Lambda Expressions and **Functional Programming** in Java were introduced as part of **Java 8** to support functional programming paradigms, improve code readability, and simplify the development of certain constructs.

---

1. What are Lambda Expressions?

Lambda Expressions provide a concise way to represent **anonymous functions** (functions without a name). They are often used to implement **functional interfaces**.

Syntax of a Lambda Expression

The general syntax is:

java

Copy code

```
(parameters) -> { body of the expression }
```

- If only **one parameter** → parameter -> expression
- If only **one statement** → {} can be omitted.

Example

java

Copy code

```
// Without Lambda (Anonymous class)
Runnable r1 = new Runnable() {
   @Override
   public void run() {
      System.out.println("Hello World!");
   }
};


// With Lambda Expression
Runnable r2 = () -> System.out.println("Hello World!");
```

---

2. Functional Interface

A functional interface is an interface with **only one abstract method**. Lambda expressions can be used to implement these interfaces.

@FunctionalInterface Annotation

You can annotate functional interfaces to enforce the rule of a single abstract method.

Example:

java

Copy code

```
@FunctionalInterface
interface MyFunctionalInterface {
   void display();
}
```

Using Lambda:

java

Copy code

```
MyFunctionalInterface mfi = () -> System.out.println("Hello from Lambda!");
mfi.display();
```

---

3. Built-In Functional Interfaces

Java provides several functional interfaces under the java.util.function package.

1. **Predicate**: Used for conditions and filtering.
2. **Function**: Used for transformations or mapping.
3. **Consumer**: Used for actions or side effects.
4. **Supplier**: Used to generate or supply values.

Interface	**Abstract Method**	**Description**
Predicate<T>	boolean test(T t)	Takes an input and returns true/false.
Consumer<T>	void accept(T t)	Performs an action on the given input.
Supplier<T>	T get()	Supplies a value without any input.
Function<T,R>	R apply(T t)	Takes an input and returns a result.

Examples

1. **Predicate Example**

java

Copy code

```
Predicate<Integer> isEven = (n) -> n % 2 == 0;
System.out.println(isEven.test(4));  // true
```

2. **Consumer Example**

java

Copy code

```
Consumer<String> printUpperCase = (str) -> System.out.println(str.toUpperCase());
printUpperCase.accept("hello");  // HELLO
```

3. **Supplier Example**

java

Copy code

```
Supplier<Double> randomNumber = () -> Math.random();

System.out.println(randomNumber.get()); // Random number
```

4. **Function Example**

java

Copy code

```
Function<String, Integer> stringLength = (str) -> str.length();

System.out.println(stringLength.apply("Java")); // 4
```

---

4. Stream API and Functional Programming

The **Stream API** uses functional programming principles, like **map**, **filter**, **reduce**, etc., to process collections in a declarative way.

Stream Example

java

Copy code

```
import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;


public class StreamExample {

  public static void main(String[] args) {

    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);


    // Filtering and mapping using Lambda

    List<Integer> evenSquares = numbers.stream()

                    .filter(n -> n % 2 == 0)  // Predicate

                    .map(n -> n * n)        // Function

                    .collect(Collectors.toList());


    System.out.println(evenSquares); // [4, 16]

  }

}
```

---

5. Method References

Method References provide a shorthand notation to write lambda expressions for methods.

Syntax

- Class::staticMethod
- object::instanceMethod
- Class::instanceMethod
- Class::new (Constructor reference)

Example

java

Copy code

```
import java.util.Arrays;
import java.util.List;

public class MethodReferenceExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Jane", "Smith");

        // Using Lambda
        names.forEach(name -> System.out.println(name));

        // Using Method Reference
        names.forEach(System.out::println);
    }
}
```

---

6. Benefits of Lambda Expressions

1. **Simpler code**: Reduces boilerplate code.
2. **Improved readability**: Code is more concise and expressive.
3. **Functional style programming**: Allows declarative programming via Stream API.
4. **Encourages immutability**: Better suited for multi-threading.

---

7. Key Notes

1. **Only for functional interfaces**: Lambdas can only be used where a functional interface is expected.

2. **Effectively final variables**: Variables used in lambdas must be "effectively final."
3. **Scope**: Lambda expressions can access outer class members and local variables.

Conclusion

Lambda expressions and functional programming in Java provide a powerful way to simplify and express code. They are particularly useful in collections, Stream API operations, and cases where functional interfaces are needed.

Java IO, NIO, and Networking

Java provides comprehensive support for **Input/Output (I/O)** operations and **Networking** using its core libraries. With the introduction of **NIO (New IO)**, Java has made I/O operations faster and more flexible.

Let's break this into three parts:

1. **Java IO (Stream-based IO)**
2. **Java NIO (Non-blocking IO)**
3. **Java Networking**

---

1. Java IO (Input/Output)

Java IO is stream-based, synchronous, and blocking. It uses **streams** to perform reading and writing operations.

Key Concepts in IO

- **Stream**: A sequence of data (InputStream for reading, OutputStream for writing).
- **Blocking IO**: Execution blocks the thread until I/O is complete.
- **Hierarchy**: Java IO includes classes for **byte streams**, **character streams**, and **buffered streams**.

Main Classes in IO

Type	Input (Read)	Output (Write)	Purpose
Byte Stream	InputStream	OutputStream	Handles binary data (bytes).
Character Stream	Reader	Writer	Handles text data (characters).
Buffered Stream	BufferedInputStream	BufferedOutputStream	Efficient reading/writing using buffers.
File Operations	FileInputStream	FileOutputStream	File-based byte operations.
Object Serialization	ObjectInputStream	ObjectOutputStream	Reads/Writes objects.

IO Example: File Reading/Writing

Byte Stream Example:

java

Copy code

```
import java.io.FileInputStream;

import java.io.FileOutputStream;
import java.io.IOException;


public class ByteStreamExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("input.txt");
             FileOutputStream fos = new FileOutputStream("output.txt")) {
            int content;
            while ((content = fis.read()) != -1) {
                fos.write(content); // Write to output file
            }
            System.out.println("File copied successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Character Stream Example:

java

Copy code

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


public class CharStreamExample {
    public static void main(String[] args) {
        try (FileReader reader = new FileReader("input.txt");
             FileWriter writer = new FileWriter("output.txt")) {
            int data;
            while ((data = reader.read()) != -1) {
                writer.write(data); // Write character by character
            }
            System.out.println("File copied successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

---

2. Java NIO (New IO)

Java NIO was introduced in **Java 1.4** to improve I/O performance. It provides a **non-blocking, buffer-based** I/O system and better support for file operations and networking.

Key Features of NIO

1. **Buffers**: Data is read/written through buffers (ByteBuffer, CharBuffer).
2. **Channels**: A channel represents a connection to a data source (e.g., file, network).
3. **Selectors**: For managing multiple channels using a single thread (non-blocking I/O).
4. **Non-Blocking**: I/O operations do not block threads.

Main NIO Classes

Feature	Classes	Description
Buffers	ByteBuffer, CharBuffer	Data containers for NIO operations.
Channels	FileChannel, SocketChannel	Connections to I/O sources (file, socket).
Selectors	Selector, SelectionKey	Multiplex multiple channels using a thread.
Files	Files, Paths (Java NIO 2)	Improved file handling.

NIO Example: File Copy using Channels

java

Copy code

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;

public class NIOFileCopy {
    public static void main(String[] args) {
        Path source = Path.of("input.txt");
        Path destination = Path.of("output.txt");
        try (FileChannel sourceChannel = FileChannel.open(source, StandardOpenOption.READ);
             FileChannel destChannel = FileChannel.open(destination, StandardOpenOption.WRITE,
StandardOpenOption.CREATE)) {


            ByteBuffer buffer = ByteBuffer.allocate(1024); // Allocate buffer
            while (sourceChannel.read(buffer) > 0) {
                buffer.flip(); // Switch to read mode
                destChannel.write(buffer); // Write to destination
                buffer.clear(); // Clear buffer for next read
            }
            System.out.println("File copied successfully using NIO.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

---

3. Java Networking

Java provides the java.net package to handle networking. It supports the following protocols:

- **TCP/IP** (Transmission Control Protocol / Internet Protocol)
- **UDP** (User Datagram Protocol)

Key Networking Classes

Class	**Purpose**
Socket	Represents a TCP client socket.
ServerSocket	Represents a TCP server socket.
DatagramSocket	Represents a UDP socket.
InetAddress	Represents an IP address.
URL and URLConnection	Handles URLs and HTTP connections.

TCP Client-Server Example

Server Side:

java

Copy code

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class TCPServer {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(12345)) {
            System.out.println("Server is listening on port 12345...");
            Socket clientSocket = serverSocket.accept(); // Wait for a connection

            BufferedReader input = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
            PrintWriter output = new PrintWriter(clientSocket.getOutputStream(), true);

            String message = input.readLine(); // Read message from client
            System.out.println("Received: " + message);

            output.println("Hello, client!"); // Respond to client

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

Client Side:

java

Copy code

```
import java.io.*;
import java.net.Socket;

public class TCPClient {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 12345)) {
            BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);
            output.println("Hello, server!"); // Send message to server

            String response = input.readLine(); // Receive response from server

            System.out.println("Server Response: " + response);


        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

---

4. Java IO vs NIO

Aspect	**Java IO**	**Java NIO**
Mode	Blocking	Non-blocking
Data Handling	Stream-based	Buffer-based
Performance	Slower for large data	Faster due to buffer and channels
Concurrency	Requires multiple threads	Single thread with Selector
Ease of Use	Simpler and straightforward	Complex but more powerful

---

Conclusion

- **Java IO**: Suitable for simpler, stream-based operations.
- **Java NIO**: Designed for high-performance, non-blocking I/O operations and scalability.
- **Java Networking**: Facilitates TCP/IP, UDP-based communication, and HTTP connections.

Java 8 Date and Time API

Java 8 introduced a new Date and Time API in the java.time package to address the limitations of the older java.util.Date and java.util.Calendar classes. The new API is more immutable, thread-safe, and fluent, allowing for more intuitive and efficient date and time manipulation.

Here are some of the core classes of the Java 8 Date and Time API:

LocalDate

LocalTime

LocalDateTime

ZonedDateTime

Instant

Duration and Period

DateTimeFormatter

1. LocalDate

Represents a date without a time component (e.g., year, month, day).

Creating a LocalDate

java

Copy code

```
import java.time.LocalDate;

public class LocalDateExample {
    public static void main(String[] args) {
        // Current date
        LocalDate today = LocalDate.now();

        // Specific date
        LocalDate specificDate = LocalDate.of(2024, 12, 17);

        System.out.println("Today's Date: " + today);
        System.out.println("Specific Date: " + specificDate);
    }
}
```

Methods

now(): Returns the current date.

of(int year, int month, int day): Creates a LocalDate from specific year, month, and day.

plusDays(long days): Adds days to the date.

minusMonths(long months): Subtracts months from the date.

2. LocalTime

Represents a time without a date (e.g., hours, minutes, seconds).

Creating a LocalTime

java

Copy code

```
import java.time.LocalTime;

public class LocalTimeExample {
    public static void main(String[] args) {
        // Current time
        LocalTime now = LocalTime.now();

        // Specific time
        LocalTime specificTime = LocalTime.of(14, 30, 45);

        System.out.println("Current Time: " + now);
        System.out.println("Specific Time: " + specificTime);
    }
}
```

Methods

now(): Returns the current time.

of(int hour, int minute, int second): Creates a LocalTime from a specific hour, minute, and second.

plusHours(long hours): Adds hours to the time.

minusMinutes(long minutes): Subtracts minutes from the time.

3. LocalDateTime

Represents a date and time without a time zone.

Creating a LocalDateTime

java

Copy code

```
import java.time.LocalDateTime;

public class LocalDateTimeExample {
    public static void main(String[] args) {
        // Current date and time
        LocalDateTime now = LocalDateTime.now();

        // Specific date and time
        LocalDateTime specificDateTime = LocalDateTime.of(2024, 12, 17, 14, 30, 45);

        System.out.println("Current Date and Time: " + now);
        System.out.println("Specific Date and Time: " + specificDateTime);
    }
}
```

Methods

now(): Returns the current date and time.

of(int year, int month, int day, int hour, int minute, int second): Creates a LocalDateTime with specific values.

plusDays(long days): Adds days to the date-time.

minusHours(long hours): Subtracts hours from the date-time.

4. ZonedDateTime

Represents a date and time with a specific time zone.

Creating a ZonedDateTime

java

Copy code

```
import java.time.ZonedDateTime;
import java.time.ZoneId;

public class ZonedDateTimeExample {
    public static void main(String[] args) {
        // Current date and time with time zone
        ZonedDateTime zonedNow = ZonedDateTime.now();

        // Specific date and time with a specific time zone
        ZonedDateTime zonedDateTime = ZonedDateTime.of(2024, 12, 17, 14, 30, 45, 0,
ZoneId.of("America/New_York"));

        System.out.println("Zoned Date and Time (Current): " + zonedNow);
        System.out.println("Zoned Date and Time (Specific): " + zonedDateTime);
    }
}
```

Methods

now(): Returns the current date, time, and time zone.

of(int year, int month, int day, int hour, int minute, int second, int nanosecond, ZoneId zone): Creates a ZonedDateTime with a specific time zone.

5. Instant

Represents a specific point in time (similar to System.currentTimeMillis() but with nanosecond precision).

Creating an Instant

java

Copy code

```java
import java.time.Instant;


public class InstantExample {
    public static void main(String[] args) {
        // Current instant (UTC)
        Instant now = Instant.now();

        System.out.println("Current Instant: " + now);
    }
}
```

Methods

now(): Returns the current instant (in UTC).

ofEpochSecond(long epochSecond): Creates an Instant from a specified epoch second.

plus(Duration duration): Adds a duration to the instant.

6. Duration and Period

These classes represent time-based amounts between two points.

Duration: Measures time in terms of seconds and nanoseconds. Useful for calculating differences between Instant or LocalTime.

Period: Measures time in terms of years, months, and days. Useful for calculating differences between LocalDate objects.

Duration Example

java

Copy code

```java
import java.time.Duration;

import java.time.LocalTime;
```

```
public class DurationExample {
    public static void main(String[] args) {
        LocalTime start = LocalTime.of(14, 30);
        LocalTime end = LocalTime.of(15, 45);

        Duration duration = Duration.between(start, end);

        System.out.println("Duration in minutes: " + duration.toMinutes());
    }
}
```

Period Example

java

Copy code

```
import java.time.LocalDate;
import java.time.Period;

public class PeriodExample {
    public static void main(String[] args) {
        LocalDate startDate = LocalDate.of(2020, 1, 1);
        LocalDate endDate = LocalDate.of(2024, 12, 17);

        Period period = Period.between(startDate, endDate);

        System.out.println("Period: " + period.getYears() + " years, " + period.getMonths() + " months, " +
period.getDays() + " days");
    }
}
```

7. DateTimeFormatter

A class for formatting and parsing dates and times.

Formatting Example

java

Copy code

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class DateTimeFormatterExample {
    public static void main(String[] args) {
        LocalDate date = LocalDate.now();

        // Define a custom format
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");

        String formattedDate = date.format(formatter);
        System.out.println("Formatted Date: " + formattedDate);
    }
}
```

Parsing Example

java

Copy code

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class DateTimeParseExample {
    public static void main(String[] args) {
        String dateString = "17/12/2024";

        // Define a custom format
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");

        LocalDate date = LocalDate.parse(dateString, formatter);
        System.out.println("Parsed Date: " + date);
    }
}
```

Summary of Key Classes

Class	Purpose
LocalDate	Date without time.

LocalTime	Time without date.
LocalDateTime	Date and time without timezone.
ZonedDateTime	Date and time with timezone.
Instant	A specific point in time (UTC).
Duration	Time-based amounts in seconds and nanoseconds.
Period	Date-based amounts in years, months, and days.
DateTimeFormatter	Formatting and parsing date/time objects.

Conclusion

Java 8's Date and Time API provides more powerful and flexible tools for working with dates, times, durations, and time zones. It solves many of the limitations of the old Date and Calendar classes by offering immutable and thread-safe classes. This makes date and time manipulation in Java easier and more efficient.