Exception Handling in Java

Exception Handling in Java is a powerful mechanism that handles runtime errors, preventing program termination and allowing the application to continue its normal flow. When an error occurs, an exception object is created, and Java uses this object to represent the error.

1. What are Exceptions?

An exception is an event that disrupts the normal flow of the program. It occurs during the execution of a program and can be caused by a variety of factors such as invalid user input, loss of a network connection, or hardware failure.

Terminology:

Exception: An abnormal condition that occurs during runtime, disrupting the normal flow of the application.

Exception Object: An object that contains information about the error, such as its type and the state of the program when the error occurred.

2. Types of Exceptions

Java exceptions are divided into three broad categories:

a. Checked Exceptions (Compile-time exceptions)

Checked exceptions are exceptions that are checked at compile-time. They must be handled by the programmer, either by using a try-catch block or by declaring them using the throws keyword.

These are exceptions that are foreseeable and usually caused by external factors like file operations, database connectivity, etc.

Examples of Checked Exceptions:

IOException

SQLException

ClassNotFoundException

Example:

```
import java.io.*;
```

```
public class Main {
    public static void main(String[] args) {
        try {
            FileInputStream file = new FileInputStream("file.txt");
        } catch (FileNotFoundException e) {
            System.out.println("File not found!");
        }
    }
}
```

b. Unchecked Exceptions (Runtime exceptions)

Unchecked exceptions occur at runtime and are not checked at compile-time. These exceptions result from programming errors, such as logic errors or improper use of an API.

They inherit from the RuntimeException class.

Examples of Unchecked Exceptions:

ArithmeticException

NullPointerException

ArrayIndexOutOfBoundsException

Example:

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

c. Errors

Errors are not exceptions but serious problems that arise due to issues beyond the control of the application. Errors typically represent system-level problems like OutOfMemoryError, StackOverflowError, or VirtualMachineError.

Errors should not be caught or handled in a normal program, as they indicate critical failures that the application cannot recover from.

Example of Error:

```
public class Main {
    public static void main(String[] args) {
        // Recursive call without base condition
        // Causes StackOverflowError
        main(args);
    }
}
```

3. Exception Hierarchy

In Java, exceptions are represented by a hierarchy of classes in the java.lang package. The Throwable class is the root of this hierarchy.

Throwable: Root class for all exception types.

Error: Represents serious system-level problems (e.g., OutOfMemoryError).

Exception: Represents general exceptions that applications might want to catch.

Checked Exceptions: Subclasses of Exception (except RuntimeException).

RuntimeException: Subclasses of RuntimeException (unchecked).

Exception Hierarchy Diagram:

```php
Throwable
  /   \
 Error  Exception
         /      \
  Checked     RuntimeException
```

4. Handling Exceptions

Java provides three main keywords for handling exceptions: try, catch, and finally.

a. try-catch Block

The try block contains code that may throw an exception, while the catch block handles that exception. If an exception occurs, the catch block is executed. If no exception occurs, the catch block is skipped.

Syntax:

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Handle the exception
}
```

Example:

```
public class Main {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};
            System.out.println(arr[5]);  // Throws ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of bounds!");
        }
    }
}
```

b. finally Block

The finally block is optional and always executes, regardless of whether an exception occurs or not. It's commonly used to close resources like file streams or database connections.

Syntax:

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

Example:

```
public class Main {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};
            System.out.println(arr[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of bounds!");
        } finally {
            System.out.println("This block always executes.");
        }
    }
}
```

c. throw Keyword

The throw keyword is used to explicitly throw an exception.

Syntax:

```
throw new ExceptionType("Error message”);
```

Example:

```
public class Main {
    public static void checkAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Not eligible to vote.");
        } else {
            System.out.println("Eligible to vote.");
        }
    }

    public static void main(String[] args) {
        checkAge(15);  // Throws ArithmeticException
    }
}
```

d. throws Keyword

The throws keyword is used in method signatures to declare exceptions that can be thrown by the method.

Syntax:

```
void method() throws ExceptionType {
    // Code that may throw an exception
}
```

Example:

```
import java.io.IOException;

public class Main {
    public static void checkFile() throws IOException {
        throw new IOException("File error occurred.");
    }

    public static void main(String[] args) {
        try {
            checkFile();
        } catch (IOException e) {
            System.out.println("Caught IOException: " + e.getMessage());
        }
    }
}
```

5. Multiple Catch Blocks

You can catch multiple exceptions using multiple catch blocks. The most specific exception should be caught first, followed by more general exceptions.

Example:

```
public class Main {
    public static void main(String[] args) {
        try {
            int a = 10;
            int b = 0;
            int result = a / b;  // ArithmeticException
            int[] arr = new int[5];
            arr[10] = 50;        // ArrayIndexOutOfBoundsException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero.");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of bounds.");
        }
    }
}
```

6. Nested Try-Catch Blocks

Try-catch blocks can be nested, meaning one try block can contain another try-catch inside it.

```
public class Main {
    public static void main(String[] args) {
        try {
            try {
                int a = 10;
                int b = 0;
                int result = a / b;
            } catch (ArithmeticException e) {
                System.out.println("Inner catch: Cannot divide by zero.");
            }

            int[] arr = new int[5];
            arr[10] = 50;
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Outer catch: Array index out of bounds.");
        }
    }
}
```

7. Custom Exceptions

Java allows you to create your own exceptions by extending the Exception class or any of its subclasses.

Example:

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

public class Main {
    public static void checkAge(int age) throws AgeException {
        if (age < 18) {
            throw new AgeException("Age is less than 18.");
        } else {
            System.out.println("Age is valid.");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(16);
        } catch (AgeException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Summary of Exception Handling:

Checked exceptions: Must be handled or declared in the method signature.

Unchecked exceptions: Occur at runtime and are not checked at compile time.

Errors: Represent critical system issues that cannot be recovered from.

Handling exceptions: Use try-catch blocks to handle exceptions and ensure the program doesn't crash.

Custom exceptions: You can create your own exceptions by extending the Exception class.