Exception Handling in Java and Selenium - Complete Notes

# 1. What is Exception Handling?

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

* Exceptions are abnormal conditions that disrupt program execution.
* They are objects representing errors.

# 2. Types of Exceptions

Java exceptions fall into two main categories:

## 2.1 Checked Exceptions

* Checked at compile time.
* Handled using try-catch or declared using throws.

## 2.2 Unchecked Exceptions

* Checked at runtime.
* Include programming bugs like logic errors or improper API usage.

# 3. Java Exception Class Hierarchy

* Throwable
* ├── Exception
* │ ├── IOException, SQLException, etc.
* ├── RuntimeException
* │ ├── NullPointerException, IndexOutOfBoundsException
* └── Error (not meant to be caught)

# 4. try-catch-finally Structure

The try-catch-finally block is the standard structure to handle exceptions.

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

# 5. throws vs throw

## 5.1 throw keyword

Used to explicitly throw an exception.

throw new IllegalArgumentException("Invalid input");

## 5.2 throws keyword

Used in method signature to declare exceptions.

public void readFile() throws IOException {  
 // Code that may throw IOException  
}

# 6. Common Java Exceptions

* NullPointerException
* ArrayIndexOutOfBoundsException
* ClassNotFoundException
* ArithmeticException
* IllegalArgumentException
* FileNotFoundException

# 7. Exception Handling in Selenium

Selenium can throw various exceptions related to element handling, waits, or browser actions.

## 7.1 Common Selenium Exceptions

* NoSuchElementException
* TimeoutException
* ElementNotInteractableException
* StaleElementReferenceException
* WebDriverException
* InvalidSelectorException

## 7.2 Example of Handling WebDriver Exception

try {  
 WebElement element = driver.findElement(By.id("submit"));  
 element.click();  
} catch (NoSuchElementException e) {  
 System.out.println("Element not found: " + e.getMessage());  
}

# 8. Best Practices for Exception Handling

* Catch only specific exceptions.
* Avoid empty catch blocks.
* Use finally block to release resources.
* Log exceptions using a logging framework.
* Use custom exceptions for domain-specific errors.

# 9. Creating Custom Exceptions

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

# 10. Real Selenium Use Case Example

public void loginTest() {  
 try {  
 driver.findElement(By.id("username")).sendKeys("admin");  
 driver.findElement(By.id("password")).sendKeys("admin123");  
 driver.findElement(By.id("loginBtn")).click();  
 } catch (NoSuchElementException e) {  
 System.out.println("Login field missing: " + e.getMessage());  
 } catch (Exception e) {  
 System.out.println("Unexpected error: " + e.getMessage());  
 }  
}

# 11. Summary Table

|  |  |  |
| --- | --- | --- |
| Concept | Description | Example |
| try-catch | Handle exception with recovery | try { ... } catch (Exception e) { ... } |
| finally | Always runs after try/catch | Close DB/file connections |
| throw | Throw specific exception | throw new Exception("error") |
| throws | Declare exceptions | void method() throws IOException |
| Custom Exception | User-defined error class | class MyException extends Exception |

# 12. Final Notes

*Exception handling is essential for building robust applications. In Selenium, handling specific exceptions ensures stability in dynamic and unpredictable environments.*