What is an Exception?

An exception is an unexpected event that interrupts the normal flow of a program.

Why Do We Need to Handle Exceptions?

- To **prevent the program from crashing** during errors.
- To ensure the program can handle issues **gracefully** (e.g., showing user-friendly messages).
- To identify and fix **bugs or predictable failures** effectively.
- Helps in maintaining **stability** and a good **user experience** in applications.

Summary: Handling exceptions makes programs reliable and ensures they can deal with errors without breaking.

Learning for Checked Exceptions

1. **What are Checked Exceptions?**
   - Checked exceptions happen in **predictable situations** where something might fail.
   - They **must be handled** (using `try-catch`) or declared with `throws`.
2. **Why They Occur:**
   - Examples include:
     - `IOException`: Issues in file operations (e.g., file not found).
     - `SQLException`: Problems during database queries.
3. **Key Points to Remember:**
   - **Mandatory to Handle:** The compiler forces you to handle checked exceptions to ensure proper error management.
   - **Extends Exception:** ***All checked exceptions inherit from the `Exception` class.***
4. **How to Manage Checked Exceptions:**
   - **Use try-catch:**
     Example:

```
try {
    FileReader reader = new FileReader("file.txt"); // File might
not exist
} catch (IOException e) {
    System.out.println("File not found! Check the file path.");
}
```

- **Declare with `throws`:** Let the calling method handle the exception.
  Example:

  ```
  public void readFile() throws IOException {
      FileReader reader = new FileReader("file.txt");
  }
  ```

5. **Real-Life Example:**

   - In a banking app, reading customer details from a file might throw an `IOException`. Instead of crashing, you handle it to show "Unable to fetch details. Try again later."

Summary:
Checked exceptions occur in predictable cases like file or database operations. The compiler forces handling them to ensure your program runs smoothly even when errors happen.

Learning for Unchecked Exceptions

1. **What are Unchecked Exceptions?**
   - Unchecked exceptions happen due to **coding mistakes or bugs**.
   - They are **not mandatory** to handle (no `throws` needed).
2. **Why They Occur:**
   - Examples include:
     - `NullPointerException`: Using `null` where an object is expected.
     - `ArrayIndexOutOfBoundsException`: Accessing an invalid array index.
     - `ArithmeticException`: Dividing by zero.
3. **Key Points to Remember:**
   - **Optional to Handle:** You don’t have to handle unchecked exceptions, but it’s good to prevent or manage them in critical cases.
   - ***RuntimeException Class: All unchecked exceptions are part of the `RuntimeException` class.***
4. **How to Manage Unchecked Exceptions:**
   - **Prevent them:** Write clean code (e.g., check for `null` before using objects).
   - **Use try-catch (when needed):**
     Example:

     ```
     try {
         int[] numbers = {1, 2, 3};
     ```

```
    System.out.println(numbers[5]); // Invalid index
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Invalid index! Check your array size.");
}
```

- **Global Handling (for apps):** Use frameworks or custom handlers to log and manage errors gracefully.

5. **Real-Life Example:**

    - In a shopping app, if a user enters an invalid product ID, it might cause a `NullPointerException`. Instead of crashing, the app can show a "Product Not Found" message.

Summary:
Unchecked exceptions are bugs that you can avoid with good coding practices. Handle them only when they can affect user experience or system stability.