
Core Java 8
Lesson 16 : Multithreading
Capgemini

## Lesson Objectives

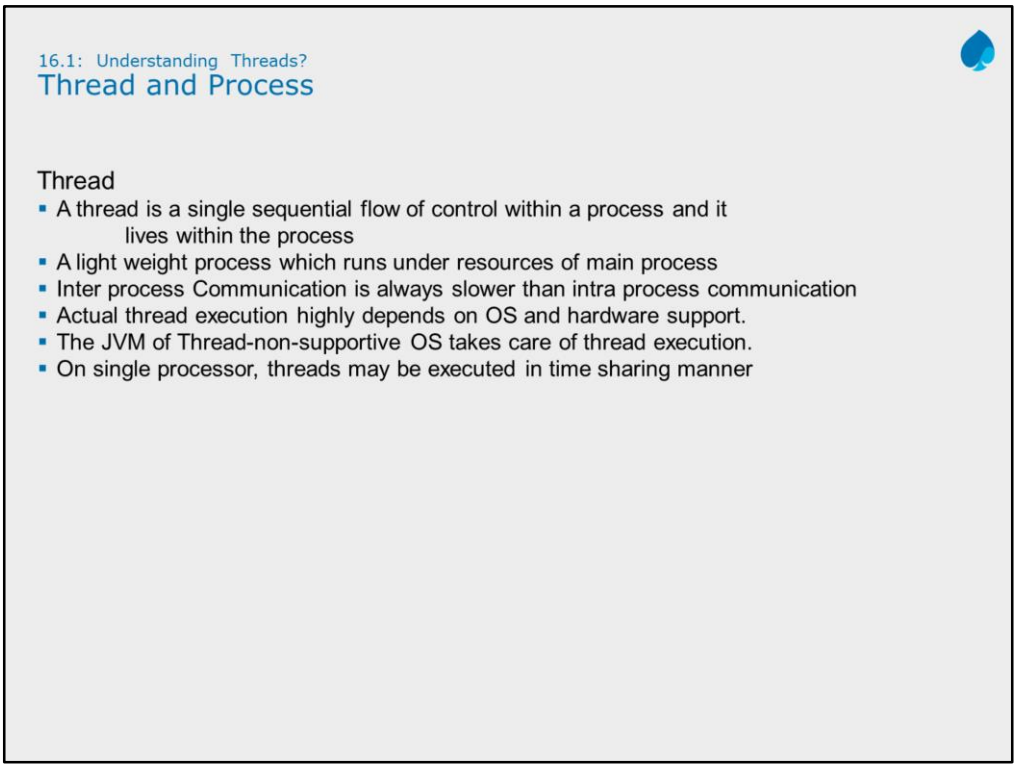
After completing this lesson, participants will be able to

- Understanding threads
- Thread life cycle
- Scheduling threads- Priorities
- Controlling threads using sleep(),join()
- Synchronization concept
- Inter Thread Communications
- Implementations of wait(),notify(),notifyAll() in Producer Consumer problem

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

Lesson outline:

16.1: Understanding Threads.
16.2: Thread Life cycle
16.3: Scheduling Thread Priorities
16.4: Controlling thread using sleep() and join()
16.5:Synchronization
16.6: Inter Thread Communication
16.7 : Producer Consumer Problem




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.

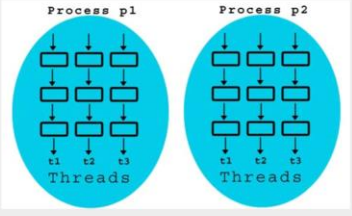
In a uni-processor system, a thread scheduling algorithm is applied and the processor is scheduled to run each thread one at a time.

16.1: 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.



16.1: 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();
        }
}
```

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.

}

16.1: 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();
}
```

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.

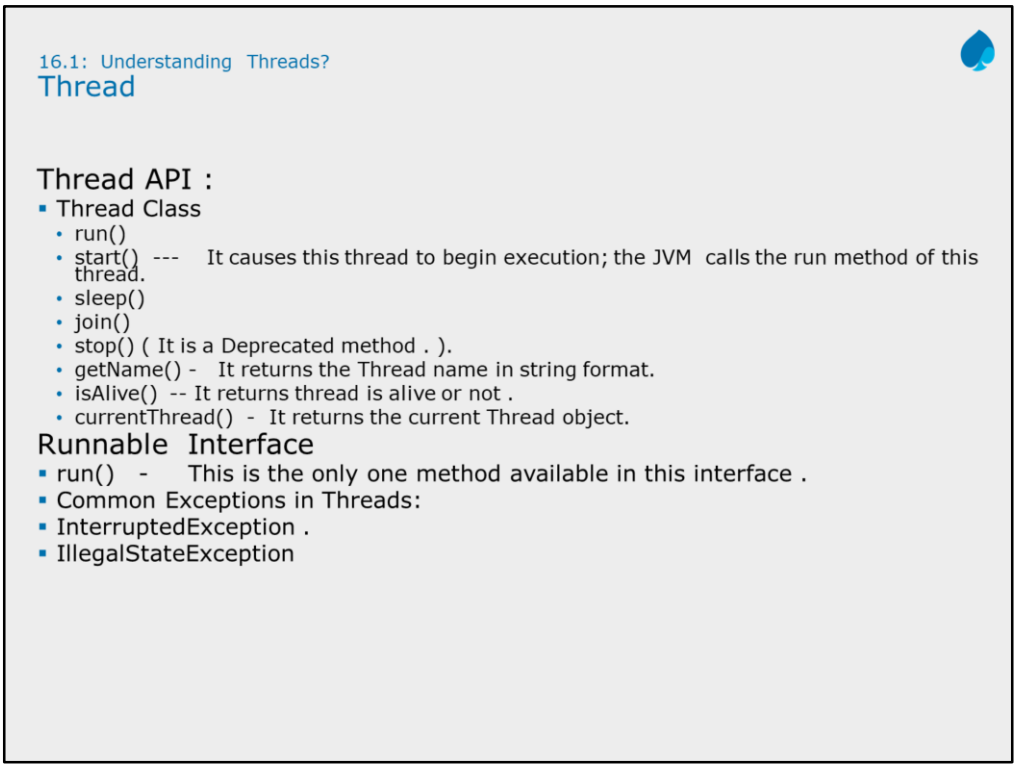
**Find below the main class to use HelloRunnable Thread**

```
public class HelloMain {

public static void main(String[] args) {


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

helloThread.start();

}

}
```

16.1: 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. |




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**

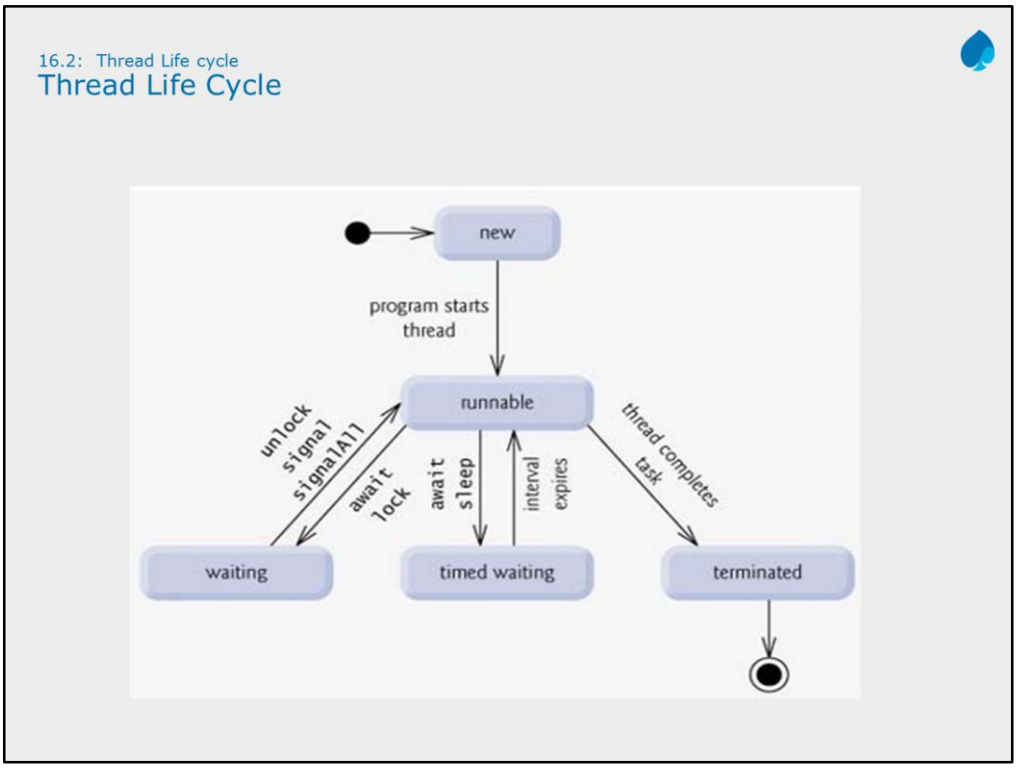
1. **InterruptedException :** It is thrown when the waiting or sleeping state is disrupted by another thread.

2. **IllegalStateException :** It is thrown when a thread is tried to start that is already started.

16.1: Understanding Threads?

## Demo

HelloThread.java
HelloRunnable.java




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 .
```

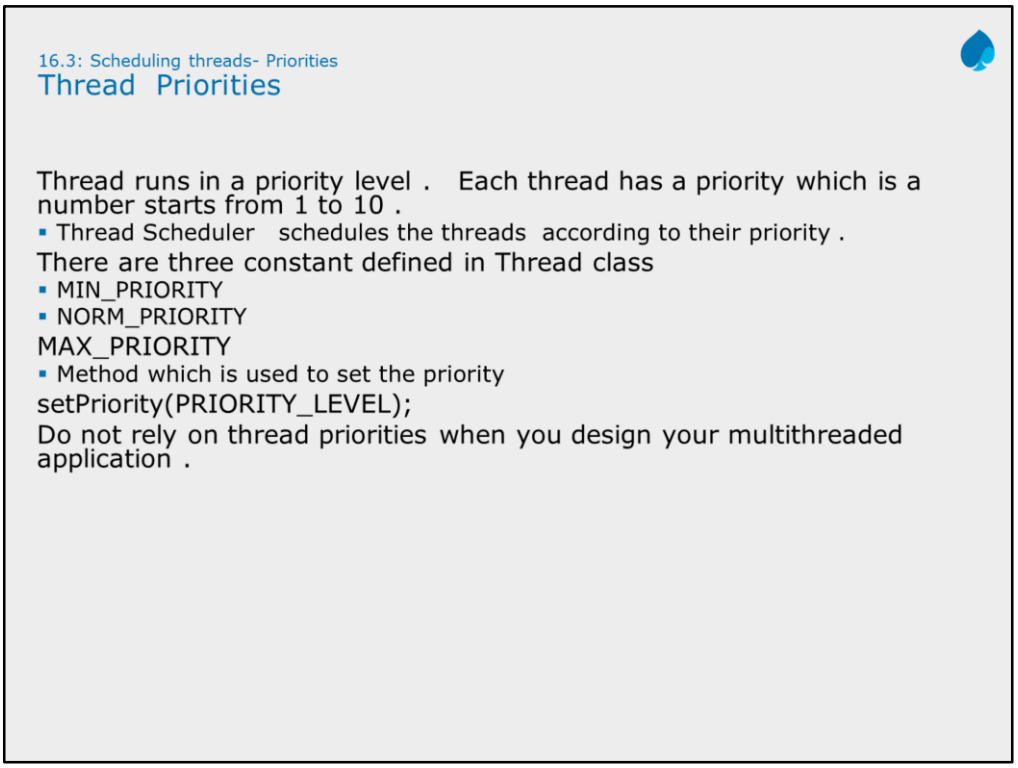
•**Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

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.




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 .

16.3: Scheduling threads- Priorities

## Demo

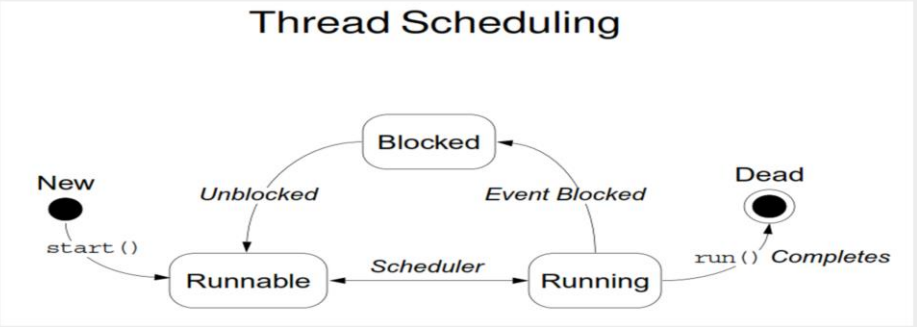
ThreadPriorityDemo

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

## Threads in Blocked Stage

Number of ways a thread can be brought to blocked state.

1. By calling sleep() method
2. By calling suspend() method
3. By calling wait() method




**Number of ways a thread can be brought to blocked state**

A normal running thread can be brought, a number of ways, into blocked state, listed hereunder.

1. By calling sleep ( ) method.
2. By calling suspend ( ) method.
3. When wait() method is called as in synchronization
4. When an I/O operation is performed, the thread is implicitly blocked by JVM.

16.4: 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 :---**

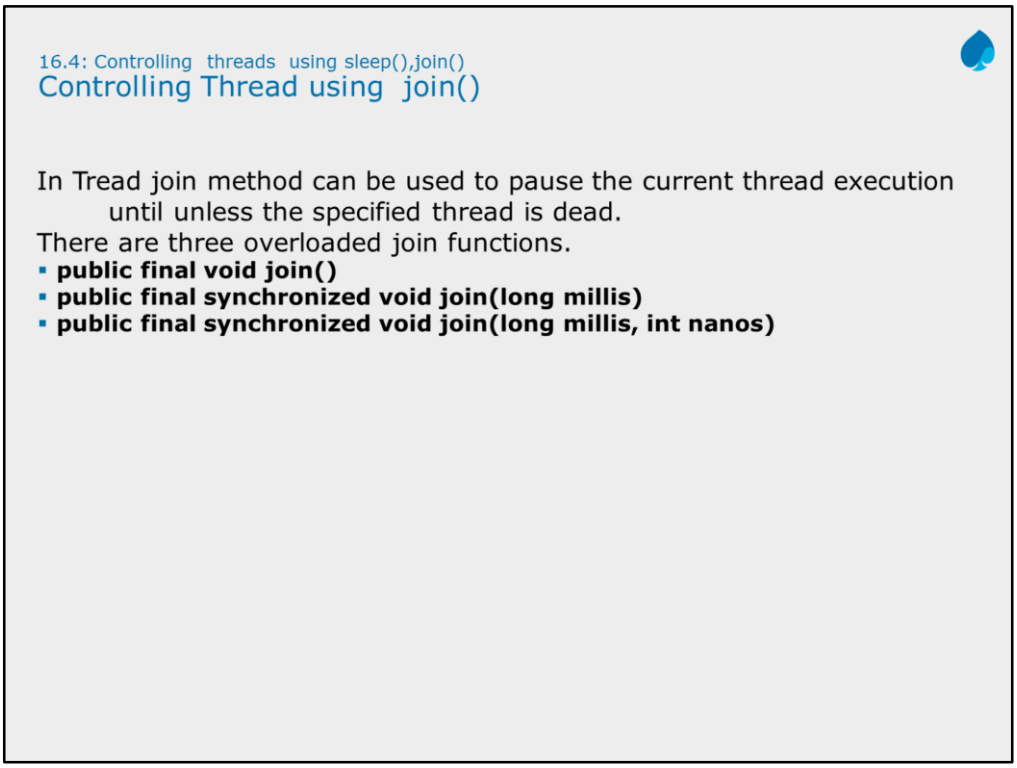
```
package com.capgemini.lesson16;

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);
       }
    }


}
```

16.2: Thread Life cycle

## Demo

ThreadLifeCycleDemo.java


```
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());
}


}
```




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 .

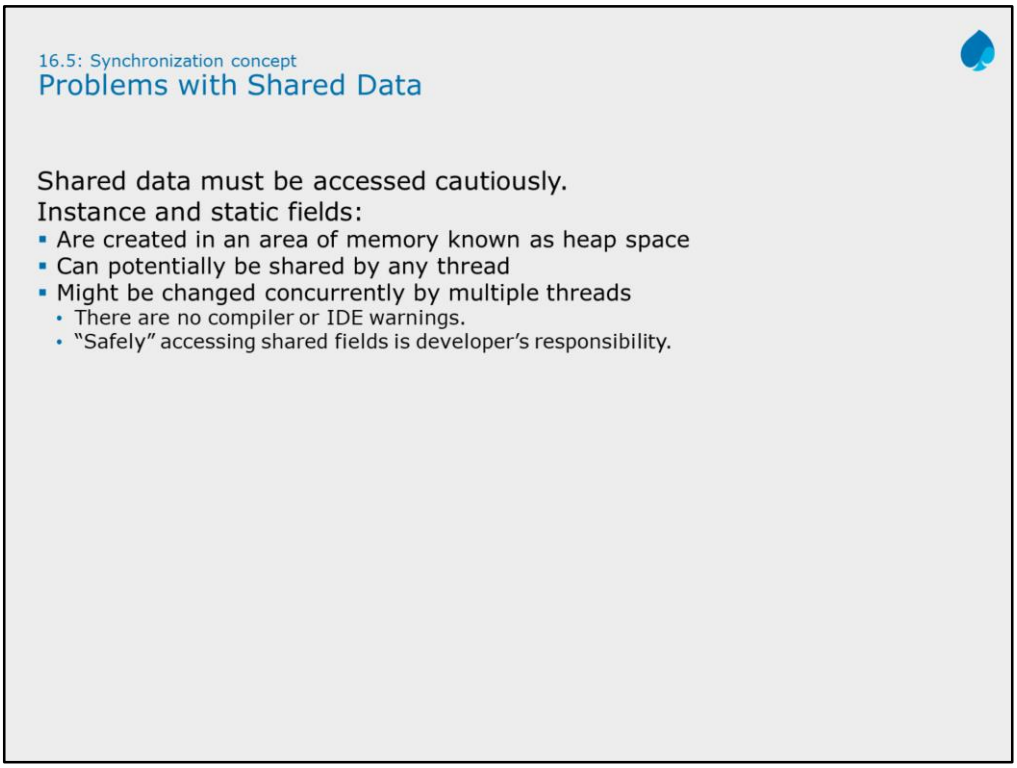
**public final synchronized void join(long millis, int nanos)**: This method is used to wait for thread to die for given milliseconds plus nanoseconds.

16.4: 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");


    }

}
```




Nonshared Data

Some variable types are never shared. The following types are always thread-safe:

- Local variables
- Method parameters

Shared Thread-Safe Data

Any shared data that is immutable, such as `String` objects or final fields, are thread-safe because they can only be read and not written.

16.5: Synchronization concept

## Multithreading without Synchronization

Demo

## Without Synchronization Demo

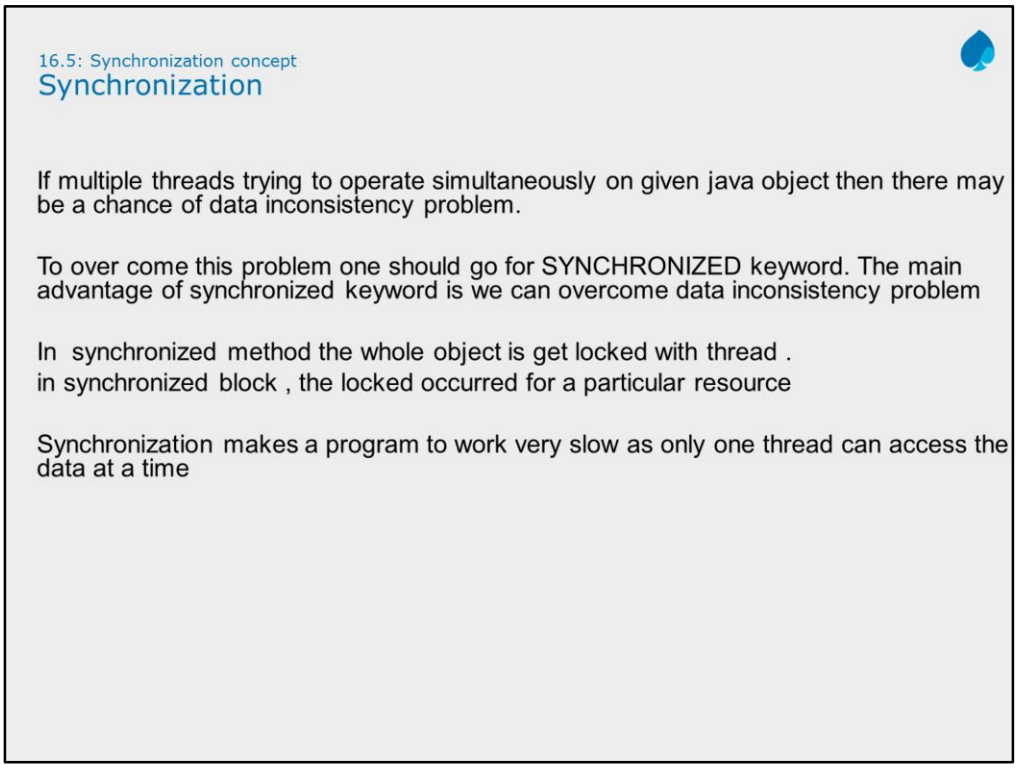
```
public class Display {
  public void wish(String name) {
    for (int i = 0; i < 10; i++) {
      System.out.print("Hello ");
      try {
        Thread.sleep(2000);
      } catch (InterruptedException ie) {
        ie.printStackTrace();
      }
      System.out.println(name);
    }
  }
}
```

```
public class MyThread extends Thread {
  private Display display;
  private String name;
  public MyThread(Display display,String name) {
    this.display=display;
    this.name=name;
  }
  public void run(){
    display.wish(name);
  }
}
```


```
public class Client {
    public static void main(String[] args) {
        Display display=new Display();
        MyThread myThread1=new MyThread(display, "Dhoni");
        MyThread myThread2=new MyThread(display, "Kohli");
        myThread1.start();
        myThread2.start();
    }
}
```

## Without Synchronization Demo output

```
Hello Hello Kohli
Hello Dhoni
Hello Dhoni
Hello Kohli
Hello Dhoni
Hello Kohli
Hello Kohli
Hello Dhoni
Hello Kohli
Dhoni
Hello Hello Dhoni
Kohli
Hello Hello Dhoni
Hello Kohli
Hello Kohli
Hello Dhoni
Hello Kohli
Dhoni
Hello Hello Dhoni
Kohli
```




If multiple threads trying to operate simultaneously on given java object then there may be a chance of data inconsistency problem. To over come this problem one should go for SYNCHRONIZED keyword.

If a method or block declared as synchronized only one thread is allowed to operate on the given object.

Synchronization can be implemented by

- **Synchronized method**
- **Synchronized block**

- In synchronized method the whole object is get locked with thread where in synchronized block , the locked occurred for a particular resource .

- The main advantage of synchronized keyword is we can overcome data inconsistency problem.

- Synchronization makes a program to work very slow as only one thread can access the data at a time. Designers advice to synchronize only the important and critical statements of the code. Do not synchronize unnecessarily.

But the main disadvantage of synchronized keyword is it increases waiting time of threads and creates performance problem. Hence unless it is a specific requirement we do not use synchronization.

16.5: Synchronization concept

## Object Monitor Locking

### Every object in Java has a lock

Using synchronization enables the lock and allows only one thread to access that part of code.

Each object in Java is associated with a monitor, which a thread can lock or unlock.

This is one of the (happen before )mechanism which is implemented in multithreading environment, for top control the access of multiple threads to any shared resource. If we want only one thread should access the resource at a time in parallel environment we use synchronization .

It is used to prevent thread interference and to avoid consistency problem .

16.5: Synchronization concept

## Object Monitor Locking

Each object in Java is associated with a monitor, which a thread can lock or unlock.

- `synchronized` methods use the monitor for the `this` object.
- `static synchronized` methods use the classes' monitor.
- `synchronized` blocks must specify which object's monitor to lock or unlock and it can be nested .

```
synchronized ( this ) {

}
```

This is one of the (happen before )mechanism which is implemented in multithreading environment, for top control the access of multiple threads to any shared resource. If we want only one thread should access the resource at a time in parallel environment we use synchronization .

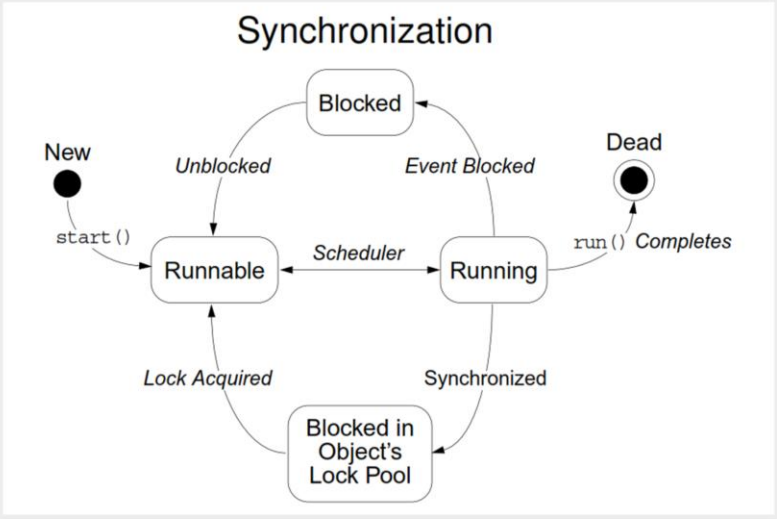
It is used to prevent thread interference and to avoid consistency problem .

Nested `synchronized` Blocks

A thread can lock multiple monitors simultaneously by using nested `synchronized` blocks.

16.5: Synchronization concept

## Synchronization



## Synchronized Method

The synchronized keyword in the method declaration. This tells Java that the method is synchronized.
A synchronized instance method in Java is synchronized on the instance (object) owning the method.

```
public synchronized void add(int value){
    this.count += value;
 }
```

Synchronized methods are an elegant variation on a time-tested model of interprocess-synchronization: the monitor

The monitor is a thread control mechanism

When a thread enters a monitor (synchronized method), all other threads, that are waiting for the monitor of same object, must wait until that thread exits the monitor

The monitor acts as a concurrency control mechanism

16.5: Synchronization concept

## Synchronized Method

Implementation of concurrency control mechanism is very simple because every Java object has its own implicit monitor associated with it.

If a thread wants to enter an object's monitor, it has to just call the synchronized method of that object

While a thread is executing a synchronized method, all other threads that are trying to invoke that particular synchronized method or any other synchronized method of the same object, will have to wait

If a thread wants to execute any synchronized method on a given object, once it has to get the lock. Then it is allowed to execute any synchronized method on that object.

Once synchronized method execution completed then lock is released automatically by thread.


Demo

16.5: Synchronization concept

## With Synchronization – Static Method Level

The synchronized keyword in the method declaration. This tells Java that the method is synchronized.
Synchronized static methods are synchronized on the class object of the class where the synchronized static method present.

```
public static synchronized void add(int value){
    count += value;
 }
```

- Synchronized static methods are synchronized on the class object of the class the synchronized static method belongs to. Since only one class object exists in the Java VM per class, only one thread can execute inside a static synchronized method in the same class.
- If the static synchronized methods are located in different classes, then one thread can execute inside the static synchronized methods of each class. One thread per class regardless of which static synchronized method it calls.

## With Synchronization – Method Level Demo

```
public class Display {
  public synchronized void wish(String name) {
    for (int i = 0; i < 10; i++) {
      System.out.print("Hello ");
      try {
        Thread.sleep(2000);
      } catch (InterruptedException ie) {
        ie.printStackTrace();
      }
      System.out.println(name);
    }
  }
}
```

```
public class MyThread extends Thread {
  private Display display;
  private String name;
  public MyThread(Display display,String name) {
    this.display=display;
    this.name=name;
  }
  public void run(){
    display.wish(name);
  }
}
```


```
public class Client {
    public static void main(String[] args) {
        Display display=new Display();
        MyThread myThread1=new MyThread(display, "Dhoni");
        MyThread myThread2=new MyThread(display, "Kohli");
        myThread1.start();
        myThread2.start();
    }
}
```

## With Synchronization – Method Level Demo output

```
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Dhoni
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
Hello Kohli
```

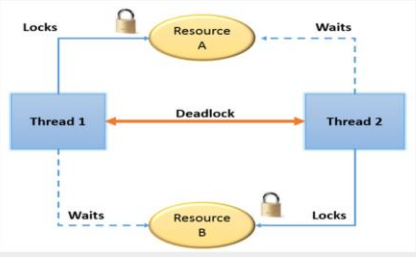
16.6: Inter thread communication

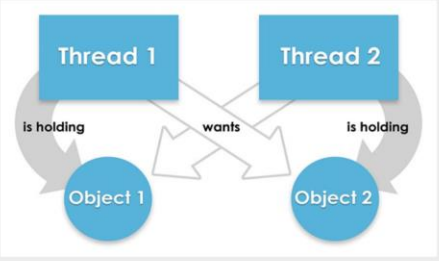
## Deadlock

Deadlock results when two or more threads are blocked forever, waiting for each other.

A deadlock has the below characteristics

- It is two threads , each waiting for a lock from the other.
- It is not detected or avoided




Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

**Starvation**

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by “greedy” threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

**Livelock**

A thread often acts in response to the action of another thread. If the other thread’s action is also a response to the action of another thread, *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked; they are simply too busy responding to each other to resume work.

16.6: Inter thread communication

## Inter-Thread Communication

When more than one thread uses a shared resource they need to synchronize with each other.

While using a shared resource the threads need to communicate with each other, to get the expected behaviour of the application.

Java provides some methods for the threads to communicate

16.7: Implementations of wait(),notify(),notifyAll()

## Inter-Thread Communication

| ***Methods for Interthread Communication*** |
|---|
| `public final void wait()` |
| Causes this thread to wait until some other thread calls the *notify* or *notifyAll* method on this object. May throw *InterruptedException*. |
| `public final void notify()` |
| Wakes up a thread that called the *wait* method on the same object. |
| `public final void notifyAll()` |
| Wakes up all threads that called the *wait* method on the same object. |

16.7: Implementations of wait(),notify(),notifyAll()

## Inter-Thread Communication

Threads are often interdependent - one thread depends on another thread to complete an operation, or to service a request.

The words wait and notify encapsulate the two central concepts to thread communication
- A thread waits for some condition or event to occur.
- You notify a waiting thread that a condition or event has occurred.

To avoid polling, Java's elegant inter-thread communication mechanism uses:
- wait( )
- notify( ), and notifyAll( )

16.7: Implementations of wait(),notify(),notifyAll()

## Inter-Thread Communication (Contd.).

wait( ), notify( ) and notifyAll( ) are:
- Declared as final in Object
- Hence, these methods are available to all classes
- These methods can only be called from a synchronized context

wait( ) directs the calling thread to surrender the monitor, and go to sleep until some other thread enters the monitor of the same object, and calls notify( )

notify( ) wakes up the other thread which was waiting on the same object(*that had called wait() previously on the same object*)

16.7: Implementations of wait(),notify(),notifyAll()

## Producer-Consumer Problem

Producing thread may write to buffer (shared memory)
Consuming thread reads from buffer
If not synchronized, data can become corrupted

- Producer may write before consumer read last data
  - Data lost
- Consumer may read before producer writes new data
  - Data "doubled"

16.7: Implementations of wait(),notify(),notifyAll()

## Producer-Consumer Problem (Contd.).

**Using synchronization**

- If producer knows that consumer has not read last data, calls wait (awaits a notify command from consumer)
- If consumer knows producer has not updated data, calls wait (awaits notify command from producer)

16.7: Implementations of wait(),notify(),notifyAll()

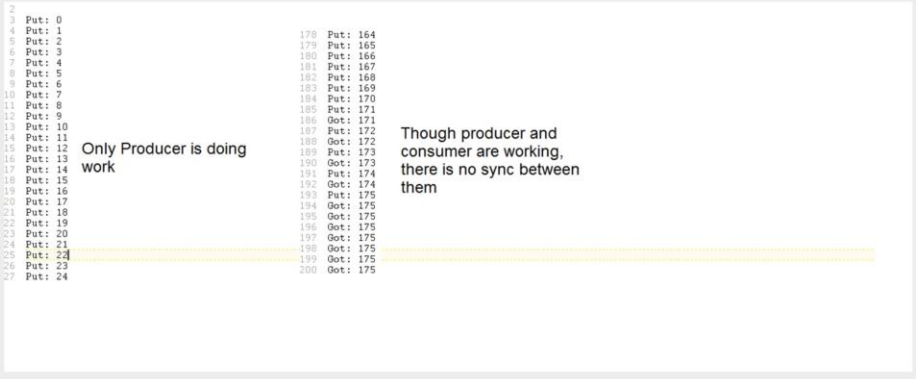
## Producer-Consumer Problem (Contd.).

Incorrect Implementation

```
synchronized int get() {
        System.out.println("Got: " + n);
        return n;
}
synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
}
```

16.7: Implementations of wait(),notify(),notifyAll()

## Producer-Consumer Problem (Contd.).



16.7: Implementations of wait(),notify(),notifyAll()

## Producer-Consumer Problem (Contd.).

### Correct Implementation

```
synchronized int get() {
if(!valueSet)
try {
wait();
} catch(InterruptedException e)
{ System.out.println("InterruptedExce
ption caught");
}
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
```

16.7: Implementations of wait(),notify(),notifyAll()

## Producer-Consumer Problem (Contd.).

```
Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
Put: 6
Got: 6
Put: 7
Got: 7
Put: 8
Got: 8
Put: 9
Got: 9
Put: 10
Got: 10
Put: 11
Got: 11
Put: 12
Got: 12
Put: 13
Got: 13
Put: 14
Got: 14
```

put and get are in synch

16.7: Implementations of wait(),notify(),notifyAll()

## Inter-Thread Communication (Contd.).

The following sample program implements a simple form of the Producer/Consumer problem

It consists of four classes namely:

- Factory , that you are trying to synchronize
- Producer, the threaded object that is producing data
- Consumer, the threaded object that is consuming data
- Main, the class that creates the single Factory, Producer, and Consumer

16.1: Understanding Threads?

## Demo

Factory.java
Producer.java
Consumer.java
Main.java

## Lab

Lab 8: 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()
- Synchronization concept
- Different types of Synchronization
- Inter thread communication
- Producer – consumer problem by using wait(),notify(), notifyAll()


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.**

## Review Question

Question 3 which of the following is not a type of synchronization.

- **Option 1 :** by block
- **Option 2 :** by method()
- **Option 3 :** by variable

Question 4: Which of the following Some variable types are never shared.

- **Option 1:** Local variables**.**
- **Option 2 :** Instance variables
- **Option 3 :** Method parameters