

Introduction to Exception Handling

Exception Handling in Java is a mechanism to handle runtime errors, maintaining the normal flow of the application.

It allows for:

1. Managing error-prone code.
2. Preventing program crashes due to unexpected runtime issues.
3. Providing alternate solutions to handle errors gracefully.

Why Handle Exceptions?

Prevents abrupt termination of programs.

Allows the application to recover from errors and continue.

Crucial in real-world applications (e.g., handling user inputs, file operations, network connections).

Types of Exceptions

Java exceptions are classified into three main categories:

1. Checked Exceptions:

Exceptions known at compile-time.

Must be handled using **try-catch** or declared in the method signature with **throws**.

Examples: **IOException**, **SQLException**.

2. Unchecked Exceptions:

Occur during runtime.

Do not need to be declared or caught.

Examples: **ArithmaticException**, **NullPointerException**.

3. Errors:

Critical issues that arise from the system (not application-specific).

Usually, should not be handled in the code.

Examples: **OutOfMemoryError**, **StackOverflowError**.

Checked vs Unchecked Exceptions

1. Checked Exceptions are enforced by the compiler. If not handled, the code will fail to compile.
2. Unchecked Exceptions are not checked by the compiler. They occur at runtime and may lead to program crashes if unhandled.
3. Errors usually signal serious issues (like memory or hardware failures) and should not be caught by applications.

Exception Hierarchy in Java

Root class of all exceptions: **Throwable**.

- **Exception**: Represents errors that programs should handle.
 - Subdivided into checked and unchecked exceptions.
- **Error**: Represents severe issues outside the program's control.

The typical hierarchy:

Throwable

Exception

RuntimeException (unchecked)

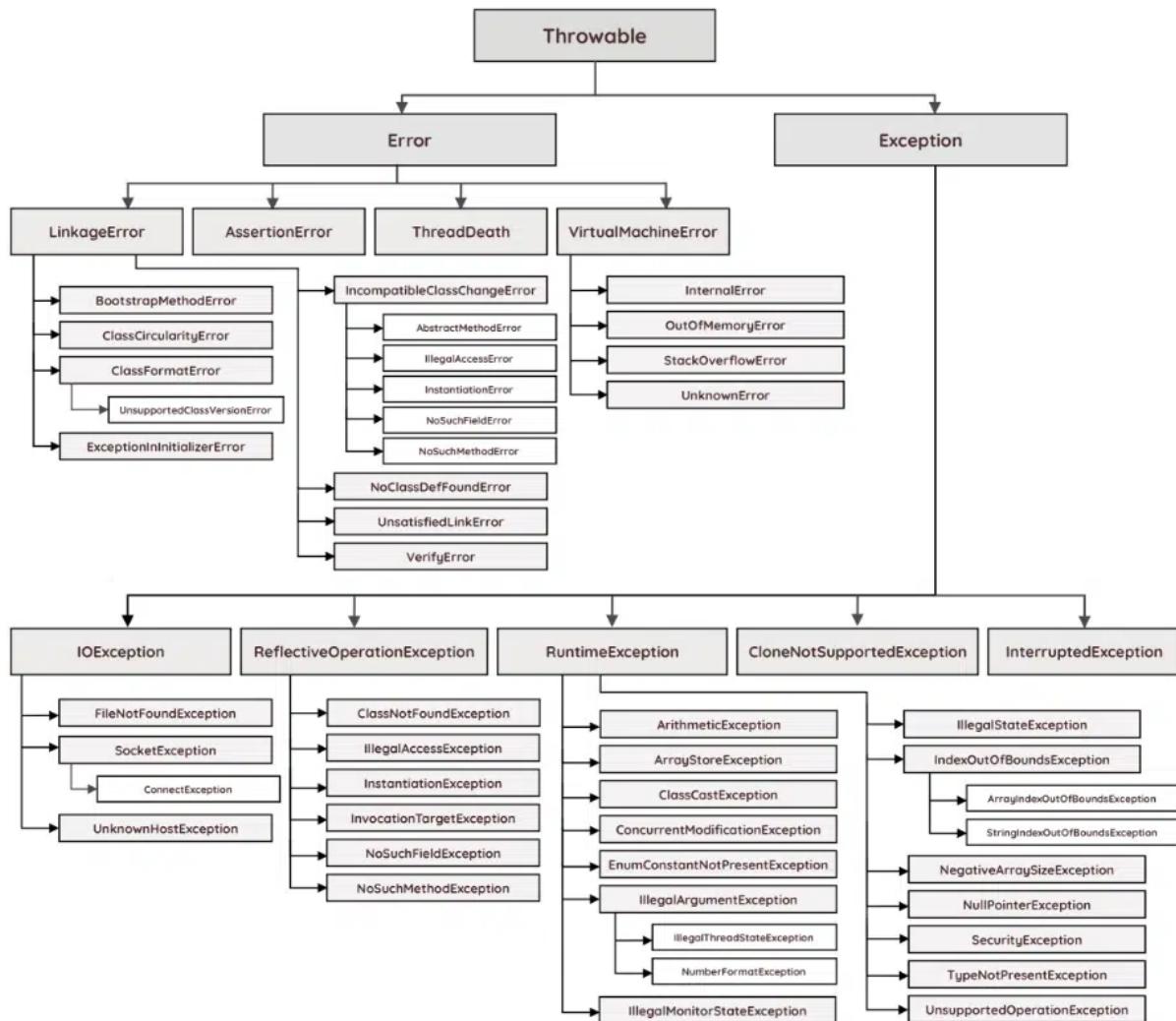
Other exceptions (checked)

Error (unchecked)

Exception Handling Keywords

Java uses the following key constructs for exception handling:

1. **try**: A block of code where exceptions can occur.
 2. **catch**: Used to handle the exception thrown in the try block.
 3. **finally**: Code that always executes after try-catch, used for cleanup.
 4. **throw**: Manually throws an exception.
- throws**: Declares that a method may throw one or more exceptions.



Caught and Uncaught Exceptions

1. Caught Exception: Handled using **try-catch** blocks.
2. The program catches and processes the exception.
3. Uncaught Exception: If an exception is not caught, it propagates up the method call stack and may eventually terminate the program.

Using try-catch-finally

The **try-catch-finally** structure allows you to handle exceptions gracefully and execute cleanup code regardless of whether an exception occurs.

Example:

```
try {  
    // Code that may throw an exception  
} catch (ExceptionType e) {  
    // Handle the exception  
} finally {  
    // Cleanup code that always runs  
}
```

try: The code prone to exceptions.

catch: If an exception occurs, control moves to the catch block.

finally: Always executed, even if an exception is not caught.

The throw Keyword

The **throw** keyword is used to explicitly throw an exception, either predefined or custom.

Example: `throw new ArithmeticException("Division by zero");`

The throws Clause

The **throws** keyword is used in a method signature to declare that the method might throw exceptions.

Example:

```
public void readFile() throws IOException {  
    // Code that might throw an IOException  
}
```

This tells the caller of the method that it should either handle or declare this exception.

Built-in Exceptions

Some common built-in exceptions in Java:

1. **NullPointerException:** Accessing an object with a null reference.
2. **ArrayIndexOutOfBoundsException:** Accessing an array with an invalid index.
3. **ClassCastException:** Invalid casting of objects.

Custom Exceptions

Java allows you to create your own exceptions by extending the `Exception` class, providing more meaningful error messages for specific situations in your application.

Example:

```
class CustomException extends Exception {  
    CustomException(String message) {  
        super(message);  
    }  
}
```

Usage: `throw new CustomException("Custom error message");`

Custom Exception Example

A custom exception example to validate age:

```
class AgeException extends Exception {  
    AgeException(String message) {  
        super(message);  
    }  
  
    public class TestAgeException {  
        static void validate(int age) throws AgeException {  
            if (age < 18) {  
                throw new AgeException("Not eligible for voting");  
            }  
        }  
  
        public static void main(String[] args) {  
            try {  
                validate(16);  
            } catch (AgeException e) {  
                System.out.println("Caught: " + e.getMessage());  
            }  
        }  
    }  
}
```

Best Practices for Exception Handling

1. Catch specific exceptions instead of using a generic **Exception** to make the code more understandable.
2. Avoid using exceptions for control flow as it can degrade performance.
3. Always clean up resources (e.g., file streams, database connections) in the **finally** block.
4. Properly document exceptions that your methods might throw using the **throws** keyword.
5. Use custom exceptions to add more context and clarity to error messages.

Conclusion

1. Exception handling in Java ensures that your programs can handle errors gracefully.
2. Always use the proper handling mechanisms (try-catch-finally, throw, throws) to maintain code reliability.
3. Understand the differences between checked and unchecked exceptions and use custom exceptions when necessary for better error reporting.

LINK:

<https://rollbar.com/blog/java-exceptions-hierarchy-explained/>

<https://www.geeksforgeeks.org/exceptions-in-java/>