

**Day-11**

# Agenda

- IO Streams
- Functional Programming
- Lambda
- Functional interfaces
- default & static methods

# IO Streams

- Java performs I/O through **Streams**.
- A Stream is linked to a physical layer by java I/O system to make input and output operation in java.
- An **input stream** is used to read data from the source. And, an **output stream** is used to write data to the destination.
- A stream can be defined as a sequence of data. There are two kinds of Streams –
  - **InPutStream** – The InputStream is used to read data from a source.
  - **OutPutStream** – The OutputStream is used for writing data to a destination.

- In general, a stream means continuous flow of data. Streams are clean way to deal with input/output without having every part of your code understand the physical.
- Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O.
- Java encapsulates Stream under **java.io** package. Java defines two types of streams. They are,
  - 1.Byte Stream** : It provides a convenient means for handling input and output of byte.
  - 2.Character Stream** : It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

- A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.
- In Java, 3 streams are created for us automatically. All these streams are attached with the console.
  - 1) System.out: standard output stream
  - 2) System.in: standard input stream
  - 3) System.err: standard error stream

➤ Example:-

```
System.out.println("simple message");
```

```
System.err.println("error message");
```


# InputStream Class

- The InputStream class of the java.io package is an abstract superclass that represents an input stream of bytes.
- Since InputStream is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.
- The Java.io.InputStream class is the superclass of all classes representing an input stream of bytes.
- Applications that need to define a subclass of InputStream must always provide a method that returns the next byte of input.
- InputStream class is the superclass of all the io classes i.e. representing an input stream of bytes. Applications that are defining subclass of InputStream must provide method, returning the next byte of input.



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

- `FileInputStream`
- `ByteArrayInputStream`
- `ObjectInputStream`

The picture can't be displayed.

## ➤ Create an InputStream

- In order to create an InputStream, we must import the java.io.InputStream package first.
- Once we import the package, here is how we can create the input stream.

## ➤ Syntax:-

```
InputStream object1 = new FileInputStream();
```

- We have created an input stream using FileInputStream. It is because InputStream is an abstract class. Hence we cannot create an object of InputStream.

- We can also create an input stream from other subclasses of InputStream.
- The InputStream class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:
  - read() - reads one byte of data from the input stream
  - read(byte[] array) - reads bytes from the stream and stores in the specified array
  - available() - returns the number of bytes available in the input stream
  - mark() - marks the position in the input stream up to which data has been read
  - reset() - returns the control to the point in the stream where the mark was set

- `markSupported()` - checks if the `mark()` and `reset()` method is supported in the stream
- `skip()` - skips and discards the specified number of bytes from the input stream
- `close()` - closes the input stream

# OutputStream

- Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.
- OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes.
- An output stream accepts output bytes and sends them to some sink.
- The `Java.io.OutputStream` class is the superclass of all classes representing an output stream of bytes.
- Applications that need to define a subclass of OutputStream must always provide at least a method that writes one byte of output.

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

- `FileOutputStream`
- `ByteArrayOutputStream`
- `ObjectOutputStream`

The picture can't be displayed.

## ➤ **Create an OutputStream**

- In order to create an OutputStream, we must import the java.io.OutputStream package first.
- Once we import the package, here is how we can create the output stream.

## ➤ **Syntax:-**

OutputStream object = new FileOutputStream();

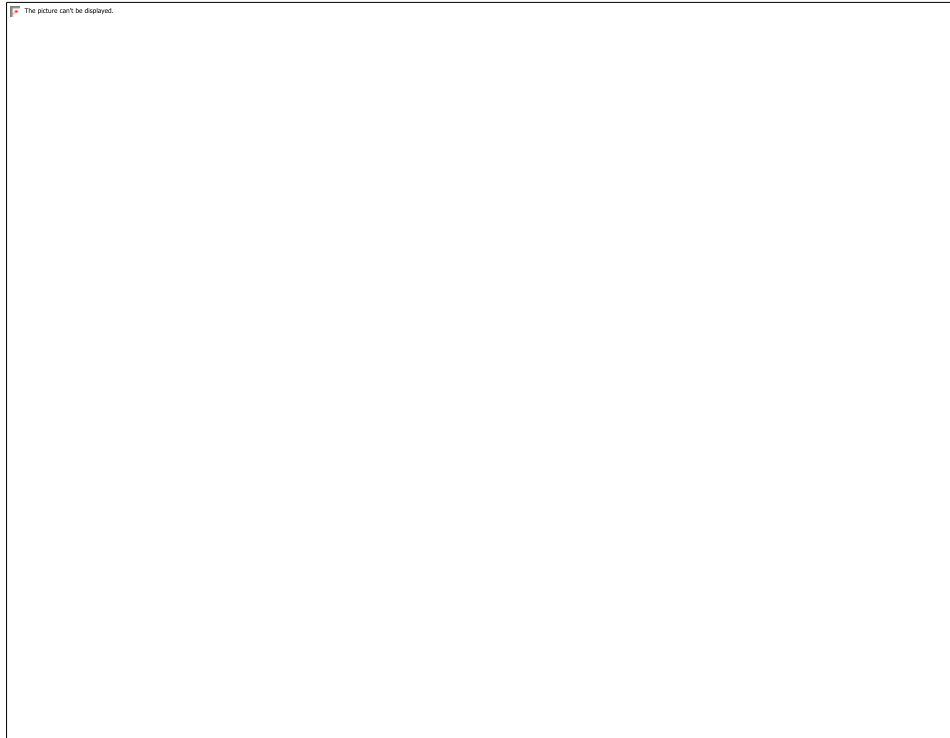
- we have created an object of output stream using FileOutputStream. It is because OutputStream is an abstract class, so we cannot create an object of OutputStream.

- We can also create the output stream from other subclasses of the OutputStream class.
  
- The OutputStream class provides different methods that are implemented by its subclasses. Here are some of the methods:
  - write() - writes the specified byte to the output stream
  - write(byte[] array) - writes the bytes from the specified array to the output stream
  - flush() - forces to write all data present in output stream to the destination
  - close() - closes the output stream



# FileInputStream

- The FileInputStream class of the java.io package can be used to read data (in bytes) from files.
- It extends the InputStream abstract class.



## Create a FileInputStream

- In order to create a file input stream, we must import the `java.io.FileInputStream` package first.
- Once we import the package, here is how we can create a file input stream in Java.
  - 1. Using the path to file**  
`FileInputStream input = new FileInputStream(stringPath);`
  - 2. Using an object of the file**  
`FileInputStream input=new FileInputStream(File fileObject);`

➤ The `FileInputStream` class provides implementations for different methods present in the `InputStream` class.

➤ `read()` Method

- `read()` - reads a single byte from the file
- `read(byte[] array)` - reads the bytes from the file and stores in the specified array
- `read(byte[] array, int start, int length)` - reads the number of bytes equal to length from the file and stores in the specified array starting from the position start

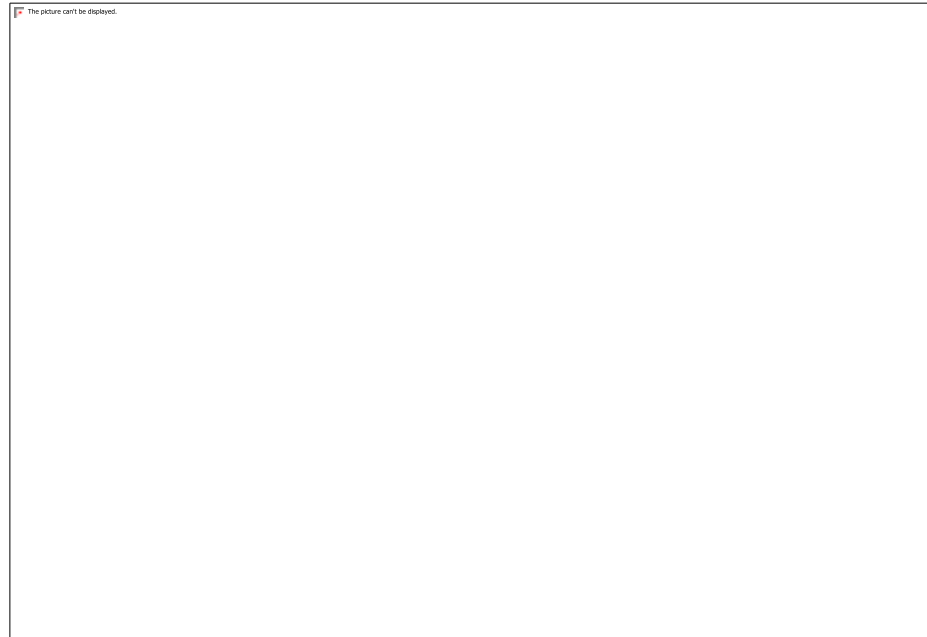
➤ `available()` Method

➤ `skip()` Method

➤ `close()` Method

# FileOutputStream

- The FileOutputStream class of the java.io package can be used to write data (in bytes) to the files.
- It extends the OutputStream abstract class.



## Create a FileOutputStream

➤ In order to create a file output stream, we must import the `java.io.FileOutputStream` package first. Once we import the package, here is how we can create a file output stream in Java.

### 1. Using the path to file

```
FileOutputStream output=new FileOutputStream(String path, boolean value);
```

```
FileOutputStream output = new FileOutputStream(String path);
```

### 2. Using an object of the file

```
FileOutputStream output = new FileOutputStream(File fileObject);
```

➤ The `FileOutputStream` class provides implementations for different methods present in the `OutputStream` class.

➤ `write()` Method

- `write()` - writes the single byte to the file output stream
- `write(byte[] array)` - writes the bytes from the specified array to the output stream
- `write(byte[] array, int start, int length)` - writes the number of bytes equal to length to the output stream from an array starting from the position start

➤ `flush()` Method

➤ `close()` Method

➤ `finalize()`

➤ `getChannel()`

➤ `getFD()`

# Java ByteArrayInputStream Class

- The ByteArrayInputStream class of the java.io package can be used to read an array of input data (in bytes).
- It extends the InputStream abstract class.
- In ByteArrayInputStream, the input stream is created using the array of bytes. It includes an internal array to store data of that particular byte array.



## Create a ByteArrayInputStream

- In order to create a byte array input stream, we must import the `java.io.ByteArrayInputStream` package first. Once we import the package, here is how we can create an input stream.

```
ByteArrayInputStream input = new ByteArrayInputStream(byte[] arr);
```

- we can also create the input stream that reads only some data from the array.

```
ByteArrayInputStream input=new ByteArrayInputStream(byte[] arr, int start, int length);
```



➤ The `ByteArrayInputStream` class provides implementations for different methods present in the `InputStream` class.

➤ `read()` Method

- `read()` - reads the single byte from the array present in the input stream
- `read(byte[] array)` - reads bytes from the input stream and stores in the specified array
- `read(byte[] array, int start, int length)` - reads the number of bytes equal to length from the stream and stores in the specified array starting from the position start

➤available()

➤skip()

➤close()

➤finalize()

➤mark()

➤reset()

➤markSupported()

# Java ByteArrayOutputStream Class

- The ByteArrayOutputStream class of the java.io package can be used to write an array of output data (in bytes).
- It extends the OutputStream abstract class.
- In ByteArrayOutputStream maintains an internal array of bytes to store the data.



## Create a ByteArrayOutputStream

- In order to create a byte array output stream, we must import the `java.io.ByteArrayOutputStream` package first. Once we import the package, here is how we can create an output stream.

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
```

- Here, we have created an output stream that will write data to an array of bytes with default size 32 bytes. However, we can change the default size of the array.

```
ByteArrayOutputStream out = new ByteArrayOutputStream(int size);
```

- Here, the size specifies the length of the array.

➤ The `ByteArrayOutputStream` class provides the implementation of the different methods present in the `OutputStream` class.

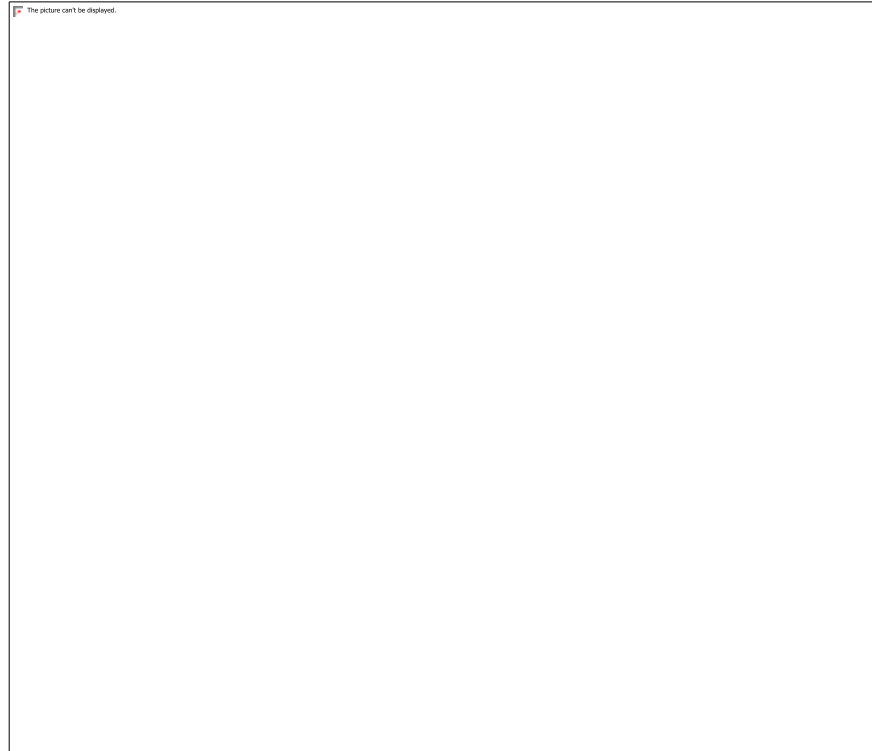
➤ `write()` Method

- `write(int byte)` - writes the specified byte to the output stream
- `write(byte[] array)` - writes the bytes from the specified array to the output stream
- `write(byte[] arr, int start, int length)` - writes the number of bytes equal to length to the output stream from an array starting from the position start
- `writeTo(ByteArrayOutputStream out1)` - writes the entire data of the current output stream to the specified output stream

- `toByteArray()` - returns the array present inside the output stream
- `toString()` - returns the entire data of the output stream in string form
- `close()` Method- To close the output stream, we can use the `close()` method.
- `size()` returns the size of the array in the output stream
- `flush()` clears the output stream

# Java ObjectInputStream Class

- The ObjectInputStream class of the java.io package can be used to read objects that were previously written by ObjectOutputStream.
- It extends the InputStream abstract class.



- Create an `ObjectInputStream`
- In order to create an object input stream, we must import the `java.io.ObjectInputStream` package first. Once we import the package, here is how we can create an input stream.

```
FileInputStream fileStream = new FileInputStream(String file);  
ObjectInputStream objStream = new ObjectInputStream(fileStream);
```

- In the above example, we have created an object input stream named `objStream` that is linked with the file input stream named `fileStream`.
- Now, the `objStream` can be used to read objects from the file.



➤ The `ObjectInputStream` class provides implementations of different methods present in the `InputStream` class.

➤ `read()` Method

- `read()` - reads a byte of data from the input stream
- `readBoolean()` - reads data in boolean form
- `readChar()` - reads data in character form
- `readInt()` - reads data in integer form
- `readObject()` - reads the object from the input stream

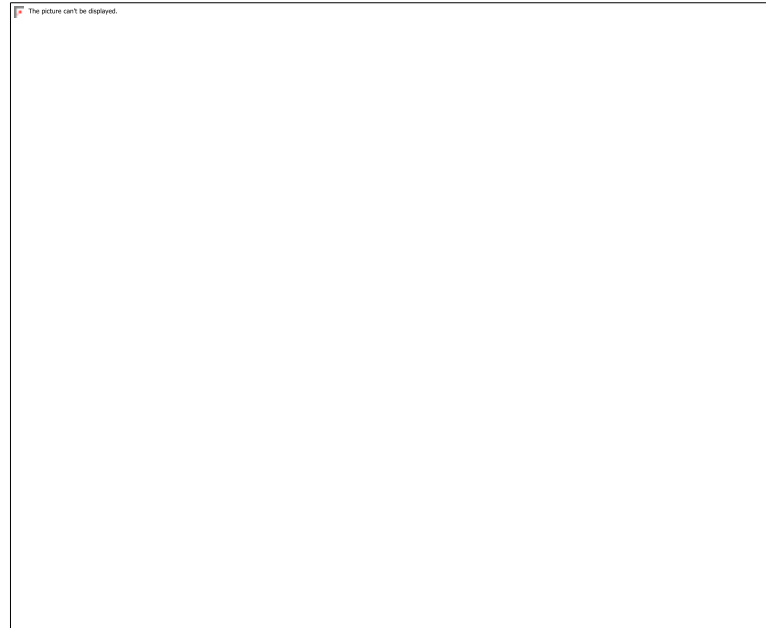
➤ `available()` returns the available number of bytes in the input stream

➤ `mark()` marks the position in input stream up to which data has been read

- `reset()` :-returns the control to the point in the input stream where the mark was set
- `skipBytes()` :-skips and discards the specified bytes from the input stream
- `close()` :-closes the object input stream

# Java ObjectOutputStream

- The ObjectOutputStream class of the java.io package can be used to write objects that can be read by ObjectInputStream.
- It extends the OutputStream abstract class.



## Create an ObjectOutputStream

- In order to create an object output stream, we must import the `java.io.ObjectOutputStream` package first. Once we import the package, here is how we can create an output stream.

```
// Creates a FileOutputStream where objects from ObjectOutputStream are written  
FileOutputStream fileStream = new FileOutputStream(String file);
```

```
// Creates the ObjectOutputStream
```

```
ObjectOutputStream objStream = new ObjectOutputStream(fileStream);
```

➤ The `ObjectOutputStream` class provides implementations for different methods present in the `OutputStream` class.

➤ `write()` Method

- `write()` - writes a byte of data to the output stream
- `writeBoolean()` - writes data in boolean form
- `writeChar()` - writes data in character form
- `writeInt()` - writes data in integer form
- `writeObject()` - writes object to the output stream

➤ `flush()`

➤ `drain()`

➤ `close()`

# Functional Programming

- The term Java functional programming refers to functional programming in Java.
- Functional programming in Java has not been easy historically, and there were even several aspects of functional programming that were not even really possible in Java.
- In Java 8 Oracle made an effort to make functional programming easier, and this effort did succeed to some extent.
- Functional programming contains the following key concepts:
  - Functions as first class objects
  - Pure functions
  - Higher order functions

- Pure functional programming has a set of rules to follow too:
  - No state
  - No side effects
  - Immutable variables
  - Favour recursion over looping
  
- Functional programming is a programming style in which computations are codified as functional programming functions.
  
- Functional programming languages are declarative, meaning that a computation's logic is expressed without describing its control flow. In declarative programming, there are no statements. Instead, programmers use expressions to tell the computer what needs to be done, but not how to accomplish the task.



# **Lambda expression**

- It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.
- The Lambda expression is used to provide the implementation of an interface which has functional interface.
- It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.
- Java lambda expression is treated as a function, so compiler does not create .class file.
- Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an annotation `@FunctionalInterface`, which is used to declare an interface as functional interface.

## ➤ Java Lambda Expression Syntax

(argument-list) -> {body}

## ➤ Java lambda expression is consisted of three components.

- 1) Argument-list: It can be empty or non-empty as well.
- 2) Arrow-token: It is used to link arguments-list and body of expression.
- 3) Body: It contains expressions and statements for lambda expression.

- **Example:-**

```
public class Main {  
    public static void main(String[] args) {  
        ArrayList<Integer> numbers = new ArrayList<Integer>();  
        numbers.add(5);  
        numbers.add(9);  
        numbers.add(8);  
        numbers.add(1);  
        Consumer<Integer> method = (n) -> { System.out.println(n); };  
        numbers.forEach( method );  
    }  
}
```

# Functional interfaces

- A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit.
- From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.
- A functional interface can have any number of default methods. **Runnable**, **ActionListener**, **Comparable** are some of the examples of functional interfaces.
- A functional interface in Java is an interface that contains only a single abstract (unimplemented) method.
- A functional interface can contain default and static methods which do have an implementation, in addition to the single unimplemented method.

➤ Here is a Java functional interface example:

```
public interface MyFunctionalInterface {  
    public void execute();  
}
```

# static methods

- The static keyword is used to create methods that will exist independently of any instances created for the class.
- Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.
- Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.



- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

*Thanks.....*