Unit4: Exception Handling and multithreading

An Exception can be anything that interrupts the normal flow of the program. When an exception occurs program gets terminated and doesn't continue further. It is an object which is thrown at runtime.

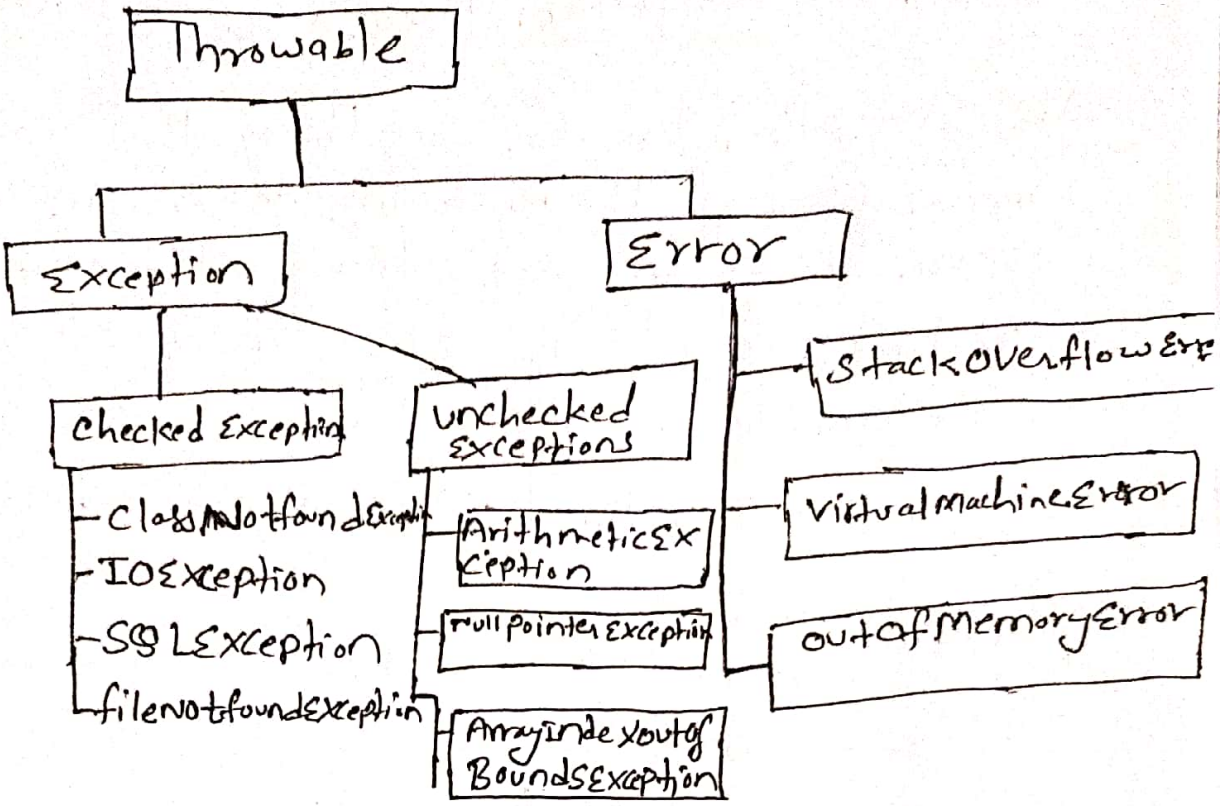
Let's understand from a scenario:

```
Statement1;
Statement2;
Statement3;
statement4; // exception occurs
Statement5;
Statement6;
:
statementn;
```

Suppose there are 10 statements in java program and an exception occurs at statement 4; the rest of the code will not be executed, i.e., Statement 5 to 10 will not be executed. However, when we perform exception handling, the rest of the statemen will be executed. That is why we use exception handling in java.

4.1 The Exception Hierarchy

In java, exceptions are represented in a class hierarchy rooted at java.lang.Throwable, which is divided into two branches:



1 Exception : Represents exceptions that can be caught and handled by the program.

(1) Checked Exception: Exceptions that are checked at compile time.
IOException, SQLException are example of Checked exceptions that must be either caught or declared in the method.

(2) unchecked Exception: An exception that occurs at the time of (Runtime) execution.
Also known as Runtime Exceptions.
NullPointerException, ArrayIndex
- outofBoundsException, ArithmeticExce
-ption are the Example of unchecke
Exception.

2 Error: Represents serious system errors (e.g outofmemory Error, stackoverflowError) that usually cannot be handled.

Errors are the critical conditions that occur due to the lack of the system resources, and it cannot be handled by the code of the program. The code is not responsible for such errors. The result of the occurrence of the error is that the program gets terminated abnormally.

4.2 Exception Handling fundamentals

Exception Handling is a mechanism to handle the exceptions. Java provides five keywords that are used to handle the exception.
try, catch, finally, throw, throws.

1. Syntax of try-catch-finally

```
try
{
  //code that might throw an exception
}
catch (ExceptionType1 e1)
{ code to handle ExceptionType1
}
catch (ExceptionType2 e2)
{ //code to handle ExceptionType2
}
finally
{ //code that will always execute (optional, but used for cleanup)
}
```

2. Using throw to manually Throw an exception

Syntax:

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

3. Method Declaration with throws

Syntax:

```
returntype methodName() throws ExceptionType1,
                                ExceptionType2
{
    // method code that might throw exceptions
}
```

All keywords using syntax!

```
public return_type method_Name() throws Exception
Type1, ExceptionType2
{
  try
  {
    //code that might throw an exception
    throw new ExceptionType1("exception message");
  }
  Catch(ExceptionType1 e1)
    {
      // Handle ExceptionType1
    }
    catch(ExceptionType2 e2)
    {
      // Handle exception Type2
    }
    finally
    {
      // Code that will always run(e.g. resource clean up)
    }
}
```

① Divide by Zero program occurs Arithmetic Exception

```
Class ExceptionExample
{
  public static void main (String args[])
  {
     int a=10;
     int b=5;
     int c=5;
     int res = a/(b-c); // exception occurs
     System.out.println (res);
  }
}
```

Now, we are handling this Arithmetic exception using Try, catch blocks.

```
Class ExceptionHandle
{
  public static void main(String args[])
  { int a=10,b=5, c=5;
    try
    {
      int res = a/(b-c);
    }
    catch(ArithmeticException e)
    {
      System.out.println("Division by zero");
    }
```

```
finally
{
    System.out.println("This block is always
                        executed");
}
}
```

output: Division by zero
This block is always executed

Example program using throw keyword:

→ throw keyword is used to throw an exception message explicitly.

```
public class ThrowAE
{
    public static int divide(int x, int y)
    {
        if(y==0)
        {
            throw new ArithmeticException("cannot
            divide by zero");
        }
        return x/y;
    }
    public static void main(String args[])
    {
        try
        {
            int x = 10;
            int y = 0;
            int result = divide(x, y);
            System.out.println(result);
        }
```

```
    Catch (ArithmeticException e)
        System.out.println("Error: " + e.getMessage());
    }
}
```

Output:

Error: Cannot divide by zero.

Example program for ArrayIndexOutOfBoundsException Handling:

```
public class ArrayOutOfBoundExample
{
    public static void main(String args[])
    {
        try
        {
            int[] numbers = {1,2,3,4,5};
            System.out.println("Element at index 10: " + numbers[10]);
        }
        Catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Error: " + e.getMessage());
            System.out.println("Array index is out of bounds.
                                Please check the index value.");
        }
        finally
        {
            System.out.println("Array access attempt completed.");
        }
    }
}
```

.output:

~~Error: Element at index 10:~~

Error: Index 10 out of bounds for length 5

Array index is out of bounds. please check the index value.

Array access attempt Completed.

Keywords Concepts:

try: Block of code where exceptions might occur.

Catch: Block of code that handles the exception.

finally: Block of code that always executes, regardless of whether an exception was thrown or caught.

throws: used to declare that a method may throw an exception.

Example:

```
try
{
  int result = 10/0;
}
catch (ArithmeticException e)
{
  System.out.println("Error: "+ e.getMessage());
}
finally
{
  System.out.println("End of Exception handling");
}
```

output: Error: / by zero
End of Exception handling.

4.3 Throwing, Re-throwing, and Catching Exceptions

↳ Throwing an exception: you can use the throw keyword to manually throw an exception.

Syntax: `throw new ArithmeticException("custom division by zero error");`

↳ Re-throwing an exception: After catching an exception, you can rethrow it to let the calling method handle it.

Syntax:

```
try
{
  // code
}
catch (Exception e)
{
  throw e;
}
```

↳ Cathing exceptions: Exceptions are caught using the catch block, which could match the type of the thrown exception.

```
try
{
  // code that might throw an exception
}
catch (IOException e)
{
  //handle the specific exception
}
```

4.5 Multithreading fundamentals

Process and Thread are two basic units of java program execution. A program in an execution is process. process can be seen as a program or application.

Thread is a segments of process. It is lightweight proless. Thread requires less resources to create and exists in the process. Thread shares the process resources.

multithreading in java is a process of executing multiple processes simultaneously.

A program is divided into two or more subprograms, which can be implemented at the same time in parallel.

multiprocessing & multithreading both are used to achieve multitasking.

Threads are independent so it doesn't affect other threads.

multithreadingcan perform many operations together so it saves time.

Threads are implemented in the form of objects.

↳ The run() and start() are two inbuilt methods which helps to thread implementation.

↳ The run() method can be initiating by the calling of start() method.

Thread is a class found in java.lang package.

Thread can be Created in two ways:

(i) By extending Thread class

(ii) By implementing Runnable interface.

1. By Extending Thread class

```
Class Multi extends Thread
{
public void run()
{
System.out.println("thread is running");
}
public static void main(String args[])
{
Multi t1 = new Multi(); // object initiated
t1.start(); // run() method called through
}}                    start()
```

output: thread is running

2. By implementing Runnable interface

Example

```
Class multi implements Runnable
{
public void run()
{
System.out.println("thread is running");
}
public static void main(String args[])
{
multi m1 = new multi();
Thread t1 = new Thread(m1);
t1.start();
}}
```

output: thread is running.

Multithreading is a proless of executing multiple threads concurrently. A thread is a lightweight process, and java supports multithreating to make better use of cpu resources.

Benefits

→ faster execution by performing multiple tasks simultaneously
→ Better utilization of cpu resources.