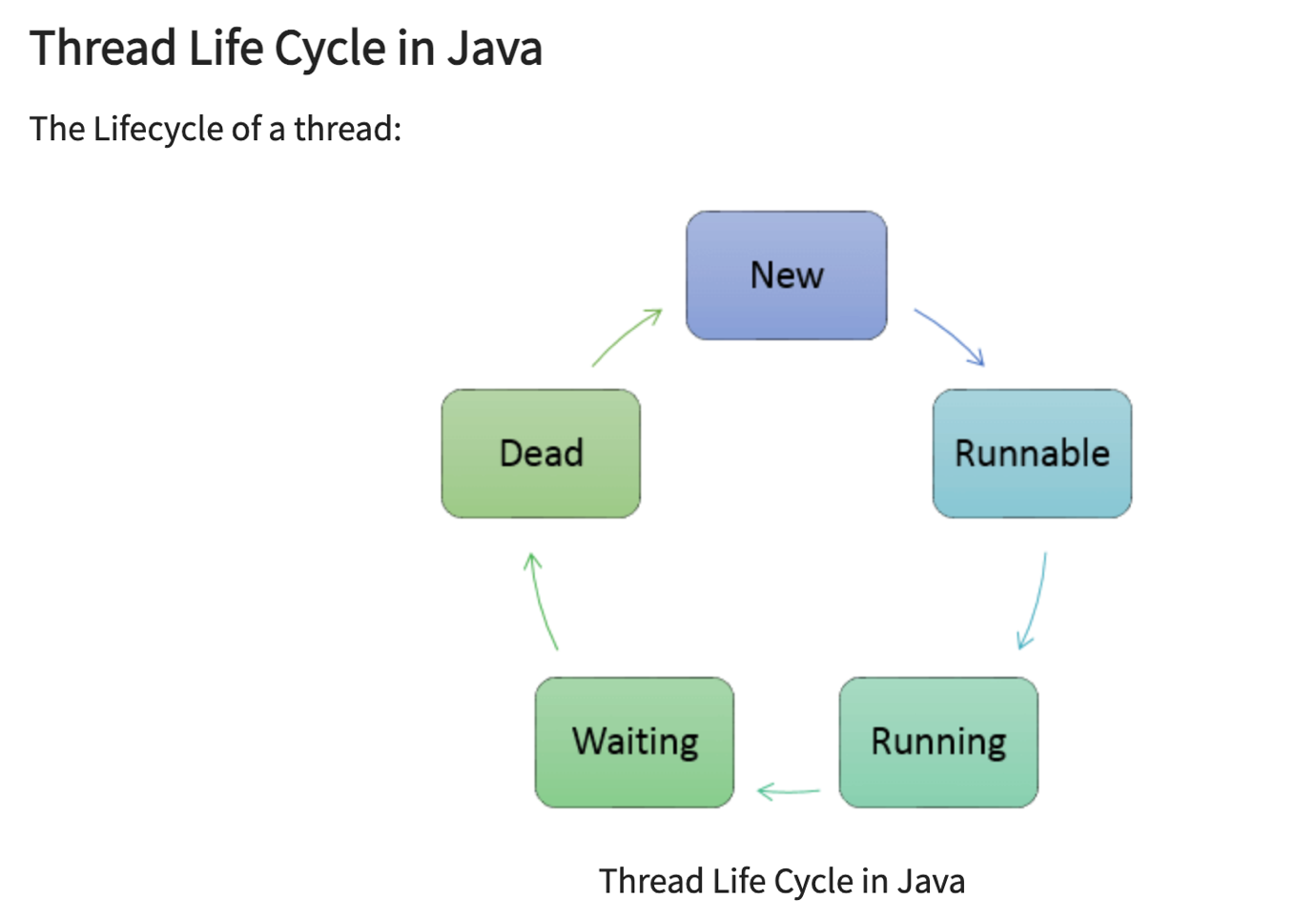
**Multithreading in Java**

A thread is an extremely lightweight process, or the smallest component of the process, that enables software to work more effectively by doing numerous tasks concurrently.

**Thread Life Cycle**



* **New**: In this phase, the thread is created using class “Thread class”.It remains in this state till the program starts the thread. It is also known as born thread.
* **Runnable**: In this page, the instance of the thread is invoked with a start method. The thread control is given to scheduler to finish the execution. It depends on the scheduler, whether to run the thread.
* **Running**: When the thread starts executing, then the state is changed to “running” state. The scheduler selects one thread from the thread pool, and it starts executing in the application.
* **Waitng**: This is the state when a thread has to wait. As there multiple threads are running in the application, there is a need for synchronization between threads. Hence, one thread has to wait, till the other thread gets executed. Therefore, this state is referred as waiting state.
* **Dead**: This is the state when the thread is terminated. The thread is in running state and as soon as it completed processing it is in “dead state”.

**Multithreading** in Java is a process of executing **multiple threads** simultaneously. The main reason for incorporating threads into an application is to **improve its performance**. Games and animations can also be made using threads.

**What is Multithreading in Java?**

* Multithreading is a feature in Java that concurrently executes two or more parts of the program for utilizing the CPU at its maximum. The part of each program is called Thread which is a lightweight process.
* Each Thread operates concurrently and permits the execution of multiple tasks inside the same application.

**How does Java Support Multithreading?**

* Java supports multithreading through its built-in features for creating and managing threads.
* It provides a **Thread** class that can be extended to create custom threads or **Runnable** interface to define tasks for threads. To use it, you can either extend the **Thread** class and override its **run() method** to define the thread's task or **implement the Runnable interface** and pass an instance of the class to the Thread constructor.
* The **start**() method is then called to begin the execution of the thread, which runs concurrently with other threads in the JVM, enabling multithreading capabilities.

**What is the Use of Multi-Thread in Java?**

* Because each Thread is managed individually and several operations can be carried out at once, the user is not blocked.
* It is used to save time as multiple operations are performed concurrently.
* Since threads are independent, other threads don’t get affected even if an exception occurs in a single thread.

**Methods of Multithreading in Java**

The following are methods used for Multithreading in Java.

| **Methods** | **Description** |
| --- | --- |
| void start() | Causes the Thread to begin execution |
| void run() | Utilised to carry out a thread's action. |
| long getId() | Returns the identifier of the thread |
| final String getName() | Returns the threads name |
| final void setName(String name) | Sets the name of the thread |
| final int getPriority() | Returns thread’s priority |
| final void setPriority(int newPriority) | Changes the priority if the thread |
| boolean isAlive() | It checks to see if the Thread is alive. |
| void suspend() | It is used to suspend the Thread. |
| void resume() | The suspended Thread is resumed using it. |
| void stop() | The Thread is stopped using it. |
| void destroy() | The thread group and all of its subgroups are destroyed using it. |
| static void sleep(long millis) throws InterruptedException | Causes the currently executing Thread to sleep (temporarily cease execution) for the specified number of milliseconds. |
| static Thread currentThread() | Returns a reference to the currently executing thread object. |
| static void yield() | A hint to the scheduler that the current Thread is willing to yield its current use of a processor. |
| final void join() throws InterruptedException | Waits for the Thread to die |
| boolean isDaemon() | It checks to see if a thread is a daemon thread. |
| void setDaemon() | It marks the Thread as daemon or user thread. |
| void interrupt() | It interrupts the Thread. |
| boolean isinterrupted() | It checks to see if the Thread has been interrupted. |
| static boolean interrupted() | It checks to see whether the current Thread has been interrupted. |
| static int activeCount() | It gives back how many threads are currently active in the thread group of the current Thread. |
| void checkAccess() | It determines if the currently running Thread has permission to modify the Thread. |
| static boolean holdLock() | It returns true if and only if the current Thread holds the monitor lock on the specified object. |
| static void dumpStack() | It is used to print a stack trace of the current Thread to the standard error stream. |
| StackTraceElement[] getStackTrace() | It returns an array of stack trace elements representing the stack dump of the thread. |
| static int enumerate() | It is used to copy every active Thread's thread group and its subgroup into the specified array. |
| Thread.State getState() | It is used to return the state of the Thread. |
| ThreadGroup getThreadGroup() | It is used to return the thread group to which this Thread belongs. |
| String toString() | It is used to return a string representation of this Thread, including the Thread's name, priority, and thread group. |
| void notify() | It is used to give the notification for only one Thread that is waiting for a particular object. |
| void notifyAll() | It is used to give notification to all waiting threads of a particular object. |

To facilitate thread programming, Java offers the **Thread class**. To generate and manage threads, the Thread class offers constructors and methods. The **Runnable interface** is implemented by the Thread class, which extends the Object class.

**Examples:**

Example:1

 In this example, the class Test implements the class Runnable. The method run() prints the current Thread and its ID. If any exception occurs, then it is caught.

**public** **class** MultithreadingTest {

**public** **static** **void** main(String[] args)

{

**int** n = 6; // Number of threads

// Creating and starting n number of threads

**for** (**int** i = 0; i < n; i++) {

Thread obj

= **new** Thread(**new** Test());

obj.start();

}

}

}

**class** Test **implements** Runnable {

// run method to execute the thread

**public** **void** run()

{

**try** {

// Displaying the running Thread

System.***out***.println(

"Thread " + Thread.*currentThread*().getId()

+ " is running");

}

**catch** (Exception e) {

// exception is caught if occurred

System.***out***.println("Exception has occurred and is caught");

}

}

}

Example 2:

The class RunnableTest implements Runnable interface which has two private variables, Thread t and String threadName. The method run() prints the currently running Thread. After printing the threadName (Thread's name), the Thread sleeps for 50 microseconds.

**public** **class** TestThread {

**public** **static** **void** main(String args[]) {

// Create a new object of RunnableTest class with the name "Thread1"

RunnableTest obj1 = **new** RunnableTest( "Thread1");

// Start the thread with name "Thread1"

obj1.start();

// Create a new object of RunnableTest class with the name "Thread2"

RunnableTest obj2 = **new** RunnableTest( "Thread2");

// Start the thread with name "Thread2"

obj2.start();

}

}

**class** RunnableTest **implements** Runnable {

**private** Thread t;

**private** String threadName;

// Constructor that sets the name of the thread

RunnableTest( String name) {

threadName = name;

System.***out***.println("Created " + threadName );

}

// Method that is executed when the thread is started

**public** **void** run() {

System.***out***.println("Currently Running " + threadName );

**try** {

**for**(**int** i = 4; i > 0; i--) {

System.***out***.println("Thread is: " + threadName + ", " + i);

// Let the thread sleep for a while.

Thread.*sleep*(50);

}

} **catch** (InterruptedException e) {

System.***out***.println("Thread is " + threadName + " interrupted.");

}

System.***out***.println("Thread is " + threadName + " exiting!!!");

}

// Method to start the thread

**public** **void** start () {

System.***out***.println("Starting now " + threadName );

**if** (t == **null**) {

t = **new** Thread (**this**, threadName);

t.start ();

}

}

}

Example 3 : **Program to print Fibonaaci series till a number and printing numbers in reverse order.**

**Without Threads:**

**public** **class** FibonaaciAndPrintNumbersWithoutThread {

**public** **static** **void** main(String[] args) {

FibonacciWot fib = **new** FibonacciWot();

fib.run();

ReverseWoT rev = **new** ReverseWoT();

rev.run();

}

}

**class** FibonacciWot

{

**public** **void** run()

{

**try**

{

**int** a=0, b=1, c=0;

Scanner sc=**new** Scanner(System.***in***);

System.***out***.print("Enter the Limit for fibonacci: ");

**int** n = sc.nextInt();

System.***out***.println("\n=================================");

System.***out***.println("Fibonacci series:");

**while** (n>0)

{

System.***out***.print(c+" ");

a=b;

b=c;

c=a+b;

n=n-1;

}

sc.close();

}

**catch** (Exception ex)

{

ex.printStackTrace();

}

}

}

**class** ReverseWoT

{

**public** **void** run()

{

**try**

{

System.***out***.println("\n=================================");

System.***out***.println("\nReverse is: ");

System.***out***.println("=================================");

**for** (**int** i=10; i >= 1 ;i-- )

{

System.***out***.print(i+" ");

}

System.***out***.println("\n=================================\n\n");

}

**catch** (Exception ex)

{

ex.printStackTrace();

}

}

}

**With Threads:**

public class FibonaaciAndPrintNumbers {

public static void main(String[] args) throws InterruptedException {

Fibonacci fib = new Fibonacci();

fib.start();

fib.*sleep*(4000);

Reverse rev = new Reverse();

rev.start();

}

}

class Fibonacci extends Thread

{

public void run()

{

try

{

int a=0, b=1, c=0;

Scanner sc=new Scanner(System.*in*);

System.*out*.print("Enter the Limit for fibonacci: ");

int n = sc.nextInt();

System.*out*.println("\n=================================");

System.*out*.println("Fibonacci series:");

while (n>0)

{

System.*out*.print(c+" ");

a=b;

b=c;

c=a+b;

n=n-1;

if(n == 10) {

System.*out*.println("Sleeping for 6 seconds in Fib class");

Thread.*sleep*(6000);

}

}

sc.close();

}

catch (Exception ex)

{

ex.printStackTrace();

}

}

}

class Reverse extends Thread

{

public void run()

{

try

{

System.*out*.println("\n=================================");

System.*out*.println("\nReverse is: ");

System.*out*.println("=================================");

for (int i=10; i >= 1 ;i-- )

{

System.*out*.print(i+" ");

if(i == 5) {

System.*out*.println("Sleeping for 10 seconds in Reverse class");

Thread.*sleep*(10000);

}

}

System.*out*.println("\n=================================\n\n");

}

catch (Exception ex)

{

ex.printStackTrace();

}

}

}

**Executor Service:**

package multithreading;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ExecutorServiceDemo {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(3);

for (int i = 0; i < 10; i++) {

Runnable worker = new WorkerThread("" + i);

executor.execute(worker);

}

}

}

class WorkerThread implements Runnable {

private String command;

public WorkerThread(String s){

this.command=s;

}

@Override

public void run() {

System.out.println(Thread.currentThread().getName()+" Start. Command = "+command);

processCommand();

System.out.println(Thread.currentThread().getName()+" End.");

}

private void processCommand() {

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

@Override

public String toString(){

return this.command;

}

}

**Where can thread create a problem and how to solve it?**

**package** multithreading;

**public** **class** BankingExampleDemo {

**public** **static** **void** main(String[] args) {

BankAccount account = **new** BankAccount(100);

**int** slaveCount = 4;

Thread[] slaves = **new** Thread[slaveCount];

**for**(**int** i = 0; i < slaveCount; i++) {

**if** (i % 2 == 0) {

slaves[i] = **new** Producer(account);

} **else** {

slaves[i] = **new** Consumer(account);

}

}

**for**(**int** i = 0; i < slaveCount; i++) {

slaves[i].start();

}

**for**(**int** i = 0; i < slaveCount; i++) {

**try** {

slaves[i].join();

} **catch**(InterruptedException ie) {

System.***err***.println(ie.getMessage());

} **finally** {

System.***out***.println("slave "+ i + " has died");

}

}

System.***out***.print("Closing balance = ");

System.***out***.println("$" + account.balance);

}

}

**class** BankAccount{

**double** balance;

**public** BankAccount(**double** balance) {

**this**.balance=balance;

}

**public** **void** deposit(**double** amt) {

**double** temp = balance;

temp = temp + amt;

**try** {

Thread.*sleep*(300); // simulate production time

} **catch** (InterruptedException ie) {

System.***err***.println(ie.getMessage());

}

System.***out***.println("after deposit balance = $" + temp);

balance = temp;

}

**public** **void** withdraw(**double** amt) {

**if** (balance < amt) {

System.***out***.println("Insufficient funds!");

**return**;

}

**double** temp = balance;

temp = temp - amt;

**try** {

Thread.*sleep*(200); // simulate consumption time

} **catch** (InterruptedException ie) {

System.***err***.println(ie.getMessage());

}

System.***out***.println("after withdrawl balance = $" + temp);

balance = temp;

}

}

**class** Producer **extends** Thread {

**private** BankAccount account;

**public** Producer(BankAccount acct) { account = acct; }

**public** **void** run() {

**for**(**int** i = 0; i < 5; i++) {

account.deposit(10);

}

}

}

**class** Consumer **extends** Thread {

**private** BankAccount account;

**public** Consumer(BankAccount acct) { account = acct; }

**public** **void** run() {

**for**(**int** i = 0; i < 5; i++) {

account.withdraw(10);

}

}

}

**Use synchronized blocks**

**class** Producer **extends** Thread {

**private** BankAccount account;

**public** Producer(BankAccount acct) { account = acct; }

**public** **void** run() {

**for**(**int** i = 0; i < 5; i++) {

**synchronized**(account) {

account.deposit(10);

}

}

}

}

**class** Consumer **extends** Thread {

**private** BankAccount account;

**public** Consumer(BankAccount acct) { account = acct; }

**public** **void** run() {

**for**(**int** i = 0; i < 5; i++) {

**synchronized**(account) {

account.withdraw(10);

}

}

}

}