Java Exceptions and Errors: A Beginner's Guide

Problems in Java

When you run a Java application, problems or issues can occur. These problems interrupt the normal control flow of your program. They can be caused by:

- **Wrong input**: User provides data your program doesn't expect
- **Coding errors**: Bugs in your code
- **Unforeseen circumstances**: Situations that weren't anticipated (like a file not existing when you try to open it)

When a problem occurs, **Java throws an exception** to signal that something has gone wrong. An exception is an event that disrupts the normal control flow of your program.

The good news is that Java gives you tools to detect these problems and handle them gracefully, so your program doesn't just crash.

Errors vs. Exceptions

Java distinguishes between two types of problems: **Errors** and **Exceptions**. Understanding this distinction is crucial because you handle them very differently.

Errors

An error is a serious issue or problem.

- Errors represent problems that applications **cannot** handle or recover from
- Errors cause the application to "crash" or stop executing
- Example: An **OutOfMemoryError** occurs when Java runs out of memory — your program cannot recover from this
- You should **not** try to catch errors in your code

Exceptions

Exceptions are problems that applications can handle and recover from.

- Exceptions represent issues that occur during normal program execution
- Your program can catch exceptions and continue running
- Example: Trying to access an element outside the size of an array — your program can detect this and take corrective action
- You **should** handle exceptions in your code

Visual Hierarchy

```
                    Throwable
                   /         \
```

```
                  Error      Exception
                (Serious)    (Recoverable)
          OutOfMemoryError   ArithmeticException
          StackOverflowError ArrayIndexOutOfBoundsException
```

Common Exceptions

The following are common **RuntimeException** subclasses that you will encounter. RuntimeExceptions are unchecked, meaning Java doesn't require you to explicitly handle them (though it's often good practice to do so).

1. ArrayIndexOutOfBoundsException

This occurs when you try to access an array element at an index that doesn't exist.

```
int[] numbers = {1, 2, 3};
System.out.println(numbers[5]);  // ArrayIndexOutOfBoundsException!
// Array only has valid indices: 0, 1, 2
```

What the exception tells you: You tried to access an index that is out of bounds.

How to handle it:

```
try {
    int[] numbers = {1, 2, 3};
    int index = 5;
    System.out.println(numbers[index]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Error: Index " + index + " is out of bounds!");
    System.out.println("Array length is: " + numbers.length);
}
```

2. NullPointerException

This occurs when you try to use an object that is `null` (hasn't been initialized).

```
String text = null;
System.out.println(text.length());  // NullPointerException!
// text is null, so it has no methods
```

What the exception tells you: You tried to call a method or access a property on a null object.

How to handle it:

```
try {
    String text = null;
    System.out.println(text.length());
} catch (NullPointerException e) {
    System.out.println("Error: You tried to use an object that is null!");
}
```

3. ArithmeticException

This occurs when you perform an invalid arithmetic operation, most commonly dividing by zero.

```
int result = 10 / 0;  // ArithmeticException!
```

How to handle it:

```
try {
    int a = 10;
    int b = 0;
    int result = a / b;
    System.out.println("Result: " + result);
} catch (ArithmeticException e) {
    System.out.println("Error: Cannot divide by zero!");
}
```

4. NumberFormatException

This is a **RuntimeException** subclass that occurs when you try to convert a string to a number, but the string isn't a valid number.

```
String number = "abc";
int value = Integer.parseInt(number);  // NumberFormatException!
// "abc" cannot be converted to an integer
```

How to handle it:

```
try {
    String number = "abc";
    int value = Integer.parseInt(number);
} catch (NumberFormatException e) {
    System.out.println("Error: '" + number + "' is not a valid number!");
}
```

5. StringIndexOutOfBoundsException

This occurs when you try to access a character in a string at an invalid index.

```
String word = "Hello";
char letter = word.charAt(10);  // StringIndexOutOfBoundsException!
// "Hello" only has indices 0-4
```

How to handle it:

```
try {
    String word = "Hello";
    char letter = word.charAt(10);
} catch (StringIndexOutOfBoundsException e) {
    System.out.println("Error: That character position doesn't exist!");
}
```

Checked Exceptions

Checked exceptions are exceptions that Java requires you to handle. They are exceptions that are NOT subclasses of `RuntimeException`. The compiler will force you to either catch them in a try-catch block or declare them with the `throws` keyword.

FileNotFoundException

A common checked exception. This occurs when you try to open or read a file that doesn't exist.

```
// This will not compile without handling the exception!
FileReader reader = new FileReader("file.txt");
```

The compiler gives an error because `FileNotFoundException` is a checked exception.

How to handle it:

```
try {
    FileReader reader = new FileReader("file.txt");
    // use the file
} catch (FileNotFoundException e) {
    System.out.println("Error: File not found!");
    System.out.println("Details: " + e.getMessage());
}
```

Key Point: For checked exceptions, Java requires that you deal with them one way or another. You cannot ignore them like you can with unchecked exceptions.

Common Errors

Errors represent serious problems that your application cannot recover from. Unlike exceptions, you cannot reliably catch and handle errors. If an error occurs, your program will crash.

1. OutOfMemoryError

Occurs when the Java program runs out of heap memory.

```
// This might cause OutOfMemoryError (on some systems)
int[] hugeArray = new int[Integer.MAX_VALUE];
```

Why you cannot handle it: If your program runs out of memory, there is no memory available for recovery. The program will crash.

2. StackOverflowError

Occurs when the program runs out of stack space, usually due to infinite recursion or deeply nested method calls.

```
public void infiniteRecursion() {
    infiniteRecursion();  // Calls itself forever without a base case
    // Eventually causes StackOverflowError
}
```

Why you cannot handle it: This indicates a fundamental logic error in your code. The stack cannot recover from this condition.

3. NoClassDefFoundError

Occurs when the Java Virtual Machine (JVM) cannot find a class that your program needs at runtime.

```
// If the JVM can't find a compiled class your code references,
// it throws NoClassDefFoundError
```

Why you cannot handle it: This is a configuration or setup problem with your classpath or deployment, not something your code can fix while running.

Checked vs. Unchecked Exceptions

Java distinguishes between two categories of exceptions based on whether the compiler requires you to handle them.

Unchecked Exceptions (RuntimeExceptions)

Unchecked exceptions are subclasses of `RuntimeException`. The compiler does **not** require you to handle them.

- **Why unchecked?** These exceptions typically result from your code logic and programming errors
- **Handling is optional** — you can write code without try-catch blocks for these
- **Best practice:** You should still handle them, but Java doesn't force you to
- **Examples:** `ArrayIndexOutOfBoundsException`, `NullPointerException`, `ArithmeticException`, `NumberFormatException`

Example without handling (compiles fine):

```
int[] numbers = {1, 2, 3};
System.out.println(numbers[5]);  // No try-catch needed - but will crash
if index is wrong
```

Example with handling (better practice):

```
int[] numbers = {1, 2, 3};
try {
    System.out.println(numbers[5]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Index out of bounds!");
}
```

Checked Exceptions

Checked exceptions are exceptions that are NOT subclasses of `RuntimeException`. The compiler **requires** you to handle them.

- **Why checked?** These exceptions represent recoverable situations often outside your code's control
- **Handling is mandatory** — the code will not compile if you don't handle them
- **Best practice:** Handle them in a try-catch block or declare them with `throws`
- **Examples:** `FileNotFoundException`, `IOException`, `SQLException`

Example that will NOT compile:

```
// This won't compile! FileNotFoundException is a checked exception
FileReader reader = new FileReader("file.txt");
```

Error: `unreported exception FileNotFoundException; must be caught or declared to be thrown`

Solution 1: Use try-catch

```
try {
    FileReader reader = new FileReader("file.txt");
    // use the reader
} catch (FileNotFoundException e) {
    System.out.println("File not found: " + e.getMessage());
}
```

Solution 2: Use throws (see section below)

```
public void readFile() throws FileNotFoundException {
    FileReader reader = new FileReader("file.txt");
    // use the reader
}
```

Quick Comparison

Aspect	Unchecked (RuntimeException)	Checked Exception
Compiler enforcement	Optional - no compile error	Required - compile error if not handled
Cause	Usually programming errors	External factors or system conditions
Handling requirement	No (but recommended)	Yes - mandatory
Examples	ArrayIndexOutOfBoundsException, NullPointerException	FileNotFoundException, IOException

How to Handle Exceptions: Try, Catch, and Finally

Try Block

The **try block** wraps code that might throw an exception. It says to Java: "Try this code, and if there is a problem, pass it to the catch block."

```
try {
    // Exception-generating code goes here
    int result = 10 / 0;
    // If an exception occurs, the rest of this block is skipped
    System.out.println("This won't print if an exception occurs");
} catch (ArithmeticException e) {
    // Handle the exception here
}
```

Catch Block

The **catch block** acts as a handler to address exception problems. It is placed immediately after the try block. The exception that is thrown is passed into the catch block as a parameter.

Important: For some exception types (checked exceptions like `FileNotFoundException`), try/catch blocks are **required**. For others (unchecked exceptions), they are **optional** but recommended.

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    // This block only executes if an ArithmeticException is thrown
    System.out.println("Error: Cannot divide by zero!");
}
```

Multiple Catch Blocks

You can use **multiple catch blocks** to handle different types of exceptions. Each catch gets its own exception type and exception variable. When an exception occurs, Java checks the catch blocks in order until it finds a match.

```
try {
    String input = "abc";
    int number = Integer.parseInt(input);  // Might throw
NumberFormatException
    int[] array = {1, 2, 3};
    int value = array[10];  // Might throw ArrayIndexOutOfBoundsException
} catch (NumberFormatException e) {
    System.out.println("Error: Invalid number format");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Error: Array index out of bounds");
}
```

Catch-All Exception Handler

You can use `catch (Exception variable)` as a catch-all handler. This will handle any exception not handled by the other, more specific catch blocks.

```
try {
    // risky code
} catch (NumberFormatException e) {
    System.out.println("NumberFormatException caught");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("ArrayIndexOutOfBoundsException caught");
} catch (Exception e) {
    // This catches ANY other exception type
    System.out.println("Some other exception occurred: " + e);
}
```

Finally Block

The **finally block** always executes, whether or not an exception occurred. It runs after the try block (if no exception) or after the catch block (if an exception was caught).

```
try {
    int result = 10 / 2;
    System.out.println("Result: " + result);
} catch (ArithmeticException e) {
    System.out.println("Error: Math problem!");
} finally {
    System.out.println("This always runs!");
}

// Output:
// Result: 5
// This always runs!
```

When to use finally:

- Closing files or resources
- Releasing database connections
- Cleanup operations that must happen regardless of whether an exception occurred

Exception Object Methods

The exception object passed into the catch block is rich with information. You can use methods on the object to learn more about what went wrong:

- `.getMessage()` — Returns a human-readable description of what went wrong

```
catch (NumberFormatException e) {
    System.out.println(e.getMessage());
    // Output: For input string: "abc"
}
```

- `.getClass()` — Returns the exception type/class name

```
catch (Exception e) {
    System.out.println(e.getClass());  // class
java.lang.ArithmeticException
}
```

- `.printStackTrace()` — Displays the stack trace showing the execution flow to the problem

```
catch (Exception e) {
    e.printStackTrace();  // Prints the full stack trace
}
```

Stack Trace

The **stack trace** provides a record of the method calls (execution flow) that led to the exception. When you call `.printStackTrace()`, you get output like:

```
java.lang.ArithmeticException: / by zero
    at MyProgram.divide(MyProgram.java:10)
    at MyProgram.main(MyProgram.java:5)
```

This tells you:

- What exception occurred: `ArithmeticException`
- What caused it: "/ by zero"
- Where it happened: Line 10 of MyProgram.java in the `divide` method
- How you got there: Called from line 5 in the `main` method

Complete Example

```
public class BankAccount {
    private double balance = 100;

    public void withdraw(String amountString) {
        try {
            // Try to convert the string to a number
            double amount = Double.parseDouble(amountString);

            // Try to withdraw the amount
            if (amount > balance) {
                System.out.println("Error: Insufficient funds!");
                return;
            }

            balance -= amount;
            System.out.println("Withdrawal successful! New balance: $" +
balance);

        } catch (NumberFormatException e) {
            System.out.println("Error: Please enter a valid amount");
            System.out.println("Details: " + e.getMessage());
        } finally {
            System.out.println("Transaction attempt completed.");
        }
    }
}
```

Throwing Exceptions: The `throws` Keyword

Instead of handling an exception in the method where it occurs, you can **throw** the exception to the calling method. This is called "throwing an exception" or "passing-the-buck."

Using the `throws` Keyword

Use the `throws` keyword on any method to pass an exception back to the calling method instead of handling it locally. This passes the responsibility for handling the exception to the caller.

```
public void readFile() throws FileNotFoundException {
    FileReader reader = new FileReader("file.txt");
    // ... use the reader
    // Don't catch the exception - pass it to the caller
}
```

Syntax: Use a comma-separated list of exception types after `throws` to indicate what types of exceptions the caller method might anticipate.

```
public void riskyMethod() throws FileNotFoundException, IOException {
    // This method might throw FileNotFoundException or IOException
    // The caller must handle these
}
```

Throws vs. Try-Catch: When to Use Each

Try-Catch is the preferred approach:

- Handle the exception where it happens
- Write code to recover from the exception
- Your method is "self-contained" and doesn't burden the caller

Example (preferred):

```
public void readFile(String filename) {
    try {
        FileReader reader = new FileReader(filename);
        // use the reader
    } catch (FileNotFoundException e) {
        System.out.println("File not found: " + filename);
        // Handle the problem and continue
    }
}
```

When to use throws:

- The method where the exception occurs may not know how to recover from it
- Centralizing exception handling to reduce redundant try-catch blocks across multiple methods
- You want the calling code to make decisions about how to handle the error

Example (when throws is appropriate):

```
// Each method just throws - handling is centralized in main
public void processData() throws FileNotFoundException, IOException {
    readFile("data.txt");
    parseData();
}

public static void main(String[] args) {
    try {
        processor.processData();
    } catch (FileNotFoundException e) {
        System.out.println("Data file not found");
    } catch (IOException e) {
        System.out.println("Error reading data");
    }
}
```

Important: If a checked exception is thrown from `main` without being caught, the application will crash.

Catching and Rethrowing

You can catch an exception, do something with it, and then **rethrow** it to pass it up the call stack.

```
try {
    FileReader reader = new FileReader("file.txt");
    // use the reader
} catch (FileNotFoundException e) {
    System.out.println("Logging: File not found at " + new Date());
    throw e;  // Rethrow the exception to the caller
}
```

When to use catch and rethrow:

- You want to log the exception or perform some action when it occurs
- But you don't actually know how to recover from it
- You want another method up the call stack to handle it

Example: Throws vs. Try-Catch

Option 1: Handle with try-catch

```
public void loadFile(String filename) {
    try {
        FileReader reader = new FileReader(filename);
        // read from file
    } catch (FileNotFoundException e) {
        System.out.println("Could not find: " + filename);
    }
}

// Caller doesn't need to worry about exceptions
loadFile("data.txt");
```

Option 2: Handle with throws

```
public void loadFile(String filename) throws FileNotFoundException {
    FileReader reader = new FileReader(filename);
    // read from file
    // Let the caller deal with the exception
}

// Caller must handle the exception
try {
    loadFile("data.txt");
} catch (FileNotFoundException e) {
    System.out.println("Could not find: " + filename);
}
```

Best Practices for Exception Handling

1. Be Specific with Your Exception Types

Don't do this (too broad):

```
try {
    // code here
} catch (Exception e) {  // Catches EVERYTHING
    System.out.println("Something went wrong");
}
```

Catching `Exception` is too broad. You might catch exceptions you didn't anticipate, and it makes debugging harder.

Do this instead:

```
try {
    // code here
} catch (ArithmeticException e) {  // Specific exception type
    System.out.println("Math error: " + e.getMessage());
} catch (ArrayIndexOutOfBoundsException e) {  // Specific exception type
    System.out.println("Array access error");
}
```

Being specific allows you to handle different exceptions appropriately.

2. Always Log or Display Exception Information

Don't do this (silent failure):

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    // Empty catch block - exception is silently ignored!
}
```

Silent failures are hard to debug. You won't know what went wrong.

Do this instead:

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Error: " + e.getMessage());
    e.printStackTrace();  // Shows the full stack trace for debugging
}
```

3. Use Exceptions for Exceptional Cases, Not Normal Control Flow

Don't use exceptions for normal program logic:

```
// BAD: Using exception for normal control flow
try {
    int i = 0;
    while (true) {
        System.out.println(myArray[i]);
        i++;
    }
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Reached end of array");
}
```

This is slow and bad practice. Exceptions are expensive to throw.

Do this instead:

```
// GOOD: Use normal control flow
for (int i = 0; i < myArray.length; i++) {
    System.out.println(myArray[i]);
}
```

4. Check Before You Access (Prevention)

Do this to prevent exceptions:

```
// Check bounds before accessing
if (index >= 0 && index < array.length) {
    int value = array[index];
} else {
    System.out.println("Invalid index");
}

// Check for null before using an object
if (text != null) {
    System.out.println(text.length());
}

// Check for zero before dividing
if (divisor != 0) {
    result = dividend / divisor;
}
```

Prevention is better than having to catch exceptions.

5. Use Finally for Cleanup

Always use finally for resource cleanup:

```
try {
    FileReader reader = new FileReader("file.txt");
    // read from file
} catch (FileNotFoundException e) {
    System.out.println("File not found");
} finally {
    // This always runs, even if an exception occurs
    // Perfect place to close files and release resources
    reader.close();
}
```

6. Choose the Right Handling Strategy

Use try-catch when:

- You can actually recover from the exception
- You know what to do if the exception occurs
- The method is responsible for the error

Use throws when:

- You don't know how to recover from the exception
- Multiple methods have the same issue (centralize handling)
- You want the caller to decide how to handle it

Summary

Errors vs. Exceptions

Aspect	Error	Exception
Severity	Serious, usually unrecoverable	Problems that can be recovered
Cause	JVM or system-level issues	Program logic or data issues
Should you catch?	No (generally)	Yes
Examples	OutOfMemoryError, StackOverflowError	ArithmeticException, NullPointerException

Checked vs. Unchecked Exceptions

Aspect	Unchecked (RuntimeException)	Checked Exception
Compiler requirement	Optional - compiles without handling	Required - compile error if not handled
Handling	Try-catch not required (but recommended)	Try-catch or throws is mandatory
Examples	ArrayIndexOutOfBoundsException, NullPointerException	FileNotFoundException, IOException

Exception Handling Strategies

Strategy	Use When	Example
Try-Catch	You can recover from the error	Catch invalid user input
Throws	Caller should handle it	Database methods throw exceptions to calling code

Strategy	Use When	Example
Catch and Rethrow	You want to log but pass to caller	Log error, then rethrow
Finally	Need cleanup regardless of outcome	Close files or resources

Key Takeaways:

- Errors are serious and unrecoverable; exceptions can be handled
- Checked exceptions require handling; unchecked exceptions don't
- Use try-catch to handle exceptions where they occur (preferred)
- Use throws to pass responsibility to the calling method
- Always log exception information for debugging
- Use finally for cleanup operations
- Catch specific exceptions, not generic `Exception`
- Don't use exceptions for normal program flow

Quick Reference: Common Exceptions

RuntimeExceptions (Unchecked)

Exception	Cause	Handling Required
`ArithmeticException`	Invalid math operation (like dividing by zero)	No, but recommended
`ArrayIndexOutOfBoundsException`	Accessing invalid array index	No, but recommended
`NullPointerException`	Using a null reference	No, but recommended
`NumberFormatException`	Converting invalid string to number	No, but recommended
`StringIndexOutOfBoundsException`	Accessing invalid string index	No, but recommended

Checked Exceptions

Exception	Cause	Handling Required
`FileNotFoundException`	File you're trying to open doesn't exist	**Yes - Mandatory**
`IOException`	I/O operation fails	**Yes - Mandatory**

Common Errors

Error	Cause
`OutOfMemoryError`	Program runs out of memory
`StackOverflowError`	Stack space exhausted (usually infinite recursion)
`NoClassDefFoundError`	Required class not found at runtime