UNIT-2

Course Outcome:

CO 2: Implement exception handling; file handling, and multi-threading in Java

Syllabus:

Exception Handling: The Idea behind Exception, Exceptions & Errors, Types of Exception, Control

Flow in Exceptions, JVM Reaction to Exceptions, Use of try, catch, finally, throw, throws in

Exception Handling, In-built and User Defined Exceptions, Checked and Un-Checked Exceptions**.**

Input /Output Basics: Byte Streams and Character Streams, Reading and Writing File in Java.

Multithreading: Thread, Thread Life Cycle, Creating Threads, Thread Priorities, Synchronizing Threads, Inter-thread Communication.

Lecture Delivery Plan:

Lecture-11

11.1 The Idea behind Exception

11.2 Exceptions & Errors

11.3 Types of Exception

Lecture-12

12.1Control Flow in Exceptions

12.2 JVM Reaction to Exceptions

12.3 Use of try

12.4 Catch

12.5, finally

12.6 Throw

12.7 throws in

12.8 Exception Handling,

Lecture-13

13.1 In-built and User Defined Exceptions

13.2 Checked and Un-Checked Exceptions

Lecture-14

14.1 Input /Output Basics:

14.2 Byte Streams and Character Streams,

Lecture -15

15.1 Reading and Writing File in Java.

\

Lecture-16

16.1 Thread

16.2 Thread Life Cycle

16.3 Creating Threads,

Lecture-17

17.1 Thread Priorities

17.2 Synchronizing Threads

Lecture-18

18.1 Inter-thread Communication

Important questions

1.What is a thread in Java? Describe the life cycle of a thread.

2.Explain the difference between reading and writing files using byte streams and character streams.

3.Discuss the role of the call stack in exception handling.

4.Discuss the difference between checked and unchecked exceptions.

5.When should checked exceptions be used in Java programs?

Exception Handling

The Idea behind Exception,- The core idea behind exceptions is to create a **controlled mechanism** for handling **unexpected events** that occur during program execution. These events, also called exceptions, can disrupt the normal flow of the program and potentially lead to crashes.

Here are some key aspects of the exception concept:

- **Identifying unexpected events:** Exceptions represent situations that deviate from the program's expected behavior. This could be due to various reasons like:
  - **User errors:** Entering invalid data, attempting forbidden actions.
  - **Resource limitations:** Running out of memory, network issues.
  - **System errors:** Hardware failures, software bugs.
- **Signaling the event:** When an exception occurs, the program "throws" an exception object. This object contains information about the error, such as its type and details.
- **Handling the event:** The program can define specific code blocks called "try-catch" blocks to catch and handle different types of exceptions.
  - The `try` block contains the code that might potentially throw an exception.
  - The `catch` block specifies how to handle the exception if it occurs within the `try` block.
- **Maintaining program flow:** By using exceptions, the program can gracefully recover from errors or provide informative messages to the user, instead of simply crashing. This helps maintain the program's overall stability and user experience.

Overall, exceptions provide a structured approach to dealing with unexpected situations, making programs more robust, maintainable, and user-friendly

What is Exception in Java?

Dictionary Meaning: Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Opening an unavailable file

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

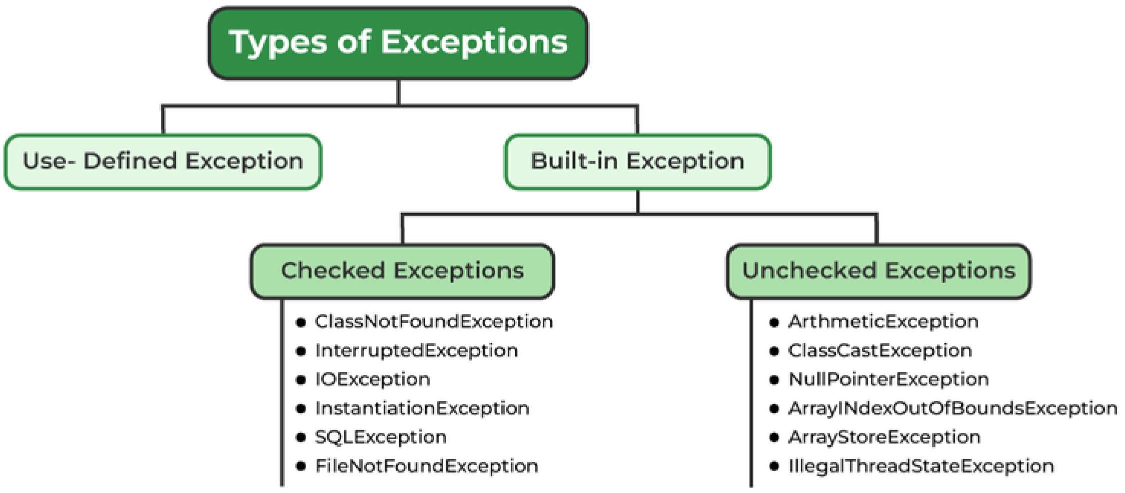
Difference between Error and Exception

Let us discuss the most important part which is the **differences between Error and Exception** that is as follows:

- **Error:** An Error indicates a serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Exceptions can be categorized in two ways:

1. **Built-in Exceptions**
   - Checked Exception
   - Unchecked Exception
2. **User-Defined Exceptions**

Let us discuss the above-defined listed exception that is as follows:

1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

Note: *For checked vs unchecked exception, see* [*Checked vs Unchecked Exceptions*]

2. User-Defined Exceptions:

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
Try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
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 necessary 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 specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

```
public class JavaExceptionExample{
public static void main(String args[]){
try{
//code that may raise exception
int data=100/0;
}catch(ArithmeticException e){
System.out.println(e);
}
//rest code of the program
System.out.println("rest of the code...");
}
}
```

OUTPUT

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Let's see a simple example of java multi-catch block.

MultipleCatchBlock1.java

```
public class MultipleCatchBlock1 {

public static void main(String[] args) {

try{
```

```
int a[]=new int[5];
a[5]=30/0;
}
catch(ArithmeticException e)
{
System.out.println("Arithmetic Exception occurs");
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("ArrayIndexOutOfBounds Exception occurs");
}
catch(Exception e)
{
System.out.println("Parent Exception occurs");
}
System.out.println("rest of the code");
}
}
```

Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

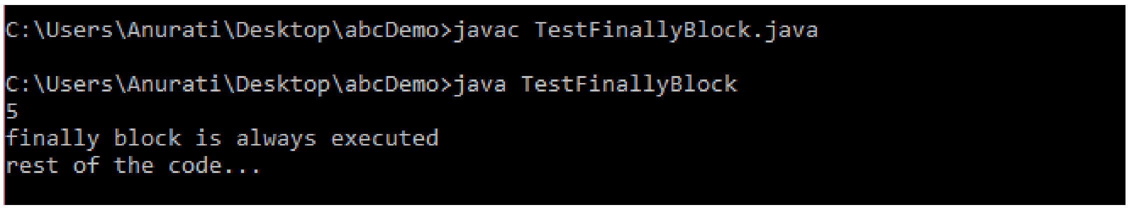
- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

TestFinallyBlock.java

```
class TestFinallyBlock {
public static void main(String args[]){
try{
//below code do not throw any exception
int data=25/5;
System.out.println(data);
```

```java
}
//catch won't be executed
catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
finally {
System.out.println("finally block is always executed");
}

System.out.println("rest of phe code...");
}
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock
5
finally block is always executed
rest of the code...
```

When an exception occurr but not handled by the catch block

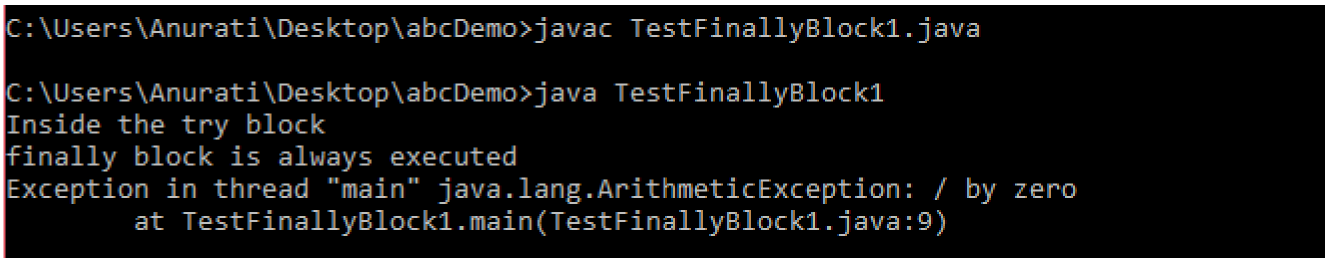
Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

TestFinallyBlock1.java

```java
public class TestFinallyBlock1{
public static void main(String args[]){

try {

System.out.println("Inside the try block");

//below code throws divide by zero exception
int data=25/0;
System.out.println(data);
}
//cannot handle Arithmetic type exception
//can only accept Null Pointer type exception
catch(NullPointerException e){
```

```
System.out.println(e);
}

//executes regardless of exception occured or not
finally {
System.out.println("finally block is always executed");
}
System.out.println ("rest of the code...");
}
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

Java throw keyword

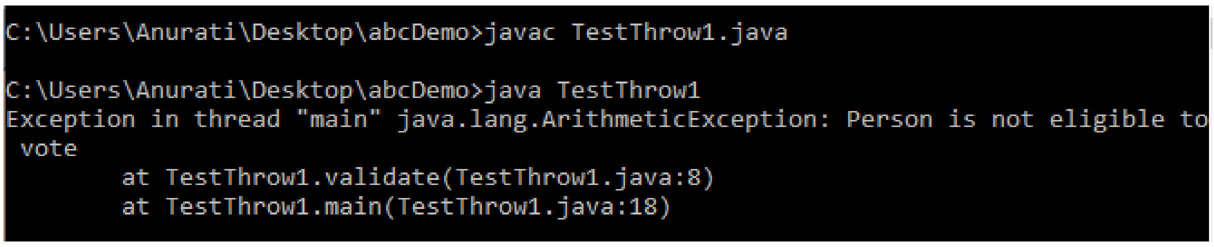
The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

```
public class TestThrow1 {
//function to check if person is eligible to vote or not
public static void validate(int age) {
if(age<18) {
//throw Arithmetic exception if not eligible to vote
throw new ArithmeticException("Person is not eligible to vote");
}
else {
System.out.println("Person is eligible to vote!!");
}
}
//main method
public static void main(String args[]){
//calling the function
15.         validate(13);
16.         System.out.println("rest of the code...");
17.         }
18.         }
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

Exceptions as Control Flow

Using exceptions as control flow is generally considered a bad practice. While exceptions are a mechanism to stop unexpected behavior in software development, abusing them in expected behaviors can lead to negative consequences.

This practice can reduce the performance of the code and make it less readable

.

Reasons to Avoid Using Exceptions as Control Flow

1. **Performance Impact**: Using exceptions as control flow can reduce the performance of the code as a response per unit time
2. **Readability**: It makes the code less readable and hides the programmer's intention, which is considered a bad practice

Alternatives to Using Exceptions for Control Flow

Instead of using exceptions for expected behaviours, it is recommended to use control flow statements to handle the logic.

If the behaviours are expected, using control flow statements is generally considered better than throwing an exception and handling it by yourself

JVM Reaction to Exceptions,

Certainly! Here is a diagram illustrating the Java exception hierarchy and the various types of exceptions:

```
                              +----------------------+
                       |      Throwable      |
                       +----------+----------+
                                  |
                         +--------+--------+
                         |     Error       |
                         +-----------------+
                         |     Exception   |
                         +--------+--------+
                                  |
                         +--------+--------+
                         |  RuntimeException|
                         +--------+--------+
                        |  Checked Exception|
                         +--------+--------+
                        |Unchecked Exception|
                        +------------------+
```

In this hierarchy:

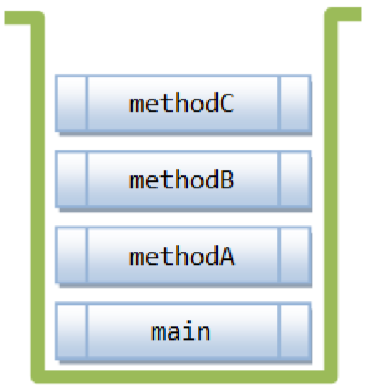
- **Throwable** is the base class for all exceptions and errors in Java.
- **Error** is used by the Java runtime system (JVM) to indicate errors related to the runtime environment itself.
- **Exception** is used for exceptional conditions that user programs should catch. It has two main branches:
    - **RuntimeException**: These are unchecked exceptions that occur at runtime and are not checked by the compiler.
    - **Checked Exception**: These are exceptions that occur at compile time and are checked by the compiler.

This diagram provides a clear overview of the Java exception hierarchy and the different types of exceptions.

```
Exit
methodA(
) Exit
main()
```

As seen from the output, the sequence of events is:

1. JVM invoke the main().
2. main()pushed onto call stack, before invoking methodA().
3. methodA()pushed onto call stack, before invoking methodB().
4. methodB()pushed onto call stack, before invoking methodC().
5. methodC() completes.
6. methodB()popped out from call stack and completes.
7. methodA()popped out from the call stack and completes.
8. main()popped out from the call stack and completes. Program exits.



Method Call Stack
(Last-in-First-out Queue)

suppose that we modify methodC() to carry out a "divide-by-0" operation, which triggers a ArithmeticException:

```
public static void methodC() {

     System.out.println("Enter  methodC()");

     System.out.println(1 / 0);   // divide-by-0 triggers an ArithmeticException
     System.out.println("Exit methodC()");

}
```

The exception message clearly shows the *method call stack trace* with the relevant statement line numbers:

```
Enter main()

Enter      methodA()
Enter      methodB()
Enter methodC()

Exception in thread "main" java.lang.ArithmeticException: / by zero

            at       MethodCallStackDemo.methodC(MethodCallStackDemo.java:22)        at
```

MethodC() triggers an ArithmeticException. As it does not handle this exception, it popped off from the call stack immediately. MethodB() also does not handle this exception and popped off the call stack. So does methodA() and main() method. The main() method passes back to JVM, which abruptly terminates the program and print the call stack trace, as shown.

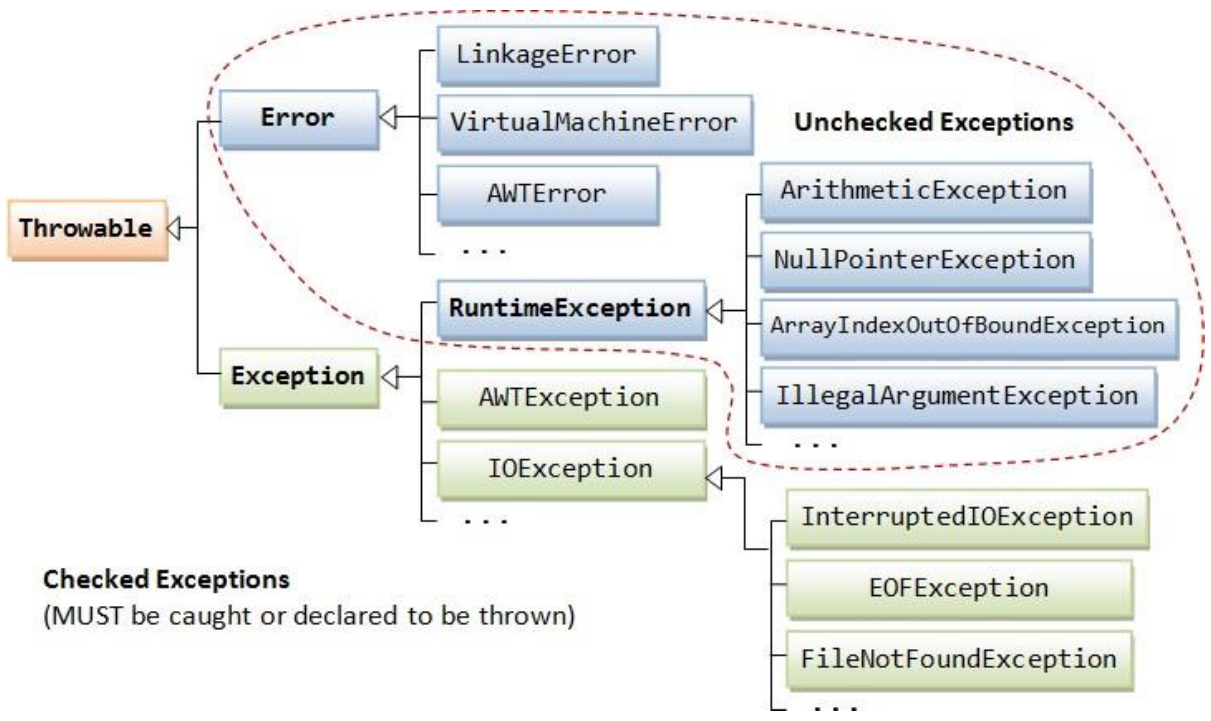
1.3 Exception & Call Stack

When an exception occurs inside a Java method, the method creates an Exception object and passes the Exception object to the JVM (in Java term, the method "throw" an Exception). The Exception object contains the type of the exception, and the state of the program when the exception occurs. The JVM is responsible for finding an *exception handler* to process the Exception object. It searches backward through the call stack until it finds a matching exception handler for that particular class of Exception object (in Java term, it is called "catch" the Exception). If the JVM cannot find a matching exception handler in all the methods in the call stack, it terminates the program.

This process is illustrated as follows. Suppose that methodD() encounters an abnormal condition and throws a XxxException to the JVM. The JVM searches backward through the call stack for a matching exception handler. It finds methodA() having a XxxException handler and passes the exception object to the handler. Notice that methodC() and methodB() are required to declare "throws XxxException" in their method signatures in order to compile the program.

1.4 Exception Classes - Throwable, Error, Exception & RuntimeException

The figure below shows the hierarchy of the Exception classes. The base class for all Exception objects is java.lang.Throwable, together with its two subclasses java.lang.Exception and java.lang.Error.



The Error class describes internal system errors (e.g., VirtualMachineError, LinkageError) that rarely occur. If such an error occurs, there is little that you can do and the program will be terminated by the Java runtime.

The Exception class describes the error caused by your program (e.g. FileNotFoundException, IOException). These errors could be caught and handled by your program (e.g., perform an alternate action or do a graceful exit by closing all the files, network and database connections).

Use of try

Certainly! Here's a diagram illustrating the use of the try...catch block in Java:

```java
public class Main {
    public static void main(String[] args) {
        try {
            // Code that may throw an exception
            int[] myNumbers = {1, 2, 3};
            System.out.println(myNumbers[10]); // This line may
throw an exception
        } catch (Exception e) {
            // Code to handle the exception
            System.out.println("Something went wrong.");
        } finally {
            // Code that always executes, regardless of the result
            System.out.println("The 'try catch' is finished.");
        }
    }
}
```

In this diagram:

- The `try` block contains the code that may throw an exception.
- The `catch` block is used to catch and handle the exception. It contains the code that executes when an exception is thrown.

- The `finally` block contains the code that always executes, regardless of whether an exception was thrown or caught.

CATCH

1.7 try-catch-finally

The syntax of try-catch-finally is:

```
try {
    // main logic, uses methods that may throw Exceptions
    ......
} catch (Exception1 ex) {
    // error handler for Exception1
    ......
} catch (Exception2 ex) {
    // error handler for Exception1
    ......
}  finally  {     // finally is optional
    // clean up codes, always executed regardless of exceptions
    ......
}
```

If no exception occurs during the running of the try-block, all the catch-blocks are skipped, and finally-block will be executed after the try-block. If one of the statements in the try-block throws an exception, the Java runtime ignores the rest of the statements in the try-block, and begins searching for a matching exception handler. It matches the exception type with each of the catch-blocks *sequentially*. If a catch-block catches that exception class or catches a *superclass* of that exception, the statement in that catch-block will be executed. The statements in the finally-block are then executed after that catch-block. The program continues into the next statement after the try-catch-finally, unless it is pre-maturely terminated or branch-out.

If none of the catch-block matches, the exception will be passed up the call stack. The current method executes the finally clause (if any) and popped off the call stack. The caller follows the same procedures to handle the exception.

The finally block is almost certain to be executed, regardless of whether or not exception occurs (unless JVM encountered a severe error or a System.exit()is called in the catchblock).

Example 1

```
import java.util.Scanner; import
java.io.File;
import java.io.FileNotFoundException; public class
TryCatchFinally {
    public static void main(String[] args) { try {   // main
        logic
            System.out.println("Start of the main logic"); System.out.println("Try
            opening a file ..."); Scanner in = new Scanner(new File("test.in"));

            System.out.println("File Found, processing the file ..."); System.out.println("End of the main logic");

        } catch (FileNotFoundException ex) {                    // error handling separated from the main logic
            System.out.println("File Not Found caught ...");

        } finally {              // always run regardless of exception status
            System.out.println("finally-block runs regardless of the state of exception");
        }
        // after the try-catch-finally
        System.out.println("After try-catch-finally, life goes on...");
} finally {      // finally is optional
    // clean up codes, always executed regardless of exceptions
    ......
}
```

Catch

In Java, the `catch` keyword is used as part of the try...catch block to handle exceptions. Here's a summary of its usage based on the provided search results:

- The `catch` statement allows you to define a block of code to be executed if an error occurs in the try block. It comes in pairs with the `try` statement.
  - Syntax:

```
try {
    // Block of code to try
} catch(Exception e) {
    // Block of code to handle errors
}
```

  - The `catch` block catches and handles the exceptions by declaring the type of exception within the parameter. It includes the code that is executed if an exception inside the try block occurs.
  - It is used to handle uncertain conditions of a try block and must be followed by the try block.
  - Multiple catch blocks can be used with a single try block to handle different types of exceptions.

The `catch` block is an essential part of exception handling in Java, allowing developers to gracefully handle errors and prevent the abnormal termination of the program.

Finally

The `finally` block is used to execute important code such as releasing resources, regardless of whether an exception is thrown or not in the `try` block.

  - Syntax:

```
try {
    // Block of code to try
} catch(Exception e) {
    // Block of code to handle errors
} finally {
    // Block of code to execute regardless of an exception
}
```

  - The `finally` block always executes, even if an exception is thrown and caught.

    - It is often used to release resources like file handles, database connections, or network connections, to ensure that these resources are properly closed or released.
    - The `finally` block is optional, but when used, it ensures that certain code will always be executed, making it useful for cleanup or finalization tasks.

Throw

the `throw` keyword is used to explicitly throw an exception. Here's a summary of its usage based on the provided search results:

- The `throw` statement is used to throw an exception explicitly within a method or block of code.
    - Syntax:

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

    - The `throw` statement is followed by the keyword `new` and the constructor of the exception type to create and throw an instance of that exception.
    - It is typically used to handle exceptional situations where the program encounters an error or an unexpected condition.
    - The exception that is thrown can be a built-in exception class provided by Java, or it can be a custom exception class created by the programmer.
-

throws in Exception Handling,

`throws` keyword is used in method declarations to specify which exceptions are not handled within the method but are instead propagated to the calling code to be handled. Here's an overview of its usage:

Usage of the **`throws`** Keyword:

- **Syntax:**

```
returnType methodName(parameters) throws ExceptionType1, ExceptionType2, ... {
    // Method body
}
```

- The `throws` keyword is used in the method signature to indicate that the method may throw one or more exceptions of the specified types.

- When a method defines a `throws` clause, it informs the calling code that it is not handling the specified exceptions internally, and it's the responsibility of the calling code to handle or propagate these exceptions.
- The calling code must either handle the exceptions using a `try-catch` block or declare the exceptions to be thrown further up the call stack.
- Example:

```
public void readFile(String fileName) throws IOException {
    // Method implementation that may throw IOException
}
```

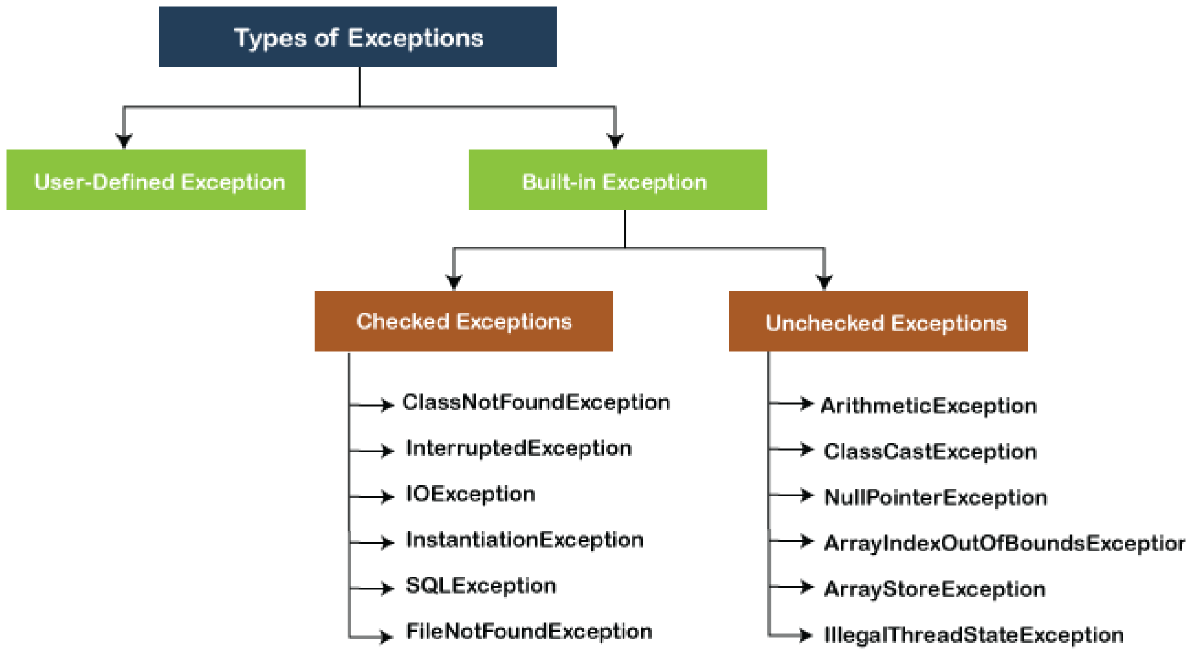
In the example above, the `readFile` method declares that it may throw an `IOException`. This means that any code calling the `readFile` method must either handle the `IOException` or declare it to be thrown further up the call stack

Types of Exception in Java

In Java, **exception** is an event that occurs during the execution of a program and disrupts the normal flow of the program's instructions. Bugs or errors that we don't want and restrict our program's normal execution of code are referred to as **exceptions**. In this section, we will focus on the **types of exceptions in Java** and the differences between the two.

Exceptions can be categorized into two ways:

1. Built-in Exceptions
    - Checked Exception
    - Unchecked Exception
2. User-Defined Exceptions



Built-in Exception

Exceptions that are already available in **Java libraries** are referred to as **built-in exception**. These exceptions are able to define the error situation so that we can understand the reason of getting this error. It can be categorized into two broad categories, i.e., **checked exceptions** and **unchecked exception**.

Checked Exception

Checked exceptions are called **compile-time** exceptions because these exceptions are checked at compile-time by the compiler. The compiler ensures whether the programmer handles the exception or not. The programmer should have to handle the exception; otherwise, the system has shown a compilation error.

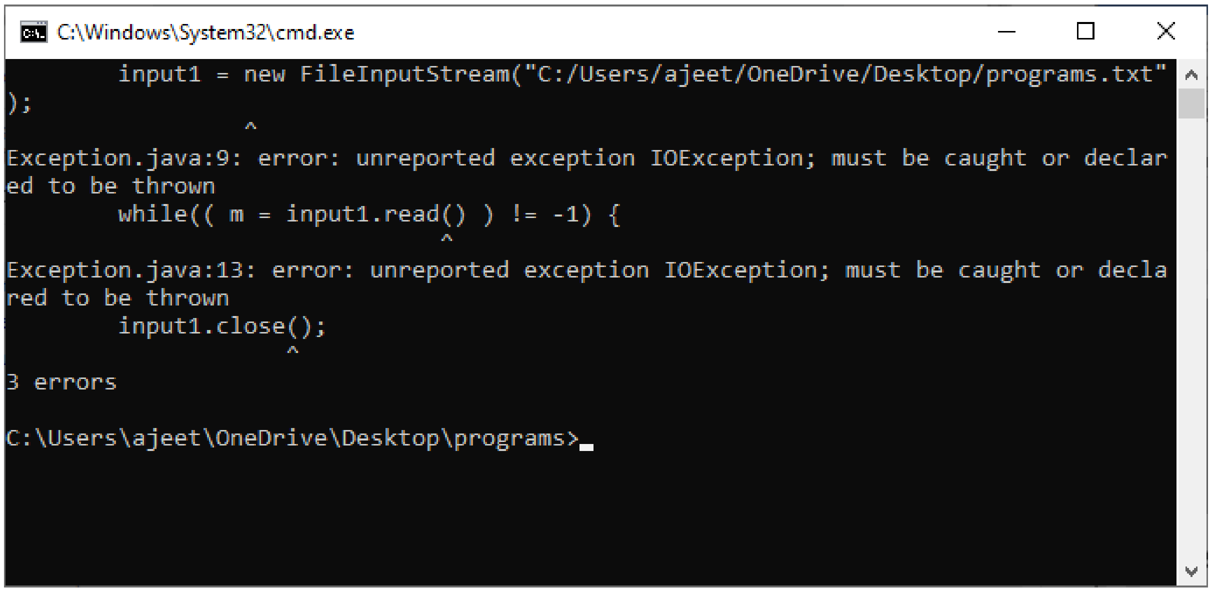
CheckedExceptionExample.java

```java
import java.io.*;
class CheckedExceptionExample {
   public static void main(String args[]) {
      FileInputStream file_data = null;
      file_data = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/Hello.txt");
      int m;
      while(( m = file_data.read() ) != -1) {
         System.out.print((char)m);
      }
      file_data.close();
   }
}
```

In the above code, we are trying to read the **Hello.txt** file and display its data or content on the screen. The program throws the following exceptions:

1. The **FileInputStream(File filename)** constructor throws the **FileNotFoundException** that is checked exception.
2. The **read()** method of the **FileInputStream** class throws the **IOException**.
3. The **close()** method also throws the IOException.

Output:



```
        input1 = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/programs.txt"
);
                 ^
Exception.java:9: error: unreported exception IOException; must be caught or declar
ed to be thrown
        while(( m = input1.read() ) != -1) {
                               ^
Exception.java:13: error: unreported exception IOException; must be caught or decla
red to be thrown
        input1.close();
                    ^
3 errors

C:\Users\ajeet\OneDrive\Desktop\programs>
```

How to resolve the error?

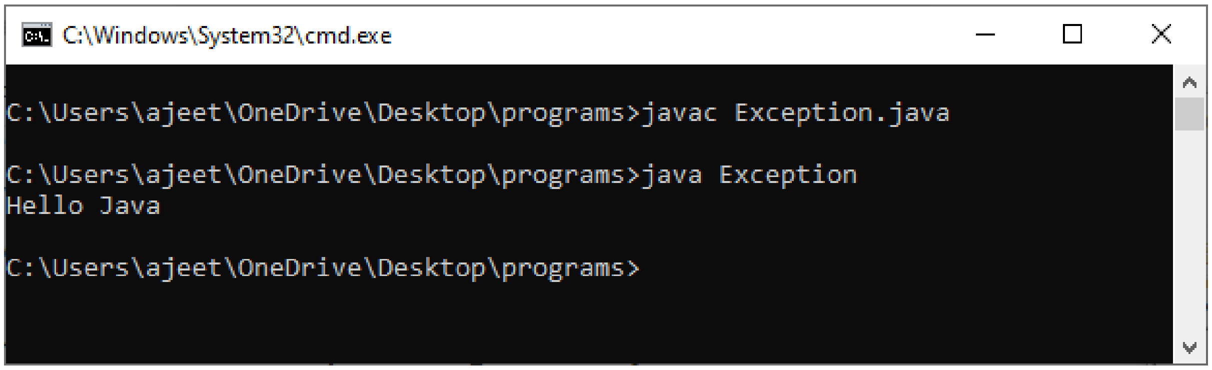
There are basically two ways through which we can solve these errors.

1) The exceptions occur in the main method. We can get rid from these compilation errors by declaring the exception in the main method using **the throws** We only declare the IOException, not FileNotFoundException, because of the child-parent relationship. The IOException class is the parent class of FileNotFoundException, so this exception will automatically cover by IOException. We will declare the exception in the following way:

```
class Exception{
  public static void main(String args[])  throws IOException {
      ...
     ...
}
```

If we compile and run the code, the errors will disappear, and we will see the data of the file.

ADVERTISEMENT



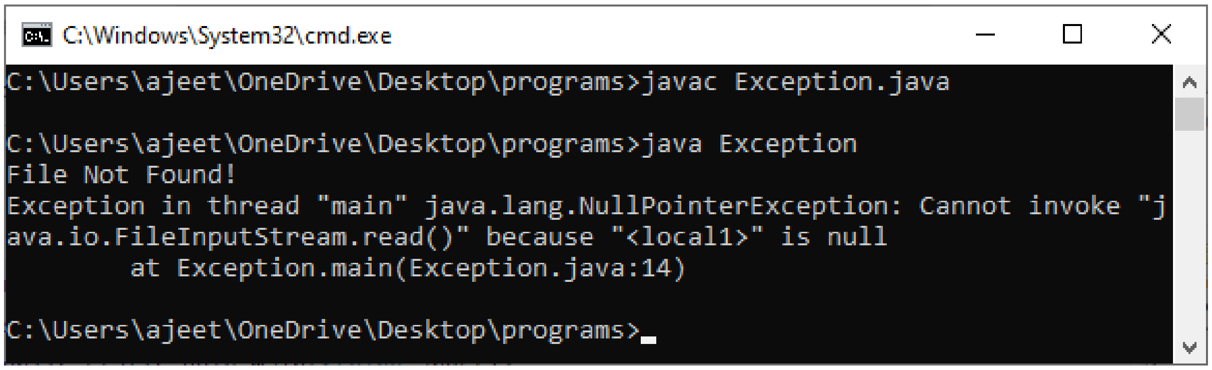
```
C:\Users\ajeet\OneDrive\Desktop\programs>javac Exception.java

C:\Users\ajeet\OneDrive\Desktop\programs>java Exception
Hello Java

C:\Users\ajeet\OneDrive\Desktop\programs>
```

2) We can also handle these exception using **try-catch** However, the way which we have used above is not correct. We have to a give meaningful message for each exception type. By doing that it would be easy to understand the error. We will use the try-catch block in the following way:

Exception.java

```java
import java.io.*;
class Exception{
    public static void main(String args[]) {
        FileInputStream file_data = null;
        try{
            file_data = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/programs/Hell.txt");
        }catch(FileNotFoundException fnfe){
            System.out.println("File Not Found!");
        }
        int m;
        try{
            while(( m = file_data.read() ) != -1) {
                System.out.print((char)m);
            }
            file_data.close();
        }catch(IOException ioe){
            System.out.println("I/O error occurred: "+ioe);
        }
    }
}
```

We will see a proper error message **"File Not Found!"** on the console because there is no such file in that location.



```
C:\Users\ajeet\OneDrive\Desktop\programs>javac Exception.java

C:\Users\ajeet\OneDrive\Desktop\programs>java Exception
File Not Found!
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "j
ava.io.FileInputStream.read()" because "<local1>" is null
        at Exception.main(Exception.java:14)

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

Unchecked Exceptions

The **unchecked** exceptions are just opposite to the **checked** exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Usually, it occurs when the user provides bad data during the interaction with the program.

Note: The RuntimeException class is able to resolve all the unchecked exceptions because of the child-parent relationship.

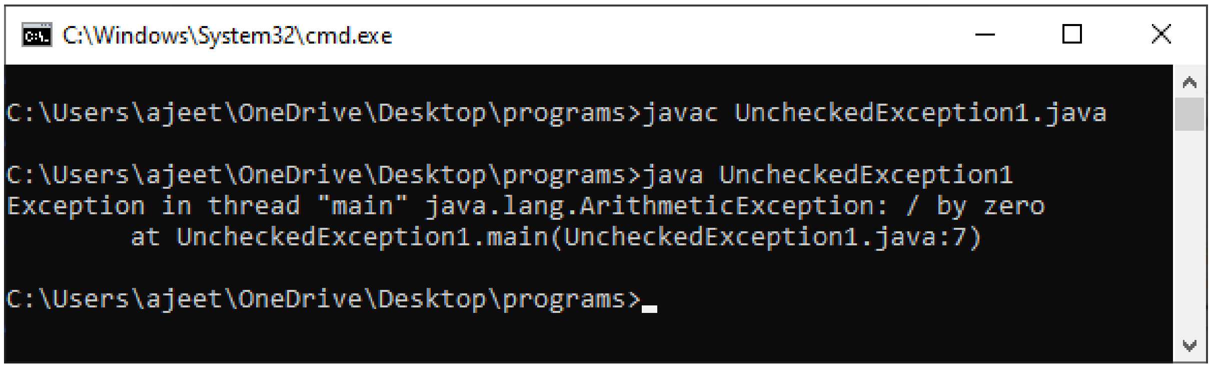
UncheckedExceptionExample1.java

```
class UncheckedExceptionExample1 {
  public static void main(String args[])
  {
   int postive = 35;
   int zero = 0;
   int result = positive/zero;
   //Give Unchecked Exception here.
System.out.println(result);
  }
}
```

In the above program, we have divided 35 by 0. The code would be compiled successfully, but it will throw an ArithmeticException error at runtime. On dividing a number by 0 throws the divide by zero exception that is a uncheck exception.

Output:

ADVERTISEMENT

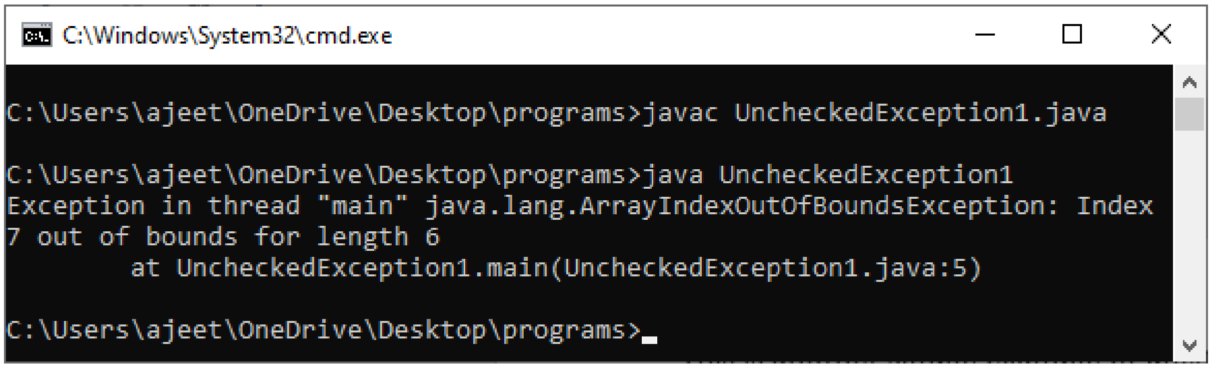


```
C:\Users\ajeet\OneDrive\Desktop\programs>javac UncheckedException1.java

C:\Users\ajeet\OneDrive\Desktop\programs>java UncheckedException1
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at UncheckedException1.main(UncheckedException1.java:7)

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

UncheckedException1.java

```
class UncheckedException1 {
  public static void main(String args[])
  {
   int num[] ={10,20,30,40,50,60};
   System.out.println(num[7]);
  }
}
```

Output:



```
C:\Users\ajeet\OneDrive\Desktop\programs>javac UncheckedException1.java

C:\Users\ajeet\OneDrive\Desktop\programs>java UncheckedException1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index
7 out of bounds for length 6
        at UncheckedException1.main(UncheckedException1.java:5)

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

In the above code, we are trying to get the element located at position 7, but the length of the array is 6. The code compiles successfully, but throws the ArrayIndexOutOfBoundsException at runtime.

User-defined Exception

In Java, we already have some built-in exception classes like ArrayIndexOutOfBoundsException, **NullPointerException**, and **ArithmeticException**. These exceptions are restricted to trigger on some predefined conditions. In Java, we can write our own exception class by extends the Exception class. We can throw our own exception on a particular condition using the throw keyword. For creating a user-defined exception, we should have basic knowledge of **the** try-catch block and **throw** keyword.

Let's write a Java program and create user-defined exception.

Difference between Checked and Unchecked Exception

S.No	Checked Exception	Unchecked Exception
1.	These exceptions are checked at compile time. These exceptions are handled at compile time too.	These exceptions are just opposite to the checked exceptions. These exceptions are not checked and handled at compile time.
2.	These exceptions are direct subclasses of exception but not extended from RuntimeException class.	They are the direct subclasses of the RuntimeException class.
3.	The code gives a compilation error in the case when a method throws a checked exception. The compiler is not able to handle the exception on its own.	The code compiles without any error because the exceptions escape the notice of the compiler. These exceptions are the results of user-created errors in programming logic.
4.	These exceptions mostly occur when the probability of failure is too high.	These exceptions occur mostly due to programming mistakes.
5.	Common checked exceptions include IOException, DataAccessException, InterruptedException, etc.	Common unchecked exceptions include ArithmeticException, InvalidClassException, NullPointerException, etc.
6.	These exceptions are propagated using the throws keyword.	These are automatically propagated.
7.	It is required to provide the try-catch and try-finally block to handle the checked exception.	In the case of unchecked exception it is not mandatory.

Thread

What is Thread in java

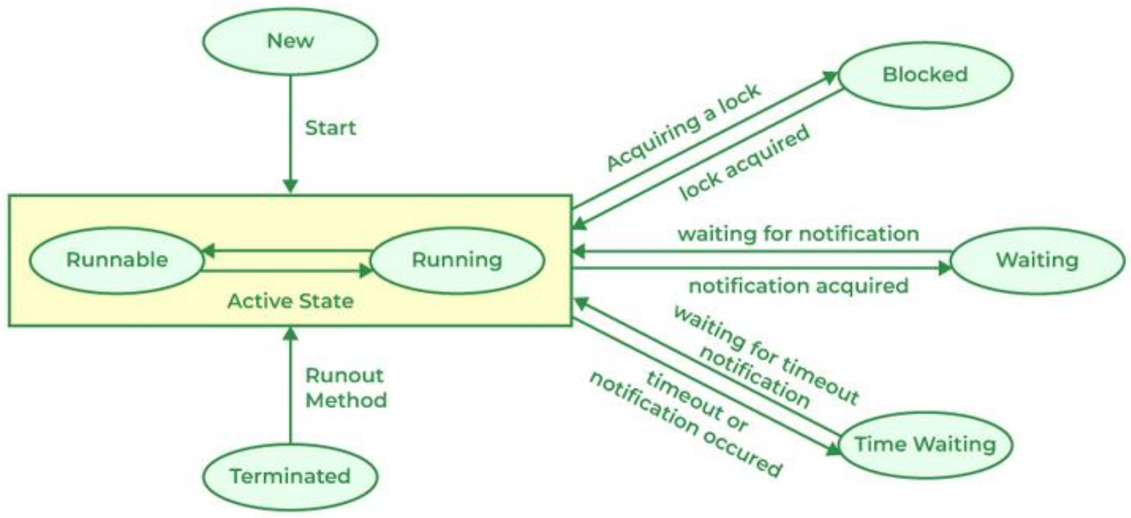
A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New State
2. Runnable State
3. Blocked State
4. Waiting State
5. Timed Waiting State
6. Terminated State

The diagram shown below represents various states of a thread at any instant in time.



States of Thread in its Lifecycle

Life Cycle of a Thread

There are multiple states of the thread in a lifecycle as mentioned below:

1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.
2. **Runnable State:** A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
3. **Blocked:** The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.
4. **Waiting state:** The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.
5. **Timed Waiting:** A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
6. **Terminated State:** A thread terminates because of either of the following reasons:
   - Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
   - Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.

Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws

IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example

- Java

```
// Java Program to Illustrate Priorities in Multithreading

// via help of getPriority() and setPriority() method



// Importing required classes

import java.lang.*;
```

```
// Main class
class ThreadDemo extends Thread {

    // Method 1
    // run() method for the thread that is called
    // as soon as start() is invoked for thread in main()
    public void run()
    {
        // Print statement
        System.out.println("Inside run method");
    }
```

```
// Main driver method

public static void main(String[] args)

{

    // Creating random threads

    // with the help of above class

    ThreadDemo t1 = new ThreadDemo();

    ThreadDemo t2 = new ThreadDemo();

    ThreadDemo t3 = new ThreadDemo();




    // Thread 1

    // Display the priority of above thread
```

```
        // using getPriority() method

        System.out.println("t1 thread priority : "

                           + t1.getPriority());




        // Thread 1

        // Display the priority of above thread

        System.out.println("t2 thread priority : "

                           + t2.getPriority());




        // Thread 3

        System.out.println("t3 thread priority : "

                           + t3.getPriority());
```

```
// Setting priorities of above threads by

// passing integer arguments

t1.setPriority(2);

t2.setPriority(5);

t3.setPriority(8);



// t3.setPriority(21); will throw

// IllegalArgumentException



// 2

System.out.println("t1 thread priority : "
                   + t1.getPriority());

// 5

System.out.println("t2 thread priority : "

                   + t2.getPriority());

// 8

System.out.println("t3 thread priority : "

                   + t3.getPriority());

// Main thread
```

```
        // Displays the name of

        // currently executing Thread

        System.out.println(

            "Currently Executing Thread : "

            + Thread.currentThread().getName());




        System.out.println(

            "Main thread priority : "

            + Thread.currentThread().getPriority());




        // Main thread priority is set to 10

        Thread.currentThread().setPriority(10);
        System.out.println(

            "Main thread priority : "

            + Thread.currentThread().getPriority());

    }

}
```

Output

```
t1 thread priority : 5

t2 thread priority : 5

t3 thread priority : 5

t1 thread priority : 2

t2 thread priority : 5

t3 thread priority : 8

Currently Executing Thread : main

Main thread priority : 5

Main thread priority : 10
```

Output explanation:

- Thread with the highest priority will get an execution chance prior to other threads. Suppose there are 3 threads t1, t2, and t3 with priorities 4, 6, and 1. So, thread t2 will execute first based on maximum priority 6 after that t1 will execute and then t3.

- The default priority for the main thread is always 5, it can be changed later. The default priority for all other threads depends on the priority of the parent thread.

Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.

    3. Static synchronization.

2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

TestSynchronization1.java

```java
class Table{
void printTable(int n){//method not synchronized
  for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
  }

```

```
 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

```
5
```

```
100
10
200
15
300
20
400
25
500
```

Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

TestSynchronization2.java

```java
//example of java synchronized method
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Inter-thread Communication in Java

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

```
public final void notify()
```

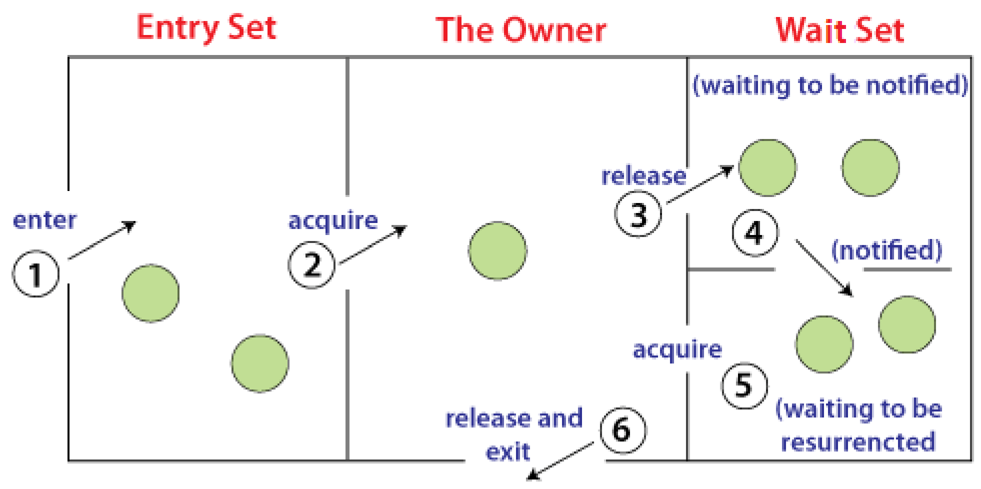
3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

1. public final void notifyAll()

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	**sleep()**
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

Test.java

```
class Customer{
int amount=10000;

synchronized void withdraw(int amount){
System.out.println("going to withdraw...");

if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}

synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}

class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();

}}
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```

peration:

This operation is used to rename the existing file.

Important questions

1.What is a thread in Java? Describe the life cycle of a thread.

2.Explain the difference between reading and writing files using byte streams and character streams.

3.Discuss the role of the call stack in exception handling.

4.Discuss the difference between checked and unchecked exceptions.

5.When should checked exceptions be used in Java programs?