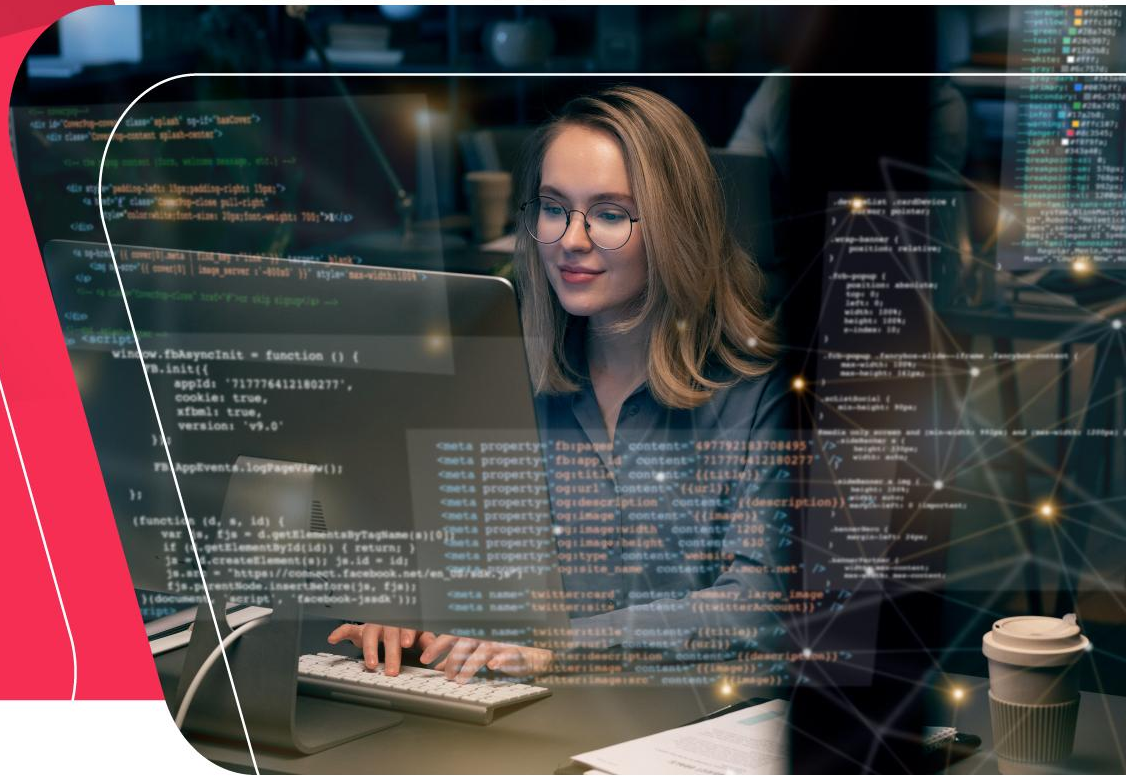
TIA-Academy

Java Bootcamp

Day 19



Technologies will be Use

- JDK 8/**11**/15
- JRE 8/**11**/15
- Writing Code using Notepad++ (For Windows) or Vim (For Linux and Mac)
- Compiling with Command Prompt (For Windows) or Terminal (For Linux and Mac)
- ***Forbidden of using IDE Based Development Tools as of Intellij IDEA or Eclipse***

TIA-Academy

Regular Expressions

Regular Expressions (Regex)

- The **Java Regex** or Regular Expression is an API to *define pattern for searching or manipulating strings*.
- It is widely used to define constraint on strings such as password and email validation. After learning java regex tutorial, you will be able to test your own regular expressions by the Java Regex Tester Tool.
- Java Regex API provides 1 interface and 3 classes in **java.util.regex** package.

java.util.regex package

It provides following classes and interface for regular expressions. The Matcher and Pattern classes are widely used in java regular expression.

1. MatchResult interface
2. Matcher class
3. Pattern class
4. *PatternSyntaxException class*

Matcher class

It implements **MatchResult** interface. It is a *regex engine* i.e. used to perform match operations on a character sequence.

No.	Method	Description
1	boolean matches()	test whether the regular expression matches the pattern.
2	boolean find()	finds the next expression that matches the pattern.
3	boolean find(int start)	finds the next expression that matches the pattern from the given start number.
4	String group()	returns the matched subsequence.
5	int start()	returns the starting index of the matched subsequence.
6	int end()	returns the ending index of the matched subsequence.
7	int groupCount()	returns the total number of the matched subsequence.

Pattern class

It is the *compiled version of a regular expression*. It is used to define a pattern for the regex engine.

No.	Method	Description
1	static Pattern compile(String regex)	compiles the given regex and return the instance of pattern.
2	Matcher matcher(CharSequence input)	creates a matcher that matches the given input with pattern.
3	static boolean matches(String regex, CharSequence input)	It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern.
4	String[] split(CharSequence input)	splits the given input string around matches of given pattern.
5	String pattern()	returns the regex pattern.

Regular Expressions example

```
import java.util.regex.*;

public class RegexExample1{
     public static void main(String args[]){
          //1st way
          Pattern p = Pattern.compile(".s");//. represents singl
          e character
          Matcher m = p.matcher(”as");
          boolean b = m.matches();

          //2nd way
          boolean b2=Pattern.compile(".s").matcher("as").matches
          ();

          //3rd way
          boolean b3 = Pattern.matches(".s", "as");

          System.out.println(b+" "+b2+" "+b3);
     }
}
```

- Output:

```
true true true
```

Regular Expressions example

- The . (dot) represents a single character.

```
import java.util.regex.*;
class RegexExample2{
      public static void main(String args[]){
            //true (2nd char is s)
            System.out.println(Pattern.matches(".s", "as"));

            //false (2nd char is not s)
            System.out.println(Pattern.matches(".s", "mk"));

            //false (has more than 2 char)
            System.out.println(Pattern.matches(".s", "mst"));

            //false (has more than 2 char)
            System.out.println(Pattern.matches(".s", "amms"));

            //true (3rd char is s)
            System.out.println(Pattern.matches("..s", "mas"));
      }
}
```

Regex Character classes

No.	Character Class	Description
1	[abc]	a, b, or c (simple class)
2	[^abc]	Any character except a, b, or c (negation)
3	[a-zA-Z]	a through z or A through Z, inclusive (range)
4	[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
5	[a-z&&[def]]	d, e, or f (intersection)
6	[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
7	[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)

Regex Character classes Example

```
import java.util.regex.*;
class RegexExample3{
     public static void main(String args[]){
          //false (not a or m or n)
          System.out.println(Pattern.matches("[amn]", "abcd"));

          //true (among a or m or n)
          System.out.println(Pattern.matches("[amn]", "a"));

          //false (m and a comes more than once)
          System.out.println(Pattern.matches("[amn]", "ammmna"));
     }
}
```

Regex Quantifiers

Regex	Description
X?	X occurs once or not at all
X+	X occurs once or more times
X*	X occurs zero or more times
X{n}	X occurs n times only
X{n,}	X occurs n or more times
X{y,z}	X occurs at least y times but less than z times

Regex Character classes and Quantifiers Example

```
import java.util.regex.*;

class RegexExample4{
      public static void main(String args[]){
            System.out.println("? quantifier ....");
            System.out.println(Pattern.matches("[amn]?", "a"));//true (a or m or n comes one time)
            System.out.println(Pattern.matches("[amn]?", "aaa"));//false (a comes more than one time)
            System.out.println(Pattern.matches("[amn]?", "aammmnn"));//false (a m and n comes more than one t
            ime)
            System.out.println(Pattern.matches("[amn]?", "aazzta"));//false (a comes more than one time)
            System.out.println(Pattern.matches("[amn]?", "am"));//false (a or m or n must come one time)

            System.out.println("+ quantifier ....");
            System.out.println(Pattern.matches("[amn]+", "a"));//true (a or m or n once or more times)
            System.out.println(Pattern.matches("[amn]+", "aaa"));//true (a comes more than one time)
            System.out.println(Pattern.matches("[amn]+", "aammmnn"));//true (a or m or n comes more than once
            )
            System.out.println(Pattern.matches("[amn]+", "aazzta"));//false (z and t are not matching pattern
            )
```

Regex Metacharacters

The regular expression metacharacters work as a short codes.

Regex	Description
.	Any character (may or may not match terminator)
\d	Any digits, short of [0-9]
\D	Any non-digit, short for [^0-9]
\s	Any whitespace character, short for [\t\n\x0B\f\r]
\S	Any non-whitespace character, short for [^\s]
\w	Any word character, short for [a-zA-Z_0-9]
\W	Any non-word character, short for [^\w]
\b	A word boundary
\B	A non word boundary

Regex Metacharacters Example

```
import java.util.regex.*;

class RegexExample5{
        public static void main(String args[]){
                System.out.println("metacharacters d....");\\d means digit

                System.out.println(Pattern.matches("\\d", "abc"));//false (non-digit)
                System.out.println(Pattern.matches("\\d", "1"));//true (digit and comes once)
                System.out.println(Pattern.matches("\\d", "4443"));//false (digit but comes more than once)
                System.out.println(Pattern.matches("\\d", "323abc"));//false (digit and char)

                System.out.println("metacharacters D....");\\D means non-digit

                System.out.println(Pattern.matches("\\D", "abc"));//false (non-digit but comes more than once)
                System.out.println(Pattern.matches("\\D", "1"));//false (digit)
                System.out.println(Pattern.matches("\\D", "4443"));//false (digit)
                System.out.println(Pattern.matches("\\D", "323abc"));//false (digit and char)
                System.out.println(Pattern.matches("\\D", "m"));//true (non-digit and comes once)

                System.out.println("metacharacters D with quantifier....");
                System.out.println(Pattern.matches("\\D*", "mak"));//true (non-digit and may come 0 or more times)

        }
}
```

TIA-Academy

ASSIGNMENT 01

TIA-Academy

Exception Handling

Exception Handling in Java

- The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.
- In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.
- Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

Advantage of Exception Handling

- The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;      //exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

- Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. **Checked Exception**

   The classes that extend 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 that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3. **Error**

   Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1. Scenario where ArithmeticException occurs

   If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2. Scenario where NullPointerException occurs

   If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
String s=null;

System.out.println(s.length());//NullPointerException
```

Common scenarios where exceptions may occur

3. Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";

int i=Integer.parseInt(s);//NumberFormatException
```

4. Scenario where `ArrayIndexOutOfBoundsException` occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];

a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Exception Handling Keywords

There are 5 **keywords** used in java exception handling.

1. **try**
2. **catch**
3. ***finally***
4. throw
5. throws

try-catch

- Java try block is used to enclose the code that might throw an exception. It must be used within the method.
- Java try block must be followed by either catch or finally block.
- Java catch block is used to handle the Exception. It must be used after the try block only
- You can use multiple catch block with a single try.

- Syntax of java try-catch

```
try{
 //code that may throw exception
}catch(Exception_class_Name ref){}
```

Problem without exception handling

- Let's try to understand the problem if we don't use try-catch block.
- Output :

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

```
public class Testtrycatch1{
  public static void main(String args[]){
      int data=50/0;//may throw exception
      System.out.println("rest of the code...");
  }
}
```

Problem without exception handling

- As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

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

- There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

- Let's see the solution of problem by java try-catch block.
- Output :

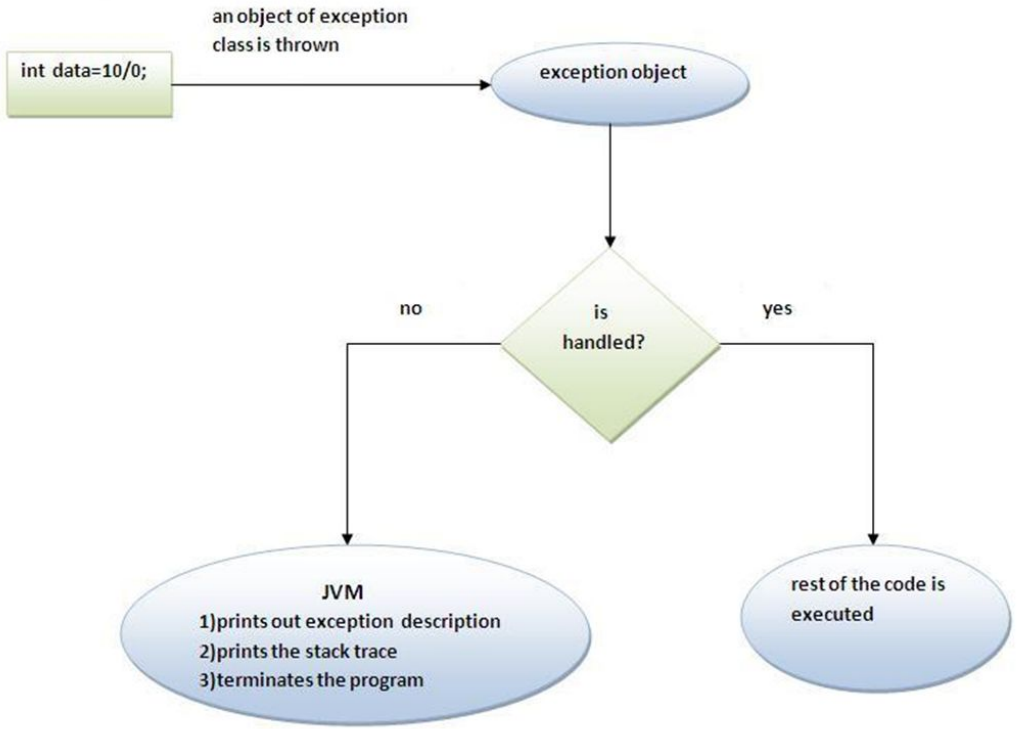
```
Exception in thread main
java.lang.ArithmeticException:/ by
zero
rest of the code...
```

- Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

```
public class Testtrycatch2{
  public static void main(String args[]){
   try{
      int data=50/0;
   }catch(ArithmeticException e){System.out.println(e);}

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

Internal working of java try-catch block



Internal working of java try-catch block

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

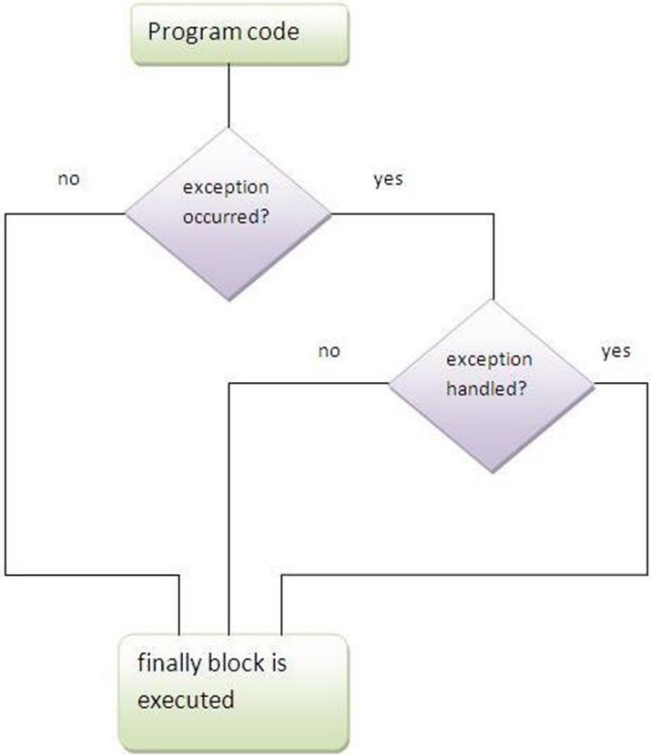
- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

finally block

- Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.
- If you don't handle exception, before terminating the program, JVM executes finally block(if any).
- Finally block in java can be used to put "cleanup" code such as closing a file, closing

finally block



Case 1

- Let's see the java finally example where **exception doesn't occur.**
- Output :

```
5
finally block is always executed
rest of the code...
```

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}

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

Case 2

- Let's see the java finally example where **exception occurs and not handled.**
- Output :

```
finally block is always executed
Exception in thread main
java.lang.ArithmeticException:/ by
zero
```

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e)
      {System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

Case 3

- Let's see the java finally example where **exception occurs and handled.**
- Output :

```
Exception in thread main
java.lang.ArithmeticException:/ by
zero
finally block is always executed
rest of the code...
```

```
public class TestFinallyBlock2{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(ArithmeticException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

finally block

- For each try block there can be zero or more catch blocks, but only one finally block.
- The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

throw exception

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or uncheked exception in java by throw keyword.
- The throw keyword is mainly used to throw custom exception.
- The syntax of java throw keyword is given below.

```
throw exception;
```

- Let's see the example of throw IOException.

```
throw new IOException("sorry device error);
```

throw example

- In this example, we have created the validate method that takes integer value as a parameter.
- If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.
- Output :

```
public class TestThrow1{
   static void validate(int age){
     if(age<18)
      throw new ArithmeticException("not valid");
     else
      System.out.println("welcome to vote");
   }
   public static void main(String args[]){
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

```
Exception in thread main
java.lang.ArithmeticException:not valid
```

throws exception

- The **Java throws keyword** is used to declare an exception.
- It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- Exception Handling is mainly used to handle the checked exceptions.
- If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.
- The syntax of java throws keyword is given below.

```
return_type method_name() throws exception_class_name{
 //method code
}
```

throws exception

Which exception should be declared? checked exception only, because

- **unchecked Exception:** under your control so correct your code.

- **error:** beyond your control e.g. you are unable to do anything if there occurs

  VirtualMachineError or StackOverflowError.

throws example

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}

  }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

- Output :

```
exception handled
normal flow...
```

throws exception

If you are calling a method that declares an exception, you must either caught or declare the exception.

1. **Case1:** You caught the exception i.e. handle the exception using try/catch.

2. **Case2:** You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
public class Testthrows2{
   public static void main(String args[]){
    try{
     M m=new M();
     m.method();
    }catch(Exception e){System.out.println("exception handled");}


    System.out.println("normal flow...");
  }
}
```

- Output :

```
exception handled
normal flow...
```

Case2: You declare the exception

A. In case you declare the exception, if exception does not occur, the code will be executed fine.

B. In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

Program if exception does not occur

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

- Output :

```
device operation performed
normal flow...
```

Program if exception occurs

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

- Output :

```
Runtime Exception
```

Difference between throw and throws in Java

- There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Difference between throw and throws in Java

- **Java throw example**

```
void m(){
 throw new ArithmeticException("sorry");
}
```

- **Java throws example**

```
void m()throws ArithmeticException{
 //method code
}
```

- **Java throw and throws example**

```
void m()throws ArithmeticException{
 throw new ArithmeticException("sorry");
}
```

TIA-Academy

ASSIGNMENT 02

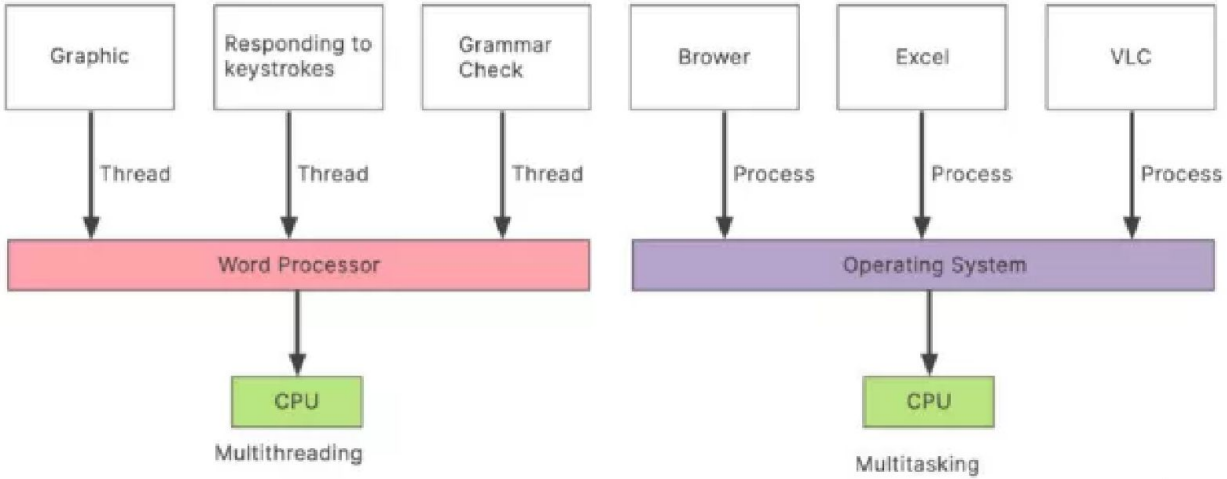
TIA-Academy

Multithreading

Multithreading

- **Multithreading in java** is a process of executing multiple threads simultaneously.

- **Thread** is basically a lightweight sub-process, a smallest unit of processing.



Multithreading

- Multiprocessing and multithreading, both are used to achieve multitasking.
- But we use multithreading than multiprocessing because threads share a common memory area.
- They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Advantages of Multithreading

1. It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2. You **can perform many operations together so it saves time**.

3. Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

1. Process-based Multitasking(Multiprocessing)
2. Thread-based Multitasking(Multithreading)

Process-based Multitasking (Multiprocessing)

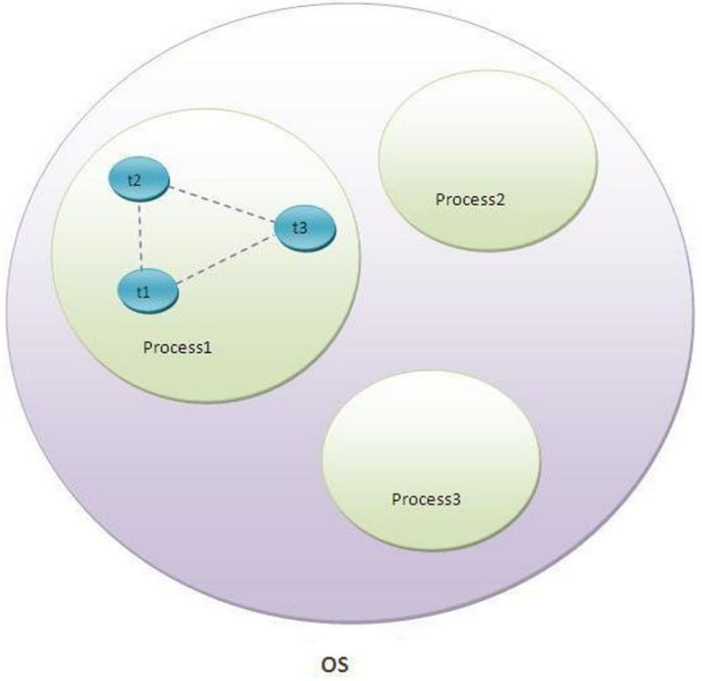
- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.
- At least one process is required for each thread.

What is Thread

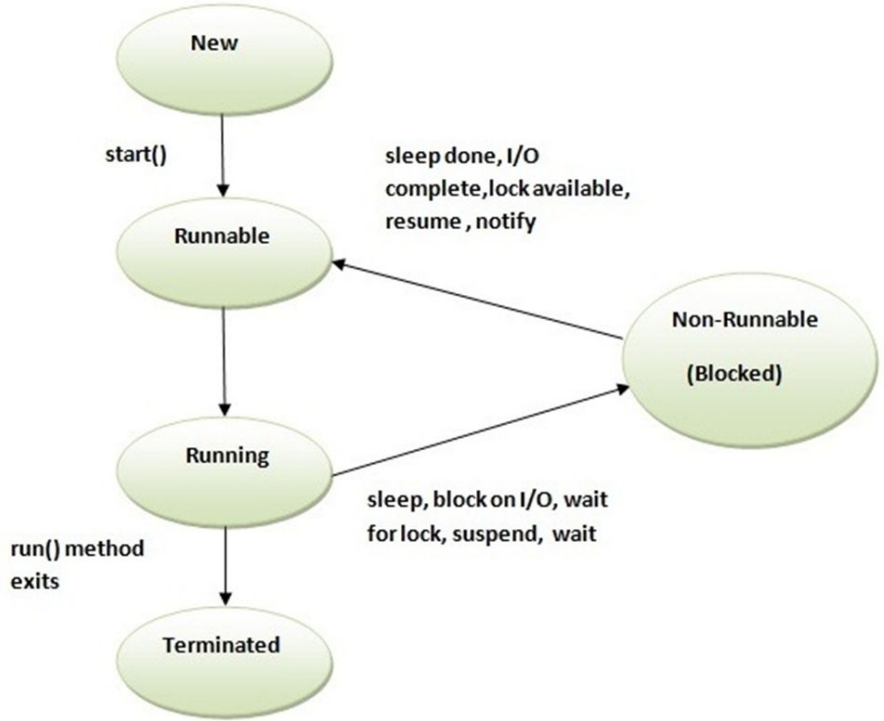
- A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.
- Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.
- As shown in the side figure, thread is executed inside the process. There is context-switching between the threads.
- There can be multiple processes inside the OS and one process can have multiple threads.



Life cycle of a Thread

- A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.
- But for better understanding the threads, we are explaining it in the 5 states.
- The life cycle of the thread in java is controlled by JVM. The java thread states are as follows.
    1. New
    2. Runnable
    3. Running
    4. Non-Runnable (Blocked)
    5. Terminated

Life cycle of a Thread



Life cycle of a Thread

1. New

   The thread is in **new state if you create an instance of Thread class** but before the invocation of start() method.

2. Runnable

   The thread is in runnable state **after invocation of start() method**, but the thread scheduler has not selected it to be the running thread.

3. Running

   The thread is in running state if the **thread scheduler has selected it**.

4. *Non-Runnable (Blocked)*

   *This is the state when the **thread is still alive, but is currently not eligible to run**.*

5. Terminated

   A thread is in terminated or **dead state when its run() method exits**.

How to create thread

There are two ways to create a thread:

- By extending Thread class
- By implementing Runnable interface.

Thread class

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Commonly used methods of Thread class

- **public void run():** is used to perform action for a thread.
- **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
- **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **public void join():** waits for a thread to die.
- **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
- **public int getPriority():** returns the priority of the thread.

Commonly used methods of Thread class

- **public int setPriority(int priority):** changes the priority of the thread.
- **public String getName():** returns the name of the thread.
- **public void setName(String name):** changes the name of the thread.
- **public Thread currentThread():** returns the reference of currently executing thread.
- **public int getId():** returns the id of the thread.
- **public Thread.State getState():** returns the state of the thread.

Commonly used methods of Thread class

- **public boolean isAlive():** tests if the thread is alive.
- **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **public void suspend():** is used to suspend the thread(depricated).
- **public void resume():** is used to resume the suspended thread(depricated).
- **public void stop():** is used to stop the thread(depricated).

Commonly used methods of Thread class

- **public boolean isDaemon():** tests if the thread is a daemon thread.
- **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
- **public void interrupt():** interrupts the thread.
- **public boolean isInterrupted():** tests if the thread has been interrupted.
- **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

Starting a thread

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Thread Example by extending Thread class

- Output :

```
thread is running...
```

```
class Multi extends Thread{
      public void run(){
           System.out.println("thread is running...");
      }
      public static void main(String args[]){
            Multi t1=new Multi();
            t1.start();
            Multi t2=new Multi();
            t2.start();
      }
}
```

Thread Example by implementing Runnable interface

- Output :

```
thread is running...
```

```
class Multi3 implements Runnable{
      public void run(){
            System.out.println("thread is running...");
      }
      public static void main(String args[]){
            Multi3 m1=new Multi3();
            Thread t1 =new Thread(m1);
            t1.start();
      }
}
```

Sleep method

- The sleep() method of Thread class is used to sleep a thread for the specified amount of time.
- The Thread class provides two methods for sleeping a thread:
  - public static void sleep(long miliseconds)throws InterruptedException
  - public static void sleep(long miliseconds, int nanos)throws InterruptedException

Example of sleep method

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();

  t1.start();
  t2.start();
 }
}
```

```
class TestSleepMethod2{
  public static void main(String args[]){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
```

Example of sleep method

- As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread schedular picks up another thread and so on.

Output:

```
1
1
2
2
3
3
4
4
```

TIA-Academy

ASSIGNMENT 03

TIA-Academy

Threadpool

Background

- Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks.
- An approach for building a server application would be to create a new thread each time a request arrives and service this new request in the newly created thread.
- While this approach seems simple to implement, it has significant disadvantages. A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.
- Since active threads consume system resources, a JVM creating too many threads at the same time can cause the system to run out of memory. This necessitates the need to limit the number of threads being created.

What is ThreadPool in Java?

- **A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.**
- Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

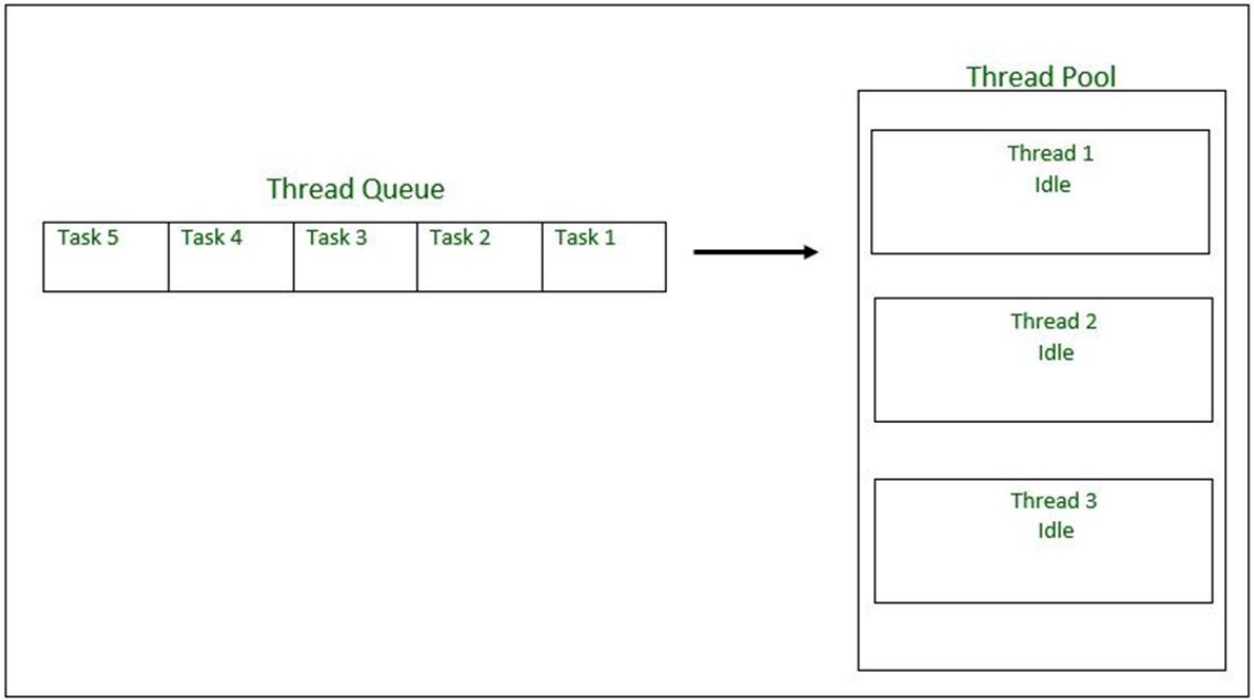
What is ThreadPool in Java?

- Java provides the Executor framework which is centered around the Executor interface, its sub-interface –**ExecutorService** and the class-**ThreadPoolExecutor**, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.

- They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics.

- To use thread pools, we first create a object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.

What is ThreadPool in Java?



Executor Thread Pool Methods

Method	Description
newFixedThreadPool(int)	Creates a fixed size thread pool.
newCachedThreadPool()	Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available
newSingleThreadExecutor()	Creates a single thread.

- In case of a fixed thread pool, if all threads are being currently run by the executor then the pending tasks are placed in a queue and are executed when a thread becomes idle.

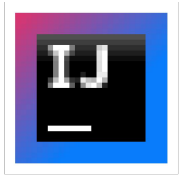
Thread Pool Example

- In the following tutorial, we will look at a basic example of thread pool executor- FixedThreadPool.

- **Steps to be followed**

```
1. Create a task(Runnable Object) to execute
2. Create Executor Pool using Executors
3. Pass tasks to Executor Pool
4. Shutdown the Executor Pool
```

Thread Pool Example

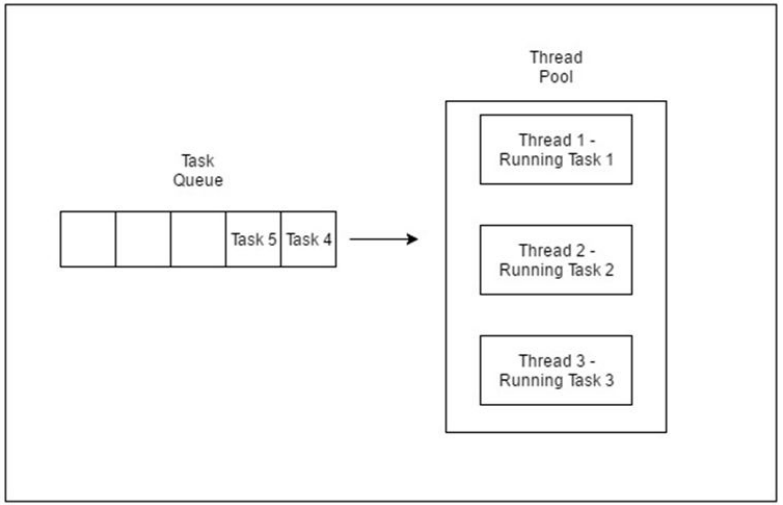


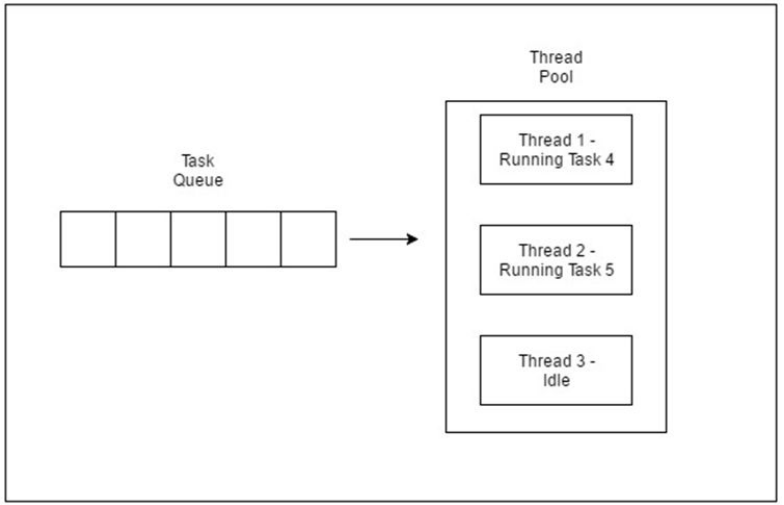
Test.java

```
Output:
Initialization Time for task name - task 2 = 02:32:56
Initialization Time for task name - task 1 = 02:32:56
Initialization Time for task name - task 3 = 02:32:56
Executing Time for task name - task 1 = 02:32:57
Executing Time for task name - task 2 = 02:32:57
Executing Time for task name - task 3 = 02:32:57
Executing Time for task name - task 1 = 02:32:58
Executing Time for task name - task 2 = 02:32:58
Executing Time for task name - task 3 = 02:32:58
Executing Time for task name - task 1 = 02:32:59
Executing Time for task name - task 2 = 02:32:59
Executing Time for task name - task 3 = 02:32:59
Executing Time for task name - task 1 = 02:33:00
Executing Time for task name - task 3 = 02:33:00
Executing Time for task name - task 2 = 02:33:00
Executing Time for task name - task 2 = 02:33:01
Executing Time for task name - task 1 = 02:33:01
Executing Time for task name - task 3 = 02:33:01
task 2 complete
task 1 complete
task 3 complete
Initialization Time for task name - task 5 = 02:33:02
Initialization Time for task name - task 4 = 02:33:02
Executing Time for task name - task 4 = 02:33:03
Executing Time for task name - task 5 = 02:33:03
Executing Time for task name - task 5 = 02:33:04
Executing Time for task name - task 4 = 02:33:04
Executing Time for task name - task 4 = 02:33:05
Executing Time for task name - task 5 = 02:33:05
Executing Time for task name - task 5 = 02:33:06
Executing Time for task name - task 4 = 02:33:06
Executing Time for task name - task 5 = 02:33:07
Executing Time for task name - task 4 = 02:33:07
task 5 complete
task 4 complete
```

Thread Pool Example

- As seen in the execution of the program, the task 4 or task 5 are executed only when a thread in the pool becomes idle. Until then, the extra tasks are placed in a queue.





Risks in using Thread Pools

- **Deadlock :** While deadlock can occur in any multi-threaded program, thread pools introduce another case of deadlock, one in which all the executing threads are waiting for the results from the blocked threads waiting in the queue due to the unavailability of threads for execution.

- **Thread Leakage :**Thread Leakage occurs if a thread is removed from the pool to execute a task but not returned to it when the task completed. As an example, if the thread throws an exception and pool class does not catch this exception, then the thread will simply exit, reducing the size of the thread pool by one. If this repeats many times, then the pool would eventually become empty and no threads would be available to execute other requests.

- **Resource Thrashing :**If the thread pool size is very large then time is wasted in context switching between threads. Having more threads than the optimal number may cause starvation problem leading to resource thrashing as explained.

TIA-Academy

ASSIGNMENT 04
(HOME ASSIGNMENT)

TIA-Academy

Thank You