In this tutorial, we will learn the fundamentals of exception handling in Java. We will cover the following topics:

1. What is an Exception?
2. Types of Exceptions.
3. Java Exception Hierarchy.
4. Exception Handling Keywords in Java.
5. The try-with-resources Statement.
6. Create Custom Exceptions.
7. Java Chained Exceptions.
8. Advantages of Java Exceptions.
9. Java Exception Handling Best Practices.

1. What is an Exception?

In Java, an exception is an unwanted or unexpected event that occurs during the execution of a program. Exceptions can occur for a variety of reasons – from user input errors to system failures.

The essence of exception handling is to provide a mechanism to handle these unexpected events, allowing the program to either recover from the error or notify the user gracefully.

2. Types of Exceptions

2.1 Checked Exceptions

These are exceptions that a method can throw but must either catch them or declare them using the *`throws`* keyword. They are direct subclasses of the *`Exception`* class, except *`RuntimeException`*.

Java built-in checked exceptions:

- FileNotFoundException
- EOFException
- ClassNotFoundException
- SQLException

- NoSuchMethodException
- InterruptedException

2.2 Unchecked Exceptions (Runtime Exceptions)

These exceptions are not checked at compile-time. They're a direct subclass of the RuntimeException class.

Java built-in unchecked exceptions:

- NullPointerException
- ArrayIndexOutOfBoundsException
- StringIndexOutOfBoundsException
- ArithmeticException
- IllegalArgumentException
- NumberFormatException
- IllegalStateException
- ClassCastException

2.3 Errors

These are exceptional conditions that are external to the application and it cannot anticipate or recover from them.

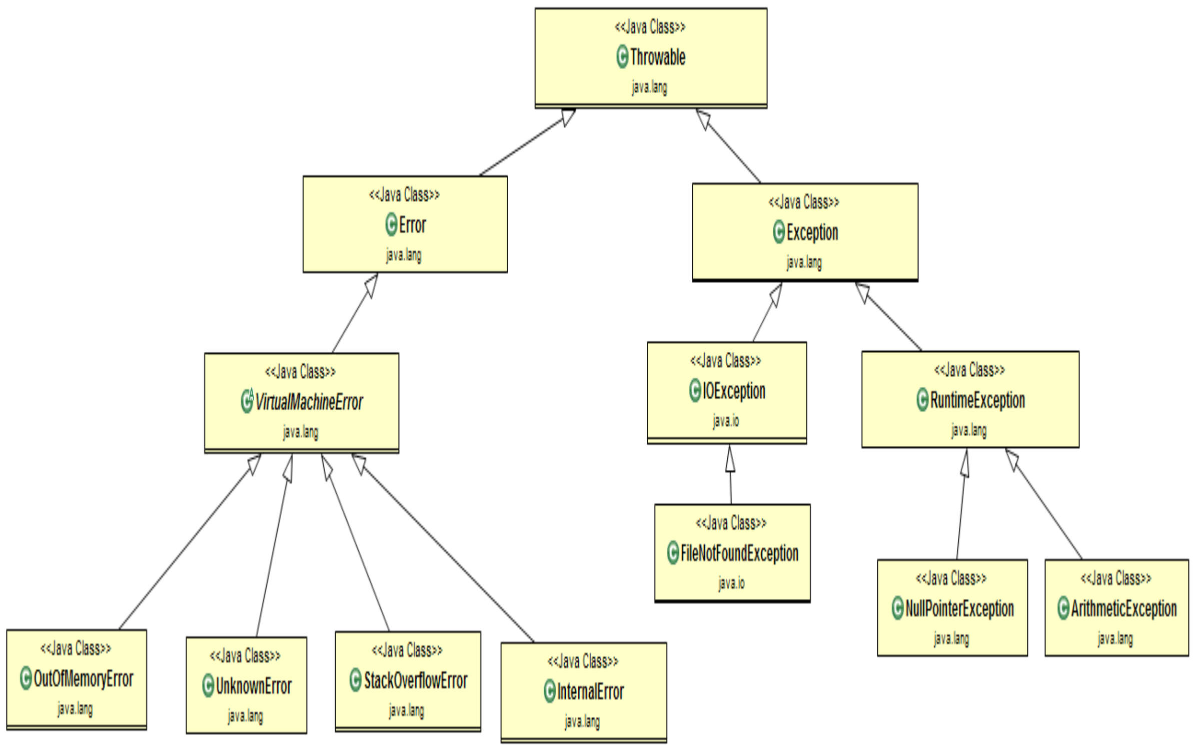
Java built-in errors:

- OutOfMemoryError
- StackOverflowError
- NoClassDefFoundError

3. Java Exception Hierarchy

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. **Throwable** are two subclasses that partition exceptions into two distinct branches that are:

- Error Class
- Exception Class



4. Exception Handling Keywords in Java

Java provides several keywords specifically designed for exception handling. Understanding and using these keywords correctly is crucial for writing robust and error-resilient programs. Here are the primary exception-handling keywords in Java:

try:

This block encloses the code that might throw an exception. Must be followed by either *catch* or *finally* or both.

```
try {
    // code that may cause an exception
}
```

catch:

Follows the *try* block and contains the code that handles the exception. Multiple catch blocks can follow a single try block to handle different types of exceptions separately.

```
catch (ExceptionType e) {
    // handling exception
}
```

finally:

Used to execute important code that must run whether an exception occurs or not. Usually used for cleanup activities, like closing streams or connections.

```
finally {
    // cleanup code
}
```

throw:

Used to manually throw an exception from a method or a code block.

```
if (someCondition) {
    throw new SomeException("Description");
}
```

throws:

Declares the exceptions that a particular method might throw, allowing the calling method to handle or propagate it. Often used for checked exceptions.

```
public void someMethod() throws SomeCheckedException {
    // method body
}
```

Complete Example

Here is an example that demonstrates the usage of *try*, *catch*, *finally*, *throw*, and *throws* keywords:

```
public class ExceptionHandlingKeywords {

    public static void main(String[] args) {
        try {
            int result = divide(10, 0);
            System.out.println(result);
        } catch (ArithmeticException e) {
            System.out.println("Caught exception: " + e.getMessage());
        } finally {
            System.out.println("This will always execute.");
        }
    }

    public static int divide(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Division by zero");
        }
        return a / b;
    }
}
```

Output:

```
Caught exception: Division by zero
This will always execute.
```

5. The try-with-resources Statement

The *try-with-resources* is a syntactic enhancement in exception handling where you can declare resources, like streams, connections, and sockets, directly in the try block. The main advantage is that these resources will be automatically closed after the program is done using them, whether the execution was successful or an exception occurred.

How to Use the try-with-resources Statement:

For a class to be used with *try-with-resources*, it should implement the *AutoCloseable* (or its descendant Closeable) interface, which mandates the *close()* method. Here's the syntax:

```
try (ResourceType resourceName = new ResourceType()) {
    // Use the resource
} catch (ExceptionType e) {
    // Handle exception
}
```

Example:

Consider reading from a file using the **FileInputStream** class:

```
import java.io.FileInputStream;
import java.io.IOException;

public class TryWithResourcesExample {

    public static void main(String[] args) {
        try (FileInputStream input = new FileInputStream("file.txt")) {
            int data = input.read();
            while(data != -1) {
                System.out.print((char) data);
                data = input.read();
            }
        } catch (IOException e) {
            System.out.println("File error: " + e.getMessage());
        }
    }
}
```

In the *try* block's parentheses, the resource (FileInputStream) is instantiated. This resource will be automatically closed when the try block is exited, either normally after reading the file or due to an exception.

6. Create Custom Exceptions

Creating custom exceptions in Java allows you to define specific exception types that can convey more meaningful information about the nature of the problem when compared to standard exceptions. These custom exceptions can also encapsulate additional data or functionality related to the exceptional circumstance.

Steps to Create a Custom Exception:

Define a New Exception Class: You typically extend the *Exception* class for checked exceptions or the *RuntimeException* class for unchecked exceptions.

Constructors: Provide a constructor that accepts a message as its argument. You can also override other constructors based on your needs.

Add Custom Fields and Methods (Optional): You can add additional fields and methods that provide more context or functionality for your exception.

Example: Here's how you can define a simple custom-checked exception:

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

For a custom unchecked exception:

```
public class CustomUncheckedException extends RuntimeException {
    public CustomUncheckedException(String message) {
        super(message);
    }
}
```

You can even add more details to your exception:

```
public class DetailedException extends Exception {
    private int errorCode;

    public DetailedException(String message, int errorCode) {
        super(message);
        this.errorCode = errorCode;
    }

    public int getErrorCode() {
        return errorCode;
    }
}
```

How to Use Custom Exceptions:

You can throw the custom exception like any other exception:

```
public void customCheckedExceptionMethod() throws CustomCheckedException {
    if (someCondition) {
        throw new CustomCheckedException("An error occurred.");
    }
}

public void customUncheckedExceptionMethod() {
    if (anotherCondition) {
        throw new CustomUncheckedException("A runtime error occurred.");
    }
}
```

Benefits of Custom Exceptions:

Expressiveness: They convey specific error conditions more clearly.

Flexibility: You can add extra fields or methods to encapsulate more information.

Maintainability: With specific exceptions for specific problems, debugging and maintenance become more straightforward.

7. Java Chained Exceptions

Chained exceptions, also known as "exception wrapping" or "exception nesting," allow developers in Java to handle a new exception that arises while processing another exception. By chaining exceptions, you can capture both the original exception and the secondary exception, providing a more detailed context when diagnosing issues.

Why Use Chained Exceptions?

Preserve Original Exception: It allows you to preserve the original exception even if you want to throw a new type of exception.

Provide More Context: It gives more context to what went wrong. For example, if a higher-level exception is thrown because of a lower-level exception, both can be captured.

Improved Debugging: The stack trace of the chained exception provides a more in-depth view of the problem, showing both exceptions in the trace.

Chained Exceptions Example

The `Throwable` class, which is the superclass of all exceptions and errors in Java, provides constructors that can take another `Throwable` as a parameter for chaining.

```java
public class ChainedExceptionDemo {
    public static void main(String[] args) {
        try {
            method1();
        } catch (HighLevelException e) {
            e.printStackTrace();
        }
    }
    static void method1() throws HighLevelException {
        try {
            method2();
        } catch (LowLevelException lle) {
            throw new HighLevelException("Method1 faced an issue.", lle);
        }
    }

    static void method2() throws LowLevelException {
        throw new LowLevelException("A low-level exception occurred.");
    }
}

class HighLevelException extends Exception {
    public HighLevelException(String message, Throwable cause) {
        super(message, cause);
    }
}

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

In the example above, *`method2`* throws a *`LowLevelException`*. This exception is caught in *`method1`*, and a new *`HighLevelException`* is thrown, wrapping the original *`LowLevelException`*. When catching the *`HighLevelException`* in the main method and printing its stack trace, both exceptions' messages and stack traces will be displayed.

Output:

```
com.javaguides.net.HighLevelException: Method1 faced an issue.
        at
com.javaguides.net.ChainedExceptionDemo.method1(ChainedExceptionDemo.java:16)
        at com.javaguides.net.ChainedExceptionDemo.main(ChainedExceptionDemo.java:6)
Caused by: com.javaguides.net.LowLevelException: A low-level exception occurred.
        at
com.javaguides.net.ChainedExceptionDemo.method2(ChainedExceptionDemo.java:21)
        at
com.javaguides.net.ChainedExceptionDemo.method1(ChainedExceptionDemo.java:14)
        ... 1 more
```

8. Advantages of Java Exception Handling

Here are the primary advantages of using Java exceptions:

Separation of Error Handling from Regular Code: Exception handling in Java separates error-handling code from regular code.

Program Robustness: Java's exception mechanism ensures that a program can gracefully handle unexpected situations, making the program more robust.

Hierarchical Exception Classes: Java exceptions are objects and inherit from the Throwable class. This hierarchy allows developers to design high-level exception handling that can catch various related exceptions in a single catch block.

Checked vs. Unchecked Exceptions: Java distinguishes between checked and unchecked exceptions. Checked exceptions must be either caught or declared in the method signature, ensuring that developers handle or propagate potential error conditions. Unchecked exceptions (which inherit from RuntimeException) are typically programming errors and do not have this requirement.

Descriptive Error Messages: Exception objects can carry detailed error messages, providing context about what went wrong, which aids in debugging.

Chained Exceptions: Java allows exceptions to be chained, meaning that one exception can be the cause of another. This feature provides a deeper context when trying to understand the root cause of an error.

Standardized Handling: Java provides a standardized method of handling errors via the try-catch-finally blocks. This consistent approach means that developers can quickly understand and manage error-handling scenarios across different codebases.

Resource Management: With the introduction of the try-with-resources statement in Java 7, resource management has been significantly improved. It ensures that resources, such as files or network connections, are closed automatically, reducing the risk of resource leaks.

Propagation: If a method does not want to handle an exception, it can propagate it upwards in the call stack, allowing a higher-level method to handle it.

Custom Exceptions: Java allows developers to define custom exception classes, enabling them to create specific error types for their applications. This specificity can make error diagnosis and recovery more straightforward.

9. Java Exception Handling Best Practices

Here are some best practices to follow when dealing with exceptions in Java:

Be Specific When Catching Exceptions:

Instead of catching the general `Exception` class, catch the most specific exception that you expect. It makes the error handling more targeted and avoids unintentionally catching and masking other unexpected issues.

```
try {
    // ...
} catch (FileNotFoundException e) {
    // Handle this specific exception
}
```

Avoid Empty Catch Blocks:

Never leave a catch block empty as it swallows the exception, making issues hard to trace.

```
catch (SomeException e) {
    // Don't do this
}
```

Always Log Exceptions:

Even if you think an exception is minor, always log it. It will save you a lot of time during debugging.

```
catch (SomeException e) {
    logger.error("An error occurred: ", e);
}
```

Use Checked Exceptions for Recoverable Conditions:

If a client can reasonably be expected to recover from an exception, make it a checked exception.

Use Runtime Exceptions for Programming Errors:

Use unchecked exceptions (subtypes of RuntimeException) for programming errors, such as null pointers or illegal arguments.

Avoid Overusing Checked Exceptions:

If a method throws too many checked exceptions, it can become cumbersome to use and can reduce code readability.

Document Exceptions Using JavaDoc:

If your method throws an exception, use the *`@throws`* or *`@exception`* tags in its JavaDoc to describe the conditions under which it is thrown.

Clean Up Resources in a Finally Block:

Ensure resources like streams, connections, and files are closed, preferably in a *`finally`* block or using *`try-with-resources`*.

```
try {
    // Use resources
} finally {
    // Close resources
}
```

Or using try-with-resources:

```
try (ResourceType resource = new ResourceType()) {
    // Use the resource
}
```

Avoid Throwing Raw Exception Types:

Instead of throwing Exception or RuntimeException, use a more specific exception type or create a custom one if needed.

Use Meaningful Messages:

When throwing exceptions, always provide a meaningful message that gives context about the problem.

```
throw new IllegalArgumentException("Parameter x must be > 0");
```