Types of Errors in Java

1. Compile-time Errors

- These errors occur during compilation, i.e., before the program runs.
- The compiler checks for **syntax and structural mistakes** in the code.
- Common causes:
    - Missing semicolon `;`
    - Misspelled keywords
    - Type mismatch (assigning `int` to `String`)
    - Missing return statement in a non-void method

Example:

```
public class Main {
    public static void main(String[] args) {
        int x = "hello"; // Compile-time error: incompatible types
    }
}
```

2. Run-time Errors

- These occur **while the program is executing**.
- Even if the code compiles successfully, it can still fail at runtime.
- Common causes:
    - Dividing a number by zero
    - Accessing an array index out of bounds
    - Null pointer exception
    - File not found

Example:

```
public class Main {
    public static void main(String[] args) {
        int a = 5 / 0; // Runtime Error: ArithmeticException
    }
}
```

3. Logical Errors

- The program compiles and runs, but **the output is incorrect**.
- These errors happen due to **wrong logic** applied by the programmer.
- The compiler cannot detect these.

Example:

```
public class Main {
    public static void main(String[] args) {
        int a = 5, b = 10;
        int avg = (a + b) / 2;
        System.out.println("Average: " + avg); // Correct output: 7
```

```
        // If programmer mistakenly writes:
        int wrongAvg = (a - b) / 2;
        System.out.println("Average: " + wrongAvg); // Logical error: -2
    }
}
```

4. Linker Errors (Rare in Java)

- Usually happen if **external libraries, classes, or methods are missing** during linking.
- Example: Calling a method from a library that is not included in the classpath.

5. Exceptions vs Errors

- In Java, **both are part of Throwable class**, but they differ:

  Exception: Conditions that can be handled (e.g., file not found, divide by zero).

  Error: Serious problems that cannot be handled by the program (e.g., `OutOfMemoryError`, `StackOverflowError`).

Example:

```
public class Main {
    public static void main(String[] args) {
        // Example of Error (cannot handle)
        // throw new OutOfMemoryError("Memory exhausted");

        // Example of Exception (can handle using try-catch)
        try {
            int a = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
    }
}
```

Try-Catch Block in Java

- `try-catch` is used in **exception handling** to prevent program termination when an error occurs.
- The **`try` block** contains code that might throw an exception.
- The **`catch` block** handles the exception.

Example 1: Simple `try-catch`

```
public class Main {
    public static void main(String[] args) {
        try {
            int a = 10 / 0; // risky code
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
        System.out.println("Program continues...");
    }
```

```
}
```

Output:

```
Cannot divide by zero!
Program continues...
```

Example 2: Multiple Catch Blocks

```
public class Main {
    public static void main(String[] args) {
        try {
            int a = 10 / 0;  // ArithmeticException
            int[] arr = new int[5];
            arr[10] = 20;   // ArrayIndexOutOfBoundsException
        }
        catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of range!");
        }
        catch (Exception e) { // generic handler
            System.out.println("Some other error occurred: " + e);
        }
    }
}
```

Output:

```
Cannot divide by zero!
```

Nested Try-Catch Block

- A `try` block **inside another try block** is called a nested try-catch.
- Useful when different parts of code may throw different exceptions.
- Inner `try` handles its own exceptions, outer `try` handles others.

Example 2: Nested `try-catch`

```
public class Main {
    public static void main(String[] args) {
        try {
            int[] arr = new int[5];

            try {
                arr[10] = 50; // ArrayIndexOutOfBoundsException
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Inner catch: Array index out ofrange");
            }

            int a = 10 / 0; // ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Outer catch: Cannot divide by zero");
        }
    }
```

```
}
```

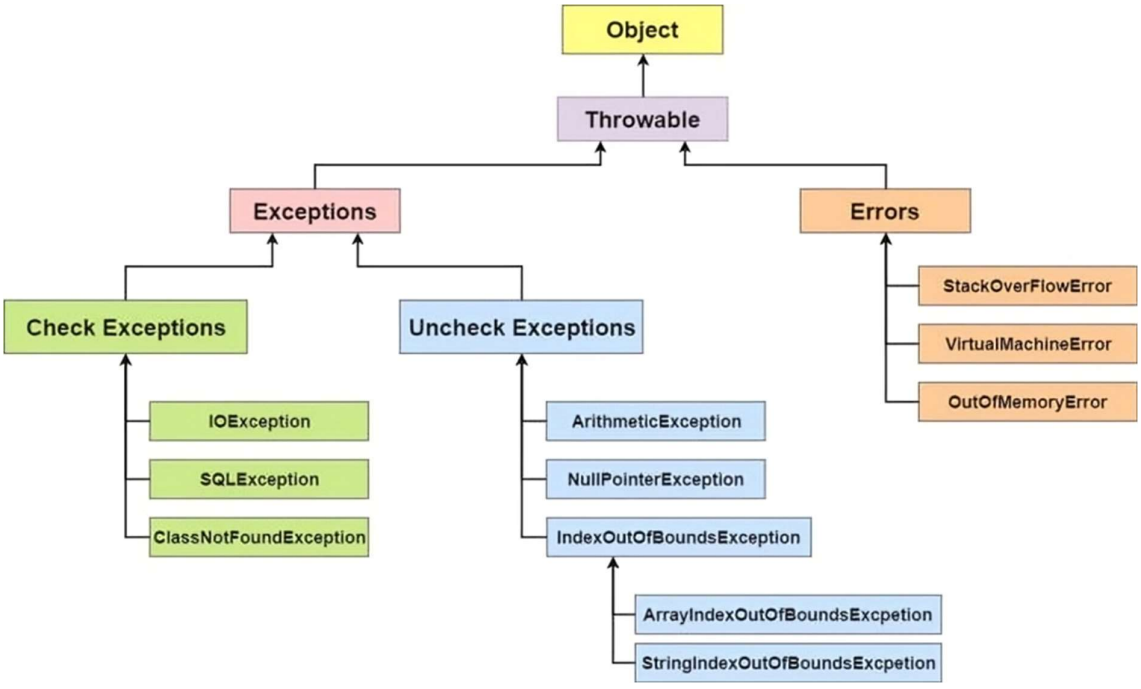
Output:

```
Inner catch: Array index out of range
Outer catch: Cannot divide by zero
```

Exception Class in Java

An *Exception* is an unwanted or unexpected event that occurs during the execution of a program, disrupting the normal flow of instructions. In Java, all exceptions are objects and are derived from the **`Throwable`** class.

Exception Hierarchy:



Key Points:

- **Throwable** → Parent class of all errors and exceptions.
- **Exception** → Represents conditions a program might want to catch.
- **RuntimeException** → Subclass of Exception (unchecked exceptions).
- **Error** → Serious problems that a program should not try to handle (like JVM crashes).

`throw` Keyword

The `throw` keyword is used to **explicitly throw** an exception from a method or block of code.

Syntax:

```
throw new ExceptionType("Error Message");
```

Points to Remember:

Only **one exception** can be thrown at a time using `throw`.
The object thrown must be of type **Throwable** (Exception or subclass).
Commonly used to throw custom or predefined exceptions.

Example:

```
class Test {
    public static void main(String[] args) {
        int age = 15;
        if (age < 18) {
            throw new ArithmeticException("Not eligible for voting");
        }
        System.out.println("Eligible for voting");
    }
}
```

`throws` Keyword

The `throw` keyword is used to **explicitly throw** an exception from a method or block of code.

Syntax:

```
throw new ExceptionType("Error Message");
```

Points to Remember:

Only **one exception** can be thrown at a time using `throw`.
The object thrown must be of type **Throwable** (Exception or subclass).
Commonly used to throw custom or predefined exceptions.

```
class Division {
    // Method declares it might throw an exception
    int divide(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Division by zero is not
allowed");
        }
        return a / b;
    }
}

public class ThrowsExample {
    public static void main(String[] args) {
        Division obj = new Division();
        try {
            int result = obj.divide(10, 0);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Output:

```
Exception caught: Division by zero is not allowed
```

Default Behavior of getMessage() and toString()

- `getMessage()` → returns the string passed in `super(message)`.
- `toString()` → returns `"ClassName: message"`.

Example (Default):

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String msg) {
        super(msg); // Parent Exception stores the message
    }
}

public class Test {
    public static void main(String[] args) {
        try {
            throw new InvalidAgeException("Age must be 18+");
        } catch (InvalidAgeException e) {
            System.out.println("getMessage(): " + e.getMessage());
            System.out.println("toString(): " + e);
        }
    }
}
```

Output:

```
getMessage(): Age must be 18+
toString(): InvalidAgeException: Age must be 18+
```

Overriding getMessage() and toString()

If you want **custom formatting** for error messages, you can override these methods.

Example (Overridden):

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String msg) {
        super(msg);
    }

    @Override
    public String getMessage() {
        return "Custom Message → " + super.getMessage();
    }

    @Override
    public String toString() {
        return "InvalidAgeException occurred → " + super.getMessage();
    }
}

public class Test {
    public static void main(String[] args) {
        try {
            throw new InvalidAgeException("Age is below 18");
        } catch (InvalidAgeException e) {
            System.out.println("getMessage(): " + e.getMessage());
            System.out.println("toString(): " + e);
```

```
        }
    }
}
```

Output:

```
getMessage(): Custom Message → Age is below 18
toString(): InvalidAgeException occurred → Age is below 18
```

`finally` block in Java

- The `finally` block is used in exception handling.
- It contains **cleanup code** that **always executes** after `try` and `catch`, whether an exception occurs or not.

Key Points

1. Runs after `try` / `catch` — no matter what.
2. Executes even if there is a `return` in `try` or `catch`.
3. **Does not run** if JVM exits (`System.exit()`) or crashes.
4. Best used for **closing files, DB connections, releasing resources**.
5. Avoid writing `return` inside `finally` (it overrides the original return/exception).

Example

```
public class FinallyDemo {
    public static void main(String[] args) {
        try {
            System.out.println("In try");
            int x = 10 / 0;  // exception
        } catch (ArithmeticException e) {
            System.out.println("In catch");
        } finally {
            System.out.println("In finally");
        }
    }
}
```

Output:

```
In try
In catch
In finally
```