()	Long	This method returns the size of a file in bytes.
9.	list()	String[]	This method returns an array, listing all the files present in the present working directory(PWD).
10.	mkdir()	Boolean	This Method stands for make directory. This method helps us create a new directory(Not a file).

Java File Operation - Using java FileWriter

We can use the createNewFile() method to create a new file using Java. If the method returns true, the file has been created successfully, else the file creation was unsuccessful.

```
package com.iit.java.FileHandling;
import java.io.File;
import java.io.IOException;
public class CreatingNewFile
{
    public static void main(String args[])
    {
        try {
            File fcreate = new File("G:\\Internship\\File Handling\\NewFile.txt");
            if (fcreate.createNewFile()) {
                System.out.println("File " + fcreate.getName() + " is created successfully.");
            }
            else {
                System.out.println("File is already exist in the directory.");
            }
        } catch (IOException exception) {
            System.out.println("An unexpected error is occurred.");
            exception.printStackTrace();
        }
    }
}
```

Java File Operation – Java Get File Information:

Through the various methods given in File class, we are able to get all sorts of information about a file. These methods can give information like name, length, path, read-only, write-only, etc.

```
package com.itt.java.FileHandling;
import java.io.File;
public class FileInformation
{
    public static void main(String[] args) {
        File fileinfo = new File("G:\\Internship\\File Handling\\NewFile.txt");
        if (fileinfo.exists()) {
            System.out.println("The name of the file is: " + fileinfo.getName());
            System.out.println("The absolute path of the file is: " + fileinfo.getAbsolutePath());
            System.out.println("Is file writeable: " + fileinfo.canWrite());
            System.out.println("Is file readable: " + fileinfo.canRead());
            System.out.println("The size of the file is: " + fileinfo.length());
        } else {
            System.out.println("The file does not exist.");
        }
    }
}
```

Java File Operation - Write into a Java File

write into a file using the byte stream class OutputStreamWriter or Character stream class FileWriter.

Using OutputStreamWriter

We can use the OutputStreamWriter class of the byte stream to write into a file. This class writes 8 bits of data at a time.
We should always remember to close the stream or else it might create dump memory.

Using java FileWriter

The character stream contains the FileWriter class, which can write 16-bits of data at a time into a file. This is a much quicker technique compared to OutputStreamWriter as 16 bits of data is written at a time.

Write into a File using OutputStreamWriter - Byte Stream

```java
package com.iit.java.FileHandling;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;

public class ByteStreamWrite
{
    public static void main(String[] args) {
        try {
            //Object Reference form Generic Class
           OutputStream outputStream =new FileOutputStream("D:\\mohan\\IIT\\Java\\files\\NewFile.txt");
            //Object Reference form FileOutputStream Class
           //FileOutputStream fileOutputStream =new FileOutputStream("D:\\mohan\\IIT\\Java\\files\\NewFile.txt");
           Writer fileWrite =new OutputStreamWriter(outputStream);
           fileWrite.write("Writing Using OutputStreamWriter!!!");
           fileWrite.close();

        } catch (Exception e) {
            e.getMessage();
        }
    }
}
```

Write into a file using FileWriter – Character stream

```java
package com.iit.java.FileHandling;
import java.io.FileWriter;
import java.io.IOException;
public class CharacterStreamWrite
{
    public static void main(String[] args) {
        try {
            FileWriter fwrite = new FileWriter("D:\\mohan\\IIT\\Java\\files\\NewFile.txt");
            fwrite.write("Written using FileWriter!!!");
            fwrite.close();
        } catch (IOException e) {
            System.out.println("Error While Writing!!!");
            e.printStackTrace();
        }
    }
}
```

Java File Operation - Reading from file

Similarly, like write, we can read a file using byte stream and character stream. In the byte-stream we use InputStreamreader and in the character stream, we have FileReader to read the contents of a file.

Using Java InputStreamReader

The InputStreamReader class is part of the java byte stream, it can read 8 bits of data at a time. After reading the data the file object should always be closed.

Using java FileReader

The FileReader is a class of the character stream, thus it reads 16 bits of data at a time. It is faster than InputStreamReader.

File Reading using InputStreamReader : byte stream

By creating an InputStreamReader on a FileInputStream and specifying the character encoding and byte-buffer size, you have more control over how the data is read and processed,

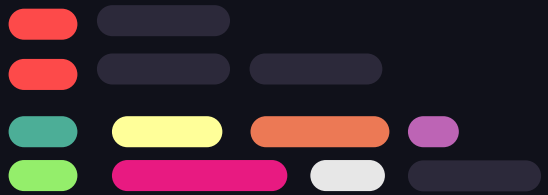
```
package com.iit.java.FileHandling;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.Reader;
import java.io.InputStreamReader;
public class ByteStreamRead
{
    public static void main(String[] args) {
        try  {
            InputStream fread = new FileInputStream("D:\\mohan\\IIT\\Java\\files\\NewFile.txt");
            String charsetName = "UTF-8"; // Specify your desired character encoding
            Reader fileReader = new InputStreamReader(fread,charsetName);
            int character;
            // Read characters one at a time and print them
            while ((character = fileReader.read()) != -1) {
                System.out.print((char) character);
            }
            fileReader.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

File Reading file using FileReader – character stream

```java
package com.iit.java.FileHandling;
import java.io.FileReader;
public class CharacterStreamRead
{
    public static void main(String args[])throws Exception
    {
        FileReader fileReader=new FileReader("D:\\mohan\\IIT\\Java\\files\\NewFile.txt");
        String charsetName = "UTF-8"; // Specify your desired character encoding
        int bufferSize = 1024; // Specify your desired byte-buffer size
        int character;
        // Read characters one at a time and print them
        while ((character = fileReader.read()) != -1) {
            System.out.print((char) character);
        }
        fileReader.close();
    }
}
```

{..

BufferedReader & BufferedWriter

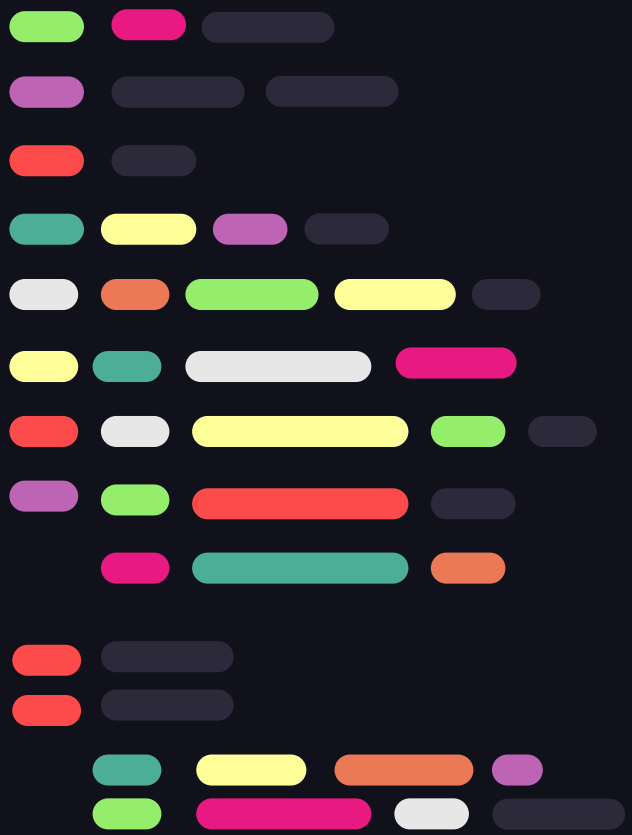


..}

Java BufferedWriter

The BufferedWriter class of the java.io package can be used with other writers to write data (in characters) more efficiently.

It extends the abstract class Writer.



Java BufferedWriter

The BufferedWriter maintains an internal buffer of 8192 characters.

During the write operation, the characters are written to the internal buffer instead of the disk. Once the buffer is filled or the writer is closed, the whole characters in the buffer are written to the disk.

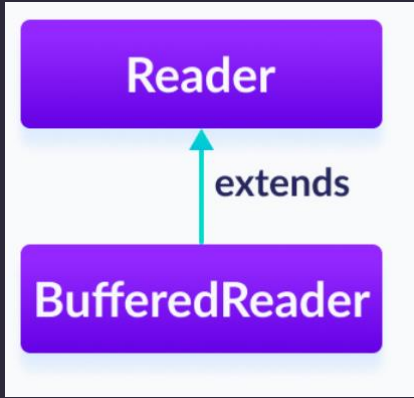
Hence, the number of communication to the disk is reduced. This is why writing characters is faster using BufferedWriter.

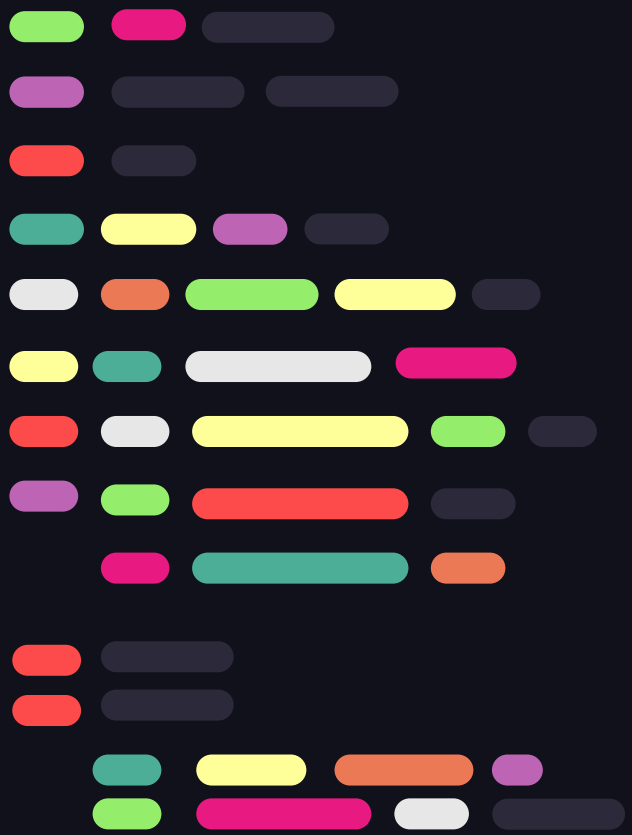
BufferedWriter Example

```
package com.iit.java.FileHandling;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class BufferWrite {

    public static void main(String []arg) throws IOException {
            BufferedWriter writer = new BufferedWriter(new FileWriter("D:\\mohan\\IIT\\Java\\files\\NewFile.txt”));
            writer.write("The greatest glory in living lies not in never falling,");
            writer.newLine();
            writer.write("but in rising every time we fall");
            writer.close();
        }
}
```

Java BufferedReader

The BufferedReader class of the java.io package can be used with other readers to read data (in characters) more efficiently.
It extends the abstract class Reader.





Java BufferedReader

The BufferedReader maintains an internal buffer of 8192 characters.

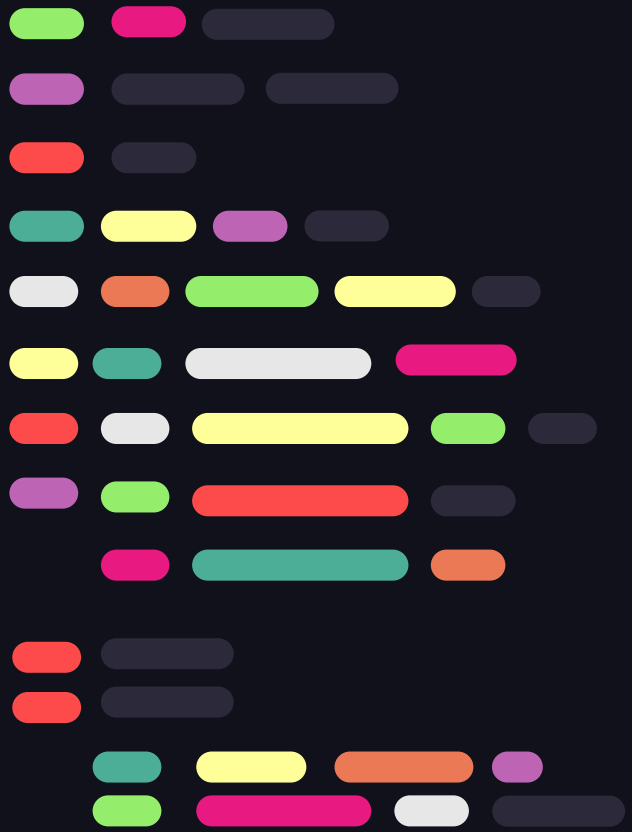
During the read operation in BufferedReader, a chunk of characters is read from the disk and stored in the internal buffer. And from the internal buffer characters are read individually.

Hence, the number of communication to the disk is reduced. This is why reading characters is faster using BufferedReader.

BufferedReader Example

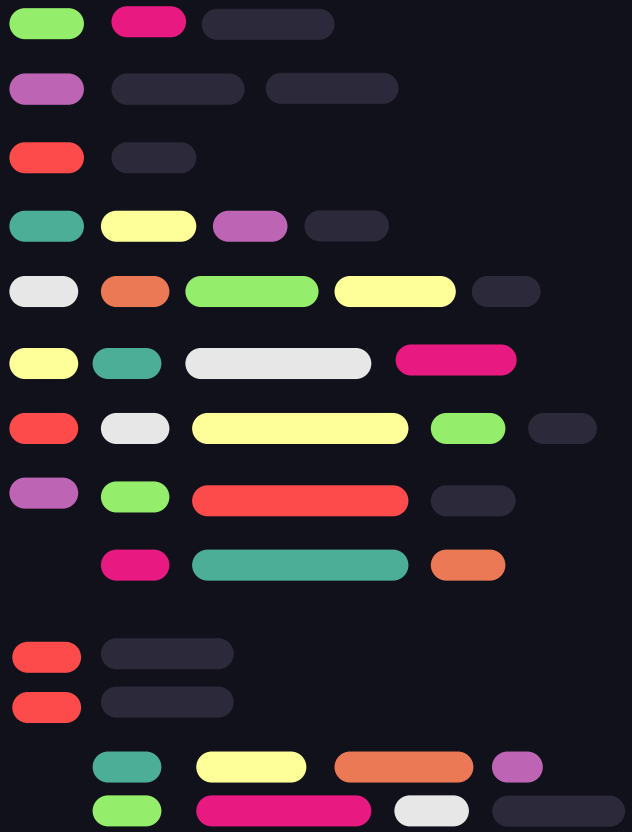
```
package com.iit.java.FileHandling;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class BufferRead {
    public static void main(String []arg) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader("D:\\mohan\\IIT\\Java\\files\\NewFile.txt"));
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
    }
}
```


Serialization and deserialization



Serialization and deserialization

Serialization and deserialization are processes used in Java (and in other programming languages) to convert complex objects or data structures into a format that can be easily stored, transmitted, or reconstructed later. These processes are commonly used for tasks like saving objects to files, sending objects over a network, or storing them in a database.



Serialization

Purpose:

Serialization is the process of converting an object's state (its fields and data) into a byte stream. The resulting byte stream can be saved to a file, sent over a network, or stored in a database.

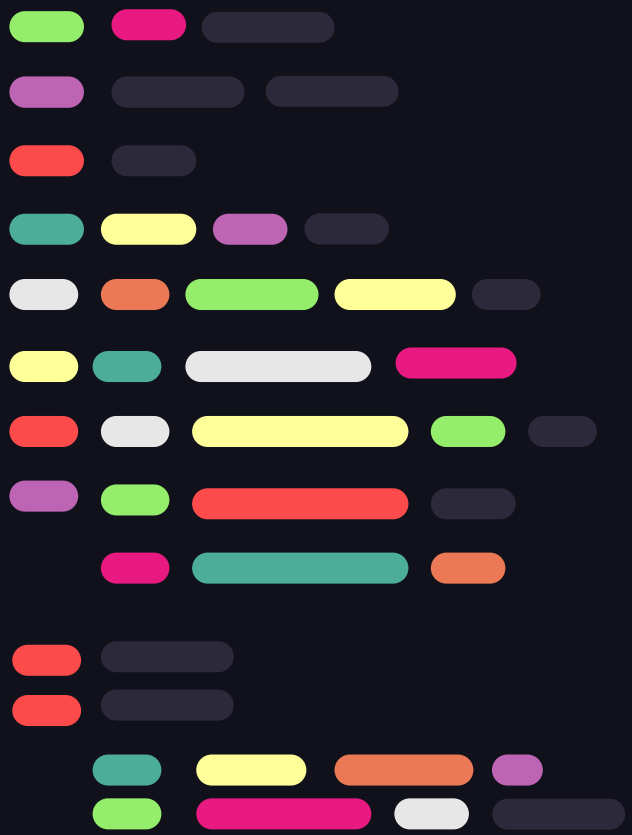
Implementation

In Java, serialization is achieved by implementing the Serializable interface. This interface marks a class as serializable and provides methods for the serialization process.

The ObjectOutputStream class is used to serialize objects. It writes the object's state to an output stream, which can be a file, network socket, or any other output source.

Serialization Example

```
package com.iit.java.FileHandling;
import java.io.*;
public class DemoSerialization {
    public static void main(String []arg) throws IOException, ClassNotFoundException {
        OutputStream outputStream = new FileOutputStream("D:\\mohan\\IIT\\Java\\files\\objectfile.txt");

        ObjectOutputStream objectOutputStream =new ObjectOutputStream(outputStream);

        Student studentOne =new Student();
        studentOne.setEmail("kevin@gmail.com");
        studentOne.setName("Kevin");

        objectOutputStream.writeObject(studentOne);
        objectOutputStream.close();
        outputStream.close();
    }
}
```



Deserialization

Purpose

Deserialization is the process of recreating an object from a previously serialized byte stream.

It allows you to reconstruct the object's state and behavior as it was when it was serialized.

Implementation

In Java, deserialization is also done using the ObjectInputStream class, which reads a byte stream and reconstructs the original object.

The class being deserialized must implement the Serializable interface for successful deserialization.

Serialization Example

```
package com.iit.java.FileHandling;
import java.io.*;
public class DemoDeSerialization {
    public static void main(String []arg) throws IOException, ClassNotFoundException {

        InputStream inputStream =new FileInputStream("D:\\mohan\\IIT\\Java\\files\\objectfile.txt");
        ObjectInputStream objectInputStream =new ObjectInputStream(inputStream);

        Student studentOne =new Student();

        studentOne= (Student)objectInputStream.readObject();
        System.out.println("Name : "+studentOne.getName());
        System.out.println("Email : "+studentOne.getEmail());

        objectInputStream.close();
        inputStream.close();
    }
}
```

Exercise -02

Exercise 4: Serialize and Deserialize Objects

Create a class Student that implements Serializable with fields such as name, age, and grade.

Write a Java program to serialize multiple Student objects into a file named "students.ser" and then deserialize them, displaying the student information on the console.

Exercise -02

CSV File Handling Write a Java program that generates a CSV file named "exercise5.csv" containing a list of products. Each product should have a name, price, and quantity.

Use commas to separate the values in each column.
Use new line for each new product

Summary

FileReader and FileWriter:

Advantages:

- Convenient for reading and writing textfiles.
- Automatically handle character encoding (such as UTF-8).

Disadvantages:

- Less efficient for reading/writing large binary files.

FileInputStream and FileOutputStream:

Advantages:

- Low-level, basic classes for reading and writing binary data.
- Suitable for reading and writing raw binary files.

Disadvantages:

- You need to handle data conversion and buffering manually.
- Not very convenient for text-based file operations.

BufferedReader and BufferedWriter:

Advantages:

- Efficient for reading and writing text files due to buffering.
- Provides methods for reading lines.

Disadvantages:

- Limited to text-based file operations.