**1. Chained Exceptions**

In Java, sometimes one exception leads to another. To capture this cause-and-effect relation, **chained exceptions** were introduced in JDK 1.4. They allow linking one exception with another using initCause() and retrieving it with getCause(). This helps in debugging complex systems by preserving the original cause even if a new exception is thrown. The main idea is that the higher-level exception can wrap the lower-level exception. This is especially important in large applications where multiple layers (e.g., I/O, database, business logic) interact. By chaining, developers can see both the immediate error and the root cause.

**General Syntax**

Throwable(Throwable cause)

Throwable(String message, Throwable cause)

Throwable initCause(Throwable cause)

Throwable getCause()

**Example Program**

class ChainedExceptionDemo {

public static void main(String[] args) {

try {

divide();

} catch (Exception e) {

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

System.out.println("Original cause: " + e.getCause());

}

}

static void divide() {

try {

int data = 10 / 0; // ArithmeticException

} catch (ArithmeticException e) {

// wrap original cause inside RuntimeException

throw new RuntimeException("Error during division", e);

}

}

}

**Output**

Caught: java.lang.RuntimeException: Error during division

Original cause: java.lang.ArithmeticException: / by zero

**2. Three Recently Added Exception Features**

* **Multi-Catch (Java 7)**

Before Java 7, handling multiple exceptions required separate catch blocks. This often led to redundant code. With **multi-catch**, Java allows combining different exception types in a single catch block using the | operator. This improves readability and reduces code duplication. The variable in a multi-catch block is implicitly final, so it cannot be reassigned. This feature is especially useful when the handling code is identical for multiple exception types. It also prevents accidental mistakes where exception variables are reused incorrectly.

**General Syntax**

try {

// code that may throw exceptions

} catch (ExceptionType1 | ExceptionType2 ex) {

// common handling code

}

**Example Program**

class MultiCatchDemo {

public static void main(String[] args) {

try {

int arr[] = new int[5];

arr[10] = 50 / 0; // may throw both exceptions

} catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {

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

}

}

}

**Output**

Caught exception: java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 5

* **Try-With-Resources (Java 7)**

Resource management in Java (like closing files, sockets, or streams) traditionally required a finally block. If not handled properly, resources could leak. The **try-with-resources** statement solves this problem by automatically closing resources when the try block exits. Any class that implements the AutoCloseable interface can be used in this block. Multiple resources can be declared, separated by semicolons. This makes code cleaner, safer, and less error-prone. It is one of the most practical enhancements in Java exception handling.

**General Syntax**

try (ResourceType resource = new ResourceType()) {

// use the resource

} catch (ExceptionType e) {

// handle exception

}

**Example Program**

import java.io.\*;

class TryWithResourcesDemo {

public static void main(String[] args) {

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

String line = br.readLine();

System.out.println("File content: " + line);

} catch (IOException e) {

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

}

}

}

**Output (if test.txt contains "Hello Java")**

File content: Hello Java

* **Re-throwing Exceptions with Type Inference (Java 7)**

In older Java versions, when re-throwing an exception, the method signature had to declare the broadest type (often Exception), even if only specific types were rethrown. From Java 7, the compiler can **infer the precise exception type** when re-throwing. This is called **more precise rethrow**. It avoids redundant and misleading exception declarations. It works with final or effectively final variables. This improves type safety and makes code cleaner. It is particularly useful in utility methods that encapsulate multiple exception sources.

**General Syntax**

void method() throws SpecificException {

try {

// risky code

} catch (Exception e) {

throw e; // compiler infers the type

}

}

**Example Program**

import java.io.\*;

import java.nio.file.\*;

class RethrowDemo {

static void readFile(String path) throws IOException {

try {

Files.readAllLines(Path.of(path));

} catch (Exception e) {

throw e; // compiler infers IOException

}

}

public static void main(String[] args) {

try {

readFile("test.txt");

System.out.println("File read successfully.");

} catch (IOException e) {

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

}

}

}

**Output (if file does not exist)**

IOException caught: java.nio.file.NoSuchFileException: test.txt