**Java Exception Handling:**

An **exception** is a problem that occurs during the normal execution of a program. When an exception arises, the program's normal flow is disrupted. Exception handling is the process of managing these events to prevent abnormal program termination.

**Exception Hierarchy & Categories**

All exceptions and errors in Java are subclasses of the java.lang.Throwable class.

A diagram of a program

AI-generated content may be incorrect.

The two main subclasses are Exception and Error.

1. **Checked Exceptions:**
   * **Definition:** These exceptions are **checked by the compiler at compile-time**. The compiler forces you to handle them using a try-catch block or declare them with the throws keyword.
   * **Reason:** They represent conditions that a well-written application should anticipate and recover from.
   * **Example:** IOException, FileNotFoundException.
2. **Unchecked Exceptions:**
   * **Definition:** These exceptions are **checked at runtime**. The compiler does not enforce handling them.
   * **Reason:** They are typically caused by programming bugs or logic errors and are not easily recoverable.
   * **Example:** NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException.
3. **Errors:**
   * **Definition:** These are serious problems that are **beyond the control of the application programmer**.
   * **Reason:** They usually indicate a failure in the Java Virtual Machine (JVM) or a hardware resource.
   * **Example:** StackOverflowError, OutOfMemoryError.

**Handling Exceptions: try, catch, finally**

A try block contains the "protected code" that might throw an exception. The catch block handles a specific exception type if it's thrown. The optional finally block always executes, regardless of whether an exception occurred.

* **try:** Contains the code that is prone to exceptions.
* **catch:** Catches and handles an exception. You can have multiple catch blocks for different exception types. Since Java 7, you can use a single catch block for multiple exceptions with the | symbol (e.g., catch(IOException | FileNotFoundException e)).
* **finally:** Contains cleanup code that must always run, such as closing files or database connections.

**The throw and throws Keywords**

* **throw:** Used to **explicitly throw** an instance of an exception. You create a new exception object and throw it from within a method.
* **throws:** Used in a method's signature to **declare** that it might throw one or more checked exceptions. This forces the calling method to handle the exception.

**try-with-resources**

Introduced in Java 7, this statement automatically manages resources (like streams and connections) that implement the AutoCloseable interface. The resources are declared within the try block's parentheses and are automatically closed at the end of the block, eliminating the need for a finally block.

**Example:**

Java

try (FileReader fr = new FileReader("file.txt")) {

// Use the resource

} catch (IOException e) {

// Handle the exception

} // The 'fr' resource is automatically closed here

**User-Defined Exceptions**

You can create your own custom exceptions to represent specific problems in your application logic.

* To create a **checked exception**, extend the java.lang.Exception class.
* To create an **unchecked exception**, extend the java.lang.RuntimeException class.

**TRY AND CATCH :**

Java exception handling is a mechanism for dealing with unexpected problems during program execution, preventing a crash. The core of this mechanism is the try-catch block.

**The try-catch Block**

* The **try block** contains the "protected code"—the part of your program that might throw an exception.
* The **catch block** follows the try block. It "catches" an exception if one occurs. A catch block must declare the type of exception it can handle. If the type matches the thrown exception, the code inside the catch block executes.

Java

try {

// Protected code that might throw an exception

} catch (ExceptionType e) {

// Code to handle the exception

}

Example:

In this example, trying to access an array element beyond its bounds causes an ArrayIndexOutOfBoundsException. The catch block handles it, allowing the program to continue running normally.

Java

public class ExcepTest {

public static void main(String args[]) {

try {

int a[] = new int[2];

System.out.println("Access element three :" + a[3]);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Exception thrown : " + e);

}

System.out.println("Out of the block");

}

}

**Output:**

Exception thrown : java.lang.ArrayIndexOutOfBoundsException: 3

Out of the block

**Handling Multiple Exceptions**

A single try block can be followed by multiple catch blocks to handle different types of exceptions. The catch blocks are checked in order from top to bottom.

Java

try {

// Protected code

} catch (ExceptionType1 e1) {

// Handle ExceptionType1

} catch (ExceptionType2 e2) {

// Handle ExceptionType2

}

Example:

If the protected code throws an ArithmeticException (e.g., division by zero), the first catch block is skipped, and the exception is handled by the more general Exception catch block.

Java

public class ExcepTest {

public static void main(String args[]) {

try {

int a[] = new int[2];

int c = 1 / 0; // Throws ArithmeticException

System.out.println("Access element three :" + a[3]);

} catch (ArrayIndexOutOfBoundsException e) {

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

} catch (Exception e) {

System.out.println("Exception thrown :" + e);

}

System.out.println("Out of the block");

}

}

**Output:**

Exception thrown :java.lang.ArithmeticException: / by zero

Out of the block

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

Starting with Java 7, you can handle multiple exception types in a single catch block using the vertical bar |. This simplifies the code when the handling logic is the same for multiple exception types.

Java

try {

// Protected code

} catch (ExceptionType1 | ExceptionType2 e) {

// Handle both exception types

}

**Example:**

Java

public class ExcepTest {

public static void main(String args[]) {

try {

int c = 1 / 0; // Throws ArithmeticException

int a[] = new int[2];

System.out.println("Access element three :" + a[3]);

} catch (ArrayIndexOutOfBoundsException | ArithmeticException e) {

System.out.println("Exception thrown :" + e);

}

System.out.println("Out of the block");

}

}

**Output:**

Exception thrown :java.lang.ArithmeticException: / by zero

Out of the block

**Java - Try with Resources**

The try-with-resources statement is a feature introduced in Java 7 to simplify resource management. It automatically closes resources like streams, file handles, or database connections, preventing resource leaks without needing a finally block. This is often referred to as "Automatic Resource Management."

**Syntax and Usage**

To use try-with-resources, you declare one or more resources inside the parentheses of the try statement. The resources must implement the **java.lang.AutoCloseable** interface, which guarantees they have a close() method. Once the try block is finished (either normally or due to an exception), the close() method is automatically called.

**Basic Syntax**

Java

try (Resource res = new Resource()) {

// Use the resource

} catch (Exception e) {

// Handle any exceptions

}

// The 'res' is automatically closed here

**Multiple Resources**

You can declare multiple resources in a single try block, separated by a semicolon. They are closed in the **reverse order** of their declaration.

Java

try (

FileReader fr = new FileReader("file1.txt");

BufferedReader br = new BufferedReader(fr)

) {

// Use both resources

} catch (IOException e) {

// Handle exception

}

// 'br' is closed first, then 'fr'

**try-with-resources vs. try-catch-finally**

| Aspect | try-with-resources | try-catch-finally |
| --- | --- | --- |
| **Syntax** | Concise; resources declared in parentheses. | Verbose; requires a finally block for cleanup. |
| **Resource Closing** | Automatic. The JVM handles it. | Manual. The developer must remember to call close() in the finally block. |
| **Safety** | Safer; less prone to resource leaks. | Prone to leaks if close() is forgotten or not handled correctly within finally. |

**Improvements in Java 9**

Prior to Java 9, a resource had to be declared within the try statement's parentheses. Java 9 improved this by allowing you to use a resource variable that was declared and initialized outside the try block, as long as it's effectively final.

**Before Java 9:**

Java

BufferedReader br = new BufferedReader(new StringReader("data"));

try (BufferedReader br1 = br) {

br1.readLine();

}

**Java 9 and later:**

Java

BufferedReader br = new BufferedReader(new StringReader("data"));

try (br) { // Use the existing 'br' variable

br.readLine();

}

**Multiple Catch Blocks in Java**

To handle different types of exceptions that a block of code might throw, you can use multiple catch blocks in a single try statement.

**Multiple Catch Blocks**

A single try block can have one or more catch blocks. When an exception occurs, the JVM checks each catch block in the order they appear. The first catch block that matches the exception type (or a superclass of the exception) is executed. After the catch block finishes, the program continues with the code that follows the entire try-catch block.

**Key Rule: Order Matters ⚠️**

You must place catch blocks for specific exception types (subclasses) before those for more general types (superclasses). If you place a general exception handler, like catch (Exception e), before a specific one, the more specific one will never be reached, and the compiler will flag a compile-time error.

**Example:**

Java

public class ExcepTest {

public static void main(String[] args) {

try {

// This line throws an ArithmeticException

int c = 1 / 0;

int a[] = new int[2];

System.out.println(a[3]); // This line is never reached

}

// Specific catch block comes first

catch (ArithmeticException e) {

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

}

// More general catch block comes second

catch (Exception e) {

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

}

System.out.println("Out of the block");

}

}

**Output:**

ArithmeticException thrown: java.lang.ArithmeticException: / by zero

Out of the block

**Handling Multiple Exceptions in a Single Block (Java 7+)**

Beginning with Java 7, you can handle multiple exception types with the same logic in a single catch block. This is done by separating the exception types with a vertical bar (|). This simplifies the code and reduces redundancy.

**Example:**

Java

public class ExcepTest {

public static void main(String[] args) {

try {

int a[] = new int[2];

int b = 0;

int c = 1 / b; // Throws ArithmeticException

System.out.println(a[3]);

}

// Single catch block for two types of exceptions

catch (ArrayIndexOutOfBoundsException | ArithmeticException e) {

System.out.println("Exception thrown: " + e);

}

System.out.println("Out of the block");

}

}

**Output:**

Exception thrown: java.lang.ArithmeticException: / by zero

Out of the block

**Java - Nested Try Block :**

A nested try block is a try block placed inside another try block. This allows for a more granular level of exception handling within a program. When an exception is thrown in an inner try block, Java first attempts to catch it with the inner catch blocks. If no matching catch block is found, the exception is propagated to the outer try block, and its catch blocks are then checked.

**How It Works**

1. **Inner try Block:** The inner try block contains code that might throw a specific, localized exception.
2. **Inner catch Block:** The inner catch block is typically used to handle a specific exception for the code in the inner try block.
3. **Exception Propagation:** If the inner catch block cannot handle the exception, or if there is no inner catch block, the exception moves up to the outer try block.
4. **Outer catch Block:** The outer catch block can handle exceptions from the outer try block as well as any exceptions that were not handled by the inner catch block. This makes it a good place to handle more general or unexpected exceptions.

This structure allows you to manage different types of exceptions at different levels of a program's logic.

**Example**

In this example, the inner try block attempts a division by zero, which throws an ArithmeticException. The inner catch block handles it and prints a message. Then, the program continues to the next line of the outer try block, which tries to access an invalid array index, throwing an ArrayIndexOutOfBoundsException. The outer catch block handles this exception.

Java

public class NestedTryExample {

public static void main(String args[]) {

try { // Outer try block

int a[] = new int[2];

try { // Inner try block

int b = 0;

int c = 1 / b; // Throws ArithmeticException

} catch(Exception e) { // Inner catch block handles ArithmeticException

System.out.println("Inner catch: Exception thrown: " + e);

}

System.out.println("Accessing array element:" + a[3]); // Throws ArrayIndexOutOfBoundsException

}

catch (ArrayIndexOutOfBoundsException e) { // Outer catch block handles it

System.out.println("Outer catch: Exception thrown: " + e);

}

System.out.println("Program finished.");

}

}

**Output:**

Inner catch: Exception thrown: java.lang.ArithmeticException: / by zero

Outer catch: Exception thrown: java.lang.ArrayIndexOutOfBoundsException: 3

Program finished.

**Java - Finally Block :**

The finally block in Java is used to execute a block of code regardless of whether an exception was thrown in the preceding try block or handled by a catch block. It's primarily used for **cleanup operations**.

**Key Principles of the finally Block**

* **Guaranteed Execution:** The code inside the finally block will always execute. The only exceptions are when the JVM exits (e.g., via System.exit()) or encounters a fatal error.
* **Purpose:** It's essential for **resource management**, such as closing files, network connections, or database connections, to prevent resource leaks.
* **Placement:** It must follow a try block, and it can optionally follow catch blocks. A try block can exist with a finally block even without a catch block.
* **Return Statements:** Even if a try or catch block contains a return statement, the finally block will execute before the method returns to the caller.

**finally Block Examples**

**Example 1: Exception Handled**

In this case, an ArrayIndexOutOfBoundsException is thrown and handled by the catch block. The program then proceeds to the finally block.

Java

public class FinallyExample {

public static void main(String args[]) {

try {

int a[] = new int[2];

System.out.println("Accessing element three: " + a[3]);

} catch (ArrayIndexOutOfBoundsException e) {

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

} finally {

System.out.println("The finally block always executes.");

}

}

}

**Output:**

Exception caught: java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 2

The finally block always executes.

**Example 2: No Exception**

When no exception occurs in the try block, the catch block is skipped, but the finally block still executes.

Java

public class FinallyExample2 {

public static void main(String args[]) {

try {

System.out.println("No exception thrown in this block.");

} finally {

System.out.println("This is the cleanup code in the finally block.");

}

}

}

**Output:**

No exception thrown in this block.

This is the cleanup code in the finally block.

**Java - Throws and Throw | Throw an Exception :**

throws and throw are two different keywords in Java used for exception handling. They're often confused because of their similar names, but they serve distinct purposes.

**throw (Singular) 💥**

The throw keyword is used to **explicitly throw an exception instance**. It creates a new exception object and immediately halts the current method's execution, transferring control to the nearest catch block that can handle the exception.

* **Syntax:** throw new ExceptionType("message");
* **Purpose:** To signal that an exceptional condition has occurred at a specific point in your code.
* **Example:** Throwing an IllegalArgumentException when an invalid argument is provided.

Java

public class ThrowExample {

public static void validateAge(int age) {

if (age < 0) {

// Throwing a new exception object

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

}

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

}

public static void main(String[] args) {

try {

validateAge(-5);

} catch (IllegalArgumentException e) {

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

}

}

}

**Output:**

Caught an exception: Age cannot be negative.

**throws (Plural) 📢**

The throws keyword is used in a method's signature to **declare** that the method might throw one or more **checked exceptions**. It's a signal to the compiler and to other developers that the method is not handling a particular exception, and therefore, the calling method must either handle it or declare it as well.

* **Syntax:** public void someMethod() throws ExceptionType1, ExceptionType2 { ... }
* **Purpose:** To postpone the handling of a checked exception to a higher level in the call stack. This satisfies the "Handle or Declare" rule for checked exceptions.
* **Example:** Declaring that a method might throw an IOException.

Java

import java.io.IOException;

public class ThrowsExample {

// This method declares that it can throw an IOException

public void readData() throws IOException {

// Code that might cause an IOException

throw new IOException("File not found.");

}

public static void main(String[] args) {

ThrowsExample obj = new ThrowsExample();

try {

// The calling method must handle the declared exception

obj.readData();

} catch (IOException e) {

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

}

}

}

**Output:**

Caught declared exception: File not found.

**Summary of Differences**

| Feature | throw | throws |
| --- | --- | --- |
| **Purpose** | Throws an exception instance. | Declares a method's potential exceptions. |
| **Usage** | Used within the method body. | Used in the method signature. |
| **Followed By** | A single Exception object. | One or more Exception class names. |

**Java - Exception Propagation :**

**Exception propagation** is the mechanism by which an exception is passed up the call stack from the method where it occurred to its calling method, and so on, until it is either caught and handled or reaches the top of the stack, causing the program to terminate.

**How Exception Propagation Works ↩️**

When a method throws an exception, the Java Virtual Machine (JVM) creates an exception object. Instead of immediately crashing the program, the JVM searches the current method for a suitable catch block to handle it.

* If a matching catch block is found, the exception is handled, and the program's normal flow resumes.
* If no matching catch block is found in the current method, the exception is "propagated" or "thrown up" to the method that called it.
* The process repeats: the JVM searches for a handler in the calling method, then its caller, and so on.
* If the exception reaches the main() method and is still not handled, the program will terminate abnormally, and the JVM will print a stack trace.

**Propagation Scenarios**

1. **Handled by Caller:** A method throws an exception, but the calling method contains a try-catch block to handle it. This is the most common and desirable scenario.

Java

// Method that throws an exception

private static int divide(int a, int b) {

return a / b; // Throws ArithmeticException if b is 0

}

public static void main(String args[]) {

try {

System.out.println("Result: " + divide(10, 0));

} catch (ArithmeticException e) {

// Catches the exception thrown by divide()

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

}

}

**Output:**

Exception caught: / by zero

1. **Uncaught Propagation:** A method throws an exception, and neither it nor any of its callers handle it. The exception propagates all the way to the top.

Java

private static int divide(int a, int b) {

return a / b; // Throws ArithmeticException

}

public static void main(String args[]) {

// No try-catch block to handle the exception

System.out.println("Result: " + divide(10, 0));

}

**Output:**

Exception in thread "main" java.lang.ArithmeticException: / by zero

... stack trace...

**Java - Built-in Exceptions :**

Java comes with a number of built-in exception classes that are automatically available to all programs. These exceptions are categorized into **checked** and **unchecked** types.

|  |  |
| --- | --- |
| **Sr.No.** | **Exception & Description** |
| 1 | **ArithmeticException**  Arithmetic error, such as divide-by-zero. |
| 2 | **ArrayIndexOutOfBoundsException**  Array index is out-of-bounds. |
| 3 | **ArrayStoreException**  Assignment to an array element of an incompatible type. |
| 4 | **ClassCastException**  Invalid cast. |
| 5 | **IllegalArgumentException**  Illegal argument used to invoke a method. |
| 6 | **IllegalMonitorStateException**  Illegal monitor operation, such as waiting on an unlocked thread. |
| 7 | **IllegalStateException**  Environment or application is in incorrect state. |
| 8 | **IllegalThreadStateException**  Requested operation not compatible with the current thread state. |
| 9 | **IndexOutOfBoundsException**  Some type of index is out-of-bounds. |
| 10 | **NegativeArraySizeException**  Array created with a negative size. |
| 11 | **NullPointerException**  Invalid use of a null reference. |
| 12 | **NumberFormatException**  Invalid conversion of a string to a numeric format. |
| 13 | **SecurityException**  Attempt to violate security. |
| 14 | **StringIndexOutOfBounds**  Attempt to index outside the bounds of a string. |
| 15 | **UnsupportedOperationException**  An unsupported operation was encountered. |
| 16 | **ClassNotFoundException**  Class not found. |
| 17 | **CloneNotSupportedException**  Attempt to clone an object that does not implement the Cloneable interface. |
| 18 | **IllegalAccessException**  Access to a class is denied. |
| 19 | **InstantiationException**  Attempt to create an object of an abstract class or interface. |
| 20 | **InterruptedException**  One thread has been interrupted by another thread. |
| 21 | **NoSuchFieldException**  A requested field does not exist. |
| 22 | **NoSuchMethodException**  A requested method does not exist. |

**Unchecked Exceptions (Runtime) ⚠️**

Unchecked exceptions are subclasses of java.lang.RuntimeException. The compiler **does not force you to handle them**. They usually indicate a programming mistake or a logical error that should be fixed in the code.

| Exception | Description |
| --- | --- |
| **ArithmeticException** | An arithmetic operation error, like division by zero. |
| **ArrayIndexOutOfBoundsException** | An array is accessed with an illegal index. |
| **NullPointerException** | An attempt is made to use a null reference as an object. |
| **ClassCastException** | An object is cast to an incompatible type. |
| **IllegalArgumentException** | A method is called with an invalid or inappropriate argument. |
| **NumberFormatException** | A string cannot be converted into a valid numeric format. |

**Example:** Using try-catch to handle an ArithmeticException.

Java

public class UncheckedExample {

public static void main(String args[]) {

try {

int result = 10 / 0; // Throws an ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Error: Cannot divide by zero.");

}

}

}

**Checked Exceptions (Compile-time) 🔒**

Checked exceptions are subclasses of java.lang.Exception. They are checked at compile time, and the compiler **requires that you either handle them with a try-catch block or declare them with throws**. They represent conditions that a well-written application should anticipate and gracefully recover from.

| Exception | Description |
| --- | --- |
| **IOException** | An I/O operation (e.g., file access, networking) has failed. |
| **FileNotFoundException** | A file cannot be accessed or found. |
| **ClassNotFoundException** | The JVM tried to load a class that doesn't exist. |
| **InterruptedException** | A thread is interrupted while waiting. |
| **SQLException** | An error occurred with a database. |

**Example:** A method that might throw a ClassNotFoundException must declare it.

Java

public class CheckedExample {

// Declares that this method can throw ClassNotFoundException

public static void loadClass(String className) throws ClassNotFoundException {

Class.forName(className);

}

public static void main(String args[]) {

try {

// The caller must handle the declared exception

loadClass("com.example.NonExistentClass");

} catch (ClassNotFoundException e) {

System.out.println("Error: The class was not found.");

}

}

}

**Java - Custom Exception :**

**Custom exceptions** in Java are user-defined exception classes created to address specific application-level errors. They help categorize exceptions and make code more readable and maintainable by providing more meaningful error messages than generic built-in exceptions.

**How to Create a Custom Exception**

To create a custom exception, you must create a new class that inherits from either java.lang.Exception or java.lang.RuntimeException. The choice determines whether your custom exception is checked or unchecked. All custom exceptions must ultimately inherit from java.lang.Throwable.

* **For a Checked Exception:** Extend the Exception class. The compiler will then enforce the "Handle or Declare" rule, requiring the calling method to either catch the exception or declare it with throws.
* **For an Unchecked Exception:** Extend the RuntimeException class. The compiler will not force you to handle it, similar to built-in exceptions like NullPointerException.

**Example: Checked Custom Exception**

Let's create a custom **checked exception** for a banking application that is thrown when a withdrawal amount exceeds the available balance.

1. **Define the Custom Exception:**

Java

class InsufficientFundsException extends Exception {

private double amount;

public InsufficientFundsException(double amount) {

this.amount = amount;

}

public double getAmount() {

return amount;

}

}

1. **Use it in a Class:** The withdraw() method declares that it might throw this checked exception. The caller must handle it.

Java

class CheckingAccount {

private double balance;

public void withdraw(double amount) throws InsufficientFundsException {

if (amount <= balance) {

balance -= amount;

} else {

double needs = amount - balance;

throw new InsufficientFundsException(needs);

}

}

}

1. **Handle the Exception in main():**

Java

public class BankDemo {

public static void main(String [] args) {

CheckingAccount c = new CheckingAccount(101);

c.deposit(500.00);

try {

c.withdraw(600.00);

} catch (InsufficientFundsException e) {

System.out.println("Sorry, you are short $" + e.getAmount());

}

}

}

**Output:**

Sorry, you are short $100.0