E-BOOK

UNIT -2

PDF

OOPS WITH JAVA

written By : Abhay Kumar Singh

2.1 The Idea Behind Exception

- Exceptions in Java are events that disrupt the normal flow of program execution.
- The primary goal of using exceptions is to handle errors and unexpected situations in a controlled manner.
- ensuring the program can continue or gracefully shut down.

2.2 Why Use Exceptions?

- **Error Handling:** To manage runtime errors effectively.
- **Separation of Concerns**: To separate error-handling code from regular code, improving readability and maintainability.
- **Error Propagation:** To propagate errors up the call stack, allowing higher-level methods to handle them if necessary.
- **Uniform Handling:** To provide a consistent way to handle different types of errors.

2.3 How Exceptions Work?

- Flow Interruption: An exception interrupts the normal program flow.
- Handler Search: The runtime system searches for an appropriate exception handler.
- Handler Found: If a handler is found, control is transferred to it.
- No Handler: If no handler is found, the program terminates.
- Graceful Shutdown: Proper exception handling ensures a controlled and graceful shutdown if necessary.
- Example:**To handle division by zero, catch the ArithmeticException using a try-catch block.**

```
public class  ExampleArithmeticException {
   public static void main(String[] args) {
      try {
         int result = divide(10, 0);
         System.out.println("Result: " + result);
      } catch (ArithmeticException e) {
         System.out.println("Exception caught: " + e.getMessage());
      }
   }

   public static int divide(int numerator, int denominator) {
      return numerator / denominator;
   }
}

output: Exception caught: / by zero
```

2.4 Explanation of Exception :

- Throwing an Exception: Creating and "throwing" an exception object to indicate an error.
- Catching an Exception: Using a try-catch block to handle exceptions.
- Finally Block: Contains code that always executes, used for cleanup.
- Exception Propagation: If not caught, exceptions move up the call stack until caught or the program terminates.

2.5 Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Opening an unavailable file

2.6 Benefits of Using Exceptions

- Improved Code Clarity
- Better Error Reporting
- Resource Management

2.7 Exceptions

- Events in Java that disrupt normal program flow due to errors or exceptional conditions.

2.8 Errors

- Serious issues usually beyond the program's control, like JVM failures or resource limitations.

2.9 finally Block

- The finally block is used in conjunction with try-catch to ensure that certain code executes, regardless of whether an exception is thrown or not.
- It is typically used for cleanup tasks, such as closing resources (files, streams, database connections) that were opened in the try block.
- **Syntax:**

```
try {
   // Code that may throw exceptions
} catch (ExceptionType1 e1) {
   // Exception handling code
} catch (ExceptionType2 e2) {
   // Exception handling code
} finally {
   // Cleanup code or code that must always execute
}
```

2..10 Throw Keyword

- The throw keyword is used to explicitly throw an exception within a method or block of code.
- It is followed by an instance of an exception class or a subclass of Throwable.
- Used to indicate exceptional conditions programmatically.
- Can be used to throw both built-in and user-defined exceptions.
- **Syntax:**

```
throw throwableInstance;
```

2.11 Throws Clause

- The throws clause is used in method signatures to indicate that the method may throw one or more types of exceptions.
- It specifies the exceptions that a method can throw, allowing callers of the method to handle those exceptions.
- **Syntax**:

```
void methodName() throws ExceptionType1, ExceptionType2, ... {
   // Method code that may throw exceptions
}
```

2.12 Types of Built-in Exceptions

2.12.1 Checked Exceptions:

- Checked exceptions are verified by the compiler at compile-time.
- Examples include IOException, SQLException.
- These exceptions must be handled using try-catch or declared using throws in the method signature.

2.12.2 Unchecked Exceptions:

- Also known as runtime exceptions.
- Not checked by the compiler at compile-time.
- Examples include NullPointerException, ArrayIndexOutOfBoundsException.
- Programs can throw unchecked exceptions without handling or declaring them, and they won't cause a compilation error.

2.12.3 User-Defined Exceptions

- User-Defined Exceptions:
- Created by Java developers when the built-in exceptions are inadequate to describe a specific situation.
- These exceptions extend from the Exception class or its subclasses to define custom error conditions in applications.

Example: Checked Exceptions(IOException)

```
import java.io.*;

public class a {
   public static void main(String[] args) {
      try {
         BufferedReader reader = new BufferedReader(new
FileReader("file.txt"));
         String line = reader.readLine();
         System.out.println(line);
         reader.close();
      } catch (IOException e) {
         System.out.println("IOException caught: " + e.getMessage());
      }
   }
}
```

Example: UnChecked Exceptions(ArithmeticException)

```
public class ArithmeticExceptionExample {
   public static void main(String[] args) {
      try {
         int numerator = 10;
         int denominator = 0;
         int result = numerator / denominator; // This line will throw
ArithmeticException
         System.out.println("Result: " + result); // This line will not be reached
      } catch (ArithmeticException e) {
         System.out.println("ArithmeticException caught: Division by zero.");
      }
   }
}
```

Example: User-Defined Exceptions(Custom Exception)

```
class CustomException extends Exception {
   public CustomException(String message) {
      super(message);
   }
}
public class CustomExceptionExample {
   public static void main(String[] args) {
      try {
         int age = 17;
         if (age < 18) {
            throw new CustomException("Underage person not allowed.");
         } else {
            System.out.println("Welcome! You are eligible.");
         }
      } catch (CustomException e) {
         System.out.println("CustomException caught: " + e.getMessage());
      }
   }  }
```

2.13 JVM Reaction to Exceptions

- When an exception is not caught, the JVM handles it by printing the stack trace and terminating the program.
- **Example:**

```
int result = 10 / 0; // JVM will throw an ArithmeticException and terminate
the program
```

2.14 Input/Output Basics in Java

- Java provides robust input/output (I/O) functionality through various classes and interfaces, primarily found in the java.io package.
- The two main types of streams in Java are Byte Streams and Character Streams, each serving different purposes in handling data.

2.15 Byte Streams

- Byte streams are used to perform input and output of 8-bit bytes.
- They are useful for handling binary data, such as image files, audio files, and other types of media.
- Byte streams are built around the InputStream and OutputStream classes and their subclasses.
- **Common Byte Stream Classes:**
  - **FileInputStream:** Reads bytes from a file.
  - **FileOutputStream:** Writes bytes to a file.
  - **BufferedInputStream:** Buffers input bytes for more efficient reading.
  - **BufferedOutputStream:** Buffers output bytes for more efficient writing.

2.15.1 Reading a File Using FileInputStream:

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

public class  FileInputStreamExample  {
   public static void main(String[] args) {
      try (FileInputStream fis = new FileInputStream("file.txt")) {
         int data;
         while ((data = fis.read()) != -1) {
            System.out.print((char) data);
            s+=(char) data;
         }

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

2.15.2 Writing to a File Using FileOutputStream:

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

public class ByteStreamWriteExample {
   public static void main(String[] args) {
      String data = "Hello, Byte Stream!";
      try (FileOutputStream fos = new FileOutputStream("output.txt")) {
         fos.write(data.getBytes());
      } catch (IOException e) {
         e.printStackTrace();
      }
   }

}
```

2.16 Character Streams

- Character streams are used to perform input and output for 16-bit Unicode characters.
- They are useful for handling text data.
- Character streams are built around the Reader and Writer classes and their subclasses.
- **Common Character Stream Classes:**
  - **FileReader**: Reads characters from a file.
  - **FileWriter**: Writes characters to a file.
  - **BufferedReader**: Buffers characters for more efficient reading.
  - **BufferedWriter**: Buffers characters for more efficient writing.

2.16.1 Reading a File Using FileReader:

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

public class CharacterStreamReadExample {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("input.txt")) {
            int data;
            while ((data = fr.read()) != -1) {
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2.16.2 Writing to a File Using FileWriter:

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

public class CharacterStreamWriteExample {
    public static void main(String[] args) {
        String data = "Hello, Character Stream!";
        try (FileWriter fw = new FileWriter("output.txt")) {
            fw.write(data);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2.16.3 Reading a File Using BufferedReader:

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

public class ReadFileUsingCharacterStream {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new
FileReader("example.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2.16.4 Writing to a File Using BufferedWriter:

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

public class WriteFileUsingCharacterStream {
    public static void main(String[] args) {
        String content = "Hello, World!";
        try (BufferedWriter bw = new BufferedWriter(new
FileWriter("example.txt"))) {
            bw.write(content);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2.17 Multithreading in Java

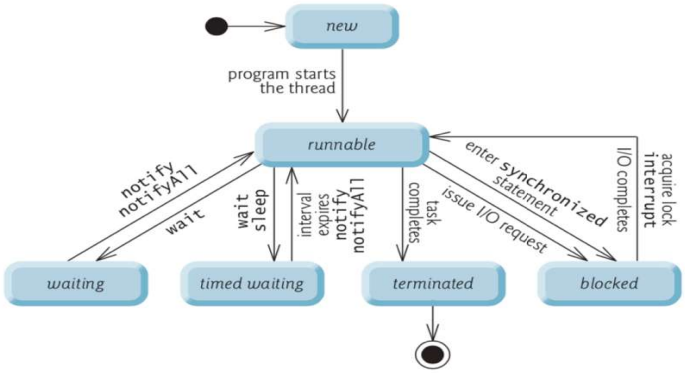
- Multithreading is a feature of Java that allows concurrent execution of two or more threads.
- It is essential for performing multiple tasks simultaneously and efficiently utilizing the CPU.

2.18 Thread

- A thread is the smallest unit of a process that can be scheduled and executed independently by the operating system.
- In Java, the Thread class represents a thread of execution.

2.19 Thread Life Cycle

- A thread in Java goes through several states in its life cycle:
- New: A thread is created but not yet started.
- Runnable: A thread is ready to run and waiting for CPU time.
- Blocked: The thread is waiting for a resource.
- Waiting: The thread is waiting indefinitely for another thread to perform a particular action.
- Timed Waiting: The thread is waiting for another thread to perform an action for up to a specified waiting time.
- Terminated: The thread has finished its execution.



2.20 Creating Threads

- There are two ways to create a thread in Java:
- **By extending the Thread class:**

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // start() method calls run() method internally
    }
}
```

- **By implementing the Runnable interface:**

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running...");
    }
     public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread t1 = new Thread(myRunnable);
        t1.start(); // start() method calls run() method internally
    }
}
```

2.21 Example of Multiple Thread Running (parallel manner)

```
class MyThread extends Thread {
    private String threadName;

    MyThread(String name) {
        threadName = name;
    }
```

```
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(threadName + " is running: " + i);
            try {
                Thread.sleep(500); // Sleep for 500 milliseconds
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(threadName + " has finished.");
    }
    public static void main(String[] args) {
        MyThread t1 = new MyThread("Thread 1");
        MyThread t2 = new MyThread("Thread 2");
        MyThread t3 = new MyThread("Thread 3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

2.22 Thread Priorities

- Each thread in Java has a priority that helps the thread scheduler determine the order of thread execution.
- The priority is an integer in the range of 1 (MIN_PRIORITY) to 10 (MAX_PRIORITY), with 5 (NORM_PRIORITY) as the default priority.

```
class PriorityThread extends Thread {
    public void run() {
        System.out.println("Thread is running with priority: " +
Thread.currentThread().getPriority());
    }

    public static void main(String[] args) {
        PriorityThread t1 = new PriorityThread();
        t1.setPriority(Thread.MAX_PRIORITY); // setting priority to 10
        t1.start();

        PriorityThread t2 = new PriorityThread();
        t2.setPriority(Thread.MIN_PRIORITY); // setting priority to 1
        t2.start();
    }
}
```

2.24 Run Two method concurrent using Thread

```
class SumThread extends Thread {
    private int[] numbers;
    private int sum;

    SumThread(int[] nums) {
        numbers = nums;
    }
    public void run() {
        sum = 0;
        for (int num : numbers) {
            sum += num;
        }
        System.out.println("Sum of numbers: " + sum);
    }
}
class EvenThread extends Thread {
    private int[] numbers;

    EvenThread(int[] nums) {
        numbers = nums;
    }

    public void run() {
        System.out.print("Even numbers in the array: ");
        for (int num : numbers) {
            if (num % 2 == 0) {
                System.out.print(num + " ");
            }
        }
        System.out.println();
    }
}
public class ConcurrentThreadsExample {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

        SumThread sumThread = new SumThread(arr);
        EvenThread evenThread = new EvenThread(arr);

        sumThread.start();
        evenThread.start();
    }
}
output: Even numbers in the array: 2 4 6 8 10
                Sum of numbers: 55
```

2.22 Synchronizing Threads

- Ensures only one thread accesses a shared resource at a time, preventing race conditions.
- Marking a method as synchronized allows only one thread to execute it at a time for a given instance.
- Each object has an intrinsic lock; a thread acquires this lock when entering a synchronized method or block, making other threads wait.
- Crucial for thread-safe operations, preventing inconsistencies and errors during concurrent access.

```
class Printing {
    synchronized void print(char ch) {
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j <= i; j++) {
                System.out.print(ch);
            }
            System.out.println();
        } }
}
class PrintThread extends Thread {
    Printing printing;
    char ch;

    PrintThread(Printing printing, char ch) {
        this.printing = printing;
        this.ch = ch;
    }

    public void run() {     printing.print(ch);   }
}
public class Synthread {

    public static void main(String[] args) {
        Printing printing = new Printing();
        PrintThread threadA = new PrintThread(printing, '1');
        PrintThread threadB = new PrintThread(printing, '2');
        threadA.start();
        threadB.start();
    }
}
```

Inter-thread communication

- Inter-thread communication in Java allows threads to coordinate and share information while working on a shared task.
- This is essential for creating efficient and synchronized multi-threaded applications.

wait(): Causes the current thread to wait until another thread calls notify() or notifyAll() on the same object.Must be called from a synchronized context.

notify(): Wakes up a single waiting thread.The waiting thread continues after reacquiring the lock.

notifyAll(): Wakes up all waiting threads.Threads continue one at a time after reacquiring the lock.