

Core Java 8 and Development Tools

Lesson 19 : Multithreading

Lesson Objectives

- After completing this lesson, participants will be able to
 - Understanding threads
 - Thread life cycle
 - Scheduling threads- Priorities
 - Controlling threads using `sleep()`, `join()`



Copyright © Capgemini 2015. All Rights Reserved 2

This lesson covers the usage of Thread in application. It explains how to create Thread program and implement multi threading.

Lesson outline:

- 19.1: Understanding Threads.
- 19.2: Thread Life cycle
- 19.3: Scheduling Thread Priorities
- 19.4: Controlling thread using `sleep()` and `join()`

19.1: Understanding Threads?

Thread and Process

■ Thread

- A thread is a single sequential flow of control within a process and it lives within the process
- A light weight process which runs under resources of main process
- Inter process Communication is always slower than intra process communication
- Actual thread execution highly depends on OS and hardware support.
- The JVM of Thread-non-supportive OS takes care of thread execution.
- On single processor, threads may be executed in time sharing manner



Copyright © Capgemini 2015. All Rights Reserved 3

Process Vs Thread :

Understanding Process : A process is nothing but an executing instance of a program which run in separate memory spaces .Processes don't share the same address space and always stored in the main memory . Once the machine is rebooted it disappears .The execution shifts between the tasks of different processes is known as **heavyweight process** .**Therefore** , Inter process communication is always slower

Exp :- Running MicrosoftWord , Notepad , Different instance of Calculator , all works on multiple process .

Understanding Thread : A thread is a single sequential flow of control within a process and it lives within the process . A Java process can be divided into a number of threads (or to say, modules) and each thread is given the responsibility of executing a block of statements . It is termed as a **lightweight process**, since it is similar to a real process but executes within the context of a process and shares the same resources allotted to the process by the kernel. Inter process Communication is always slower than intra process communication. Process are not easily created where threads are easily created .

Basically a process has only one thread of control – one set of machine instructions executing at one time. In some cases , a process may also be made up of multiple threads of execution that execute instructions concurrently.

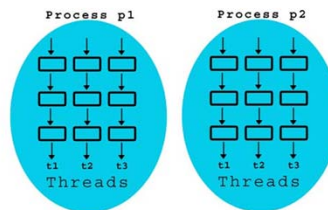
In a uni-processor system, a thread scheduling algorithm is applied and the processor is scheduled to run each thread one at a time.

19.1: Understanding Threads?

Thread Application

■ Applications of Multithreading

- Playing media player while doing some activity on the system like typing a document.
- Transferring data to the printer while typing some file.
- Running animation while system is busy on some other work
- Honoring requests from multiple clients by an application/web server.



19.1: Understanding Threads?

Create Thread using Extending Thread

- Extending Thread class to create threads :
 - Inherited class should:
 - Override the *run()* method.
 - Invoke the *start()* method to start the thread.

```
public class HelloThread extends Thread {  
  
    public void run(){  
        System.out.println("Hello .. Welcome to Capgemini.");  
    }  
    public static void main(String... args) {  
        new HelloThread().start();  
    }  
}
```



Copyright © Capgemini 2015. All Rights Reserved 5

To create threads, your class must extend the Thread class and must override the *run()* method. Call *start()* method to begin execution of the thread. *run()* method defines the code that constitutes a new thread. *run()* can call other methods, use other classes, and declare variables just like the main thread. The only difference is that *run()* establishes the entry point for another, concurrent thread of execution within your program. This ends when *run()* returns.

}

19.1: Understanding Threads?

Creating Thread Implementing Runnable

- Another way to Create Thread.
 - Create a class that implements the runnable interface.
 - Class to implement only the run method that constitutes the new thread

```

public class HelloRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("Hello .. Welcome to Capgemini
        ..");
    }

}

public static void main(String... args){

    HelloRunnable hello = new HelloRunnable();
    Thread helloThread = new Thread(hello);

    helloThread . start();

}

```

Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 6

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

Find below the main class to use HelloRunnable Thread

```

public class HelloMain {

    public static void main(String[] args) {

        HelloRunnable hello = new HelloRunnable();
        Thread helloThread = new Thread(hello);

        helloThread.start();

    }

}

```

19.1: Understanding Threads?


Thread Extending Vs. Implementing

Extending Thread	Implementing Runnable
Basically for creating worker thread.	Basically for defining task.
It itself is a Thread. Simple syntax	Thread object wraps Runnable object
Can not extend any other class	Can extend any other class
A functionality is executed only once on a thread instance.	A functionality can be executed more than once by multiple worker threads.
Concurrent framework does have limited support.	Concurrent framework provide extensive support.
Thread's life cycle methods like interrupt() can be overridden.	Only run() method can be overridden.

19.1: Understanding Threads?

Thread

- Thread API :
 - Thread Class
 - run()
 - start() --- It causes this thread to begin execution; the JVM calls the run method of this thread.
 - sleep()
 - join()
 - stop() (It is a Deprecated method .).
 - getName() - It returns the Thread name in string format.
 - isAlive() -- It returns thread is alive or not .
 - currentThread() - It returns the current Thread object.
 - Runnable Interface
 - run() - This is the only one method available in this interface .
- Common Exceptions in Threads:
 - InterruptedException .
 - IllegalStateException

 CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 8

Threads can be created either of these two ways

Creating a **worker** object of the java.lang.Thread class

Creating the **task** object which is implementing the java.lang.Runnable interface


Using Executor framework which **decouples Task submission from policy of worker threads**


1. **InterruptedException** : It is thrown when the waiting or sleeping state is disrupted by another thread.
2. **IllegalStateException** : It is thrown when a thread is tried to start that is already started.

19.1: Understanding Threads?

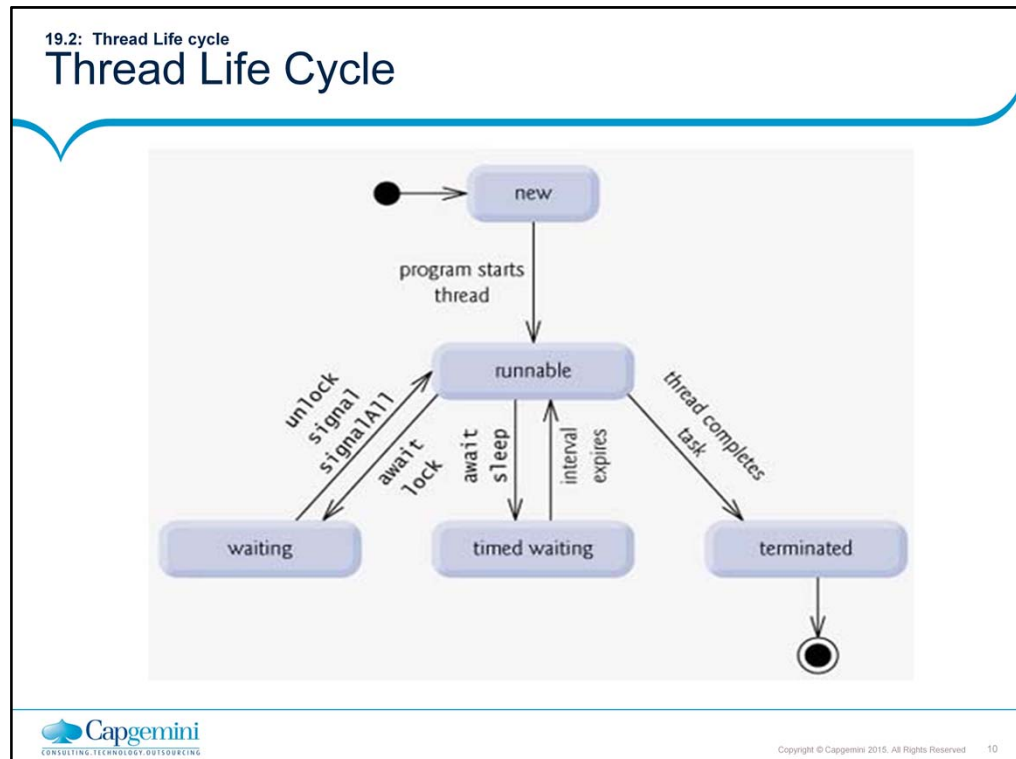
Demo

- HelloThread.java
- HelloRunnable.java



 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 9



Above-mentioned stages are explained here:

•**New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

`Thread thread = new Thread(); // New Stage`

•**Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

`thread.start(); // thread is runnable stage where run() is invoked .`

•**Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

`wait()` is used to send thread to waiting stage and `notify()` invoked by another thread to bring the waiting thread to runnable stage once again. Those Methods belong to Object class basically used with synchronization method in Multi threading program .

•**Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

`sleep()`

•**Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

19.3: Scheduling threads- Priorities

Thread Priorities

- Thread runs in a priority level . Each thread has a priority which is a number starts from 1 to 10 .
Thread Scheduler schedules the threads according to their priority .
- There are three constant defined in Thread class
 - MIN_PRIORITY
 - NORM_PRIORITY
 - MAX_PRIORITY
- Method which is used to set the priority
`setPriority(PRIORITY_LEVEL);`
- Do not rely on thread priorities when you design your multithreaded application .



Copyright © Capgemini 2015. All Rights Reserved 11

One can modify the thread priority using the **setPriority(PRIORITY_LEVEL)** method. Thread Priority level and their constant Values:

MIN_PRIORITY - 1
NORM_PRIORITY - 5
MAX_PRIORITY - 10

Default priority of a thread is always 5 .

Note: Do not rely on thread priorities when designing your multithreaded application. Because thread-scheduling priority behavior is not guaranteed, use thread priorities as a way to improve the efficiency of your program, but just be sure your program doesn't depend on that behavior for correctness.

The thread scheduler is the part of the JVM (although most JVMs map Java threads directly to native threads on the underlying OS) that decides which thread should run at any given moment, and also takes threads *out* of the run state. Assuming a single processor machine, only one thread can actually *run* at a time. Only one stack can ever be executing at one time. And it's the thread scheduler that decides *which* of the thread is eligible for the next execution.


Any thread in the *runnable* state can be chosen by the scheduler to be the one and only running thread. If a thread is not in a runnable state, then it cannot be chosen to be the *currently running* thread.


The order in which runnable threads are chosen to run is not guaranteed. Although *queue* behavior is typical, it isn't guaranteed. Queue behavior means that when a thread has finished with its "turn," it moves to the end of the line of the runnable pool and waits until it eventually gets to the front of the line, where it can be chosen again. In fact, you call it a *runnable pool*, rather than a *runnable queue*, to help reinforce the fact that threads aren't all lined up in some guaranteed order. Although you do not control the thread scheduler .

19.3: Scheduling threads- Priorities

Demo

- ThreadPriorityDemo



 Capgemini
CONSULTING TECHNOLOGY BUSINESS

Copyright © Capgemini 2015. All Rights Reserved 12

19.4: Controlling threads using sleep().join()

Controlling thread using sleep()

- The Thread class provides two methods for sleeping a thread:
 - public static void sleep(long milliseconds)throws InterruptedException
 - public static void sleep(long milliseconds, int nanos)throws InterruptedException

```
public static void sleep(long milliseconds)throws InterruptedException :
```

The sleep(long millis) method of Thread class causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers. if any thread has interrupted the current thread. The current thread interrupted status is cleared when this exception is thrown.

```
public static void sleep(long milliseconds, int nanos)throws InterruptedException :
```

The above method implementation causes the currently executing thread to sleep (temporarily pause execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers

Example given below :---

```
package com.capgemini.lesson20;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class SleepDemo {

    public static void main(String args[]) {
        List<String> seasonList = new ArrayList<>();
        seasonList = Arrays.asList(new String[]{
            "Winter",
            "Summer",
            "Spring",
            "Autumn"
        });

        for (String value : seasonList) {
            //Pause for 4 seconds
            try {
                Thread.sleep(4000);
            } catch (InterruptedException exp) {


                System.err.println(exp.getMessage());
            }

            //Print a message
            System.out.println(value);
        }
    }
}
```

19.2: Thread Life cycle

Demo

- ThreadLifeCycleDemo.java

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 15

```
public class ThreadLifeCycleDemo extends Thread {

    public void run(){

        System.out.println("In side run() Thread is alive or not "+this.isAlive());

        int num = 0;

        while (num < 4) {

            num++;

            System.out.println("num = " + num);

            try {

                sleep(500);
                System.out.println("In not runnable stage, Thread is alive or not "+this.isAlive());

            } catch (InterruptedException exp) {

                System.err.println("Thread Interrupted ...");
            }
        }

        public static void main(String[] args) {

            Thread myThread = new ThreadLifeCycleDemo();

            System.out.println("Before Runnable stage Thread is alive or not : "+myThread.isAlive());
            myThread.start();

            try{
                myThread.sleep(4000);
            }
            catch(InterruptedException exp){
                System.err.println("Thread is interrupted !");
            }

            //myThread.stop();
            System.out.println("After complete execution of Thread ,it is alive or not "+myThread.isAlive());
        }

    }
}
```


19.4: Controlling threads using sleep(),join()

Controlling Thread using join()

- In Thread join method can be used to pause the current thread execution until unless the specified thread is dead.
- There are three overloaded join functions.
 - **public final void join()**
 - **public final synchronized void join(long millis)**
 - **public final synchronized void join(long millis, int nanos)**



Copyright © Capgemini 2015. All Rights Reserved 17

public final void join() : This method puts the current thread on wait until the thread on which it's called is dead. If the thread is interrupted, it throws InterruptedException.

public final synchronized void join(long millis) : This method is used to wait for the thread on which it's called to be dead or wait for a specified milliseconds. Since thread execution depends on OS implementation, one can't guarantee that the current thread will wait only for given time .

public final synchronized void join(long millis, int nanos): This method is used to wait for thread to die for given milliseconds plus nanoseconds.

19.4: Controlling threads using sleep(),join()

Demo :

- ThreadJoinDemo . Java
- SleepDemo.java

```
public class ThreadJoinDemo {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Thread thread1 = new Thread(new MyRunnable(), "First");
        Thread thread2 = new Thread(new MyRunnable(), "Second");
        Thread thread3 = new Thread(new MyRunnable(), "Third");

        thread1.start();

        //start second thread after waiting for 10 seconds or if it's dead
        try {
            thread1.join(10000);
        } catch (InterruptedException exp) {
            System.err.println(exp.getMessage());
        }

        thread2.start();

        //start third thread only when first thread is dead
        try {
            thread1.join();
        } catch (InterruptedException exp) {
            System.err.println(exp.getMessage());
        }

        thread3.start();

        //let all threads finish execution before finishing main thread
        try {
            thread1.join();
            thread2.join();
            thread3.join();
        } catch (InterruptedException exp) {
            System.err.println(exp.getMessage());
        }

        System.out.println("All threads are dead, exiting main thread");

    }

}
```

Lab :Multithreading

- Lab 13: Multithreading



Summary

- In this lesson, you have learnt the following:
 - What is Thread and use of Multithread
 - how to create a Thread program and Lifecycle?
 - Thread Priorities Implementation in Multi Threading environment .
 - Use of sleep() , join()



Add the notes here.

Review Question

- Question 1 which method is invoked to send thread object to runnable stage.
 - **Option 1** : start()
 - **Option 2** : stop()
- Question 2: Does sleep() belongs to Thread class ? True/False.

