1. Checked Exceptions

Checked exceptions are those that the Java compiler checks at compile-time. These exceptions need to be either handled using a try-catch block or declared in the method signature using the `throws` keyword.

Example of a **checked exception** (e.g., `FileNotFoundException`):

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

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            FileInputStream file = new FileInputStream("file.txt");
            int data;
            while ((data = file.read()) != -1) {
                System.out.print((char) data);
            }
            file.close();
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("I/O error occurred: " + e.getMessage());
        }
    }
}
```

In this example:

- The `FileNotFoundException` is checked at compile-time. If the file "file.txt" doesn't exist, a `FileNotFoundException` will be thrown.
- The `IOException` is caught if an I/O error occurs while reading the file.

2. Unchecked Exceptions

Unchecked exceptions are those that are not checked at compile-time. They are subclasses of `RuntimeException`. These usually indicate programming bugs, such as logic errors or

improper use of an API.

Example of an **unchecked exception** (e.g., `ArrayIndexOutOfBoundsException`):

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};

        try {
            System.out.println(array[10]); // Attempting to access an invalid index
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index is out of bounds: " + e.getMessage());
        }
    }
}
```

In this example:

- The program tries to access an element at index 10, which is out of bounds.
- Since this is a programming error, `ArrayIndexOutOfBoundsException`, an unchecked exception, is thrown and caught in the catch block.

3. Using the `throws` Keyword

The `throws` keyword in Java is used in method declarations to indicate that the method may throw certain exceptions. This helps in propagating checked exceptions to the calling method.

Example:

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

public class ThrowsExample {
    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }

    public static void readFile() throws IOException {
        FileInputStream file = new FileInputStream("file.txt");
        int data;
        while ((data = file.read()) != -1) {
            System.out.print((char) data);
        }
        file.close();
    }
}
```

In this example:

- The `readFile` method declares that it throws `IOException`.
- The `main` method calls `readFile` and handles the exception using a try-catch block.

4. Difference Between `throw` and `throws`

- `throws` is used in the method signature to declare exceptions.
- `throw` is used to explicitly throw an exception within a method.

Example of `throw`:

```java
public class ThrowExample {
    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (IllegalArgumentException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }

    public static void validateAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or older to vote.");
        }
        System.out.println("Eligible to vote");
    }
}
```

In this example:

- The `validateAge` method throws an `IllegalArgumentException` if the age is below 18.
- This shows how `throw` is used to manually throw an exception based on a condition.

Here's an enhanced explanation of the `throw` and `throws` keywords, along with examples of the `try-catch-finally` block based on the slides you've provided:

1. Using the `throw` Keyword

The `throw` keyword in Java is used to explicitly throw an exception in a program. It can throw only one exception at a time and is typically used for custom exceptions or specific error messages.

Example using `throw`:

```
public class ThrowExample {
    public static void main(String[] args) {
        try {
            validate(10);  // Passing an invalid value
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    public static void validate(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or older.");
        }
        System.out.println("Eligible to vote");
    }
}
```

In this example:

- The `validate` method uses `throw` to manually throw an `IllegalArgumentException` with a custom message if the `age` is below 18.
- The `catch` block in `main` catches the exception and displays the error message.

2. Using the `throws` Keyword

The `throws` keyword is used in a method signature to declare the exceptions that a method might throw, thereby allowing the caller to handle or propagate them. Unlike `throw`, `throws` can declare multiple exceptions.

Example using `throws`:

```java
import java.io.IOException;

public class ThrowsExample {
    public static void main(String[] args) {
        try {
            processFile();  // Method that may throw an exception
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    public static void processFile() throws IOException {
        throw new IOException("File processing error.");
    }
}
```

In this example:

- The `processFile` method declares `throws IOException`, meaning it might throw an `IOException`.
- The `main` method handles this by catching the `IOException` and printing the error message.

3. Difference Between `throw` and `throws`

- `throw` is used within a method to actually throw an instance of an exception.
- `throws` is used in the method declaration to indicate what exceptions might be thrown by the method.

Example to illustrate:

```java
public void exampleMethod() throws IOException, ArithmeticException {
    // Declare multiple exceptions
    throw new IOException("IO error occurred");
}
```

In the above code:

- `throws` is used in the method signature to declare multiple exceptions (e.g., `IOException`, `ArithmeticException`).
- `throw` can only throw one exception at a time.

4. Using `try-catch-finally` Block

The `try-catch-finally` block is a way to handle exceptions in Java:

- `try` contains code that might throw an exception.
- `catch` handles specific exceptions.
- `finally` is always executed, regardless of whether an exception was thrown or not.

Example:

```
public class TryCatchFinallyExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            System.out.println("Execution completed.");
        }
    }

    public static int divide(int a, int b) {
        return a / b;
    }
}
```

In this example:

- The `divide` method performs a division that might throw an `ArithmeticException` (division by zero).
- The `catch` block catches and handles this exception.
- The `finally` block executes regardless, printing "Execution completed."

5. Understanding `finally` Block Execution

The `finally` block is executed:

- Whether or not an exception occurs.
- If there is a `return` statement within the `try` or `catch`.

However, if the program exits (using `System.exit()`), the `finally` block will not execute.

Example to demonstrate:

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            System.out.println("Inside try block");
            return;
        } catch (Exception e) {
            System.out.println("Inside catch block");
        } finally {
            System.out.println("Inside finally block");
        }
    }
}
```

In this example:

- The `finally` block will execute even though there is a `return` in the `try` block.
- Output will be:

  ```
  Inside try block
  Inside finally block
  ```

Here's a Java program that demonstrates error handling for each of the exceptions listed in your labsheet: `ArithmeticException`, `ArrayIndexOutOfBoundsException`, and `NumberFormatException`.

```
public class ExceptionHandlingDemo {

    public static void main(String[] args) {
        // Example for ArithmeticException
        try {
            int result = 10 / 0; // This will throw an ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught: " + e.getMessage());
        }

        // Example for ArrayIndexOutOfBoundsException
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]); // This will throw ArrayIndexOutOfBoundsExcept
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBoundsException caught: " + e.getMessage())
        }

        // Example for NumberFormatException
        try {
            int num = Integer.parseInt("ABC"); // This will throw NumberFormatException
        } catch (NumberFormatException e) {
            System.out.println("NumberFormatException caught: " + e.getMessage());
        }
    }
}
```

Explanation of Each Example:

1. **ArithmeticException**:
    - This exception occurs when there is an illegal arithmetic operation, such as division by zero.
    - In this example, `10 / 0` will cause an `ArithmeticException`, which is caught in the `catch` block.
2. **ArrayIndexOutOfBoundsException**:
    - This exception occurs when you try to access an index in an array that is outside the array's bounds.
    - Here, trying to access `numbers[5]` when the array only has indices 0

to 2 throws an `ArrayIndexOutOfBoundsException`.

3. **NumberFormatException**:
    - This exception occurs when you try to convert a string to a numeric type (like `int`) and the string doesn't have an appropriate format.
    - Trying to parse `"ABC"` as an integer with `Integer.parseInt("ABC")` results in a `NumberFormatException`.

This program demonstrates handling these specific exceptions and prints a relevant message for each when the exception is caught.