

**Exceptions:** Exception-Handling Fundamentals, Exception Types, Uncaught Exceptions, Using try and catch, Multiple catch Clauses, Nested try Statements, throw, throws, finally, Java's Built-in Exceptions, Creating Your Own Exception Subclasses, Chained Exceptions.

**Exceptions:** In Java, an **exception** is an **unwanted or unexpected event** that occurs during the execution of a program, disrupting the normal flow of the program's instructions.

- It is an **object** that describes an error or an unusual condition that has occurred.
- Java uses exceptions to handle runtime errors, ensuring the program can recover gracefully without abruptly terminating.

**Types of Exceptions:** In Java, exceptions are classified into two main categories: **Checked Exceptions** and **Unchecked Exceptions**. There is also a special category for **Errors**, which are severe issues outside the programmer's control.

### Checked Exceptions (Compile-Time Exceptions)

Checked exceptions are exceptions that are checked by the compiler at compile-time. If not handled properly (using try-catch) or declared (using throws), the program will not compile.

#### Characteristics:

- Represent recoverable conditions.
- Examples: Issues with I/O, database operations, etc.
- The programmer must explicitly handle or declare them.

#### Examples:

1. `IOException`: Occurs during input/output operations.
2. `SQLException`: Related to database operations.
3. `FileNotFoundException`: Thrown when trying to access a file that does not exist.

```
import java.io.File;
import java.io.FileReader;

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            File file = new File("nonexistent.txt");
            FileReader reader = new FileReader(file);
        } catch (Exception e) {
            System.out.println("File not found: " + e.getMessage());
        }
    }
}
```

### Unchecked Exceptions (Runtime Exceptions)

Unchecked exceptions are exceptions that occur at runtime and are not checked by the compiler. They usually result from programming errors, such as improper logic or invalid usage of data structures.

#### Characteristics:

- Represent issues that occur due to bugs in the code (e.g., logical errors).
- Examples: Division by zero, null references, invalid array access, etc.

#### Examples:

1. **ArithmeticException:** Thrown for illegal arithmetic operations, e.g., division by zero.
2. **NullPointerException:** Thrown when trying to access an object with a null reference.
3. **ArrayIndexOutOfBoundsException:** Thrown when accessing an invalid array index.

```
public class UncheckedExceptionExample {  
    public static void main(String[] args) {  
        int[] numbers = {1, 2, 3};  
        try {  
            System.out.println(numbers[5]); // Throws ArrayIndexOutOfBoundsException  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Exception: " + e.getMessage());  
        }  
    }  
}
```

## Errors

Errors represent serious issues beyond the programmer's control, often related to the environment in which the program is running. They are not meant to be handled by the program.

### Characteristics:

- Represent fatal conditions.
- Examples: System failures, resource exhaustion, etc.
- Generally, the program cannot recover from errors.

### Examples:

1. **OutOfMemoryError:** Thrown when the JVM runs out of memory.
2. **StackOverflowError:** Thrown when the stack size is exceeded due to deep recursion.
3. **VirtualMachineError:** Thrown when the JVM fails to operate.

```
public class ErrorExample {  
    public static void main(String[] args) {  
        try {  
            recursiveCall(); // Causes StackOverflowError  
        } catch (StackOverflowError e) {  
            System.out.println("Error: Stack overflow occurred.");  
        }  
    }  
  
    public static void recursiveCall() {  
        recursiveCall();  
    }  
}
```

## Uncaught Exceptions in Java

An **uncaught exception** occurs when a program throws an exception but does not handle it using a try-catch block. This results in the program terminating abruptly and displaying an error message along with a stack trace.

### Characteristics of Uncaught Exceptions

1. **Runtime Termination:** The program stops execution immediately when the exception occurs.

2. **Stack Trace:** Java prints the stack trace to show where the exception originated and its propagation through the call stack.
3. **Common Cause:** Usually results from a lack of proper exception handling or missing checks for risky operations.
4. **Examples:** Uncaught exceptions often involve runtime exceptions like `ArithmeticException`, `NullPointerException`, or `ArrayIndexOutOfBoundsException`.

```
public class UncaughtExceptionExample {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 0;  
        // Division by zero without handling  
        int result = a / b; // Throws ArithmeticException  
        System.out.println("Result: " + result); // This line is not executed  
    }  
}
```

```
C:\TestJava>javac UncaughtExceptionExample.java
```

```
C:\TestJava>java UncaughtExceptionExample  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at UncaughtExceptionExample.main(UncaughtExceptionExample.java:6)
```

### Propagation of Uncaught Exceptions

When an exception is not caught:

1. The JVM checks the call stack for a try-catch block.
2. If no try-catch block is found in the current method, the exception propagates to the calling method.
3. If no method in the stack handles the exception, the JVM terminates the program and prints the stack trace.

### How to Prevent Uncaught Exceptions

1. **Use try-catch blocks:**  
Handle exceptions at appropriate points to avoid abrupt termination.
2. **Validate Inputs:**  
Ensure proper checks are in place to prevent exceptions.
3. **Log and Monitor Exceptions:**  
Use logging frameworks to capture and analyze exceptions instead of terminating the program.

### Using try and catch in Java

The try-catch mechanism in Java is used for exception handling, allowing developers to manage runtime errors gracefully. The try block contains the code that may throw an exception, and the catch block handles the exception if it occurs.

### Structure of try and catch

1. **try block:**

- Contains code that may generate exceptions.
- Only exceptions occurring within this block will be handled by the associated catch blocks.

## 2. catch block:

- Handles specific exceptions that occur in the try block.
- You can have multiple catch blocks for different exception types.

```
try {  
    // Code that may throw an exception  
} catch (ExceptionType1 e1) {  
    // Handle ExceptionType1  
} catch (ExceptionType2 e2) {  
    // Handle ExceptionType2  
}
```

### Key Points About Using try and catch

- **Only One Exception is Caught:** When an exception occurs, only the first matching catch block is executed, and the rest are ignored.
- **Order of catch Blocks:** The catch blocks should be ordered from most specific to most general exceptions. Otherwise, the compiler will throw an error.
- **The Exception Object:** Each catch block receives an exception object (e.g., e) that contains information about the exception.
- **Optional finally Block:** The finally block, if present, executes after the try or catch block, whether an exception occurs or not.

### Advantages of Using try and catch

1. **Prevents Program Crashes:** Ensures the program does not terminate abruptly.
2. **Error Handling and Recovery:** Allows the program to recover and continue execution.
3. **Separates Error Logic:** Cleanly separates error-handling code from the main logic.

### Simple try-catch block

```
public class TryCatchExample {  
    public static void main(String[] args) {  
        try {  
            int result = 10 / 0; // This will throw ArithmeticException  
        } catch (ArithmeticException e) {  
            System.out.println("Exception caught: " + e.getMessage());  
        }  
        System.out.println("Program continues...");  
    }  
}
```

### Output:

```
C:\TestJava>java TryCatchExample
Exception caught: / by zero
Program continues...
```

## Multiple catch Blocks in Java

In Java, a try block can be followed by multiple catch blocks to handle different types of exceptions separately. This allows developers to provide specific exception-handling logic for various types of errors that may arise.

### Key Points about Multiple catch Blocks

- **Handling Specific Exceptions:** Each catch block is designed to handle a particular type of exception.
- **Order Matters:**
  - The order of the catch blocks is important.
  - Specific exceptions must be caught before more general exceptions (e.g., `ArrayIndexOutOfBoundsException` before `Exception`).
  - If a general exception is placed first, it will handle all exceptions, and subsequent catch blocks will become unreachable, causing a compilation error.
- **One Exception Per Catch:** Only the first catch block that matches the thrown exception type will execute, even if multiple exceptions occur in the try block.

Example:

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]); // Throws ArrayIndexOutOfBoundsException
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic Exception caught: " + e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array Index Out Of Bounds Exception caught: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("General Exception caught: " + e.getMessage());
        }
    }
}
```

Output:

```
C:\TestJava>javac MultipleCatchExample.java

C:\TestJava>java MultipleCatchExample
Array Index Out Of Bounds Exception caught: Index 5 out of bounds for length 3
```

## Nested try Statements in Java

A **nested try statement** refers to placing one try block inside another try block. This can be useful when different blocks of code within the try block need different exception handling logic.

### Key Points about Nested try Statements

1. **Multiple Layers of Exception Handling:**  
You can have multiple levels of exception handling, where exceptions from different parts of the try block are handled at different levels.
2. **Inner catch Block:**  
Each inner try block can have its own catch block to handle exceptions specific to that block. If an exception occurs in the inner try block, it will be caught by the corresponding

catch block. If an exception occurs in the outer try block, it can be handled by the outer catch.

### 3. Control Flow:

The flow of control depends on where the exception is thrown. If an exception occurs in an inner try, it is handled by the inner catch. If not, the outer catch will handle exceptions from the outer try.

### 4. Finally Block:

The finally block, if used, will execute after the try-catch blocks, regardless of whether an exception occurred.

#### Example:

```
public class NestedTry {  
    public static void main(String[] args) {  
        try {  
            // Outer try block  
            int[] numbers = {1, 2, 3};  
            System.out.println(numbers[2]); // This will print 3  
  
            try {  
                // Inner try block  
                int result = 10 / 0; // This will throw ArithmeticException  
            } catch (ArithmeticException e) {  
                System.out.println("Inner catch: Division by zero occurred");  
            }  
  
            // Outer try continues after inner try  
            System.out.println("Outer try continues...");  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Outer catch: Array index out of bounds");  
        }  
  
        System.out.println("Program continues...");  
    }  
}
```

#### Output:

```
C:\TestJava>javac NestedTry.java  
  
C:\TestJava>java NestedTry  
3  
Inner catch: Division by zero occurred  
Outer try continues...  
Program continues...
```

### throw, throws, and finally in Java

In Java, **throw**, **throws**, and **finally** are important keywords used in exception handling. They help developers manage errors more effectively and ensure that resources are released properly. Let's explore each of them in detail:

#### throw Keyword

The throw keyword is used to explicitly throw an exception from a method or a block of code. When an exception is thrown, the normal flow of the program is disrupted, and the control is transferred to the nearest catch block that can handle the exception.

#### Key Points about throw:

- It is followed by an instance of Throwable (either an Exception or an Error).
- Can throw both checked and unchecked exceptions.
- Once the exception is thrown, the execution stops at that point, and control is transferred to the appropriate catch block.

***throw new ExceptionType("Error message");***

#### Example:

```
public class ExceptionEx {  
    public static void main(String[] args) {  
        try {  
            checkAge(15); // Will throw an exception  
        } 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.");  
        }  
        System.out.println("Age is valid");  
    }  
}
```

#### Output:

```
C:\TestJava>javac ExceptionEx.java  
C:\TestJava>java ExceptionEx  
Exception caught: Age must be 18 or older.
```

#### throws Keyword

The throws keyword is used in a method declaration to specify that a method may throw one or more exceptions. This is used for **checked exceptions** (exceptions that are subclasses of Exception but not RuntimeException). It tells the compiler and the developer that a method might throw certain exceptions, and the caller of the method needs to handle them.

#### Key Points about throws:

- It is placed after the method signature.
- It is used to declare exceptions that a method might throw.
- It can declare multiple exceptions, separated by commas.

***public void methodName() throws ExceptionType1, ExceptionType2 { // method body }***

#### Example:

```
import java.io.IOException;

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

    public static void readFile() throws IOException {
        throw new IOException("File not found");
    }
}
```

**Output:**

```
IOException caught: File not found
```

**finally Keyword**

The finally block is used to execute code that must run after the try-catch blocks, regardless of whether an exception was thrown or not. It is typically used for clean-up activities such as closing files, releasing resources, or resetting states.

**Key Points about finally:**

- The finally block always executes, whether or not an exception occurs.
- It is optional, but if present, it will execute after the try and catch blocks.
- It can be used for code that must always execute, such as closing resources or network connections.
- If a return statement is encountered in a try or catch block, the finally block will still execute before the method returns the value.

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Exception handling code
} finally {
    // Code to clean up, close resources, etc.
}
```

**Example:**

```

public class FinallyExample {
    public static void main(String[] args) {
        try {
            System.out.println("Inside try block");
            int result = 10 / 0; // This will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        } finally {
            System.out.println("This will always be executed, even if exception occurs");
        }
        System.out.println("Program continues...");
    }
}

```

**Output:**

```

C:\TestJava>javac FinallyExample.java

C:\TestJava>java FinallyExample
Inside try block
Exception caught: / by zero
This will always be executed, even if exception occurs
Program continues...

```

**Key Differences**

Aspect	throw	throws
<b>Definition</b>	Used to explicitly throw an exception.	Used to declare that a method might throw an exception.
<b>Usage</b>	Inside the body of a method or code block.	In the method signature (declaration).
<b>Functionality</b>	Actually throws an exception.	Declares the exceptions that a method may throw.
<b>Syntax</b>	throw new ExceptionType("Message");	public void methodName() throws ExceptionType;
<b>When to Use</b>	When you want to trigger an exception manually.	When a method might throw a checked exception and needs to be handled by the caller.
<b>Handled by</b>	Handled in a catch block or propagated further.	Handled by the caller (using try-catch or further throws).
<b>Example</b>	throw new IllegalArgumentException("Invalid input");	public void readFile() throws IOException { ... }

### Java's Built-in Exceptions:

In Java, exceptions are used to handle situations that disrupt the normal flow of execution. Java provides a variety of **built-in exceptions** (also known as **predefined exceptions**), which are part of the Java class library and are subclasses of the Throwable class. These exceptions are categorized into two main types:

1. **Checked Exceptions** – Exceptions that must be explicitly handled by the programmer (either through a try-catch block or by declaring the exception using throws).
  2. **Unchecked Exceptions** – Exceptions that are not required to be handled by the programmer. These are usually runtime exceptions, which typically occur due to programming errors.
1. **ArithmeticException**
    - **Type:** Unchecked Exception (RuntimeException)
    - **Description:** Thrown when an exceptional arithmetic condition occurs, such as division by zero.
  2. **NullPointerException**
    - **Type:** Unchecked Exception (RuntimeException)
    - **Description:** Thrown when an application attempts to use null where an object is required, such as calling a method on a null object reference.
  3. **ArrayIndexOutOfBoundsException**
    - **Type:** Unchecked Exception (RuntimeException)
    - **Description:** Thrown when trying to access an array index that is out of bounds (i.e., index is less than 0 or greater than or equal to the length of the array).
  4. **ClassNotFoundException**
    - **Type:** Checked Exception
    - **Description:** Thrown when an application tries to load a class using Class.forName() or other similar methods, but the class is not found.
  5. **IOException**
    - **Type:** Checked Exception
    - **Description:** A general class for exceptions produced by failed or interrupted I/O operations, such as reading from a file or writing to a network socket.
  6. **FileNotFoundException**
    - **Type:** Checked Exception (Subclass of IOException)
    - **Description:** Thrown when an attempt to open the file denoted by a specified pathname has failed.
  7. **NumberFormatException**
    - **Type:** Unchecked Exception (RuntimeException)
    - **Description:** Thrown when an attempt to convert a string to a number fails, such as when a string is not a valid representation of a number.
  8. **StackOverflowError**
    - **Type:** Error (Not Exception)
    - **Description:** A subclass of Error, this is thrown when the stack overflows, typically due to deep recursion or an infinite recursive method call.

## Creating Your Own Exception Subclasses in Java

In Java, you can define your own custom exceptions by subclassing the existing exception classes, typically `Exception` or `RuntimeException`. Custom exceptions allow you to provide more meaningful error messages or handle specific error conditions that are relevant to your application.

### Steps to Create a Custom Exception:

1. **Subclass the `Exception` or `RuntimeException` class.**
  - If you want to create a **checked exception**, extend `Exception`.
  - If you want to create an **unchecked exception**, extend `RuntimeException`.
2. **Provide constructors for your custom exception.**
  - You typically define at least two constructors: one for the default message and one for a more detailed message or for passing a `Throwable` cause.
3. Optionally, override methods such as `getMessage()` or `toString()` for more customized behavior.

```
// Custom Exception Subclass
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class CustomExceptionDemo {
    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }

    public static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or above!");
        } else {
            System.out.println("Eligible to vote!");
        }
    }
}
```

## Chained Exceptions in Java

Chained exceptions, also known as **exception chaining**, refer to the practice of linking one exception to another. This technique is useful when an exception occurs, and you want to provide additional context by attaching the original exception (the cause) to a new one. It allows you to trace the root cause of an error, providing more useful information for debugging and logging.

Java provides built-in support for exception chaining, enabling developers to preserve the cause of an exception even when it is re-thrown in a different form.

### Why Use Chained Exceptions?

Chained exceptions are especially useful in scenarios where:

1. **Wrapping Low-Level Exceptions:** When a low-level exception (such as `IOException`) is thrown, you might want to throw a higher-level, domain-specific exception (such as `DataProcessingException`) while preserving the original cause of the error.
2. **Error Context Preservation:** When catching one exception and throwing another, you can maintain the original exception so that the root cause is not lost.
3. **Improved Debugging:** Chained exceptions provide a full stack trace of the exception chain, helping you understand how the error propagated.

### Creating Chained Exceptions

Java provides two primary mechanisms for chaining exceptions:

1. **Using Throwable constructors:** You can pass the original exception (cause) to the new exception when creating it.

*Syntax: `new Exception("New exception message", originalException);`*

2. **Using `initCause()` method:** If you've already created an exception, you can later associate a cause with it using this method. *Syntax: `exception.initCause(cause);`*

#### Example:

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

    public static void methodA() throws CustomException {
        try {
            methodB();
        } catch (ArithmeticException e) {
            throw new CustomException("Custom exception occurred while calling methodB", e);
        }
    }

    public static void methodB() {
        int result = 10 / 0; // This will throw ArithmeticException
    }
}

// Custom Exception Class
class CustomException extends Exception {
    public CustomException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

#### Using `initCause` method Example:

```
public class ChainedExceptionDemo {  
    public static void main(String[] args) {  
        try {  
            methodA();  
        } catch (CustomException e) {  
            System.out.println("Caught custom exception: " + e.getMessage());  
            System.out.println("Cause: " + e.getCause());  
        }  
    }  
  
    public static void methodA() throws CustomException {  
        try {  
            methodB();  
        } catch (ArithmeticException e) {  
            CustomException customException =  
                new CustomException("Custom exception occurred while calling methodB");  
            customException.initCause(e); // Assigning the cause  
            throw customException;  
        }  
    }  
  
    public static void methodB() {  
        int result = 10 / 0; // This will throw ArithmeticException  
    }  
}  
  
// Custom Exception Class  
class CustomException extends Exception {  
    public CustomException(String message) {  
        super(message);  
    }  
}
```

**Output:**

```
C:\TestJava>javac ChainedExceptionDemo.java  
  
C:\TestJava>java ChainedExceptionDemo  
Caught custom exception: Custom exception occurred while calling methodB  
Cause: java.lang.ArithmeticException: / by zero
```