```
Object
 └── Throwable
      ├── Error (Unchecked - serious problems, usually not handled)
      │    ├── VirtualMachineError
      │    │    ├── OutOfMemoryError
      │    │    ├── StackOverflowError
      │    │    ├── InternalError
      │    │    └── UnknownError
      │    ├── LinkageError
      │    │    ├── NoClassDefFoundError
      │    │    ├── ClassFormatError
      │    │    ├── UnsupportedClassVersionError
      │    │    └── VerifyError
      │    ├── AssertionError
      │    ├── AWTError
      │    └── IOError
      │
      └── Exception (Handled problems, recoverable)
           ├── RuntimeException (Unchecked - program bugs)
           │    ├── ArithmeticException
           │    ├── NullPointerException
           │    ├── IndexOutOfBoundsException
           │    │    ├── ArrayIndexOutOfBoundsException
           │    │    └── StringIndexOutOfBoundsException
           │    ├── ClassCastException
           │    ├── IllegalArgumentException
           │    │    ├── NumberFormatException
           │    ├── IllegalStateException
           │    ├── ConcurrentModificationException
           │    ├── UnsupportedOperationException
           │    ├── SecurityException
           │    └── MissingResourceException
           │
           └── (Checked exceptions)
                ├── IOException
                │    ├── FileNotFoundException
                │    ├── EOFException
                │    ├── InterruptedIOException
                │    ├── ObjectStreamException
                │    │    ├── InvalidClassException
                │    │    └── NotSerializableException
                │    └── SocketException
                │
                ├── SQLException
                ├── ClassNotFoundException
                ├── InvocationTargetException
                │
```

```
├── InterruptedException
├── NoSuchMethodException
├── NoSuchFieldException
├── CloneNotSupportedException
├── ParseException
├── TimeoutException
├── InstantiationException
└── ReflectiveOperationException
```

📋 Java Exception & Error Cheat Sheet

Type	Class Name	Meaning / Common Use
Checked Exception	`IOException`	Input/Output failure
	`FileNotFoundException`	File does not exist
	`EOFException`	End of file unexpectedly reached
	`InterruptedIOException`	I/O operation interrupted
	`ObjectStreamException`	Problems with object serialization
	`InvalidClassException`	Incompatible serialized class
	`NotSerializableException`	Class does not support serialization
	`SocketException`	Socket/network errors
	`SQLException`	Database access error
	`ClassNotFoundException`	Requested class not found
	`InstantiationException`	Cannot instantiate abstract class/interface
	`IllegalAccessException`	Cannot access class member

`InvocationTargetException`	Reflection method exception
`InterruptedException`	Thread interrupted
`NoSuchMethodException`	No such method via reflection
`NoSuchFieldException`	No such field via reflection
`ParseException`	Parsing problem (like dates)
`TimeoutException`	Operation timed out
`CloneNotSupportedException`	Object does not support clone

Type	Class Name	Meaning / Common Use
Unchecked Exception (RuntimeException)	`ArithmeticException`	Division by zero, bad math
	`NullPointerException`	Using object with null reference
	`ArrayIndexOutOfBoundsException`	Wrong array index access
	`StringIndexOutOfBoundsException`	Wrong string index access
	`ClassCastException`	Bad casting between classes
	`IllegalArgumentException`	Method received wrong argument
	`NumberFormatException`	Bad conversion to number
	`IllegalStateException`	Bad method call at wrong time
	`ConcurrentModificationException`	Concurrent modification on collections

	`UnsupportedOperationException`	Operation not supported
	`SecurityException`	Security manager violation
	`MissingResourceException`	Missing resource like locale files

Type	Class Name	Meaning / Common Use
Errors	`OutOfMemoryError`	JVM memory exhausted
	`StackOverflowError`	Stack overflow (too deep recursion)
	`InternalError`	JVM internal error
	`UnknownError`	Unknown JVM error
	`NoClassDefFoundError`	Class definition not found at runtime
	`ClassFormatError`	Class file format incorrect
	`UnsupportedClassVersionError`	Class compiled with newer version
	`VerifyError`	Bytecode verification failed
	`AssertionError`	Failed assert statement
	`AWTError`	GUI system error
	`IOError`	General I/O error
	`VirtualMachineError`	Serious JVM error like memory issues

🧠 Quick Mnemonics:

Memory Tip	Meaning
"Checked = Controlled"	External issues you must handle (like files, networks).
"Runtime = Risky Code"	Programming bugs you can (but don't have to) check.

"Error = Endgame"	Something went *seriously* wrong, usually cannot recover.

🎯 Fast Visual Map:

- Throwable
  - Error (Red ❌) → Very serious → OutOfMemoryError, StackOverflowError...
  - Exception
    - Checked (Green ✅) → IOException, SQLException...
    - Unchecked (Orange ⚡) → NullPointerException, ClassCastException...

✨ Final Tip for Interview:

- Always mention Java forces you to handle *Checked Exceptions* (or declare them).
- Unchecked Exceptions and Errors are not forced — but good code defends against them.

Would you also like:

- A PDF of this table 📄?
- A mind map image version? 🧠🖼️

👉 Tell me — I can prepare it if you want!

What is an Error?

- Error refers to serious problems that a reasonable application should not try to catch.
- Errors usually happen outside the control of the program (e.g., hardware failure, memory overflow).
- Example in Java: `OutOfMemoryError`, `StackOverflowError`.

What is an Exception?

- Exception refers to problems that a program should try to catch and handle.
- They are conditions that an application might want to catch (like wrong user input, file not found, network issues).
- Example: `IOException`, `NullPointerException`, `ArithmeticException`.

Difference between Error and Exception

Aspect	Error	Exception
Definition	Serious problems mostly outside application control.	Issues within the application logic that can be handled.
Handling	Generally cannot be handled in the program.	Can be caught and handled using `try-catch`.
Examples	`OutOfMemoryError`, `VirtualMachineError`	`ArrayIndexOutOfBoundsException`, `IOException`
When Happens	Resource failures, JVM crashes	User mistakes, logical errors, failed operations

Recovery	Program usually cannot recover.	Program can recover and continue running.

Can We Handle Errors?

- In general, no, we should not handle Errors because they usually represent conditions that are too severe to recover from.
- Technically, yes, you *can* catch an `Error` using `catch (Error e)` — but it's strongly discouraged unless you have a very specific, expert-level reason.

Example:

```
try {
    // Some risky code
} catch (Error e) {
    System.out.println("Caught an error, but this is not recommended.");
}
```

But again: handling errors is not a good practice. It's better to let them terminate the program and investigate the root cause.

Would you like me to also explain this with a simple real-world analogy? 🚀

How to Handle Exceptions

In most programming languages (like Java, Python, C++), exceptions are handled using three main keywords:

- `try`
- `catch`

- `finally` (optional)

Steps to Handle Exception

1. Try Block:
    - Put the risky code (the code that might cause an exception) inside a `try` block.
2. Catch Block:
    - If an exception happens inside `try`, the program jumps to the `catch` block.
    - Here, you can define how you want to handle the problem (show a message, retry, log it, etc.).
3. Finally Block (Optional):
    - Code in the `finally` block always runs, whether an exception occurs or not.
    - It's usually used to release resources (like closing files, closing database connections).

Simple Example (Java)

```
public class Example {
    public static void main(String[] args) {
        try {
            int a = 5 / 0; // risky code, will throw
ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        } finally {
            System.out.println("Finally block: Always runs.");
        }
    }
}
```

Output:

```
Cannot divide by zero!
Finally block: Always runs.
```

Key Points:

- You can have multiple catch blocks for different types of exceptions.
- The `finally` block always executes, even if there is a `return` inside `try` or `catch`.
- If an exception is not handled, the program terminates abnormally.

Types of Exception Handling Techniques

1. Try-Catch: Catch exceptions and handle them.
2. Try-Catch-Finally: Handle exceptions and clean up resources.
3. Throws Keyword: Declare an exception and let the caller handle it.
4. Custom Exception: Create your own exception classes when needed.

Real-World Example

Imagine you are withdrawing money from an ATM:

- `try`: You request ₹10,000 withdrawal.
- `catch`: If your balance is too low, ATM catches this problem and shows "Insufficient Balance."
- `finally`: ATM always prints a receipt at the end (whether withdrawal succeeds or fails).

Would you like me to also show you how to handle multiple exceptions together or create custom exceptions? 🌟
It's quite useful!

Why Use Multiple Catch Blocks?

- Different types of exceptions might occur inside the same `try` block.
- Each `catch` block can handle a specific exception separately.

Important Rule:

- Only one matching `catch` block will execute (the first matching one).
- Catch from most specific to most general exception types.

Syntax for Multiple Catch Blocks

```
try {
    // Risky code that might throw different exceptions
} catch (ExceptionType1 e1) {
    // Handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle ExceptionType2
} catch (Exception e) {
    // Handle any other exception (general)
}
```

Example Program

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]);  //
ArrayIndexOutOfBoundsException

            int result = 10 / 0;
```

```
 // ArithmeticException (this won't happen because first exception
occurs)
        }
catch (ArrayIndexOutOfBoundsException e)
{
            System.out.println("Array index is out of bounds!");
 }
catch (ArithmeticException e)
{
            System.out.println("Cannot divide by zero!");
}
catch (Exception e)
{
            System.out.println("Some other exception occurred.");
 }
    }
}
```

Output:

```
Array index is out of bounds!
```

Important Points:

- Once an exception occurs, the program immediately jumps to the matching catch block and skips the rest of try.
- Other `catch` blocks will not be checked if one `catch` is already executed.
- Always keep more specific exceptions before more general exceptions (`Exception` class should be caught last).

Real-life Example Analogy

Imagine you're driving:

- If a tire bursts (`catch TireBurstException`), you fix it.
- If engine fails (`catch EngineFailureException`), you call a mechanic.
- If any other unexpected issue happens (`catch GeneralException`), you call emergency services.

Would you also like me to show an advanced version with multi-catch (single catch for multiple exceptions)? 🚀 (Java 7+ feature)
It's very handy too!

You are asking for a Level 3 Stack Trace example — meaning an exception that occurs deep inside nested method calls (method inside method inside method), and how the stack trace shows the chain of method calls when the exception happens.

Let's do it step-by-step:

Simple Java Example: Stack Trace with 3 Levels

```
public class StackTraceExample {

    public static void main(String[] args) {
        method1();  // Level 1
    }

    public static void method1() {
        method2();  // Level 2
    }

    public static void method2() {
        method3();  // Level 3
    }

    public static void method3() {
        int result = 10 / 0;  // This will cause ArithmeticException
    }
}
```

What Happens Here?

- `main()` calls `method1()`
- `method1()` calls `method2()`
- `method2()` calls `method3()`
- Inside `method3()`, we divide by zero → ArithmeticException occurs.

Expected Output (Stack Trace)

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at StackTraceExample.method3(StackTraceExample.java:17)
    at StackTraceExample.method2(StackTraceExample.java:13)
    at StackTraceExample.method1(StackTraceExample.java:9)
    at StackTraceExample.main(StackTraceExample.java:5)
```

Explanation of Stack Trace:

- The exception message appears first:
  ➔ `java.lang.ArithmeticException: / by zero`
- Then the call hierarchy is printed in reverse (from where it happened back to main):
  - Line 17: Error in `method3`
  - Line 13: `method2` called `method3`
  - Line 9: `method1` called `method2`
  - Line 5: `main` called `method1`

The topmost line is where the exception actually occurred.

Diagram View (for better understanding)

```
main()
   ↓
method1()
   ↓
method2()
   ↓
method3()  ← Exception occurs here
```

Summary:

- A stack trace shows the path the program took before it crashed.
- Very useful for debugging — you can trace back to the root cause easily.

Would you also like me to show an improved version where we catch the exception and print the stack trace manually? 📚
It looks a little different and is very useful too!

Understanding `throws` in Java

In Java, the `throws` keyword is used to declare that a method might throw certain exceptions. It is a way of saying, "This method doesn't handle these exceptions; instead, it passes them to the caller for handling."

When to Use `throws`

- You use `throws` in the method signature when a method might throw a checked exception (i.e., an exception that the compiler forces you to handle, such as `IOException`, `SQLException`, etc.).
- The method that calls the method with `throws` is responsible for handling the exception (either by catching it with a `try-catch` block or by declaring it with `throws` too).

Syntax of `throws`

```
public returnType methodName() throws ExceptionType1, ExceptionType2
{
    // method body
}
```

- **Example:**
  If a method might throw `IOException` and `SQLException`, you'd declare it like this:
  `throws IOException, SQLException`

Example 1: `throws` to Declare Exceptions

```
import java.io.*;

public class ThrowsExample {
    public static void main(String[] args) {
        try {
            readFile();  // Calling method that might throw an
exception
        } catch (IOException e) {
            System.out.println("Caught an IOException: " +
e.getMessage());
        }
    }

    // This method declares that it may throw an IOException

    public static void readFile() throws IOException {

        FileReader file = new FileReader("non_existent_file.txt");
// This might throw IOException

        BufferedReader reader = new BufferedReader(file);
        reader.readLine();
    }
}
```

Explanation:

- `readFile()` method is declared with `throws IOException`, meaning it does not handle the `IOException` internally.
- `main()` method is responsible for handling the exception, which it does using a `try-catch` block.

Output:

```
Caught an IOException: non_existent_file.txt (No such file or
directory)
```

Example 2: `throws` with Multiple Exceptions

```
import java.sql.*;

public class MultipleThrowsExample {

    public static void main(String[] args) {
        try {
            connectToDatabase();  // Calling method that might throw
exceptions
        } catch (SQLException | ClassNotFoundException e) {
            System.out.println("Error occurred: " + e.getMessage());
        }
    }

    // Declaring multiple exceptions in the throws clause
    public static void connectToDatabase() throws SQLException,
ClassNotFoundException {
        Class.forName("com.mysql.jdbc.Driver");  // This might throw
ClassNotFoundException
        Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/test",
"root", "password");  // Might throw SQLException
    }
}
```

Explanation:

- `connectToDatabase()` method declares both `SQLException` and `ClassNotFoundException` using `throws`.
- `main()` handles these exceptions using a `try-catch` block that can catch both types of exceptions (`SQLException` or `ClassNotFoundException`).

Output:

```
Error occurred: No suitable driver found for
jdbc:mysql://localhost:3306/test
```

Example 3: Chaining `throws` in Multiple Methods

In this example, we have a method calling another method that also declares exceptions with `throws`.

```
public class ThrowsChainingExample {

    public static void main(String[] args) {
        try {
            method1();  // Calling method1, which will propagate
exceptions
        } catch (Exception e) {
            System.out.println("Caught an exception: " +
e.getMessage());
        }
    }

    // This method calls another method, but doesn't handle the
exception
    public static void method1() throws Exception {
        method2();  // Passing the exception to method2
    }

    // This method actually throws the exception
    public static void method2() throws Exception {
        throw new Exception("Something went wrong in method2!");
    }
}
```

Explanation:

- `method1()` calls `method2()`, but it doesn't handle the exception, so it passes it up using `throws`.
- `method2()` throws an exception, and `main()` catches and handles it.

Output:

```
Caught an exception: Something went wrong in method2!
```

Key Points:

- `throws` is used in the method signature to declare that a method might throw one or more exceptions.
- It is used for checked exceptions, i.e., exceptions that need to be either caught or declared (like `IOException`, `SQLException`).
- The calling method is responsible for handling the exception if it's declared with `throws`.

Would you like an example with custom exceptions or handling runtime exceptions with `throws`? Let me know! 🌟

The `throw` keyword in Java is used to explicitly throw an exception. Unlike `throws`, which is used to declare that a method might throw an exception, `throw` is used to actually throw an exception at a particular point in the code.

Key Points About `throw`:

- `throw` is used inside a method to throw an exception.

- It is followed by an instance of an exception (either a pre-defined exception like `NullPointerException` or a custom exception).
- After an exception is thrown using `throw`, the program will stop executing the code that follows and will immediately transfer control to the nearest `catch` block, or the method will terminate if there's no `catch`.

Syntax:

java
CopyEdit
```
throw new ExceptionType("Error message");
```

Example 1: Basic Usage of `throw`

java
CopyEdit
```
public class ThrowExample {

    public static void main(String[] args) {
        try {
            checkAge(15);  // Age is below the required limit
        } catch (IllegalArgumentException e) {
            System.out.println("Exception caught: " +
e.getMessage());
        }
    }

    public static void checkAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or
older!");  // Explicitly throw an exception
        }
        System.out.println("Age is valid.");
    }
}
```

Explanation:

- `checkAge()` method checks if the age is less than 18. If it is, it throws an `IllegalArgumentException` with a message.
- In the `main()` method, the exception is caught inside the `try-catch` block and a message is printed.

Output:

php
CopyEdit
```
Exception caught: Age must be 18 or older!
```

Example 2: Throwing Custom Exceptions

You can also create and throw your own exceptions. For example:

java
CopyEdit
```
class AgeNotValidException extends Exception {
    public AgeNotValidException(String message) {
        super(message);
    }
}

public class CustomExceptionExample {

    public static void main(String[] args) {
        try {
            checkAge(16);  // Age is below the required limit
        } catch (AgeNotValidException e) {
            System.out.println("Caught a custom exception: " +
e.getMessage());
        }
    }

    public static void checkAge(int age) throws AgeNotValidException
{
        if (age < 18) {
            throw new AgeNotValidException("Age must be 18 or
older!");  // Throw custom exception
        }
```

```java
        System.out.println("Age is valid.");
    }
}
```

Explanation:

- We've created a custom exception `AgeNotValidException` that extends `Exception`.
- `checkAge()` throws the custom exception if the age is less than 18.
- The exception is caught in the `main()` method and handled appropriately.

Output:

css
CopyEdit
```
Caught a custom exception: Age must be 18 or older!
```

Example 3: Throwing an Exception Based on Condition

You can use `throw` to control the flow of your program and raise exceptions based on specific conditions.

java
CopyEdit
```java
public class TemperatureChecker {

    public static void main(String[] args) {
        try {
            checkTemperature(-10);  // Invalid temperature
        } catch (IllegalStateException e) {
            System.out.println("Caught an exception: " +
e.getMessage());
        }
    }

    public static void checkTemperature(int temp) {
        if (temp < 0) {
            throw new IllegalStateException("Temperature cannot be below 0!");  // Throw exception if condition is met
        }
        System.out.println("Temperature is valid.");
    }
}
```

Explanation:

- `checkTemperature()` checks if the temperature is below 0.
- If the condition is met, it throws an `IllegalStateException`.

Output:

yaml
CopyEdit
```
Caught an exception: Temperature cannot be below 0!
```

Key Differences Between `throw` and `throws`:

Feature	`throw`	`throws`
Purpose	Used to explicitly throw an exception.	Used to declare that a method might throw an exception.
Used By	Inside a method, to throw an exception.	In the method signature, to declare exceptions.
Syntax	`throw new ExceptionType("message");`	`public void methodName() throws ExceptionType { }`
Example	`throw new IllegalArgumentException("Invalid input!");`	`public void someMethod() throws IOException { }`

Summary:

- `throw` is used to throw exceptions deliberately from your code, whereas `throws` is used to declare that a method can throw certain exceptions, passing the responsibility of handling them to the calling method.

- You can use `throw` for both standard and custom exceptions to handle situations that would result in program failure or incorrect behavior.

Would you like to see more examples with specific exception types or more about creating custom exceptions? Let me know! 😄

```
// Online Java Compiler
// Use this editor to write, compile and run your Java code online
import java.util.*;
 class ss extends Exception{
    ss(String msg)
    {
        super(msg);
    }
}
public class Main {
    public static void main(String[] args) {
       try{
           Scanner sc=new Scanner(System.in);
           System.out.println("give number");
           int h=sc.nextInt();
           if(h>0)
           {
               throw new ss("hi hello this is exception");
           }
       }
       catch(Exception e)
       {
           e.printStackTrace();
           e.getMessage();
       }

    }
}
```

```
give number
11
ss: hi hello this is exception
	at Main.main(Main.java:18)

=== Code Execution Successful ===
```

Real-Time Custom Exception Examples

1. Bank Account - Insufficient Funds Exception

Imagine you're building a banking application where users can withdraw money. You need a custom exception to handle the situation when a user tries to withdraw more money than their current balance.

```
// Custom Exception for Insufficient Funds
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}

public class BankAccount {
    private double balance;

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

    public void withdraw(double amount) throws
InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException("Insufficient
funds! Your balance is " + balance);
        }
        balance -= amount;
        System.out.println("Withdrawal successful! New balance: " +
balance);
    }
}

public class BankingApp {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000);
        try {
            account.withdraw(1500);  // Trying to withdraw more than
the available balance
        } catch (InsufficientFundsException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Explanation:

- `InsufficientFundsException` is a custom exception.
- When the withdrawal amount is greater than the balance, it throws the exception.
- In the `main()` method, the exception is caught and the error message is displayed.

Output:

```
Error: Insufficient funds! Your balance is 1000.0
```

---

2. User Registration - Invalid Email Exception

In a user registration system, you might want to ensure that users enter a valid email address. If the email is invalid, throw a custom exception.

```
// Custom Exception for Invalid Email
class InvalidEmailException extends Exception {
    public InvalidEmailException(String message) {
        super(message);
    }
}

public class UserRegistration {

    public static void registerUser(String email) throws
InvalidEmailException {
        if (!email.contains("@") || !email.contains(".")) {
            throw new InvalidEmailException("Invalid email address:
" + email);
        }
        System.out.println("User registered with email: " + email);
    }
}

public class RegistrationApp {
    public static void main(String[] args) {
        try {
            UserRegistration.registerUser("invalid-email.com");  //
Invalid email format
        } catch (InvalidEmailException e) {
            System.out.println("Registration failed: " +
e.getMessage());
        }
    }
}
```

Explanation:

- `InvalidEmailException` is thrown if the email address doesn't contain both `@` and `.`.
- In `main()`, the exception is caught and the error message is printed.

Output:

less
CopyEdit
```
Registration failed: Invalid email address: invalid-email.com
```

---

3. Order Management - Out of Stock Exception

In an e-commerce platform, you might want to track product inventory. If a user tries to place an order for an item that is out of stock, a custom exception can be thrown.

java
CopyEdit
```
// Custom Exception for Out of Stock
class OutOfStockException extends Exception {
    public OutOfStockException(String message) {
        super(message);
    }
}

public class Product {
    private String name;
    private int stock;
    public Product(String name, int stock) {
        this.name = name;
        this.stock = stock;
    }

    public void orderProduct(int quantity) throws
OutOfStockException {
        if (quantity > stock) {
            throw new OutOfStockException(name + " is out of stock.
Only " + stock + " left.");
        }
        stock -= quantity;
        System.out.println(quantity + " " + name + "(s) ordered.
Remaining stock: " + stock);
    }
}

public class ECommerceApp {
    public static void main(String[] args) {
        Product laptop = new Product("Laptop", 5);
        try {
            laptop.orderProduct(6);  // Trying to order more than
available stock
        } catch (OutOfStockException e) {
            System.out.println("Order failed: " + e.getMessage());
        }
    }
}
```

Explanation:

- `OutOfStockException` is thrown when the quantity ordered exceeds the available stock.
- In the `main()` method, it's caught and an error message is displayed.

Output:

pgsql
CopyEdit

```
Order failed: Laptop is out of stock. Only 5 left.
```

---

4. File Upload - Invalid File Type Exception

In a file upload system, you may want to restrict file types (e.g., allow only `.jpg` or `.png` files). If the user uploads an invalid file type, you can throw a custom exception.

java
CopyEdit
```
// Custom Exception for Invalid File Type
class InvalidFileTypeException extends Exception {
    public InvalidFileTypeException(String message) {
        super(message);
    }
}

public class FileUpload {

    public static void uploadFile(String fileName) throws
InvalidFileTypeException {
        if (!fileName.endsWith(".jpg") &&
!fileName.endsWith(".png")) {
            throw new InvalidFileTypeException("Invalid file type: "
+ fileName + ". Only .jpg and .png allowed.");
        }
        System.out.println("File " + fileName + " uploaded
successfully!");
    }
}

public class UploadApp {
    public static void main(String[] args) {
        try {
            FileUpload.uploadFile("image.txt");  // Invalid file
type
        } catch (InvalidFileTypeException e) {
            System.out.println("Upload failed: " + e.getMessage());
        }
    }
}
```

Explanation:

- `InvalidFileTypeException` is thrown if the file type isn't `.jpg` or `.png`.
- In `main()`, the exception is caught and the error message is displayed.

Output:

pgsql
CopyEdit

```
Upload failed: Invalid file type: image.txt. Only .jpg and .png
allowed.
```

Key Points:

- Custom exceptions allow you to handle domain-specific errors that are not covered by built-in Java exceptions.
- By throwing custom exceptions, you can provide more meaningful and descriptive error messages.
- They improve the maintainability and readability of your code by making it more specific to your application's needs.