# **Exceptions in Java**

An exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception -The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time
2. Unchecked Exception -The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.
3. Error- Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

## a. There are 5 keywords which are used in handling exceptions in Java.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature. |

## b. The following are the important differences between throw and throws.

| **#** | **Key** | **Throw** | **throws** |
| --- | --- | --- | --- |
| 1 | Definition | Throw is a keyword which is used to throw an exception explicitly in the program inside a function or inside a block of code. | Throws is a keyword used in the method signature used to declare an exception which might get thrown by the function while executing the code. |
| 2 | Internal implementation | Internally throw is implemented as it is allowed to throw only single exception at a time i.e we cannot throw multiple exception with throw keyword. | On other hand we can declare multiple exceptions with throws keyword that could get thrown by the function where throws keyword is used. |
| 3 | Type of exception | With throw keyword we can propagate only unchecked exception i.e checked exception cannot be propagated using throw. | On other hand with throws keyword both checked and unchecked exceptions can be declared and for the propagation checked exception must use throws keyword followed by specific exception class name. |
| 4 | Syntax | Syntax wise throw keyword is followed by the instance variable. | On other hand syntax wise throws keyword is followed by exception class names. |
| 5 | Declaration | In order to use throw keyword we should know that throw keyword is used within the method. | On other hand throws keyword is used with the method signature. |

Example of throw:

public class JavaTester{

public void checkAge(int age){

if(age<18)

throw new ArithmeticException("Not Eligible for voting");

else

System.out.println("Eligible for voting");

}

public static void main(String args[]){

JavaTester obj = new JavaTester();

obj.checkAge(13);

System.out.println("End Of Program");

}

}

Output:

Exception in thread "main" java.lang.ArithmeticException:

Not Eligible for voting

at JavaTester.checkAge(JavaTester.java:4)

at JavaTester.main(JavaTester.java:10)

Example of throws:

public class JavaTester{

public int division(int a, int b) throws ArithmeticException{

int t = a/b;

return t;

}

public static void main(String args[]){

JavaTester obj = new JavaTester();

try{

System.out.println(obj.division(15,0));

}

catch(ArithmeticException e){

System.out.println("You shouldn't divide number by zero");

}

}

}

Output:

You shouldn't divide number by zero

# **Java Custom Exception**

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need. By the help of custom exception, you can have your own exception and message. Let's see a simple example of java custom exception.

class InvalidAgeException extends Exception{

InvalidAgeException(String s){

super(s);

}

}

class TestCustomException1{

static void validate(int age)throws InvalidAgeException{

if(age<18)

throw new InvalidAgeException("not valid");

else

System.out.println("welcome to vote");

}

public static void main(String args[]){

try{

validate(13);

}catch(Exception m){System.out.println("Exception occured: "+m);}

System.out.println("rest of the code...");

} }

Output: Exception occured: InvalidAgeException:not valid

rest of the code...

## a. Practices when using exceptions in your automation code

Exception handling in Java isn’t an easy topic. There are several best practices that are used by most teams. Here are the few most important ones that help you get started or improve your exception handling.

1. Catch the Most Specific Exception First:

Always catch the most specific exception class first and add the less specific catch blocks to the end of your list. You can see an example of such a try-catch statement in the following code snippet. The first catch block handles all NumberFormatExceptions and the second one all IllegalArgumentExceptions which are not a NumberFormatException. The problem is that only the first catch block that matches the exception gets executed.

So, if you catch an IllegalArgumentException first, you will never reach the catch block that should handle the more specific NumberFormatException because it’s a subclass of the IllegalArgumentException.

public void catchMostSpecificExceptionFirst() {

try {

doSomething("A message");

} catch (NumberFormatException e) {

log.error(e);

} catch (IllegalArgumentException e) {

log.error(e)

}

}

2. Don’t Ignore Exceptions

The printStackTrace() method in Java is a tool used to handle exceptions and errors. It is a method of Java’s throwable class which prints the throwable along with other details like the line number and class name where the exception occurred. The printStackTrace() is very useful in diagnosing exceptions. For example, if one out of five methods in your code cause an exception, printStackTrace() will pinpoint the exact line in which the method raised the exception

class Program {

public static void foo() {

try {

int num1 = 5/0;

}

catch (Throwable e) {

e.printStackTrace();

}

}

public static void main( String args[] ) {

foo();

}

}

Output: java.lang.ArithmeticException: / by zero at Program.foo(main.java:4) at Program.main(main.java:12)

As seen in the output above, the entire stack trace is printed along with line numbers and class names to pinpoint the exact location of the exception. Note that the top-most function in the stack trace is the one that was executed last, hence, that is the function where the exception occurred.

[Avoid printStackTrace(); use a logger call instead](https://stackoverflow.com/questions/10477607/avoid-printstacktrace-use-a-logger-call-instead), It means you should use logging framework like [logback](https://stackoverflow.com/questions/tagged/logback" \o "show questions tagged 'logback') or [log4j](https://stackoverflow.com/questions/tagged/log4j) and instead of printing exceptions directly. you should log them using this frameworks' API: Logging frameworks give you a lot of flexibility, e.g. you can choose whether you want to log to console or file - or maybe skip some messages if you find them no longer relevant in some environment.

If you call printStackTrace() on an exception the trace is written to System.err and it's hard to route it elsewhere (or filter it). Instead of doing this you are advised to use a logging framework (or a wrapper around multiple logging frameworks, like Apache Commons Logging) and log the exception using that framework (e.g. logger.error("some exception message", e)).

Doing that allows you to:

1. write the log statement to different locations at once, e.g. the console and a file
2. filter the log statements by severity (error, warning, info, debug etc.) and origin (normally package or class based)
3. have some influence on the log format without having to change the code

public void logAnException() {

try {

// do something

} catch (NumberFormatException e) {

log.error("This should never happen: " + e);

}

}

3. Clean Up Resources in a Finally Block or Use a Try-With-Resource Statement

It happens quite often that you use a resource in your try block, like an [InputStream](https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html" \t "_blank), which you need to close afterward. A common mistake in these situations is to close the resource at the end of the try block.

public void doNotCloseResourceInTry() {

FileInputStream inputStream = null;

try {

File file = new File("./tmp.txt");

inputStream = new FileInputStream(file);

// use the inputStream to read a file

// do NOT do this

inputStream.close();

} catch (FileNotFoundException e) {

log.error(e);

} catch (IOException e) {

log.error(e);

}

}

The problem is that this approach seems to work perfectly fine as long as no exception gets thrown. All statements within the try block will get executed, and the resource gets closed. But you added the try block for a reason. You call one or more methods which might throw an exception, or maybe you throw the exception yourself. That means you might not reach the end of the try block. And as a result, you will not close the resources.

You should, therefore, put all your clean up code into the finally block or use a try-with-resource statement.

public void closeResourceInFinally() {

FileInputStream inputStream = null;

try {

File file = new File("./tmp.txt");

inputStream = new FileInputStream(file);

// use the inputStream to read a file

} catch (FileNotFoundException e) {

log.error(e);

} finally {

if (inputStream != null) {

try {

inputStream.close();

} catch (IOException e) {

log.error(e);

}

}

}

}

# **Common Exceptions in Selenium WebDriver**

Selenium has its own set of exceptions. While developing selenium scripts, a programmer has to handle or throw those exceptions. All runtime exception classes in Selenium WebDriver come under the superclass WebDriverException.

Though there are many Exception classes under WebDriverException, we commonly see the below ones.

|  |  |
| --- | --- |
| **Selenium Exceptions** | **Description** |
| ElementNotVisibleException | If selenium tries to find an element but the element is not visible within the page |
| NoAlertPresentException | If a user tries to handle an alert box but the alert is not present. |
| NoSuchAttributeException | While trying to get attribute value but the attribute is not available in DOM |
| NoSuchElementException | The exception occurs when WebDriver is unable to find and locate elements. |
| WebDriverException | Exception comes when a code is unable to initialize WebDriver |
| NoSuchWindowException | This is thrown when WebDriver tries to switch to an invalid window |
| NoSuchFrameException | This exception is thrown, when WebDriver is trying to switch to an invalid frame |
| ElementNotSelectableException | This exception indicates that the web element is present in the web page but cannot be selected. |
| TimeoutException | This exception occurs when a command completion takes more than the wait time |
| NoSuchSessionException | This exception is thrown when a method is called after quitting the browser by WebDriver.quit() |
| StaleElementReferenceException | This exception says that a web element is no longer present in the web page |

## a. How to handle various selenium exceptions

Let’s discuss Avoid-And-Handle approach for the selenium exceptions:

### *ElementNotVisibleException:*

While running Selenium scripts, sometimes we face below exception

“org.openqa.selenium.ElementNotVisibleException: Element is not currently visible and so may not be interacted with Command duration or timeout

Solution: To solve this issue, we must use wait provided by selenium [Explicit wait]

WebDriverWait wait= new WebDriverWait(driver,15);  
wait.until(ExpectedConditions.visibilityOf(element));

### *NoAlertPresntException :*

This Exception occurs, when the driver in the Selenium Program code is unable to find the Alert on the web page to switch. i.e. when the driver is switching to an invalid or non-existing Alert pop-up.

Solution: Using try and catch block we need to handle this exception

public void handleAlerts(WebElement element){

try{

element.click();

Alert alert=driver.switchTo().alert();

String alertMessage=driver.switchTo().alert().getText();

System.out.println(alertMessage);

alert.accept();

} catch(NoAlertPresentException e){

System.out.println(NoAlertMessage);

}

### *StaleElementReferenceException:*

Stale Element means an old element or no longer available element. Assume there is an element that is found on a web page referenced as a WebElement in WebDriver. If the DOM changes then the WebElement goes stale. If we try to interact with an element which is staled, then the StaleElementReferenceExceptionis thrown.

The two reasons for Stale Element Reference are

1. The element has been deleted entirely.
2. The element is no longer attached to the DOM.

How to overcome Stale Element Reference

Solution 1:

You could refresh the page and try again for the same element.

Assume you are trying to click on a link and getting the stale element exception.

driver.navigate().refersh();

driver.findElement(By.xpath("xpath here")).click();

Solution 2:

If an element is not attached to DOM then you could try using ‘try-catch block’ within ‘for loop’

// Using for loop, it tries for 3 times.

// If the element is located for the first time then it breaks from the for loop nad comeout of the loop

for(int i=0; i<=2;i++){

try{

driver.findElement(By.xpath("xpath here")).click();

break;

}

catch(Exception e){

Sysout(e.getMessage());

}

}

Solution 3:

In order overcome this, we need to explicitly wait until the DOM is in a state where we are sure that DOM won't change.

WebDriverWait wait = new WebDriverWait(driver, 10);

wait.until(elementIdentified(By.id("element")));

Solution 4:

We can handle Stale Element Reference Exception by using POM[Refer: <https://www.softwaretestingmaterial.com/page-object-model/>]. We could avoid StaleElementException using POM. In POM, we use initElements() method which loads the element but it won’t initialize elements. initElements() takes latest address. It initializes during run time when we try to perform any action on an element. This process is also known as Lazy Initialization.

Solution 5:

Here sis recursive method of handling StaleElementReferenceException using fluent wait.

// Waiting 30 seconds for an element to be present on the page, checking

// for its presence once every 5 seconds.

Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)

.withTimeout(30, SECONDS)

.pollingEvery(5, SECONDS)

.ignoring(NoSuchElementException.class);

WebElement foo = wait.until(new Function<WebDriver, WebElement>() {

public WebElement apply(WebDriver driver) {

return driver.findElement(By.id("foo"));

}

});

If the above, one does not work then estimate a number of loop that might be enough for that element to be ready for interact

public void StaleElementHandleByID (String elementID){

int count = 0;

while (count < 4){

try {

WebElement yourSlipperyElement= driver.findElement(By.id(elementID));

yourSlipperyElement.click();

} catch (StaleElementReferenceException e){

e.toString();

System.out.println("Trying to recover from a stale element :" + e.getMessage());

count = count+1;

}

count = count+4;

}

Solution 6:

Let's suppose you have a text field element which is manipulated by the application and it's redrawn.

Normally, you would get StaleElementReferenceException because when WebDriver invokes findElement method, it saves the REFERENCE to the object. If the object is redrawn, there reference to the object is no longer actual and StaleElementReferenceException is thrown.

ExpectedCondition.stalenessOf waits until the element is redrawn.

This might be helpful to wait if DOM manipulation has occured. Then, you can find your element again and perform the operation (or use element created by PageFactory instead of refinding it).

However, the element might be manipulated many times (for example via jQuery calls of the front-end). In this case, waiting until the element is stale, and trying to find it, might throw StaleElementReferenceException anyway, because the element got stale AGAIN.

In that case, you can use ExpectedCondition.refresh(<my-expected-condition>).

This will allow you to perform operations within the time frame, regardless of staleness of the element

WebDriverWait wait= new WebDriverWait(driver,15);

wait.until(ExpectedConditions.refreshed(ExpectedConditions.stalenessOf("table")));

### [*NoSuchSessionException*](https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/NoSuchSessionException.html)*:*

Session ID is null. Using WebDriver after calling quit()

Example :

public static void main(String[] args) {

WebDriverdriver = newFirefoxDriver();

driver.get("http://www.facebook.com");

driver.quit();//Closing the browser

String title = driver.getTitle();

System.out.println("Title of the page:" + title);

}

}

Solution: Once the driver is quit, the session will not be available.

Invoking any methods using the object returns an NoSuchSessionException.

### *TimeoutException:*

This exception occurs when a command completion takes more than the wait time

Solution: Using multiple catch blocks we can handle such cases.

try{

driver.findElement(By.xpath("//\*[@id='register']")).click();

}catch (TimeoutException toe) {

wait.until( ExpectedConditions.elementToBeClickable(By.xpath("//\*[@id='register']")));

driver.findElement(By.xpath("//\*[@id='register']")).click();

}catch (Exception e) {

System.out.println("Register element is not found.");

throw(e);

}

}

# **Selenium-Assertions**

Assertion determines the state of the application whether it is the same what we are expecting or not. If the assertion fails, then the test case is failed and stops the execution. To use the Assertion in Web Driver, you need to download the Testng jar file

There are two types of Assertion, Hard & Soft Assertion

## a. Hard Assertion:

Hard Assertion is an Assertion that throws the AssertException when the test case is failed. In the case of Hard Assertion, you can handle the error by using a catch block like a java exception. Suppose we have two test cases in a suite. The first test case in a suite has an assertion that fails, and if we want to run the second case in a suit, then we need to handle the assertion error. A Hard

Assertion contains the following methods:

* AssertEquals
* AssertNotEquals
* AssertTrue
* AssertFalse
* AssertNull
* AssertNotNull

Example: Here Assert.assertFalse() contains the condition which is returning false value. Therefore, it passes the test case

driver.navigate().to("https://www.spicejet.com/");

Assert.assertFalse(driver.findElement(By.cssSelector("input[id\*='SeniorCitizen']")).isSelected());

System.out.println(driver.findElement(By.cssSelector("input[id\*='SeniorCitizen']")).isSelected());

## b. Soft Assertion:

Sometimes we want to execute the whole script even if the assertion fails. This is not possible in Hard Assertion. To overcome this problem, we need to use a soft assertion in testing. SoftAssert don't throw an exception when an assert fails. The test execution will continue with the next step after the assert statement.

Example:

import org.testng.annotations.Test;

import org.testng.asserts.SoftAssert;

public class TestOne {

@Test

public void testCaseOne() {

System.out.println("\*\*\* test case one started \*\*\*");

Assert.assertEquals(5, 5, "First hard assert failed");

System.out.println("hard assert success...."); Assert.assertTrue("Hello".equals("hello"), "Second hard assert failed"); System.out.println("\*\*\* test case one executed successfully \*\*\*");

}

@Test

public void testCasetwo() {

SoftAssert softAssert = new SoftAssert();

System.out.println("\*\*\* test case two started \*\*\*"); softAssert.assertEquals("Hello", "Hello", "First soft assert failed"); System.out.println("hard assert success...."); softAssert.assertTrue("Hello".equals("hello"), "Second soft assert failed"); softAssert.assertTrue("Welcome".equals("welcomeeee"), "Third soft assert failed"); System.out.println("\*\*\* test case two executed successfully \*\*\*"); softAssert.assertAll();

}

}

Once you execute the above code, you should see output something like below: -

[RemoteTestNG] detected TestNG version 6.14.3

\*\*\* test case one started \*\*\*

hard assert success....

\*\*\* test case two started \*\*\*

hard assert success....

\*\*\* test case two executed successfully \*\*\*

FAILED: testCaseOne

java.lang.AssertionError: Second hard assert failed expected [true] but found [false]

Observation:

1.testCaseOne will fail at below step and test will fail immediately.

Assert.assertTrue("Hello".equals("hello"), "Second hard assert failed");

2.testCasetwo Will also fail at below step, but will continue and execute other assertions until the last step softAssert.assertAll();

softAssert.assertTrue("Hello".equals("hello"), "Second soft assert failed");