Exercise – 6

a) Write a JAVA program that describes exception handling mechanism

Description:

Exception handling in Java is a powerful mechanism that allows developers to manage errors and exceptional conditions in a graceful manner. The primary keywords used in exception handling are `try`, `catch`, `finally`, `throw`, and `throws`

programme

```
import java.util.Scanner;

public class ExceptionHandlingExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter a number: ");
            int num = scanner.nextInt();

            int result = 100 / num;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
                System.out.println("Error: Division by
  zero is not allowed.");
        } catch (java.util.InputMismatchException e) {
              System.out.println("Error: Please enter a
  valid integer.");
        } finally {
            System.out.println("Execution finished.");
            scanner.close();
        }
    }
}
```

output

```
Enter a number: 35
```

```
Result: 2
Execution finished.
```

c.Write a JAVA program Illustrating Multiple catch clauses

Description

Imports: The program imports the Scanner class for user input.

- **Try Block**: The `try` block contains code that may throw exceptions. It prompts the user for a number and attempts to divide 100 by that number.
- **Multiple Catch Blocks**:
- **ArithmeticException**: Catches the exception that occurs when the user enters zero, which would result in a division by zero.
- **InputMismatchException**: Catches the exception that occurs when the user enters a non-integer value.
- **General Exception**: The last `catch` block is a catch-all for any other unexpected exceptions that may occur.
- **Finally Block**: The `finally` block is executed after the `try` and `catch` blocks, regardless of whether an exception occurred. It closes the `Scanner` to prevent resource leaks.

Programme

```
import java.util.Scanner;

public class MultipleCatchExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter a number: ");
            int num = scanner.nextInt();

            int result = 100 / num;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
                System.out.println("Error: Division by
  zero is not allowed.");
        } catch (java.util.InputMismatchException e) {
             System.out.println("Error: Please enter a
  valid integer.");
        } catch (Exception e) {
               System.out.println("An unexpected error
  occurred: " + e.getMessage());
        } finally {
            System.out.println("Execution finished.");
            scanner.close();
        }
    }
}
```

out put

```
Enter a number: 45
Result: 2
Execution finished.
```

c.Write a JAVA program for creation of Java Built-in Exceptions

Description

Java's inbuilt exceptions, often called standard or built-in exceptions, are predefined exception classes in the Java API to handle common error scenarios. They derive from the Throwable class and are categorized into checked exceptions (like IOException) that must be explicitly handled, and unchecked exceptions (like NullPointerException and ArithmeticException) that derive from RuntimeException and are typically due to programming errors.

Programme

```
public class Main {

    public static void main(String[] args) {

        // ArrayIndexOutOfBoundsException
        try {
            int[] array = {1, 2, 3};
            System.out.println(array[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());
        }

        // NullPointerException
        try {
            String str = null;
            System.out.println(str.length());
        } catch (NullPointerException e) {
            System.out.println("Caught NullPointerException.");
        }

        // ArithmeticException (Division by zero)
        try {
            int result = 10 / 0;

            System.out.println(result);

        } catch (ArithmeticException e) {

            System.out.println("Caught ArithmeticException: " + e.getMessage());

        }



        // NumberFormatException

        try {

            int number = Integer.parseInt("NotANumber");

            System.out.println(number);

        } catch (NumberFormatException e) {

            System.out.println("Caught NumberFormatException: " + e.getMessage());

        }

    }

}
```

out put

```
Caught ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3

Caught NullPointerException.

Caught ArithmeticException: / by zero

Caught NumberFormatException: For input string: "NotANumber"
```

d.Write a JAVA program for creation of User Defined Exception

description

n Java, user-defined exceptions allow developers to create custom exception types tailored to specific application needs. These are typically created by extending the Exception class or its subclasses. By defining and throwing these exceptions, developers can implement more precise error-handling mechanisms in their programs.

Programme

PROGRAM:

```
class AgeValidationException extends Exception {
    public AgeValidationException(String message) {
        super(message);
    }
}


public class Main {


    public static void main(String[] args) {
        try {
            validateAgeForVoting(15);
        } catch (AgeValidationException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }
    public static void validateAgeForVoting(int age) throws AgeValidationException {
        if (age < 18) {
            throw new AgeValidationException("Age is not valid for voting. Must be at least 18.");
        } else {
            System.out.println("Age is valid for voting.");
        }
    }
}
```

out put

```
Caught exception: Age is not valid for voting. Must be at least 18.
```