**✅ What is an Exception?**

An **Exception** is an unwanted event that occurs **during runtime** and disrupts the normal flow of the program.

If not handled, it leads to **abnormal (abrupt) termination** of the program.

👉 Handling an exception means:

The program should not terminate abruptly.  
Instead, it should handle the problem and continue execution normally.

**✅ Compile-Time Error vs Runtime Error**

**🔹 Compile-Time Error**

* Occurs during compilation.
* Caused by syntax mistakes.
* Example:
  + Missing ;
  + Missing { }
  + Wrong method declaration

The program will not compile.

**🔹 Runtime Error**

* Occurs during execution.
* Happens after successful compilation.
* Example:
  + Divide by zero → ArithmeticException
  + Null reference → NullPointerException
  + Array index out of range → ArrayIndexOutOfBoundsException

The program compiles successfully but crashes while running.

**✅ Error vs Exception (Very Important Concept)**

In Java, both are subclasses of Throwable.

**🔹 Errors**

* Serious problems.
* Not usually handled by programmers.
* Caused by JVM or system issues.
* Example:
  + OutOfMemoryError
  + StackOverflowError

These are generally not recoverable.

**🔹 Exceptions**

* Problems that can be handled.
* Caused by program logic or external conditions.
* Can be recovered using try-catch.

**✅ Types of Exceptions**

**1️⃣ Checked Exceptions**

* Checked at compile time.
* Must be handled using try-catch or throws.
* Example:
  + IOException
  + SQLException

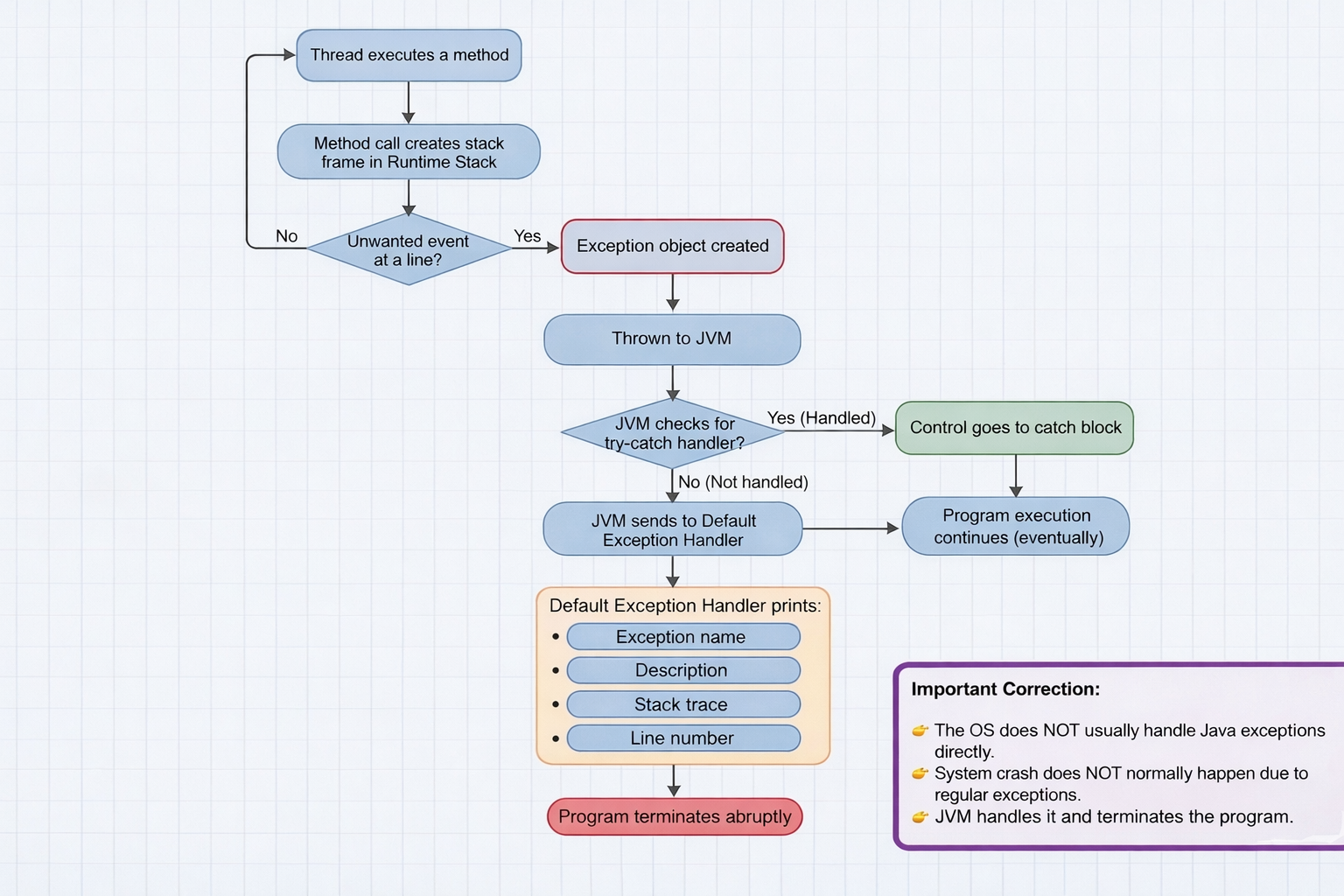
If not handled → compiler error.

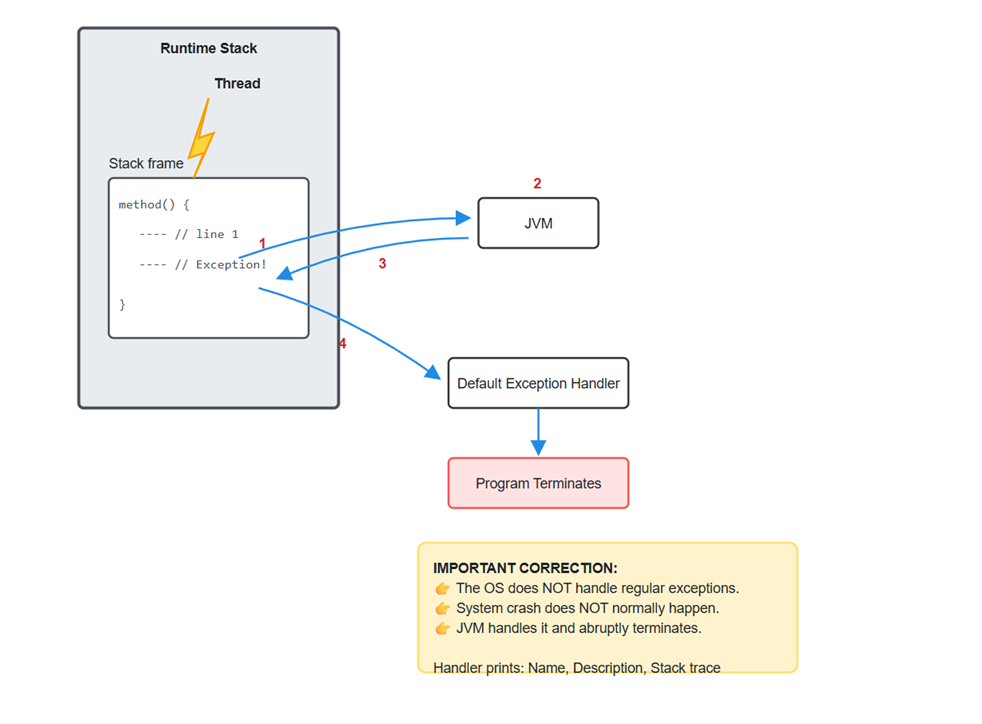
**2️⃣ Unchecked Exceptions (Runtime Exceptions)**

* Not checked at compile time.
* Occur at runtime.
* Example:
  + ArithmeticException
  + NullPointerException
  + IllegalArgumentException
  + ArrayIndexOutOfBoundsException

Handling is optional, but recommended.

**✅ How Exception Occurs (Step-by-Step Flow)**





When a thread executes a method:

1. Each method call creates a **stack frame** inside the runtime stack.

If an unwanted event happens at a line, an **Exception object** is created.

1. That object is thrown to the JVM.
2. JVM checks whether the exception is handled using try-catch.

If handled → control goes to catch block.

1. If not handled → JVM sends it to **Default Exception Handler**.

Default Exception Handler prints:

* + Exception name
  + Description
  + Stack trace
  + Line number

Program terminates abruptly.

**✅ Example**

Exception in thread "main" java.lang.ArithmeticException: / by zero  
 at Exception\_Eg1.main(Exception\_Eg1.java:10)

Meaning:

* Exception type: ArithmeticException
* Reason: divide by zero
* Occurred in main method
* Line number: 10
* Not handled → default exception handler terminated program.

Eg: Exception\_Eg1

**✅ Handling the Exception (try-catch)**

**Purpose:**

Prevent abrupt termination.

**Basic Syntax:**

try {  
 // risky code  
}  
catch (ExceptionType e) {  
 // handling code  
}

**✅ Important Rules of try-catch**

1. Risky code must be inside try.
2. After try, at least one catch or finally must be present.
3. If exception occurs:
   * Remaining code inside try is skipped.
   * Control directly goes to matching catch block.
4. Catch block executes only if exception occurs.
5. After catch block, program continues normally.
6. If exception does NOT occur:
   * Catch block will not execute.
   * Program runs normally.

**✅ What to Write in Catch Block?**

Inside catch block:

* Show user-friendly message.
* Log error.
* Take corrective action.
* Provide alternate logic.

Example:

catch (ArithmeticException e) {  
 System.out.println("Cannot divide by zero.");  
}

Eg: Exceptional\_Handling\_Try\_Catch

**✅ Using Parent Class "Exception"**

catch (Exception e)

* Exception is parent of most exception classes.
* It can catch almost all exceptions.
* But it is NOT recommended to use only generic catch.

Why?

Because:

* User will not know exact problem.
* Debugging becomes difficult.
* Not industry best practice.

**✅ Multiple Catch Blocks**

One try block can have multiple catch blocks.

Example:

try {  
 // risky code  
}  
catch (ArithmeticException e) {  
 System.out.println("Divide by zero error.");  
}  
catch (NegativeArraySizeException e) {  
 System.out.println("Array size cannot be negative.");  
}  
catch (ArrayIndexOutOfBoundsException e) {  
 System.out.println("Index out of range.");  
}

Rules:

* Only one catch block executes.
* Matching catch block executes.
* Remaining catch blocks are skipped.

Eg: Multi\_Catch\_Demo

**✅ Generic Catch Block (Best Practice)**

Sometimes unexpected exception may occur.

So we add:

catch (Exception e) {  
 System.out.println("Unexpected error occurred.");  
}

Important Rule:

👉 Generic catch block must be last.  
If placed before specific catch blocks → compiler error.

Because parent cannot come before child.

Eg: MultiCatchWithGeneric

**✅ If Programmer Handles Exception, Will Default Handler Run?**

No.

If exception is properly handled,  
Default exception handler will NOT execute.

**✅ What Happens If Exception Is Not Handled?**

* JVM default exception handler prints stack trace.
* Program terminates.
* Remaining code will not execute.

**✅ Finally Block**

finally block executes whether exception occurs or not.

Used for:

* Closing resources
* Releasing connections
* Cleaning memory

Example:

try {  
 // risky code  
}  
catch (Exception e) {  
 System.out.println("Handled.");  
}  
finally {  
 System.out.println("Cleanup code.");  
}

Finally always runs (except JVM crash or System.exit()).

Eg: FinallyDemo

**✅ Industry Recommended Structure**

1. Identify risky code.
2. Keep only risky statements inside try.
3. Use specific catch blocks.
4. Add generic catch at last.
5. Use meaningful messages.
6. Log exceptions in real projects.
7. Do not leave empty catch blocks.

Example:

try {  
 int result = a / b;  
}  
catch (ArithmeticException e) {  
 System.out.println("Invalid division operation.");  
}  
catch (Exception e) {  
 System.out.println("Unexpected error occurred.");  
}  
System.out.println("Program continues...");

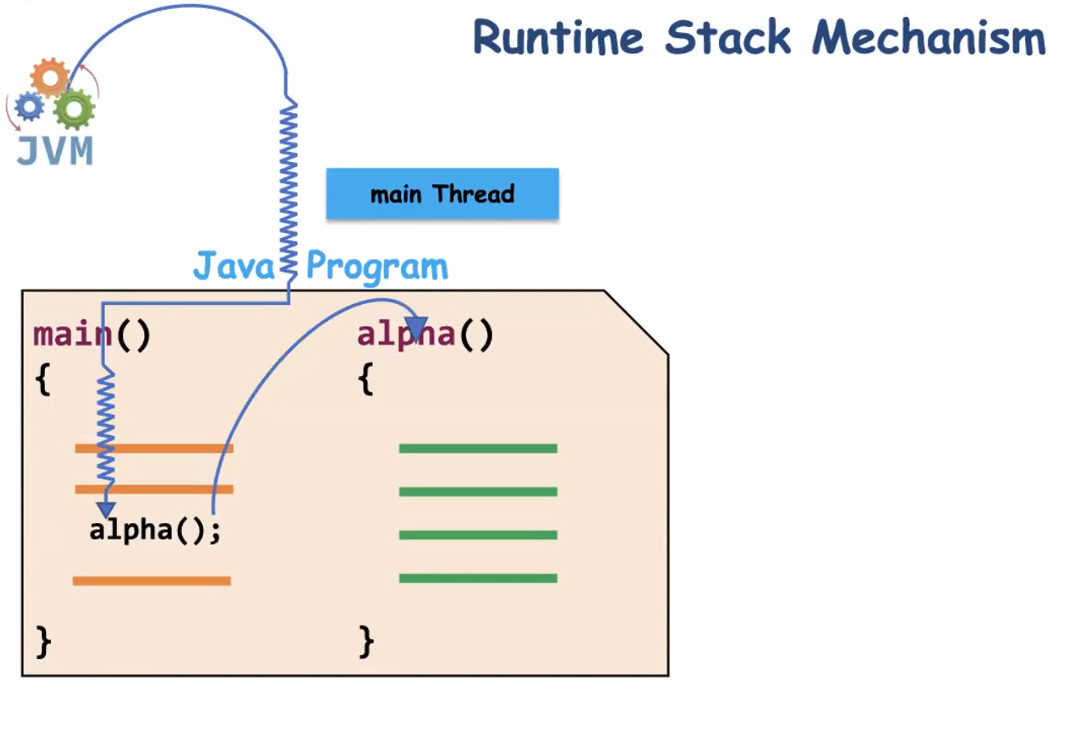
Eg: IndustryExceptionDemo

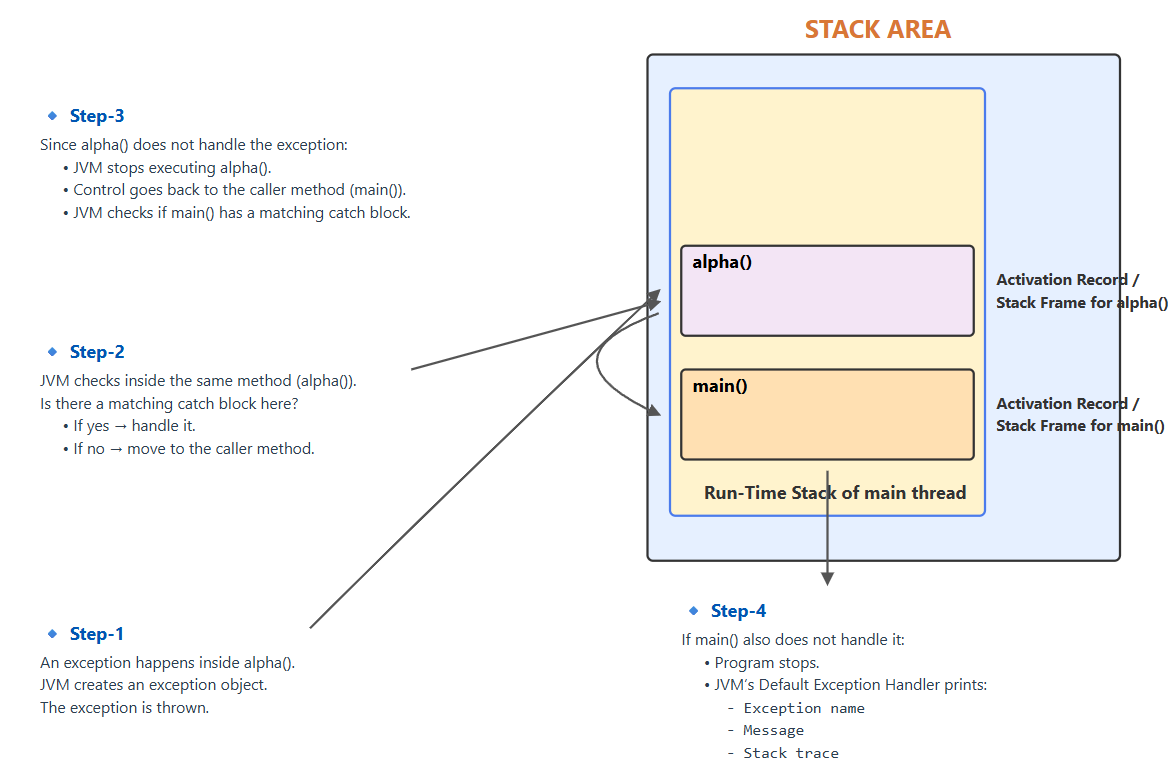
**✅ Very Important Concept – Exception Propagation**

If a method does not handle exception,  
It is propagated to the calling method.

If main method also does not handle,  
Then default exception handler executes.







**✅ Whenever an Exception Occurs, We Have Three Options**

1. **Handle the exception**
2. **Declare (Duck) the exception**
3. **Catch and Rethrow the exception**

**1️⃣ Handling the Exception (try–catch)**

Handling means:

* You write try block for risky code.
* You write catch block to deal with the problem.
* Program continues normally after handling.

try {  
 int x = 10 / 0;  
} catch (ArithmeticException e) {  
 System.out.println("Cannot divide by zero");  
}

Here:

* Exception is caught.
* Proper action is taken.
* Program does not crash.

Handling means you are taking responsibility.

**2️⃣ Ducking the Exception (Declaring using throws)**

Ducking means:

* The method does NOT handle the exception.
* It informs the caller that this method may throw an exception.
* It uses the throws keyword in the method signature.

void alpha() throws IOException {  
 FileReader fr = new FileReader("file.txt");  
}

Here:

* alpha() is not handling IOException.
* It informs the caller method.
* The caller must either handle it or declare it again.

Ducking means:

“I am not handling it. Caller, you decide what to do.”

It is **not ignoring** the exception.  
It is passing responsibility to the caller.

If no method handles it, program will terminate.

**Important Rule About Ducking**

* Ducking is mainly used for **checked exceptions**.
* For unchecked exceptions, compiler does not force you to use throws.

Eg: Ducking\_Exception

**3️⃣ Rethrowing an Exception**

Sometimes:

* You catch the exception.
* Do some work (like logging).
* Then send the same exception again to the caller.

This is called rethrowing.

void withdraw() throws Exception {  
 try {  
 int x = 10 / 0;  
 } catch (ArithmeticException e) {  
 System.out.println("Logging the error");  
 throw e; // rethrowing  
 }  
}

Here:

* Exception is caught.
* Some action is done.
* Same exception is sent to caller.

After throw, remaining code in that block will NOT execute.

If you want something to execute always, use finally.

Eg: RethrowExample

**throw vs throws**

**🔹 throw**

* Used inside method body or catch block.
* Used to actually throw an exception object.
* Execution stops immediately after it.

Example:

throw new ArithmeticException("Invalid operation");

Eg: ThrowKeywordExample

**🔹 throws**

* Used in method signature.
* Used to declare that method may throw exception.
* Does not throw anything by itself.

Example:

void test() throws IOException

**Checked vs Unchecked Exceptions**

**✅ Checked Exceptions**

* Compiler checks them at compile time.
* Compiler forces you to handle or declare them.
* Example: IOException, SQLException.

These are usually external problems:

* File not found
* Database connection failure
* Network issues

**✅ Unchecked Exceptions**

* Occur at runtime.
* Compiler does NOT force you to handle them.
* They are subclasses of RuntimeException.

Example:

* ArithmeticException
* NullPointerException
* ArrayIndexOutOfBoundsException

**Important Clarification**

❌ Checked and unchecked are NOT both runtime exceptions.

✔ Only unchecked exceptions are runtime exceptions.  
✔ Checked exceptions are checked at compile time.

**Industry Recommendation**

**🔹 Checked Exceptions**

Usually declared (ducked) or handled depending on design.

**🔹 Unchecked Exceptions**

Generally:

* Do NOT catch everywhere.
* Fix root cause.
* Let them propagate to global handler (like Spring Boot global exception handler).

In modern industry:

* Business validation errors → usually unchecked.
* System / IO errors → often checked.

**Is Ducking Equal to Handling?**

No.

Handling:

* You catch and take action.

Ducking:

* You pass responsibility to caller.

They are completely different.

**finally Block**

finally block:

* Executes whether exception occurs or not.
* Executes even if return statement is present.
* Executes even if exception is thrown.
* Used to close resources like:
  + Database connections
  + File streams
  + Network connections

Example:

try {  
 FileReader fr = new FileReader("file.txt");  
} catch (Exception e) {  
 System.out.println("Error occurred");  
} finally {  
 System.out.println("Closing resources");  
}

**Modern Alternative (Important Missing Point)**

Instead of finally, we now use:

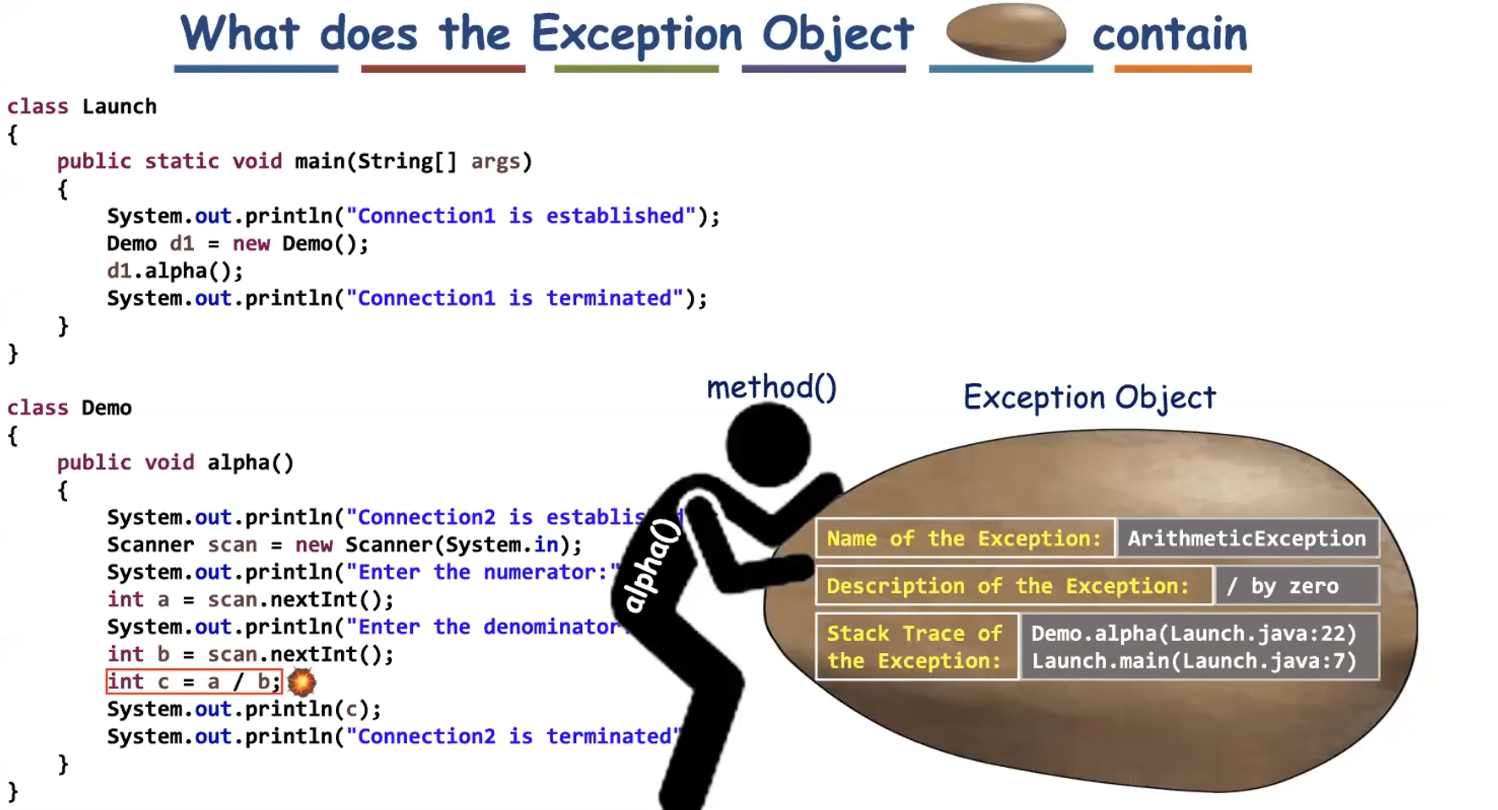
try (FileReader fr = new FileReader("file.txt")) {  
}

This is called **try-with-resources**.  
Resources are automatically closed.

This is industry preferred.

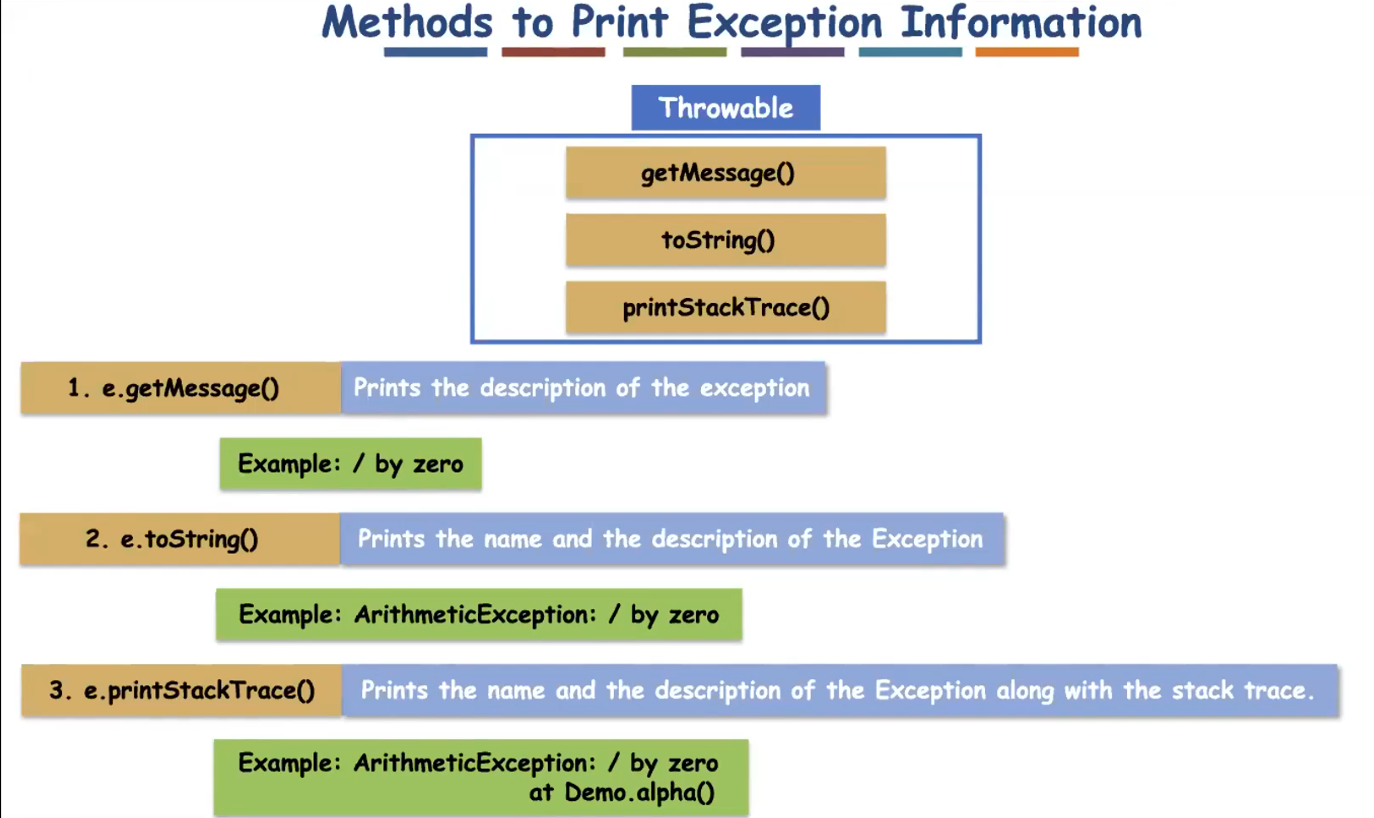
**Clean Summary**

| **Concept** | **Meaning** |
| --- | --- |
| Handling | Catch and solve the problem |
| Ducking | Declare using throws and pass responsibility |
| Rethrowing | Catch, do something, and throw again |
| throw | Actually throws exception |
| throws | Declares possible exception |
| Checked | Compiler forces handling |
| Unchecked | Compiler does not force handling |
| finally | Always executes |



Stack trace means at which stack frame the exception has occurred.

Methods used in the catch bock to print the information are



Eg: Exception\_Print\_Methods\_Demo

**✅ Important Rules About finally Block**

**1️⃣ finally cannot be written alone**

It must be written with:

* try + catch + finally  
  OR
* try + finally

❌ This is INVALID:

finally {  
 System.out.println("Hello");  
}

**2️⃣ finally block will NOT execute if System.exit() is called before it**

Because:

* System.exit(0) immediately terminates the JVM.
* When JVM stops, no further code executes.
* So finally block is skipped.

**🔎 Why Normally finally Always Executes?**

Normally finally executes in cases like:

✔ Exception occurs  
✔ No exception  
✔ Return statement inside try  
✔ Break / Continue

But ❌ Not when:

* System.exit() is called
* JVM crashes
* Power failure 😄

**🔥 Important Interview Point**

If interviewer asks:

👉 *“Does finally always execute?”*

Correct Answer:

Finally block executes in almost all situations except when JVM is terminated explicitly using System.exit() or JVM crashes.

Eg: Finally\_Block\_Exit\_Method

