Exception Handling in Java

Exception handling in Java is a powerful mechanism to handle runtime errors, ensuring the smooth execution of the program. It is managed via five keywords: try, catch, finally, throw, and throws.

1. Try and Catch Block

Try Block: The code that might throw an exception is placed inside the try block. If an exception occurs, it is handled by the catch block.

Catch Block: The catch block follows the try block and catches exceptions thrown by the try block. You can have multiple catch blocks to handle different types of exceptions.

```
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            int data = 50 / 0;
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught: " +
e.getMessage());
        }
        System.out.println("Rest of the code executes...");
    }
}
```

Exception Hierarchy

The Java exception hierarchy is as follows:

- Throwable
  - Error
    - OutOfMemoryError
    - StackOverflowError
  - Exception
    - IOException

- SQLException
- RuntimeException
  - ArithmeticException
  - NullPointerException
  - ArrayIndexOutOfBoundsException

Checked Exceptions in Java

Checked exceptions are exceptions that are checked at compile-time by the Java compiler. If a method is capable of throwing a checked exception, it must either handle the exception using a try-catch block or declare it in its method signature using the throws keyword. Failure to do so results in a compilation error.

Common Types of Checked Exceptions

1. **IOException**
2. **SQLException**
3. **FileNotFoundException**
4. **ClassNotFoundException**
5. **InstantiationException**
6. **IllegalAccessException**
7. **NoSuchMethodException**
8. **InvocationTargetException**

Examples of Common Checked Exceptions

1. IOException

Example:

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;
```

```
public class IOExceptionExample {

    public static void main(String[] args) {

        try {

            BufferedReader reader = new BufferedReader(new
FileReader("example.txt"));

            String line;

            while ((line = reader.readLine()) != null) {

                System.out.println(line);

            }

            reader.close();

        } catch (IOException e) {

            System.out.println("An IOException occurred: " +
e.getMessage());

        }

    }

}
```

This example reads lines from a file and prints them. If the file is not found or an I/O error occurs, an IOException is thrown.

ClassNotFoundException

Example:

```
public class ClassNotFoundExceptionExample {

    public static void main(String[] args) {

        try {

            Class.forName("com.mysql.jdbc.Driver");

            System.out.println("Driver found!");

        } catch (ClassNotFoundException e) {

            System.out.println("Class not found: " +
e.getMessage());

        }

    }

}
```

Unchecked Exceptions in Java

Unchecked exceptions are exceptions that are not checked at compile-time but occur at runtime. These exceptions are subclasses of RuntimeException and represent programming errors that are often avoidable, such as logic errors or improper use of an API.

Common Types of Unchecked Exceptions

1. **ArithmeticException**
2. **NullPointerException**
3. **ArrayIndexOutOfBoundsException**
4. **IllegalArgumentException**
5. **IllegalStateException**
6. **NumberFormatException**

7. **ClassCastException**

Examples of Common Unchecked Exceptions

1. ArithmeticException

Example:

```
public class ArithmeticExceptionExample {

    public static void main(String[] args) {

        try {

            int result = 10 / 0; // This will throw
ArithmeticException

        } catch (ArithmeticException e) {

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

        }

    }

}
```

This example attempts to divide a number by zero, which results in an ArithmeticException.

---

2. NullPointerException

Example:

```
public class NullPointerExceptionExample {

    public static void main(String[] args) {

        try {

            String str = null;

            System.out.println(str.length()); // This will throw
NullPointerException

        } catch (NullPointerException e) {

            System.out.println("NullPointerException caught: " +
e.getMessage());

        }

    }

}
```

This example tries to call a method on a null object reference, leading to a NullPointerException.

---

3. ArrayIndexOutOfBoundsException

Example:

```
public class ArrayIndexOutOfBoundsExceptionExample {

    public static void main(String[] args) {
```

```
        try {

            int[] arr = {1, 2, 3};

            System.out.println(arr[3]); // This will throw
ArrayIndexOutOfBoundsException

        } catch (ArrayIndexOutOfBoundsException e) {

            System.out.println("ArrayIndexOutOfBoundsException
caught: " + e.getMessage());

        }

    }

}
```

This example accesses an invalid index of an array, resulting in an ArrayIndexOutOfBoundsException.

---

4. IllegalArgumentException

Example:

```
public class IllegalArgumentExceptionExample {

    public static void main(String[] args) {

        try {

            printAge(-5); // This will throw IllegalArgumentException

        } catch (IllegalArgumentException e) {
```

```
        System.out.println("IllegalArgumentException caught: " +
e.getMessage());

        }

    }

    public static void printAge(int age) {

        if (age < 0) {

            throw new IllegalArgumentException("Age cannot be
negative");

        }

        System.out.println("Age: " + age);

    }

}
```

This example passes an invalid argument to a method, which throws an IllegalArgumentException.

---

5. IllegalStateException

Example:

```
import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;
```

```
public class IllegalStateExceptionExample {

    public static void main(String[] args) {

        List<String> list = new ArrayList<>();

        list.add("A");

        list.add("B");

        list.add("C");

        Iterator<String> iterator = list.iterator();

        while (iterator.hasNext()) {

            String value = iterator.next();

            if (value.equals("B")) {

                iterator.remove();

            }

        }

        try {

            iterator.remove(); // This will throw
IllegalStateException

        } catch (IllegalStateException e) {

            System.out.println("IllegalStateException caught: " +
e.getMessage());

        }

    }

}
```

This example improperly calls the remove method on an Iterator without a preceding call to next, resulting in an IllegalStateException.

---

6. NumberFormatException

Example:

```
public class NumberFormatExceptionExample {

    public static void main(String[] args) {

        try {

            int number = Integer.parseInt("XYZ"); // This will throw
NumberFormatException

        } catch (NumberFormatException e) {

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

        }

    }

}
```

This example attempts to parse a non-numeric string into an integer, causing a NumberFormatException.

---

7. ClassCastException

Example:

```
public class ClassCastExceptionExample {

    public static void main(String[] args) {

        try {

            Object obj = new Integer(100);

            String str = (String) obj; // This will throw
ClassCastException

        } catch (ClassCastException e) {

            System.out.println("ClassCastException caught: " +
e.getMessage());

        }

    }

}
```

This example tries to cast an object of one type to an incompatible type, resulting in a ClassCastException.

Summary

Unchecked exceptions are not required to be declared in a method or constructor's throws clause. They represent defects in the program (bugs), which should be corrected by the developer. Understanding and handling these exceptions is crucial for writing robust and error-free Java programs.

Finally Block

The finally block is used to execute important code such as closing resources. It is executed whether an exception is handled or not.

Example:

```
public class FinallyExample {

    public static void main(String[] args) {

        try {

            int data = 50 / 0; // This will throw ArithmeticException

        } catch (ArithmeticException e) {

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

        } finally {

            System.out.println("Finally block executed.");

        }

    }

}
```

Multiple Catch Block in Java

Java allows the use of multiple catch blocks to handle different types of exceptions that might be thrown by a try block. Each catch block is intended to catch and handle a specific type of exception.

Example: Multiple Catch Block

Here is an example demonstrating the use of multiple catch blocks:

```
public class MultipleCatchBlockExample {

    public static void main(String[] args) {

        try {

            int[] numbers = {1, 2, 3};

            int result = numbers[1] / 0; // This will throw ArithmeticException

            System.out.println(numbers[3]); // This will throw
ArrayIndexOutOfBoundsException

            String text = null;

            System.out.println(text.length()); // This will throw NullPointerException

        } catch (ArithmeticException e) {

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

        } catch (ArrayIndexOutOfBoundsException e) {

            System.out.println("ArrayIndexOutOfBoundsException caught: " +
e.getMessage());

        } catch (NullPointerException e) {

            System.out.println("NullPointerException caught: " + e.getMessage());

        } catch (Exception e) {

            System.out.println("General Exception caught: " + e.getMessage());

        }

        System.out.println("Rest of the code executes...");

    }

}
```

Explanation

1. **Try Block:**
    - The try block contains code that might throw different types of exceptions.
    - It attempts to divide by zero, which throws an ArithmeticException.
    - It tries to access an invalid array index, which throws an ArrayIndexOutOfBoundsException.
    - It attempts to call a method on a null reference, which throws a NullPointerException.
2. **Catch Blocks:**
    - Each catch block is designed to handle a specific type of exception.
    - The first catch block catches ArithmeticException.
    - The second catch block catches ArrayIndexOutOfBoundsException.
    - The third catch block catches NullPointerException.
    - The fourth catch block is a general Exception handler, which can catch any other exceptions not specifically caught by the previous blocks.
3. **Output:**
    - The ArithmeticException is caught first, and its message is printed.
    - The subsequent lines of code inside the try block are not executed because the first exception is caught and handled.
    - The program then continues to execute the code after the try-catch blocks.

Key Points

- Each catch block can handle a specific type of exception.
- The catch blocks are evaluated in the order they appear, and the first one that matches the thrown exception type is executed.
- If an exception is caught by one of the catch blocks, the remaining catch blocks are skipped.
- Having a general catch block for Exception at the end ensures that any other unhandled exceptions are caught, which helps in making the program more robust.

Java Nested Try

In Java, you can use nested try blocks to handle exceptions more granularly. This means placing one try block inside another try block, allowing you to catch and handle exceptions separately for different parts of your code. This can be useful when you have multiple operations that might throw exceptions, and you want to handle each operation's exceptions individually.

Example: Nested Try Block

Here is an example demonstrating the use of nested try blocks:

```
public class NestedTryExample {

    public static void main(String[] args) {

        try {

            // Outer try block

            int[] numbers = {1, 2, 3};



            try {

                // Inner try block 1

                int result = numbers[1] / 0; // This will throw
ArithmeticException

            } catch (ArithmeticException e) {

                System.out.println("ArithmeticException caught in
inner try block 1: " + e.getMessage());

            }
```

```
            try {

                // Inner try block 2

                System.out.println(numbers[3]); // This will throw
ArrayIndexOutOfBoundsException

            } catch (ArrayIndexOutOfBoundsException e) {

                System.out.println("ArrayIndexOutOfBoundsException
caught in inner try block 2: " + e.getMessage());

            }



            String text = null;

            System.out.println(text.length()); // This will throw
NullPointerException



        } catch (NullPointerException e) {

            System.out.println("NullPointerException caught in outer
try block: " + e.getMessage());

        } catch (Exception e) {

            System.out.println("General Exception caught in outer try
block: " + e.getMessage());

        }

        System.out.println("Rest of the code executes...");

    }

}
```

Explanation

1. **Outer Try Block:**
    - The outer try block contains the main logic of the program. It wraps around two inner try blocks and additional code that might throw exceptions.
2. **Inner Try Block 1:**
    - This try block attempts to perform an arithmetic operation that will result in an ArithmeticException.
    - The corresponding catch block handles this specific exception.
3. **Inner Try Block 2:**
    - This try block attempts to access an invalid index of the array, resulting in an ArrayIndexOutOfBoundsException.
    - The corresponding catch block handles this specific exception.
4. **Outer Catch Blocks:**
    - If any other exception occurs outside the inner try blocks but within the outer try block, it will be caught here.
    - A specific catch block handles NullPointerException.
    - A general catch block handles any other exceptions.
5. **Output:**
    - The ArithmeticException is caught by the first inner catch block, and its message is printed.
    - The ArrayIndexOutOfBoundsException is caught by the second inner catch block, and its message is printed.
    - The NullPointerException is caught by the outer catch block, and its message is printed.
    - Finally, the program continues executing the code after the try-catch blocks.

Key Points

- **Nested Try Blocks:** Allow handling of exceptions at different levels of code, providing more granular control over exception handling.
- **Inner Catch Blocks:** Handle exceptions specific to the inner try blocks, allowing for targeted exception handling.
- **Outer Catch Blocks:** Handle exceptions not caught by inner catch blocks, ensuring that all potential exceptions are handled.

Using nested try blocks can be useful for complex operations where different parts of the code might throw different exceptions, allowing for more precise and organized exception handling.

Throw and Throws

Throw Keyword: The throw keyword is used to explicitly throw an exception.

Example:

```
public class ThrowExample {
    public static void main(String[] args) {
        validateAge(15);
    }

    static void validateAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Not valid age");
        } else {
            System.out.println("Welcome!");
        }
    }
}
```

Throws Keyword: The throws keyword is used in a method signature to declare that a method can throw one or more exceptions.

Example:

```
import java.io.IOException;

public class ThrowsExample {
```

```
    public static void main(String[] args) {
        try {
            methodWithThrows();
        } catch (IOException e) {
            System.out.println("IOException caught: " + e.getMessage());
        }
    }

    static void methodWithThrows() throws IOException {
        throw new IOException("Example IOException");
    }
}
```

When to Use throws

1. **Checked Exceptions:**
    - Checked exceptions are exceptions that the Java compiler forces you to handle. They are subclasses of Exception but not RuntimeException.
    - If a method can throw a checked exception, it must either handle the exception within the method using a try-catch block or declare that it throws the exception using the throws keyword.
2. **Propagation of Exceptions:**
    - The throws keyword is used to declare that a method can throw one or more exceptions. This allows the calling method to handle the exception or declare it further.

Example: Method Using throws

Here's how you use the throws keyword with a method that throws a checked exception using the throw keyword:

```
import java.io.IOException;

public class ThrowsKeywordExample {
    public static void main(String[] args) {
        try {
```

```
            readFile("nonexistentfile.txt");
        } catch (IOException e) {
            System.out.println("Exception caught in main: " + e.getMessage());
        }

        System.out.println("Rest of the code executes...");
    }

    // Method declaration with `throws` keyword
    public static void readFile(String fileName) throws IOException {
        if (!fileName.equals("validfile.txt")) {
            // Throwing an IOException using `throw` keyword
            throw new IOException("File not found");
        }
        System.out.println("File read successfully");
    }
}
```

Key Points

- **Declaring Exceptions:**
  - The readFile method is declared with throws IOException, which means it can throw an IOException. This declaration tells the compiler and users of the method that they need to handle this exception.
- **Throwing Exceptions:**
  - Inside the readFile method, the throw keyword is used to actually throw the IOException when a condition is met.
- **Handling Exceptions:**
  - In the main method, the readFile method is called within a try block, and the IOException is caught and handled in the corresponding catch block.

Unchecked Exceptions

For unchecked exceptions (subclasses of RuntimeException), you are not required to use the throws keyword in the method signature, as they are not subject to the

compile-time checking that checked exceptions are. These exceptions can be thrown without being declared in the method signature.

Example: Unchecked Exception

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        try {
            divide(10, 0); // This will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Exception caught in main: " + e.getMessage());
        }

        System.out.println("Rest of the code executes...");
    }

    // No `throws` keyword needed for unchecked exceptions
    public static void divide(int a, int b) {
        if (b == 0) {
            // Throwing an ArithmeticException
            throw new ArithmeticException("Cannot divide by zero");
        }
        System.out.println("Result: " + (a / b));
    }
}
```

Summary

- **Checked Exceptions:** Use the throws keyword in the method signature to declare that a method can throw checked exceptions, allowing for proper handling or further declaration of the exception.
- **Unchecked Exceptions:** The throws keyword is not required for unchecked exceptions, although you can still use it if you want to make it explicit.

The throws keyword helps manage error handling by ensuring that methods explicitly declare the exceptions they might throw, promoting better control and readability of exception handling in your Java code.

Custom Exceptions

Custom exceptions are user-defined exceptions that extend the Exception class. They allow you to create specific error types that are more meaningful for your application.

Creating a Custom Exception

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

Using a Custom Exception

```
public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (CustomException e) {
            System.out.println("Caught custom exception: " + e.getMessage());
        }
    }

    public static void validateAge(int age) throws CustomException {
        if (age < 18) {
            throw new CustomException("Age must be 18 or older");
        }
    }
}
```

let's consider another example where we create a custom exception to handle an invalid bank transaction scenario. This example will demonstrate how to create and use a custom exception to enforce business rules, such as ensuring sufficient account balance for withdrawals.

Step 1: Define the Custom Exception

```
class InsufficientFundsException extends Exception {
    private double amount;
    public InsufficientFundsException(String message, double amount) {
        super(message);
        this.amount = amount;
    }

    public double getAmount() {
        return amount;
    }
}
```

Step 2: Implement the Bank Account Class

```
public class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public void withdraw(double amount) throws InsufficientFundsException {
        if (amount > balance) {
```

```
            throw new InsufficientFundsException("Insufficient funds for withdrawal",
amount - balance);
        }
        balance -= amount;
    }

    public double getBalance() {
        return balance;
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount(100.0);

        try {
            System.out.println("Depositing $50...");
            account.deposit(50.0);
            System.out.println("New balance: $" + account.getBalance());

            System.out.println("Withdrawing $200...");
            account.withdraw(200.0);
        } catch (InsufficientFundsException e) {
            System.out.println("Exception: " + e.getMessage());
            System.out.println("Shortfall: $" + e.getAmount());
        }
    }
}
```

Explanation

1. **Custom Exception Class**: InsufficientFundsException includes a constructor that accepts a message and an amount, representing the shortfall. It also provides a method to retrieve the shortfall amount.
2. **Bank Account Class**:
    - The BankAccount class includes methods to deposit and withdraw funds.
    - The withdraw method checks if the requested withdrawal amount is greater than the current balance. If it is, it throws an InsufficientFundsException with a message and the shortfall amount.

- The main method demonstrates the usage of the BankAccount class and handles the InsufficientFundsException.

Output

When you run the above program, you will see the following output:

```
Depositing $50...
New balance: $150.0
Withdrawing $200...
Exception: Insufficient funds for withdrawal
Shortfall: $50.0
```

Custom Exception for Invalid Login Attempts

This example demonstrates a custom exception for handling invalid login attempts.

Step 1: Define the Custom Exception

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

Step 2: Implement the User Authentication Class

```
import java.util.HashMap;
import java.util.Map;
```

```
public class UserAuthentication {
    private Map<String, String> users = new HashMap<>();

    public UserAuthentication() {
        // Add some dummy users
        users.put("user1", "password1");
        users.put("user2", "password2");
    }

    public void login(String username, String password) throws InvalidLoginException {
        if (!users.containsKey(username) || !users.get(username).equals(password)) {
            throw new InvalidLoginException("Invalid username or password");
        }
        System.out.println("User " + username + " logged in successfully");
    }

    public static void main(String[] args) {
        UserAuthentication auth = new UserAuthentication();

        try {
            auth.login("user1", "password1"); // Successful login
            auth.login("user2", "wrongpassword"); // This will throw an exception
        } catch (InvalidLoginException e) {
            System.out.println("Login failed: " + e.getMessage());
        }
    }
}
```

Explanation

1. **Custom Exception Class**: InvalidLoginException is a simple exception class with a constructor that accepts a message.
2. **User Authentication Class**:
    - The UserAuthentication class simulates a basic user authentication system using a HashMap to store user credentials.

- The login method checks the provided username and password against the stored credentials. If the credentials are invalid, it throws an InvalidLoginException with an appropriate message.
- The main method demonstrates the usage of the UserAuthentication class and handles the InvalidLoginException.

Output

When you run the above program, you will see the following output:

```
User user1 logged in successfully
Login failed: Invalid username or password
```

Summary

- Custom exceptions provide specific and meaningful error handling tailored to application-specific error conditions.
- They improve code readability and maintainability by providing clear error messages.
- Examples include handling insufficient funds in a bank account and invalid login attempts in a user authentication system.

Try-With-Resources in Java

The try-with-resources statement is a feature introduced in Java 7 that simplifies the process of managing resources such as files, database connections, sockets, etc. It ensures that each resource is closed at the end of the statement, thereby reducing the risk of resource leaks.

Key Points:

1. **Automatic Resource Management**: Resources are automatically closed after the execution of the try block.

2. **Resources Must Implement AutoCloseable**: Any resource used in a try-with-resources statement must implement the AutoCloseable interface, which includes the close() method.
3. **Simplified Code**: Reduces boilerplate code for resource management, making code more readable and less error-prone.

Example: Reading a File Using Try-With-Resources

Before Java 7 (Without Try-With-Resources):

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

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

After Java 7 (With Try-With-Resources):

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

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

Explanation

1. **Resource Declaration**:
    - In the try-with-resources statement, the resource (e.g., BufferedReader) is declared within the parentheses following the try keyword.
    - The resource is initialized when the statement is executed.
2. **Automatic Closing**:
    - When the try block finishes, whether normally or abruptly (due to an exception), the close() method is automatically called on the resource.
    - This happens even if an exception occurs, ensuring that the resource is always properly closed.
3. **Multiple Resources**:
    - You can declare multiple resources in a single try-with-resources statement by separating them with semicolons.

Example with Multiple Resources:

```
import java.io.BufferedReader;
```

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

public class MultipleResourcesExample {
    public static void main(String[] args) {
        try (
            BufferedReader reader = new BufferedReader(new FileReader("input.txt"));
            FileWriter writer = new FileWriter("output.txt")
        ) {
            String line;
            while ((line = reader.readLine()) != null) {
                writer.write(line + "\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Summary

- **Try-With-Resources**: Introduced in Java 7, it simplifies resource management by automatically closing resources.
- **Resources Must Implement AutoCloseable**: Ensures that the close() method is called automatically.
- **Reduces Boilerplate Code**: Makes code cleaner, more readable, and less prone to resource leaks.
- **Multiple Resources**: Can handle multiple resources in a single try-with-resources statement.

The try-with-resources statement is a powerful feature in Java that helps manage resources more effectively and reduces the risk of resource leaks by ensuring that all resources are closed automatically.

Best Practices for Exception Handling

✅ **Use specific exceptions instead of generic `Exception`.**
❌ **Bad:**

```
catch (Exception e) { System.out.println("Error occurred"); }
```

✅ **Good:**

```
catch (IOException e) { System.out.println("File error: " +
e.getMessage()); }
```

✅ **Close resources using `try-with-resources`.**

```
try (BufferedReader br = new BufferedReader(new
FileReader("file.txt"))) {

    System.out.println(br.readLine());

} catch (IOException e) {

    e.printStackTrace();

}
```

✅ **Use meaningful exception messages.**

```
throw new IllegalArgumentException("Invalid age: Age must be >=
18");
```

Exception Handling Best Practices

1. Only catch exceptions that you can handle.
2. Catch specific exceptions before more general ones.
3. Don't catch Throwable unless you're sure you want to catch both exceptions and errors.
4. Use try-with-resources for automatic resource management.
5. Log exceptions for debugging purposes.
6. Don't swallow exceptions (catch and do nothing).
7. Clean up resources in a finally block or use try-with-resources.

Conclusion

🔹 Java provides a robust exception handling mechanism using **try-catch, finally, throw, throws, and custom exceptions**.

🔹 Proper exception handling **prevents application crashes** and **improves readability**.

🔹 Best practices ensure **efficient debugging and maintainability**.