UNIT 3

Exceptions

- An exceptions is unaccepted/unwanted /abnormal situations that occurs in runtime called exceptions.
- Exception: An event that disrupts the normal flow of the program.
- Error: A more severe issue (like running out of memory), generally not meant to be handled by the program.
- Types of Exceptions:
- Checked Exceptions: These exceptions are checked at compile time. You must either catch these exceptions or declare them using throws. Example: IOException, SQLException.
- Unchecked Exceptions: These are not checked at compile time. They usually occur due to programming bugs. Example: NullPointerException, ArrayIndexOutOfBoundsException.
- try: Used to write code that may throw an exception.
- catch: Used to handle exceptions.
- finally: Always executed, useful for cleanup.
- throw: In Java, the throw keyword is used to throw an exception. When you want your program to intentionally raise an error (exception) during execution, you use throw
- throws: Declares exceptions in method signatures.

Example

```
public class ExceptionExample {
   public static void main(String[] args) {
      try {
         int arithmetic = 10 / 0; // This will throw ArithmeticException
         System.out.println(arithmetic);
      } catch (ArithmeticException e) {
         System.out.println("Exception caught: " + e);
```

```
        } finally {
            System.out.println("Closed security-related functions like
    database connectivity.");
        }
      }
    }
```

Throw and Throws

- The throw keyword is used inside a function. It is used when it is required to throw an Exception logically
- The throws keyword is used in the function signature. It is used when the function has some statements that can lead to exceptions.
- The throw keyword is used to throw an exception, It can throw only one exception at a time
- The throws keyword can be used to declare multiple exceptions, separated by a comma.

Example:

```
class ThrowThrowsExample {
   // Method that declares an exception using 'throws'
   static void checkAge(int age) throws ArithmeticException {
      if (age < 18) {
         throw new ArithmeticException("Access denied - You must be at
least 18 years old.");
      } else {
         System.out.println("Access granted - You are old enough.");
      }
   }

   public static void main(String[] args) {
      try {
         checkAge(15); // This will throw an exception
      } catch (ArithmeticException e) {
         System.out.println("Exception caught: " + e.getMessage());
      }
   }
}
```

User-Defined Custom Exception in Java

- In Java, an Exception is an issue (run time error) that occurred during the execution of a program.
- Java provides us the facility to create our own exceptions which are basically derived classes of Exception
- Creating our own Exception is known as a custom exception in Java or user-defined exception in Java.
- Basically, Java custom exceptions are used to customize the exception according to user needs.

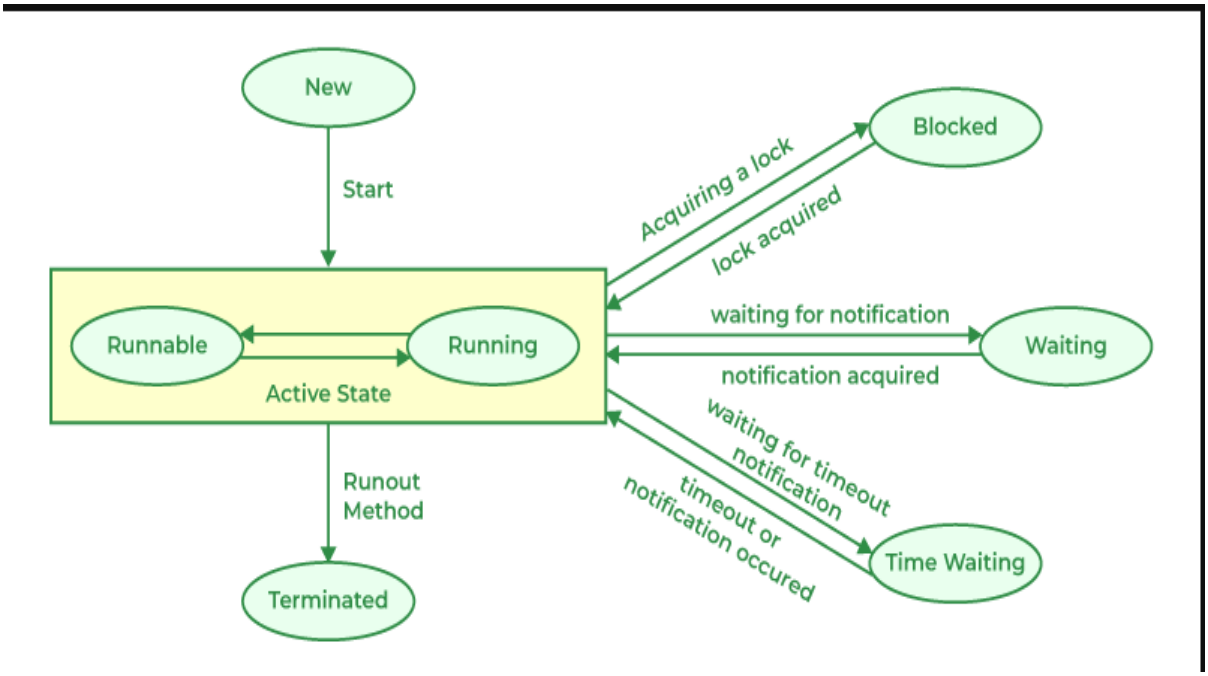
Example:

```
// Step 1: Create a custom exception class
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);  // Call the parent class constructor
    }
}

// Step 2: Use the custom exception in a program
class CustomExceptionDemo {
    // Method that throws the custom exception
    static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age is less than 18. Access
Denied!");
        } else {
            System.out.println("Access granted. You are eligible.");
        }
    }

    public static void main(String[] args) {
        try {
            validateAge(16); // This will throw the custom exception
        } catch (InvalidAgeException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Threads

- Threads are lightweight subprocesses, representing the smallest unit of execution with separate paths. The main advantage of multiple threads is efficiency (allowing multiple things at the same time). For example, in MS Word, one thread automatically formats the document while another thread is taking user input.
- A Thread in Java can exist in any one of the following states at any given time. A thread lies only in one of the shown states at any instant

Life Cycle of a Thread



- **New State**
- **Runnable State**
- **Blocked State**
- **Waiting State**
- **Timed Waiting State**
- **Terminated State**
- **New Thread**: When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state.
- public static final Thread.State NEW
- **Runnable State**: A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time.
- public static final Thread.State RUNNABLE
- **Blocked**: The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread.
- public static final Thread.State BLOCKED

- **Waiting state**: The thread will be in waiting state when it calls wait() method or join() method.
- public static final Thread.State WAITING
- **Timed Waiting**: A thread lies in a timed waiting state when it calls a method with a time-out parameter.
- **Terminated State**: A thread terminates because of either of the following reasons:
- Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
- Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.

Thread class and methods

- The Thread class in Java is part of the java.lang package and provides several methods that control the behavior of threads. Below are the most commonly used methods of the Thread class
- **start()**
- **run()**
- **sleep(long millis)**
- **join()**
- **isAlive()**
- **getPriority()**
- **getState()**

Thread class and its methods

- In Java, the Thread class is part of the java.lang package and provides several methods for creating and managing threads. Here are the key methods of the Thread class in Java

Method	Description
`start()`	Starts a new thread, executing the `run()` method.
`run()`	Defines the thread's task (can be overridden).
`join()`	Waits for the thread to finish execution.
`sleep(ms)`	Pauses the thread for a specified time (in milliseconds).
`isAlive()`	Returns `true` if the thread is still running.
`interrupt()`	Interrupts the thread.
`setName(name)` / `getName()`	Sets or retrieves the thread's name.
`setPriority(int)` / `getPriority()`	Sets or gets the thread priority (range: 1-10).
`yield()`	Suggests that the current thread gives CPU time to another thread.
`isInterrupted()`	Checks if the thread has been interrupted.
`currentThread()`	Returns a reference to the currently executing thread.

Example:

```
class MyThread extends Thread {
   public void run() {
      for (int i = 1; i <= 5; i++) {
         System.out.println(Thread.currentThread().getName() + " - Count:
" + i);
         try {
            Thread.sleep(1000); // Sleep for 1 second
         } catch (InterruptedException e) {
            System.out.println(e);
         }
      }
   }
}

public class ThreadExample {
   public static void main(String[] args) {
      MyThread t1 = new MyThread();
      MyThread t2 = new MyThread();

      t1.setName("Thread A");
      t2.setName("Thread B");

      t1.start(); // Start Thread A
      t2.start(); // Start Thread B
   }
}
```

Synchronization in Multithreading

- In Java, Synchronization is a mechanism to control the access of multiple threads to shared resources. It ensures that only one thread can access a critical section of the code at a time, preventing race conditions and ensuring data consistency.
  Why is Synchronization Needed?
- When multiple threads try to modify a shared resource simultaneously, race conditions may occur, leading to unexpected behavior and inconsistent data.
- **Types of Synchronization in Java**
- Synchronized Method
- Synchronized Block
- Static Synchronization
  Example

```
class SharedResource {
```

```
    synchronized void printTable(int num) { // Synchronized method
        for (int i = 1; i <= 5; i++) {
            System.out.println(num + " x " + i + " = " + (num * i));
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

class MyThread1 extends Thread {
    SharedResource obj;
    MyThread1(SharedResource obj) {
        this.obj = obj;
    }
    public void run() {
        obj.printTable(5);
    }
}

class MyThread2 extends Thread {
    SharedResource obj;
    MyThread2(SharedResource obj) {
        this.obj = obj;
    }
    public void run() {
        obj.printTable(10);
    }
}

public class SynchronizedMethodExample {
    public static void main(String args[]) {
        SharedResource obj = new SharedResource();
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);

        t1.start();
        t2.start();
    }
}
```

Daemon Thread vs Non-Daemon Thread in Java

- Java threads can be categorized into Daemon Threads and Non-Daemon Threads (also called User Threads). Let's explore the differences and examples.
- A Daemon Thread is a low-priority background thread that runs in the background to support other user threads.
- Important Rule: If all user threads finish execution, the JVM automatically terminates daemon threads.
- Characteristics of Daemon Threads:
- Used for background tasks (e.g., Garbage Collector, Timer, etc.).
- JVM automatically stops them when no user thread is running.
- Not recommended for critical tasks (as they may stop unexpectedly).

What is a Non-Daemon (User) Thread?

- A User Thread (Non-Daemon) is the default thread type in Java.
- The JVM waits for all user threads to finish before terminating.

Feature	Daemon Thread	Non-Daemon (User) Thread
Purpose	Runs background tasks	Executes main application logic
JVM Behavior	Stops when all user threads finish	JVM waits for it to finish
Use Case	Garbage Collection, Monitoring, Logging	Core program execution
Default Type?	No (must be set explicitly)	Yes

Daemon Thread Example

```
class MyDaemonThread extends Thread {
    public void run() {
        while (true) {
            System.out.println("Daemon thread running...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class DaemonThreadExample {
    public static void main(String[] args) {
        MyDaemonThread dt = new MyDaemonThread();
        dt.setDaemon(true);  // Setting thread as daemon
        dt.start();

        System.out.println("Main thread running...");
```

```
        try {
            Thread.sleep(3000);  // Main thread sleeps for 3 seconds
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        System.out.println("Main thread finished!");
    }
}
```

Non-Daemon Thread Example

```
class MyUserThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("User thread running...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
        System.out.println("User thread finished!");
    }
}

public class NonDaemonThreadExample {
    public static void main(String[] args) {
        MyUserThread ut = new MyUserThread();
        ut.start();

        System.out.println("Main thread finished!");
    }
}
```

Key Differences: Daemon vs Non-Daemon Threads

Feature	Daemon Thread	Non-Daemon (User) Thread
Purpose	Background support (e.g., GC)	Main application logic
Stops Execution?	Stops when all user threads finish	Runs independently
Examples	Garbage Collector, Finalizer	Main thread, worker threads
Default Type?	No, must set using `setDaemon(true)`	Yes

Streams in java

- In Java, Streams are used to perform input and output (I/O) operations. They help in reading data from various sources (files, keyboard, network, etc.)
- Java provides two main types of Streams:
- Byte Streams (Handle binary data)
- Character Streams (Handle text data)
- Byte Streams (Binary Data)
- Used to read/write binary data (images, audio, video, etc.).
- Operate on bytes (8-bit data).
- Use InputStream and OutputStream as parent classes.

Example:

```
import java.io.FileInputStream;
import java.io.IOException;

public class ByteReadExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("input.txt")) {
            int byteData;
            while ((byteData = fis.read()) != -1) {
                System.out.print((char) byteData); // Convert byte to
character
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Example 2

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class ByteCopyExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new
FileInputStream("source.jpg");
             FileOutputStream fos = new
FileOutputStream("copy.jpg")) {

            int byteData;
```

```
            while ((byteData = fis.read()) != -1) {
                fos.write(byteData);
            }
            System.out.println("File copied successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

File Class in Java

- The File class in Java is used to represent the path of a file or directory. It does not perform read/write operations but provides information about the file (existence, size, permissions, etc.).
- Common Methods of File Class

Method	Description
`exists()`	Checks if the file exists
`createNewFile()`	Creates a new empty file
`delete()`	Deletes the file
`getName()`	Returns file name
`getAbsolutePath()`	Returns full file path
`length()`	Returns file size in bytes
`canRead()`	Checks if file is readable

Example

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

public class ReadWriteExample {
    public static void main(String[] args) {
        String fileName = "sample.txt";

        // Writing to file
        try (FileWriter writer = new FileWriter(fileName)) {
            writer.write("Hello, this is a test file.\n");
            writer.write("Writing and reading in the same file!");
            System.out.println("✅ Data written to file successfully.");
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }

        // Reading from file
        try (FileReader reader = new FileReader(fileName)) {
            int ch;
            System.out.println("\n📄 File Content:");
            while ((ch = reader.read()) != -1) {
                System.out.print((char) ch);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Random Access File Class in Java

- Unlike File, the Random Access File class allows both reading and writing at any position in the file. It supports random access, meaning you can seek to a specific position and modify data without overwriting the entire file.

Mode	Description
`"r"`	Read-only mode
`"rw"`	Read and write mode
`"rws"`	Read, write, and update file contents instantly
`"rwd"`	Read, write, and update metadata instantly

Differences Between File and Random Access File

Feature	`File`	`RandomAccessFile`
Purpose	Represents file/directory	Allows reading/writing to any position in the file
Read/Write	No read/write methods	Supports both reading and writing
Random Access	No	Yes
Performance	Slower for modifications	Faster for modifying specific parts
Usage	Checking file properties	Editing large files efficiently

Example:

```
import java.io.*;

public class RandomAccessFileExample {
   public static void main(String[] args) {
      String filePath = "example.txt";

      try (RandomAccessFile raf = new
RandomAccessFile(filePath, "rw")) {
         // Write to file
         raf.writeBytes("Hello, World!\n");
         System.out.println("Data written successfully.");

         // Move to beginning and read
         raf.seek(0);
         System.out.println("Read: " + raf.readLine());
      } catch (IOException e) {
         System.err.println("Error: " + e.getMessage());
      }
   }
}
```

Reading and writing through stream classes

- Java provides character stream classes for handling text data efficiently.
- Handles text files (e.g., .txt, .csv).
- Supports Unicode characters (useful for different languages).
- More efficient than byte streams (FileInputStream, FileOutputStream) for text-based data.
- Uses Buffered Writer for faster performance.

Key Classes for Character Streams

Class	Purpose
`FileReader`	Reads character data from a file
`FileWriter`	Writes character data to a file
`BufferedReader`	Reads text efficiently, line by line
`BufferedWriter`	Writes text efficiently, line by line

Example:

```
import java.io.*;

public class FileStreamExample {
   public static void main(String[] args) throws IOException {
      String filePath = "example.txt";

      // Write to file
      try (FileOutputStream fos = new FileOutputStream(filePath))
{
         fos.write("Hello, Streams!".getBytes());
      }

      // Read from file
      try (FileInputStream fis = new FileInputStream(filePath)) {
         System.out.println(new String(fis.readAllBytes()));
      }
   }
}
```

Example: FileOutputStream

```
import java.io.*;

public class FileOutputStreamExample {
   public static void main(String[] args) {
      String filePath = "example.txt";

      try (FileOutputStream fos = new FileOutputStream(filePath))
{
         fos.write("Hello, FileOutputStream!".getBytes());
         System.out.println("Data written successfully.");
      } catch (IOException e) {
         System.err.println("Error writing to file: " +
e.getMessage());
      }
   }
}
```

Why Use BufferedReader and BufferedWriter?
Faster than FileReader & FileWriter.
Reads and writes larger chunks of data at once.
Ideal for reading large text files.

Example:

```
import java.io.*;

public class BufferedFileReadExample {
   public static void main(String[] args) {
      String filePath = "example.txt";

      try (BufferedReader reader = new BufferedReader(new
FileReader(filePath))) {
         String line;
         while ((line = reader.readLine()) != null) { // Read line by
line
            System.out.println("Read: " + line);
         }
      } catch (IOException e) {
         System.err.println("Error reading from file: " +
e.getMessage());
      }
   }
}
```

Java Pipe Streams

- Pipe streams in Java allow communication between two threads using a unidirectional data flow.
- They are used for inter-thread communication.

- Helps transfer data between threads without a shared buffer.
- Types of Pipe Streams in Java
- Byte-based Streams
- PipedInputStream (reading)
- PipedOutputStream (writing)
- Character-based Streams
- PipedReader (reading)
- PipedWriter (writing)

- **How It Works**
- Writer Thread → Writes data into a PipedOutputStream.
- Reader Thread → Reads data from a PipedInputStream (connected to the output stream).
- Data flows like a pipeline from the writer to the reader.

Applications of Java Pipe Streams

Inter-Thread Communication
Pipe Streams help two threads exchange data in real-time without using files or shared variables.
Example: A producer thread writes data, and a consumer thread reads it.
Real-Time Data Processing
Used in applications where data is continuously generated and processed at the same time.
Example: A sensor thread sends data, and a monitoring thread reads and analyses it.
Logging Systems
Logs data in a background thread while the main program continues running.
Example: A web server that logs HTTP requests without slowing down processing.
Audio & Video Streaming
Helps transfer multimedia data between threads efficiently.
Example: Live streaming, where a thread reads video frames while another sends them to a client.

Example:

```
import java.io.*;

public class PipeStreamExample {
```

```
    public static void main(String[] args) throws IOException {
        // Create piped streams
        PipedOutputStream pos = new PipedOutputStream();
        PipedInputStream pis = new PipedInputStream(pos);

        // Writer thread: sends data
        Thread writer = new Thread(() -> {
            try {
                pos.write("Hello, PipeStream!".getBytes()); // Writing data
                pos.close(); // Close after writing
            } catch (IOException e) {
                System.err.println("Error writing: " + e.getMessage());
            }
        });

        // Reader thread: receives data
        Thread reader = new Thread(() -> {
            try {
                int data;
                while ((data = pis.read()) != -1) { // Reading data
                    System.out.print((char) data);
                }
                pis.close(); // Close after reading
            } catch (IOException e) {
                System.err.println("Error reading: " + e.getMessage());
            }
        });

        // Start threads
        writer.start();
        reader.start();
    }
}
```