**Exception Handling in Core Java**

🧨 What is an **Exception**?

An exception is an event that occurs during the execution of a program and disrupts the normal flow of instructions.

Java divides exceptions into two broad types:

1. **Checked Exceptions (Compile-Time)**
   * Handled during compilation.
   * You must either handle them using try-catch or declare them using throws.
   * Examples: IO Exception, SQL Exception, FileNotFoundException
2. **Unchecked Exceptions (Runtime)**
   * Occur during execution.
   * Not mandatory to catch.
   * Examples: NullPointerException, Arithmetic Exception, ArrayIndexOutOfBoundsException

**🔸 Error vs Exception?**

* Errors: Serious issues (like OutOfMemoryError, StackOverflowError) — cannot be recovered
* Exceptions: Can be handled using code

**🧪 Interview Tricky Question Example:**

Q: Will finally block execute if there’s a return statement in try?

public int test() {

try {

return 1;

} finally {

System.out.println("finally executed");

}

}

✅ Output: finally executed  
→ And method returns 1

💡 So: finally block always executes, even after return, break, or continue — unless System.exit() is called or JVM crashes.

What is **Exception** **Handling**?

Exception handling provides a way to gracefully handle errors so the program doesn't crash unexpectedly and we can provide meaningful feedback or recovery.

🧠 2. **When to use Checked vs Unchecked Exceptions**?

☑️ Use Checked Exceptions when:

* The error is expected and recoverable
* You want to force the caller to handle the issue

Examples: FileNotFoundException, SQL Exception

☑️ Use Unchecked Exceptions when:

* The error is due to a bug or logic mistake
* Caller cannot recover from it

Examples: NullPointerException, Arithmetic Exception

🧨 3. **throw vs throws** — very **important** in interviews

🔹 throw — used to actually throw an exception

🔹 throws — used in method signature to declare an exception

🔸 throw — example:

throw new IllegalArgumentException("Invalid age");

🔸 throws — example:

public void readFile(String path) throws IO Exception {

FileReader fr = new FileReader(path); // IO Exception is checked

}

💡 Rules:

* You can only throw one exception using throw.
* You can declare multiple exceptions using throws.
* throw is followed by an instance of Exception.
* throws is followed by exception class names.

Structure:

Throwable  
├── Exception  
│ ├── Checked Exceptions  
│ └── Unchecked Exceptions (Run time Exception)  
└── Error (serious problems like OutOfMemory Error — not meant to handle)

🔷 2. **Checked vs Unchecked Exceptions**

📘 Checked Exceptions (Compile-Time) You MUST handle them (try-catch or throws). Examples:

* IOException
* FileNotFoundException
* SQLException
* ClassNotFoundException
* InterruptedException

✅ When to Use:  
If something external (file, network, DB) might fail — and you can recover or notify user.

public void readFile(String path) throws IOException {

FileReader fr = new FileReader(path); // may throw IOException

}

We can handle this

try {

readFile("abc.txt");

} catch (IOException e) {

System.out.println("Handled!");

}

📙 Unchecked Exceptions (Runtime Exceptions) No need to handle them explicitly.

Examples:

* NullPointerException
* ArrayIndexOutOfBoundsException
* ArithmeticException
* IllegalArgumentException

✅ When to Use:  
These occur due to bugs in code — like logic errors — and generally not recoverable.

String s = null;

System.out.println(s.length()); // NullPointerException

4. **Common Questions to Ask Yourself** (or Get Asked):

Q: What happens if you don’t handle a checked exception? ❌ Compile-time error!

Q: What if you don’t handle unchecked exception? ✅ Compiles, but throws error at runtime

Q: Can you create your own exception? ✅ Yes, by extending Exception or RuntimeException

Q: Can constructors throw exceptions? ✅ Yes, and they must declare it using throws

Java Exception Summary Table

| **Category** | **Type** | **Is Checked?** | **Common Examples** | **When to Use / Handle?** |
| --- | --- | --- | --- | --- |
| Compiler-checked | Checked Exception | Yes | IOException, SQLException, ParseException | When dealing with external resources like files, DB, network |
| Developer mistakes | Unchecked Exception | No | NullPointerException, ArithmeticException | Usually indicate bugs — fix the logic |
| System/critical error | Error | No | OutOfMemoryError, StackOverflowError | Not to be handled, JVM-related |

| **Keyword** | **Used For** | **Syntax** | **Notes** |
| --- | --- | --- | --- |
| throw | Actually throwing exception | throw new Exception("msg"); | Inside method, followed by an exception object |
| throws | Declaring possible exceptions | public void method() throws IOException | In method signature, can declare multiple |

throw vs throws Summary Table

💡 Key Rules:

1. Only unchecked exceptions can be thrown without declaring them.
2. Checked exceptions must be declared using throws OR handled with try-catch.
3. A method can declare multiple exceptions with throws: public void method() throws IOException, SQLException { ... }
4. throw is used with an exception object (must be an instance of Throwable or subclass).
5. throws is used with exception class names, not objects.

✅ Quick Tip to Remember:

* throw → action (you throw something)
* throws → declaration (you warn about something you might throw)

**Tricky Exception Handling Questions (Theory + Practical)**

**1️.What is the difference between throw and throws in Java?**🧠 Theory  
 Answer:

* throw is used to actually throw an exception (used inside method).
* throws is used in method signature to declare that the method may throw exception(s).
* throw is followed by an object, throws is followed by class names.

Example:

public void test() throws IOException { ... }

throw new IOException("Error occurred");

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2️. **Can we throw an exception from finally block?**  
 🧪 Practical  
 Answer: Yes, but it will override any exception thrown from try or catch.

Example:

try {

throw new Exception("from try");

} finally {

throw new Exception("from finally"); // This will be the one actually thrown

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**3️. What if both catch and finally throw exceptions? Which one is propagated?** Answer:Exception from finally block will override catch.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**4️. Can you catch multiple exceptions in one catch block?  
 🧠 Theory + Practice** Yes, using multi-catch:

try {

// code

} catch (IOException | SQLException e) {

e.printStackTrace();

}

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* 1. **Can a method have try without catch or finally?** Answer: No, it must have either catch or finally.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Can we have multiple catch blocks? In what order should they appear?**Yes, and child exceptions should come first.

Example:

try {

// code

} catch (FileNotFoundException e) {

} catch (IOException e) {

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Throwable** (class)

* It’s the superclass of all errors and exceptions in Java.
* Only instances of Throwable (or its subclasses) can be thrown using throw.
* Two main subclasses: Exception and Error

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

| **Term** | | **Type** | **Use** | **Example** |
| --- | --- | --- | --- | --- |
| throw | keyword | | to actually throw an exception | throw new IOException("error"); |
| throws | keyword | | to declare exceptions in method sig. | public void m() throws IOException |
| Throwable | class | | superclass of Exception/Error | Throwable t = new Exception(); |

1. **Can we catch Exception and Throwable?** Answer: You can catch both, but it’s a bad practice to catch Throwable unless you're writing a framework.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **What is the difference between Checked and Unchecked Exceptions?**  
    🧠 Theory  
   Checked → must handle or declare (IOException)  
   Unchecked → optional (NullPointerException)

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. **Can we write code after throw statement?**  
    Answer: No, compiler will throw “unreachable code” error.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. **Can constructors throw exceptions?**  
    Yes, constructors can throw both checked and unchecked exceptions.

Example:

public class MyClass {

public MyClass() throws IOException {

throw new IOException("constructor failure");

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Will finally block execute if System.exit(0) is called?**  
    Answer: No, JVM shuts down and finally will not run.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. **What happens if we catch an exception but do not handle it (empty catch block)?**  
    Answer: It swallows the exception. It’s a bad practice — could lead to silent failures.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. **Can a method override another method and throw a broader exception?**  
    Answer: No. Overridden method cannot throw broader (superclass) checked exceptions.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Should we use exceptions for flow control in Java?**  
    Answer: No. Exceptions are expensive (in terms of performance). They are for exceptional cases, not regular logic.

**Final, Finally, Finalize**

| **Keyword** | **Type** | **Purpose** | **Used With** |
| --- | --- | --- | --- |
| final | Keyword | To declare constants or prevent modification | Variables, methods, classes |
| finally | Block | Used in exception handling — always executes | try-catch-finally block |
| finalize() | Method | Called by Garbage Collector before object removal | Object (from Object class) |

1️. **final** — Constant/Non-Changeable

✅ **Real-time analogy:** If you say, “final price = ₹500,” you can’t change that price again.

Used for:

* final variable: value cannot be reassigned.
* final method: cannot be overridden.
* final class: cannot be extended.

📌 Examples:

a) **final variable:**

final int maxSpeed = 120;

maxSpeed = 150; // ❌ compile-time error

b**) final method:**

class Vehicle {

final void run() {

System.out.println("Vehicle is running");

}

}

class Car extends Vehicle {

// void run() { } ❌ can't override

}

**c) final class:**

final class Shape { }

// class Circle extends Shape {} ❌ compile-time error

🟩 2️.**finally** — Always Executes (Exception Handling)

✅ **Real-time analogy**: No matter what — whether your train is on time or delayed, the platform must be cleaned after every train. That’s finally — it always runs.

Used to close resources like files, DB connections.

📌 Example:

try {

int a = 5 / 0;

} catch (ArithmeticException e) {

System.out.println("Error");

} finally {

System.out.println("Cleanup"); // Always runs

}

**Even if exception is not caught — finally still executes:**

try {

int a = 5 / 0;

} finally {

System.out.println("Will always execute");

}

**❗ Note: finally does not run only in case of System.exit(0)**

**What is finalize()?**

* It's a method defined in Object class.
* Java gives every object a chance to clean up before it is garbage collected.
* If you override finalize() in your class, JVM will call it before destroying the object.

**Real-world analogy:** Imagine you’re checking out of a hotel (your object is no longer needed). The housekeeping staff comes in to clean the room before it's given to someone else — that’s like finalize(). The object is about to be deleted from memory, and Java says “Hey, do you want to do any last clean-up?”

🧪 Example:

class HotelRoom {

String roomNumber;

HotelRoom(String roomNumber) {

this.roomNumber = roomNumber;

}

@Override

protected void finalize() throws Throwable {

System.out.println("Cleaning up room: " + roomNumber);

}

}

public class TestFinalize {

public static void main(String[] args) {

HotelRoom r1 = new HotelRoom("101");

r1 = null; // Eligible for GC

System.gc(); // Request JVM to run GC

System.out.println("Main method ends");

}

}

🧾 Output (order may vary):

Main method ends

Cleaning up room: 101

**📌 Important points:**

1. finalize() is not guaranteed to run immediately after object becomes eligible for GC.
2. JVM decides when to run GC — so finalize() might run late or not at all.
3. It was used in old Java versions to release non-memory resources like file handles or DB connections.
4. Deprecated since Java 9 — because it is unreliable.

✅ **So what do we use instead of finalize() now**?

* Use **try-with-resources** for automatic cleanup.
* Implement AutoCloseable and close() method for explicit cleanup.

🛠️ Example with try-with-resources:

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

// read file

} catch (IOException e) {

e.printStackTrace();

}

**Q1. Can we declare a final method inside a final class?**

✅ Answer: Yes.

Explanation:

* A final class cannot be extended.
* A final method cannot be overridden.
* Since final class can't be inherited anyway, marking a method inside it as final is redundant but allowed.

**Tip: finally always executes, even after return, break, continue, or exceptions**.

**Q2. What will happen if an exception is thrown inside try and also inside finally?**

✅ Answer: finally's exception will override the try’s exception.

🧪 Example:

public class Test {

public static void main(String[] args) {

try {

throw new RuntimeException("From try");

} finally {

throw new RuntimeException("From finally");

}

}

}

🧾 Output: Exception: From finally

🎯 Interview Point: finally’s exception masks try’s exception.

Q3. **What if we declare a final reference to a mutable object?**

✅ Answer: You can’t change the reference, but can modify the object.

🧪 Example:

final StringBuilder sb = new StringBuilder("Hi");

sb.append(" there"); // ✅ allowed

// sb = new StringBuilder(); ❌ not allowed